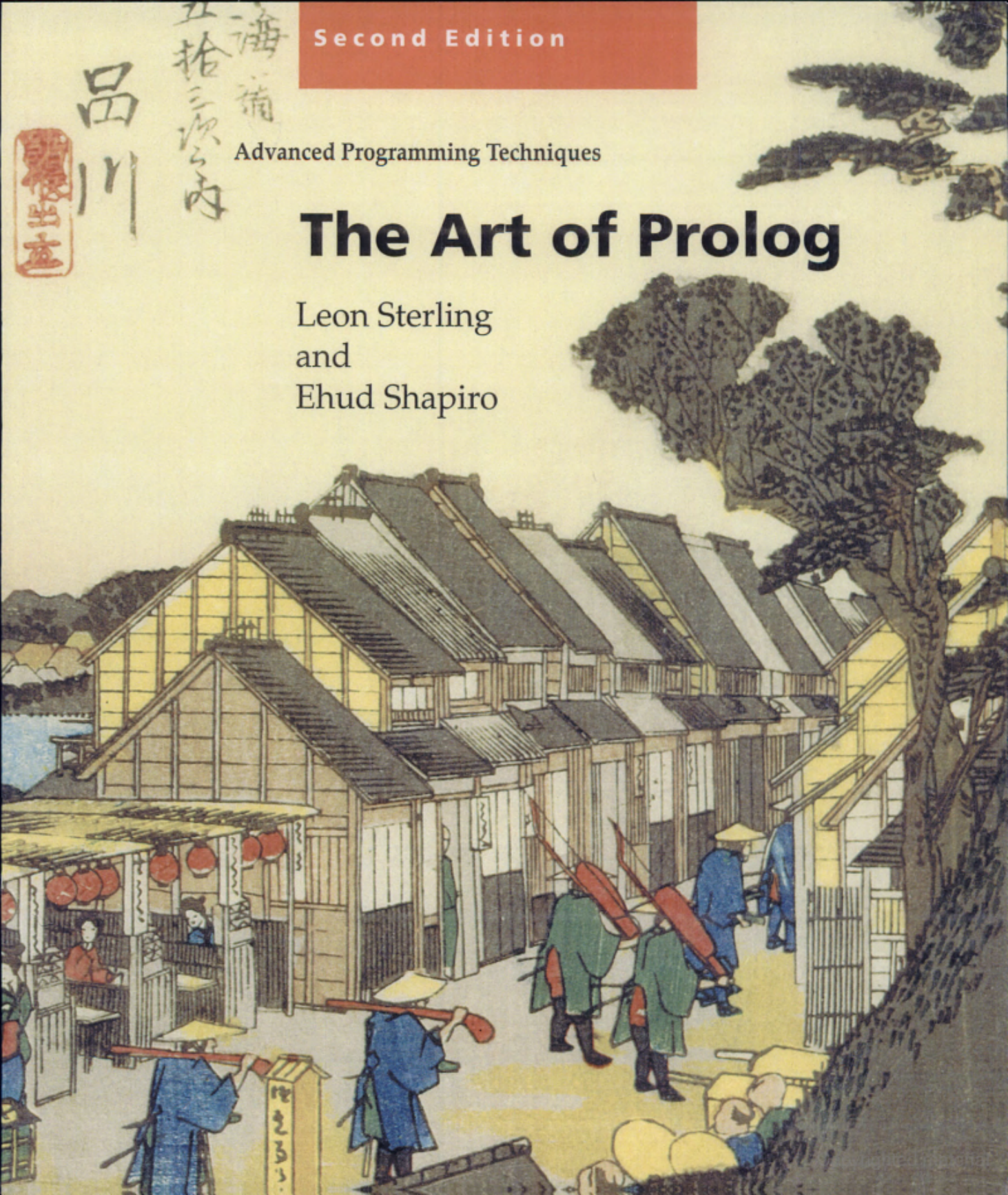
Second Edition

Advanced Programming Techniques

The Art of Prolog

Leon Sterling
and
Ehud Shapiro



Leon Sterling
Ehud Shapiro
with a foreword by David H. D. Warren

The Art of Prolog

Advanced Programming Techniques
Second Edition

The MIT Press
Cambridge, Massachusetts
London, England

This book was composed and typeset by Paul C. Anagnostopoulos and Joe Snowden using ZzTEX. The typeface is Lucida Bright and Lucida New Math created by Charles Bigelow and Kris Holmes specifically for scientific and electronic publishing. The Lucida letterforms have the large x-heights and open interiors that aid legibility in modern printing technology, but also echo some of the rhythms and calligraphic details of lively Renaissance handwriting. Developed in the 1980s and 1990s, the extensive Lucida typeface family includes a wide variety of mathematical and technical symbols designed to harmonize with the text faces.

This book was printed and bound in the United States of America.

Library of Congress Cataloging-in-Publication Data

Sterling, Leon

The art of Prolog : advanced programming techniques / Leon Sterling, Ehud Shapiro ; with a foreword by David H. D. Warren.

p. cm. — (MIT Press series in logic programming)

Includes bibliographical references and index.

ISBN 0-262-19338-8 (hc : alk. paper), 0-262-69163-9 (pb)

1. Prolog (Computer program language) I. Shapiro, Ehud Y. II. Title. III. Series.

QA76.73.P76S74 1994

005.13'3—dc20 93-49494

10 9 8 7 6 CIP

Contents

Programs

Series Foreword

The logic programming approach to computing investigates the use of logic as a programming language and explores computational models based on controlled deduction.

The field of logic programming has seen a tremendous growth in the last several years, both in depth and in scope. This growth is reflected in the number of articles, journals, theses, books, workshops, and conferences devoted to the subject. The MIT Press series in logic programming was created to accommodate this development and to nurture it. It is dedicated to the publication of high-quality textbooks, monographs, collections, and proceedings in logic programming.

Ehud Shapiro
The Weizmann Institute of Science
Rehovot, Israel

Foreword

Programming in Prolog opens the mind to a new way of looking at computing. There is a change of perspective which every Prolog programmer experiences when first getting to know the language.

I shall never forget my first Prolog program. The time was early 1974. I had learned about the abstract idea of logic programming from Bob Kowalski at Edinburgh, although the name "logic programming" had not yet been coined. The main idea was that deduction could be viewed as a form of computation, and that a declarative statement of the form

P if *Q* and *R* and *S*.

could also be interpreted procedurally as

To solve *P*, solve *Q* and *R* and *S*.

Now I had been invited to Marseilles. Here, Alain Colmerauer and his colleagues had devised the language Prolog based on the logic programming concept. Somehow, this realization of the concept seemed to me, at first sight, too simpleminded. However, Gerard Battani and Henri Meloni had implemented a Prolog interpreter in Fortran (their first major exercise in programming, incidentally). Why not give Prolog a try?

I sat at a clattering teletype connected down an ordinary telephone line to an IBM machine far away in Grenoble. I typed in some rules defining how plans could be constructed as sequences of actions. There was one important rule, modeled on the SRI planner Strips, which described how a plan could be elaborated by adding an action at the end. Another rule, necessary for completeness, described how to elaborate a plan by inserting an action in the middle of the plan. As an example for the planner to

work on, I typed in facts about some simple actions in a "blocks world" and an initial state of this world. I entered a description of a goal state to be achieved. Prolog spat back at me:

?

meaning it couldn't find a solution. Could it be that a solution was not deducible from the axioms I had supplied? Ah, yes, I had forgotten to enter some crucial facts. I tried again. Prolog was quiet for a long time and then responded:

DEBORDEMENT DE PILE

Stack overflow! I had run into a loop. Now a loop was conceivable since the space of potential plans to be considered was infinite. However, I had taken advantage of Prolog's procedural semantics to organize the axioms so that shorter plans ought to be generated first. Could something else be wrong? After a lot of head scratching, I finally realized that I had mistyped the names of some variables. I corrected the mistakes, and tried again.

Lo and behold, Prolog responded almost instantly with a correct plan to achieve the goal state. Magic! Declaratively correct axioms had assured a correct result. Deduction was being harnessed before my very eyes to produce effective computation. Declarative programming was truly programming on a higher plane! I had dimly seen the advantages in theory. Now Prolog had made them vividly real in practice. Never had I experienced such ease in getting a complex program coded and running.

Of course, I had taken care to formulate the axioms and organize them in such a way that Prolog could use them effectively. I had a general idea of how the axioms would be used. Nevertheless it was a surprise to see how the axioms got used in practice on particular examples. It was a delightful experience over the next few days to explore how Prolog actually created these plans, to correct one or two more bugs in my facts and rules, and to further refine the program.

Since that time, Prolog systems have improved significantly in terms of debugging environments, speed, and general robustness. The techniques of using Prolog have been more fully explored and are now better understood. And logic programming has blossomed, not least because of its adoption by the Japanese as the central focus of the Fifth Generation project.

After more than a decade of growth of interest in Prolog, it is a great pleasure to see the appearance of this book. Hitherto, knowledge of how to use Prolog for serious programming has largely been communicated by word of mouth. This textbook sets down and explains for the first time in an accessible form the deeper principles and techniques of Prolog programming.

The book is excellent for not only conveying what Prolog is but also explaining how it should be used. The key to understanding how to use Prolog is to properly understand the relationship between Prolog and logic programming. This book takes great care to elucidate the relationship.

Above all, the book conveys the excitement of using Prolog—the thrill of declarative programming. As the authors put it, "Declarative programming clears the mind." Declarative programming enables one to concentrate on the essentials of a problem without getting bogged down in too much operational detail. Programming should be an intellectually rewarding activity. Prolog helps to make it so. Prolog is indeed, as the authors contend, a tool for thinking.

David H. D. Warren
Manchester, England, September 1986

Preface

Seven years have passed since the first edition of *The Art of Prolog* was published. In that time, the perception of Prolog has changed markedly. While not as widely used as the language C, Prolog is no longer regarded as an exotic language. An abundance of books on Prolog have appeared. Prolog is now accepted by many as interesting and useful for certain applications. Articles on Prolog regularly appear in popular magazines. Prolog and logic programming are part of most computer science and engineering programs, although perhaps in a minor role in an artificial intelligence or programming languages class. The first conference on Practical Applications of Prolog was held in London in April 1992. A standard for the language is likely to be in place in 1994. A future for Prolog among the programming languages of the world seems assured.

In preparing for a second edition, we had to address the question of how much to change. I decided to listen to a request not to make the new edition into a new book. This second edition is much like the first, although a number of changes are to be expected in a second edition. The typography of the book has been improved: Program code is now in a distinctive font rather than in italics. Figures such as proof trees and search trees are drawn more consistently. We have taken the opportunity to be more precise with language usage and to remove minor inconsistencies with hyphenation of words and similar details. All known typographical errors have been fixed. The background sections at the end of most chapters have been updated to take into account recent, important research results. The list of references has been expanded considerably. Extra, more advanced exercises, which have been used successfully in my Prolog classes, have been added.

Let us take an overview of the specific changes to each part in turn. Part IV, Applications, is unchanged apart from minor corrections and tidying. Part I, Logic Programs, is essentially unchanged. New programs have been added to Chapter 3 on tree manipulation, including heapifying a binary tree. Extra exercises are also present.

Part II, The Prolog Langauge, is primarily affected by the imminence of a Prolog standard. We have removed all references to Wisdom Prolog in the text in preparation for Standard Prolog. It has proved impossible to guarantee that this book is consistent with the standard. Reaching a standard has been a long, difficult process for the members of the committee. Certain predicates come into favor and then disappear, making it difficult for the authors of a text to know what to write. Furthermore, some of the proposed I/O predicates are not available in current Prologs, so it is impossible to run all the code! Most of the difficulties in reaching a Prolog standard agreeable to all interested parties have been with builtin or system predicates. This book raises some of the issues involved in adding builtins to Prolog but largely avoids the concerns by using pure Prolog as much as possible. We tend not to give detailed explanations of the controversial nonlogical behaviors of some of the system predicates, and we certainly do not use odd features in our code.

Part III, Advanced Programming Techniques, is the most altered in this second edition, which perhaps should be expected. A new chapter has been added on program transformation, and many of the other chapters have been reordered. The chapters on Interpreters and Logic Grammars have extensive additions.

Many people provided us feedback on the first edition, almost all of it very positive. I thank you all. Three people deserve special thanks for taking the trouble to provide long lists of suggestions for improvements and to point out embarrassingly long lists of typos in the first edition: Norbert Fuchs, Harald Søndergaard, and Stanley Selkow. The following deserve mention for pointing out mistakes and typos in the various printings of the first edition or making constructive comments about the book that led to improvements in later printings of the first edition and for this second edition. The list is long, my memory sometimes short, so please forgive me if I forget to mention anyone. Thanks to Hani Assiryani, Tim Boemker, Jim Brand, Bill Braun, Pu Chen, Yves Deville, George Ernst, Claudia Günther, Ann Halloran, Sundar Iyengar, Gary Kacmarcik, Mansoor Khan, Sundeep Kumar, Arun Lakhotia, Jean-

Preface to First Edition

The origins of this book lie in graduate student courses aimed at teaching advanced Prolog programming. A wealth of techniques 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, has 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 computation 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 computation model of Prolog in contrast 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 computation 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 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 Parts 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 percent by homework assignments throughout the course, and 50 percent 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 and Hebrew University of Jerusalem, and in 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 might 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.

Leon Sterling
1986

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 use began. 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 that 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 led to its design, the von Neumann machine seems an arbitrary, even bizarre, device. Thinking in terms of its constrained set of operations is a nontrivial problem, which sometimes stretches the adaptiveness of the human mind to its limits.

These characteristic aspects of programming von Neumann computers led 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 use.

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, time-consuming, and tedious phase of solving a problem using a computer system, is perhaps at the very root 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 relation to or dependence on 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 that 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 that the knowledge about the problem and assumptions 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 prob-

lem, 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, "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 preceding example, 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 twentieth century. They are related to Hilbert's program, to base the entire body of mathematical knowledge on logical foundations and to provide mechanical proofs for its theories, starting from the axioms of logic and set theory alone. It is interesting to note that the failure of this program, from which ensued the incompleteness and undecidability results of Gödel and Turing, 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 a computation 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 are its body. In

addition to the declarative reading of the clause, A is true if the B_i 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, in the early 1970s, Colmerauer and his group at the University of Marseilles-Aix developed a specialized theorem prover, written in Fortran, which they used to implement natural language processing systems. The theorem prover, called Prolog (for Programmation en 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 United States 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. artificial intelligence scientists, when hearing about Prolog, thought that the Europeans were over-excited over what they, the Americans, had 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 1970s 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, still one of the finest implementations of Prolog around, delivered on pure list-processing programs a performance comparable to the best Lisp systems available at the time. Furthermore, the compiler itself was written almost entirely in Prolog, suggesting that classic programming tasks, not just sophisticated AI applications, could benefit from the power of logic programming.

The impact of this implementation cannot be overemphasized. Without it, the accumulated experience that has led 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, openly hostile, or, at best, indifferent to logic programming. By 1980 the number of researchers actively engaged in logic programming were only a few dozen in the United States and about one hundred around the world.

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

Since that time the Prolog language has undergone a rapid transition from adolescence to maturity. There are numerous commercially available Prolog implementations on most computers. A large number of Prolog programming books are directed to different audiences and emphasize 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 vices and virtues. It is time to recognize that, on the one hand, Prolog falls short of the high goals of logic programming but, on the other hand, is a powerful, productive, and practical programming formalism. Given the standard life cycle of computer programming languages, the next few years will reveal whether these properties show their merit only in the classroom or prove useful also in the field, where people pay money to solve problems they care about.

What are the current active subjects of research in logic programming and Prolog? Answers 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*

I Logic Programs

A logic program is a set of axioms, or rules, defining relations between objects. A computation of a logic program is a deduction of consequences of the program. A program defines a set of consequences, which is its meaning. The art of logic programming is constructing concise and elegant programs that have the desired meaning.

1 Basic Constructs

The basic constructs of logic programming, terms and statements, are inherited from logic. There are three basic statements: facts, rules, and queries. There is a single data structure: the logical term.

1.1 Facts

The simplest kind of statement is called a *fact*. Facts are a means of stating that a relation holds between objects. An example is

```
father(abraham,isaac).
```

This fact says that Abraham is the father of Isaac, or that the relation `father` holds between the individuals named `abraham` and `isaac`. Another name for a relation is a *predicate*. Names of individuals are known as *atoms*. Similarly, `plus(2,3,5)` expresses the relation that 2 plus 3 is 5. The familiar `plus` relation can be realized via a set of facts that defines the addition table. An initial segment of the table is

```
plus(0,0,0).    plus(0,1,1).    plus(0,2,2).    plus(0,3,3).
plus(1,0,1).    plus(1,1,2).    plus(1,2,3).    plus(1,3,4).
```

A sufficiently large segment of this table, which happens to be also a legal logic program, will be assumed as the definition of the `plus` relation throughout this chapter.

The syntactic conventions used throughout the book are introduced as needed. The first is the case convention. It is significant that the names

```
father(terach,abraham).
father(terach,nachor).
father(terach,haran).
father(abraham,isaac).
father(haran,lot).
father(haran,milcah).
father(haran,yiscah).

mother(sarah,isaac).
```

```
male(terach).
male(abraham).
male(nachor).
male(haran).
male(isaac).
male(lot).

female(sarah).
female(milcah).
female(yiscah).
```

Program 1.1 A biblical family database

of both predicates and atoms in facts begin with a lowercase letter rather than an uppercase letter.

A finite set of facts constitutes a *program*. This is the simplest form of logic program. A set of facts is also a description of a situation. This insight is the basis of database programming, to be discussed in the next chapter. An example database of family relationships from the Bible is given as Program 1.1. The predicates `father`, `mother`, `male`, and `female` express the obvious relationships.

1.2 Queries

The second form of statement in a logic program is a *query*. Queries are a means of retrieving information from a logic program. A query asks whether a certain relation holds between objects. For example, the query `father(abraham,isaac)?` asks whether the `father` relationship holds between `abraham` and `isaac`. Given the facts of Program 1.1, the answer to this query is *yes*.

Syntactically, queries and facts look the same, but they can be distinguished by the context. When there is a possibility of confusion, a terminating period will indicate a fact, while a terminating question mark will indicate a query. We call the entity without the period or question mark a *goal*. A fact *P*. states that the goal *P* is true. A query *P?* asks whether the goal *P* is true. A *simple query* consists of a single goal.

Answering a query with respect to a program is determining whether the query is a logical consequence of the program. We define logical

consequence incrementally through this chapter. Logical consequences are obtained by applying deduction rules. The simplest rule of deduction is *identity*: from P deduce P. A query is a logical consequence of an identical fact.

Operationally, answering simple queries using a program containing facts like Program 1.1 is straightforward. Search for a fact in the program that implies the query. If a fact identical to the query is found, the answer is *yes*.

The answer *no* is given if a fact identical to the query is not found, because the fact is not a logical consequence of the program. This answer does not reflect on the truth of the query; it merely says that we failed to prove the query from the program. Both the queries `female(abraham)?` and `plus(1,1,2)?` will be answered *no* with respect to Program 1.1.

1.3 The Logical Variable, Substitutions, and Instances

A logical variable stands for an unspecified individual and is used accordingly. Consider its use in queries. Suppose we want to know of whom `abraham` is the father. One way is to ask a series of queries, `father(abraham,lot)?`, `father(abraham,milcah)?`, ..., `father(abraham,isaac)?`, ... until an answer *yes* is given. A variable allows a better way of expressing the query as `father(abraham,X)?`, to which the answer is `X=isaac`. Used in this way, *variables are a means of summarizing many queries*. A query containing a variable asks whether there is a value for the variable that makes the query a logical consequence of the program, as explained later.

Variables in logic programs behave differently from variables in conventional programming languages. They stand for an unspecified but single entity rather than for a store location in memory.

Having introduced variables, we can define a *term*, the single data structure in logic programs. The definition is inductive. Constants and variables are terms. Also compound terms, or structures, are terms. A *compound term* comprises a functor (called the principal functor of the term) and a sequence of one or more arguments, which are terms. A *functor* is characterized by its *name*, which is an atom, and its *arity*, or number of arguments. Syntactically, compound terms have

the form $f(t_1,t_2,\ldots,t_n)$, where the functor has name f and is of arity n, and the t_i are the arguments. Examples of compound terms include `s(0)`, `hot(milk)`, `name(john,doe)`, `list(a,list(b,nil))`, `foo(X)`, and `tree(tree(nil,3,nil),5,R)`.

Queries, goals, and more generally terms where variables do not occur are called *ground*. Where variables do occur, they are called *nonground*. For example, `foo(a,b)` is ground, whereas `bar(X)` is nonground.

Definition
A *substitution* is a finite set (possibly empty) of pairs of the form $X_i = t_i$, where X_i is a variable and t_i is a term, and $X_i \neq X_j$ for every $i \neq j$, and X_i does not occur in t_j, for any i and j. ■

An example of a substitution consisting of a single pair is {`X=isaac`}. Substitutions can be applied to terms. The result of applying a substitution θ to a term A, denoted by $A\theta$, is the term obtained by replacing every occurrence of X by t in A, for every pair $X = t$ in θ.

The result of applying {`X=isaac`} to the term `father(abraham,X)` is the term `father(abraham,isaac)`.

Definition
A is an *instance* of B if there is a substitution θ such that $A = B\theta$. ■

The goal `father(abraham,isaac)` is an instance of `father(abraham, X)` by this definition. Similarly, `mother(sarah,isaac)` is an instance of `mother(X,Y)` under the substitution {`X=sarah,Y=isaac`}.

1.4 Existential Queries

Logically speaking, variables in queries are existentially quantified, which means, intuitively, that the query `father(abraham,X)?` reads: "Does there exist an `X` such that `abraham` is the father of `X`?" More generally, a query $p(T_1,T_2,\ldots,T_n)?$, which contains the variables $X_1,X_2,\ldots,X_k$ reads: "Are there $X_1,X_2,\ldots,X_k$ such that $p(T_1,T_2,\ldots,T_n)$?" For convenience, existential quantification is usually omitted.

The next deduction rule we introduce is *generalization*. An existential query P is a logical consequence of an instance of it, $P\theta$, for any substitution θ. The fact `father(abraham,isaac)` implies that there exists an `X` such that `father(abraham,X)` is true, namely, `X=isaac`.

This is the third deduction rule, called *instantiation*. From a universally quantified statement P, deduce an instance of it, $P\theta$, for any substitution θ.

As for queries, two unspecified objects, denoted by variables, can be constrained to be the same by using the same variable name. The fact `plus(0,X,X)` expresses that `0` is a left identity for addition. It reads that for all values of `X`, `0` plus `X` is `X`. A similar use occurs when translating the English statement "Everybody likes himself" to `likes(X,X)`.

Answering a ground query with a universally quantified fact is straightforward. Search for a fact for which the query is an instance. For example, the answer to `plus(0,2,2)?` is *yes*, based on the fact `plus(0,X,X)`. Answering a nonground query using a nonground fact involves a new definition: a common instance of two terms.

Definition
C is a *common instance* of A and B if it is an instance of A and an instance of B, in other words, if there are substitutions θ_1 and θ_2 such that $C=A\theta_1$ is syntactically identical to $B\theta_2$. ■

For example, the goals `plus(0,3,Y)` and `plus(0,X,X)` have a common instance `plus(0,3,3)`. When the substitution {`Y=3`} is applied to `plus(0,3,Y)` and the substitution {`X=3`} is applied to `plus(0,X,X)`, both yield `plus(0,3,3)`.

In general, to answer a query using a fact, search for a common instance of the query and fact. The answer is the common instance, if one exists. Otherwise the answer is *no*.

Answering an existential query with a universal fact using a common instance involves two logical deductions. The instance is deduced from the fact by the rule of instantiation, and the query is deduced from the instance by the rule of generalization.

1.6 Conjunctive Queries and Shared Variables

An important extension to the queries discussed so far is *conjunctive queries*. Conjunctive queries are a conjunction of goals posed as a query, for example, `father(terach,X),father(X,Y)?` or in general, $Q_1,\ldots,Q_n?$. Simple queries are a special case of conjunctive queries when there is a

We employ this restriction in the meantime to simplify the discussion in the coming sections.

Operationally, to solve the conjunctive query $A_1,A_2,\ldots,A_n?$ using a program P, find a substitution θ such that $A_1\theta$ and ... and $A_n\theta$ are ground instances of facts in P. The same substitution applied to all the goals ensures that instances of variables are common throughout the query. For example, consider the query `father(haran,X),male(X)?` with respect to Program 1.1. Applying the substitution {`X=lot`} to the query gives the ground instance `father(haran,lot),male(lot)?`, which is a consequence of the program.

1.7 Rules

Interesting conjunctive queries are defining relationships in their own right. The query `father(haran,X),male(X)?` is asking for a son of Haran. The query `father(terach,X),father(X,Y)?` is asking about grandchildren of Terach. This brings us to the third and most important statement in logic programming, a rule, which enables us to define new relationships in terms of existing relationships.

Rules are statements of the form:

$A \leftarrow B_1,B_2,\ldots,B_n.$

where $n \geq 0$. The goal A is the *head* of the rule, and the conjunction of goals $B_1,\ldots,B_n$ is the *body* of the rule. Rules, facts, and queries are also called *Horn clauses*, or *clauses* for short. Note that a fact is just a special case of a rule when $n = 0$. Facts are also called *unit clauses*. We also have a special name for clauses with one goal in the body, namely, when $n = 1$. Such a clause is called an *iterative clause*. As for facts, variables appearing in rules are universally quantified, and their scope is the whole rule.

A rule expressing the `son` relationship is

```
son(X,Y) ← father(Y,X), male(X).
```

Similarly one can define a rule for the `daughter` relationship:

```
daughter(X,Y) ← father(Y,X), female(X).
```

A rule for the `grandfather` relationship is

```
grandfather(X,Y) ← father(X,Z), father(Z,Y).
```

Rules can be viewed in two ways. First, they are a means of expressing new or complex queries in terms of simple queries. A query `son(X,haran)?` to the program that contains the preceding rule for `son` is translated to the query `father(haran,X),male(X)?` according to the rule, and solved as before. A new query about the `son` relationship has been built from simple queries involving `father` and `male` relationships. Interpreting rules in this way is their *procedural* reading. The procedural reading for the `grandfather` rule is: "To answer a query *Is X the grandfather of Y?*, answer the conjunctive query *Is X the father of Z and Z the father of Y?*."

The second view of rules comes from interpreting the rule as a logical axiom. The backward arrow ←is used to denote logical implication. The `son` rule reads: "*X* is a son of *Y* if *Y* is the father of *X* and *X* is male." In this view, rules are a means of defining new or complex relationships using other, simpler relationships. The predicate `son` has been defined in terms of the predicates `father` and `male`. The associated reading of the rule is known as the *declarative* reading. The declarative reading of the `grandfather` rule is: "For all *X*, *Y*, and *Z*, *X* is the grandfather of *Y* if *X* is the father of *Z* and *Z* is the father of *Y*."

Although formally all variables in a clause are universally quantified, we will sometimes refer to variables that occur in the body of the clause, but not in its head, as if they are existentially quantified inside the body. For example, the `grandfather` rule can be read: "For all *X* and *Y*, *X* is the grandfather of *Y* if there exists a *Z* such that *X* is the father of *Z* and *Z* is the father of *Y*." The formal justification of this verbal transformation will not be given, and we treat it just as a convenience. Whenever it is a source of confusion, the reader can resort back to the formal reading of a clause, in which all variables are universally quantified from the outside.

To incorporate rules into our framework of logical deduction, we need the law of modus ponens. Modus ponens states that from B and $A \leftarrow B$ we can deduce A.

Definition

The law of *universal modus ponens* says that from the rule

$R = (A \leftarrow B_1, B_2, \ldots, B_n)$

and the facts

B'_1.
B'_2.
⋮
B'_n.

A' can be deduced if

$A' \leftarrow B'_1, B'_2, \ldots, B'_n$

is an instance of R. ∎

Universal modus ponens includes identity and instantiation as special cases.

We are now in a position to give a complete definition of the concept of a logic program and of its associated concept of logical consequence.

Definition
A *logic program* is a finite set of rules. ∎

Definition
An existentially quantified goal G is a logical consequence of a program P if there is a clause in P with a ground instance $A \leftarrow B_1, \ldots, B_n, n \geq 0$ such that $B_1, \ldots, B_n$ are logical consequences of P, and A is an instance of G. ∎

Note that the goal G is a logical consequence of a program P if and only if G can be deduced from P by a finite number of applications of the rule of universal modus ponens.

Consider the query `son(S,haran)?` with respect to Program 1.1 augmented by the rule for `son`. The substitution `{X=lot,Y=haran}` applied to the rule gives the instance `son(lot,haran) ←father(haran,lot), male(lot)`. Both the goals in the body of this rule are facts in Program 1.1. Thus universal modus ponens implies the query with answer `{S=lot}`.

Operationally, answering queries reflects the definition of logical consequence. Guess a ground instance of a goal, and a ground instance of a rule, and recursively answer the conjunctive query corresponding to the body of that rule. To solve a goal A with program P, choose a rule $A_1 \leftarrow B_1, B_2, \ldots, B_n$ in P, and guess substitution θ such that $A = A_1\theta$, and

$B_i\theta$ is ground for $1 \leq i \leq n$. Then recursively solve each $B_i\theta$. This procedure can involve arbitrarily long chains of reasoning. It is difficult in general to guess the correct ground instance and to choose the right rule. We show in Chapter 4 how the guessing of an instance can be removed.

The rule given for `son` is correct but is an incomplete specification of the relationship. For example, we cannot conclude that Isaac is the son of Sarah. What is missing is that a child can be the son of a mother as well as the son of a father. A new rule expressing this relationship can be added, namely,

```
son(X,Y) ← mother(Y,X), male(X).
```

To define the relationship `grandparent` correctly would take four rules to include both cases of `father` and `mother`:

```
grandparent(X,Y) ← father(X,Z), father(Z,Y).
grandparent(X,Y) ← father(X,Z), mother(Z,Y).
grandparent(X,Y) ← mother(X,Z), father(Z,Y).
grandparent(X,Y) ← mother(X,Z), mother(Z,Y).
```

There is a better, more compact, way of expressing these rules. We need to define the auxiliary relationship `parent` as being a father or a mother. Part of the art of logic programming is deciding on what intermediate predicates to define to achieve a complete, elegant axiomatization of a relationship. The rules defining `parent` are straightforward, capturing the definition of a parent being a father or a mother. Logic programs can incorporate alternative definitions, or more technically disjunction, by having alternative rules, as for `parent`:

```
parent(X,Y) ← father(X,Y).
parent(X,Y) ← mother(X,Y).
```

Rules for `son` and `grandparent` are now, respectively,

```
son(X,Y) ← parent(Y,X), male(X).
grandparent(X,Y) ← parent(X,Z), parent(Z,Y).
```

A collection of rules with the same predicate in the head, such as the pair of parent rules, is called a *procedure*. We shall see later that under the operational interpretation of these rules by Prolog, such a collection of rules is indeed the analogue of procedures or subroutines in conventional programming languages.

Input: `son(lot,haran)?` and Program 1.2
Resolvent is `son(lot,haran)`
Resolvent is not empty
choose `son(lot,haran)` (the only choice)
choose `son(lot,haran) ← father(haran,lot), male(lot).`
new resolvent is `father(haran,lot), male(lot)`
Resolvent is not empty
choose `father(haran,lot)`
choose `father(haran,lot).`
new resolvent is `male(lot)`
Resolvent is not empty
choose `male(lot)`
choose `male(lot).`
new resolvent is empty

Output: *yes*

Figure 1.2 Tracing the interpreter

```
father(abraham,isaac).        male(isaac).
father(haran,lot).            male(lot).
father(haran,milcah).         female(milcah).
father(haran,yiscah).         female(yiscah).

son(X,Y) ← father(Y,X), male(X).
daughter(X,Y) ← father(Y,X), female(X).
```

Program 1.2 Biblical family relationships

Definition
A *reduction* of a goal G by a program P is the replacement of G by the body of an instance of a clause in P, whose head is identical to the chosen goal. ■

A reduction is the basic computational step in logic programming. The goal replaced in a reduction is *reduced*, and the new goals are *derived*. In this chapter, we restrict ourselves to *ground reductions*, where the goal and the instance of the clause are ground. Later, in Chapter 4, we consider more general reductions where unification is used to choose the instance of the clause and make the goal to be reduced and the head of the clause identical.

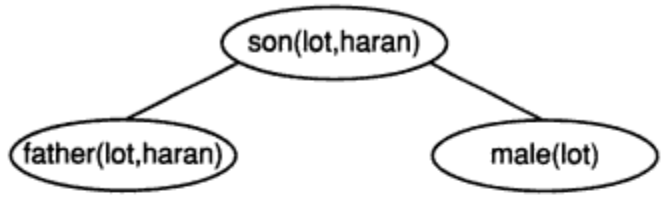


Figure 1.3 A simple proof tree

A trace of a query implicitly contains a proof that the query follows from the program. A more convenient representation of the proof is with a proof tree. A *proof tree* consists of nodes and edges that represent the goals reduced during the computation. The root of the proof tree for a simple query is the query itself. The nodes of the tree are goals that are reduced during the computation. There is a directed edge from a node to each node corresponding to a derived goal of the reduced goal. The proof tree for a conjunctive query is just the collection of proof trees for the individual goals in the conjunction. Figure 1.3 gives a proof tree for the program trace in Figure 1.2.

An important measure provided by proof trees is the number of nodes in the tree. It indicates how many reduction steps are performed in a computation. This measure is used as a basis of comparison between different programs in Chapter 3.

1.9 The Meaning of a Logic Program

How can we know if a logic program says what we wanted it to say? If it is correct, or incorrect? In order to answer such questions, we have to define what is the meaning of a logic program. Once defined, we can examine if the program means what we have intended it to mean.

Definition
The *meaning* of a logic program P, $M(P)$, is the set of ground goals deducible from P. ■

From this definition it follows that the meaning of a logic program composed just of ground facts, such as Program 1.1, is the program itself. In other words, for simple programs, the program "means just what

it says." Consider Program 1.1 augmented with the two rules defining the `parent` relationship. What is its meaning? It contains, in addition to the facts about fathers and mothers, mentioned explicitly in the program, all goals of the form `parent(X,Y)` for every pair `X` and `Y` such that `father(X,Y)` or `mother(X,Y)` is in the program. This example shows that the meaning of a program contains explicitly whatever the program states implicitly.

Assuming that we define the intended meaning of a program also to be a set of ground goals, we can ask what is the relation between the actual and the intended meanings of a program. We can check whether everything the program says is correct, or whether the program says everything we wanted it to say.

Informally, we say that a program is *correct* with respect to some intended meaning M if the meaning of P, $M(P)$, is a subset of M. That is, a correct program does not say things that were not intended. A program is *complete* with respect to M if M is a subset of $M(P)$. That is, a complete program says everything that is intended. It follows that a program P is correct and complete with respect to an intended meaning M if $M = M(P)$.

Throughout the book, when meaningful predicate and constant names are used, the intended meaning of the program is assumed to be the one intuitively implied by the choice of names.

For example, the program for the `son` relationship containing only the first axiom that uses `father` is incomplete with respect to the intuitively understood intended meaning of `son`, since it cannot deduce `son(isaac,sarah)`. If we add to Program 1.1 the rule

```
son(X,Y) ← mother(X,Y), male(Y).
```

it would make the program incorrect with respect to the intended meaning, since it deduces `son(sarah,isaac)`.

The notions of correctness and completeness of a logic program are studied further in Chapter 5.

Although the notion of truth is not defined fully here, we will say that a ground goal is *true* with respect to an intended meaning if it is a member of it, and *false* otherwise. We will say it is simply *true* if it is a member of the intended meaning implied by the names of the predicate and constant symbols appearing in the program.

1.10 Summary

We conclude this section with a summary of the constructs and concepts introduced, filling in the remaining necessary definitions.

The basic structure in logic programs is a term. A *term* is a constant, a variable, or a compound term. Constants denote particular individuals such as integers and atoms, while variables denote a single but unspecified individual. The symbol for an atom can be any sequence of characters, which is quoted if there is possibility of confusion with other symbols (such as variables or integers). Symbols for variables are distinguished by beginning with an uppercase letter.

A *compound term* comprises a functor (called the principal functor of the term) and a sequence of one or more terms called *arguments*. A *functor* is characterized by its *name*, which is an atom, and its *arity* or number of arguments. Constants are considered functors of arity 0. Syntactically, compound terms have the form $f(t_1,t_2,\ldots,t_n)$ where the functor has name f and is of arity n, and the t_i are the arguments. A functor f of arity n is denoted f/n. Functors with the same name but different arities are distinct. Terms are *ground* if they contain no variables; otherwise they are *nonground*. *Goals* are atoms or compound terms, and are generally nonground.

A *substitution* is a finite set (possibly empty) of pairs of the form $X = t$, where X is a variable and t is a term, with no variable on the left-hand side of a pair appearing on the right-hand side of another pair, and no two pairs having the same variable as left-hand side. For any substitution $\theta = \{X_1 = t_1, X_2 = t_2, \ldots, X_n = t_n\}$ and term s, the term $s\theta$ denotes the result of simultaneously replacing in s each occurrence of the variable X_i by t_i, $1 \leq i \leq n$; the term $s\theta$ is called an *instance* of s. More will be said on this restriction on substitutions in the background to Chapter 4.

A *logic program* is a finite set of clauses. A *clause* or *rule* is a universally quantified logical sentence of the form

$$A \leftarrow B_1,B_2,\ldots,B_k. \qquad k \geq 0,$$

where A and the B_i are goals. Such a sentence is read declaratively: "A is implied by the conjunction of the B_i," and is interpreted procedurally "To answer query A, answer the conjunctive query $B_1,B_2,\ldots,B_k$." A is called the clause's *head* and the conjunction of the B_i the clause's *body*. If $k = 0$,

the clause is known as a *fact* or *unit clause* and written $A.$, meaning A is true under the declarative reading, and goal A is satisfied under the procedural interpretation. If $k = 1$, the clause is known as an *iterative clause.*

A *query* is a conjunction of the form

$$A_1,\ldots,A_n? \qquad n > 0,$$

where the A_i are goals. Variables in a query are understood to be existentially quantified.

A *computation* of a logic program P finds an instance of a given query logically deducible from P. A goal G is deducible from a program P if there is an instance A of G where $A \leftarrow B_1,\ldots,B_n$, $n \geq 0$, is a ground instance of a clause in P, and the B_i are deducible from P. Deduction of a goal from an identical fact is a special case.

The *meaning* of a program P is inductively defined using logical deduction. The set of ground instances of facts in P are in the meaning. A ground goal G is in the meaning if there is a ground instance $G \leftarrow B_1,\ldots,B_n$ of a rule in P such that $B_1,\ldots,B_n$ are in the meaning. The meaning consists of the ground instances that are deducible from the program.

An intended meaning M of a program is also a set of ground unit goals. A program P is *correct* with respect to an intended meaning M if $M(P)$ is a subset of M. It is *complete* with respect to M if M is a subset of $M(P)$. Clearly, it is correct and complete with respect to its intended meaning, which is the desired situation, if $M = M(P)$.

A ground goal is *true* with respect to an intended meaning if it is a member of it, and *false* otherwise.

Logical deduction is defined syntactically here, and hence also the meaning of logic programs. In Chapter 5, alternative ways of describing the meaning of logic programs are presented, and their equivalence with the current definition is discussed.

underscores for predicate and function names, for example, `schedule_conflict`.

New relations are built from these basic relationships by defining suitable rules. Appropriate relation schemes for the relationships introduced in the previous chapter are `son(Son,Parent)`, `daughter(Daughter, Parent)`, `parent(Parent,Child)`, and `grandparent(Grandparent, Grandchild)`. From the logical viewpoint, it is unimportant which relationships are defined by facts and which by rules. For example, if the available database consisted of `parent`, `male` and `female` facts, the rules defining `son` and `grandparent` are still correct. New rules must be written for the relationships no longer defined by facts, namely, `father` and `mother`. Suitable rules are

```
father(Dad,Child) ← parent(Dad,Child), male(Dad).
mother(Mum,Child) ← parent(Mum,Child), female(Mum).
```

Interesting rules can be obtained by making relationships explicit that are present in the database only implicitly. For example, since we know the father and mother of a child, we know which couples produced offspring, or to use a Biblical term, procreated. This is not given explicitly in the database, but a simple rule can be written recovering the information. The relation scheme is `procreated(Man,Woman)`.

```
procreated(Man,Woman) ←
  father(Man,Child), mother(Woman,Child).
```

This reads: "`Man` and `Woman` procreated if there is a `Child` such that `Man` is the father of `Child` and `Woman` is the mother of `Child`."

Another example of information that can be recovered from the simple information present is sibling relationships — brothers and sisters. We give a rule for `brother(Brother,Sibling)`.

```
brother(Brother,Sib) ←
  parent(Parent,Brother), parent(Parent,Sib), male(Brother).
```

This reads: "`Brother` is the brother of `Sib` if `Parent` is a parent of both `Brother` and `Sib`, and `Brother` is male."

There is a problem with this definition of brother. The query `brother(X,X)?` is satisfied for any male child `X`, which is not our understanding of the brother relationship.

In order to preclude such cases from the meaning of the program,

Another relationship implicit in the family database is whether a woman is a mother. This is determined by using the `mother/2` relationship. The new relation scheme is `mother(Woman)`, defined by the rule

```
mother(Woman) ← mother(Woman,Child).
```

This reads: "`Woman` is a mother if she is the mother of some `Child`." Note that we have used the same predicate name, `mother`, to describe two different `mother` relationships. The `mother` predicate takes a different number of arguments, i.e., has a different arity, in the two cases. In general, the same predicate name denotes a different relation when it has a different arity.

We change examples, lest the example of family relationships become incestuous, and consider describing simple logical circuits. A circuit can be viewed from two perspectives. The first is the topological layout of the physical components usually described in the circuit diagram. The second is the interaction of functional units. Both views are easily accommodated in a logic program. The circuit diagram is represented by a collection of facts, while rules describe the functional components.

Program 2.2 is a database giving a simplified view of the logical and-gate drawn in Figure 2.2. The facts are the connections of the particular resistors and transistors comprising the circuit. The relation scheme for resistors is `resistor(End1,End2)` and for transistors `transistor(Gate,Source,Drain)`.

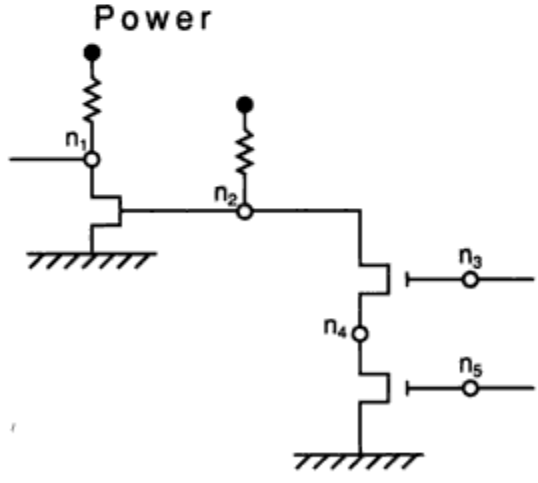


Figure 2.2 A logical circuit

```
resistor(power,n1).
resistor(power,n2).

transistor(n2,ground,n1).
transistor(n3,n4,n2).
transistor(n5,ground,n4).
```

inverter(Input,Output) ←
    Output is the inversion of *Input*.

```
inverter(Input,Output) ←
    transistor(Input,ground,Output),
    resistor(power,Output).
```

nand_gate(Input1,Input2,Output) ←
    Output is the logical nand of *Input1* and *Input2*.

```
nand_gate(Input1,Input2,Output) ←
    transistor(Input1,X,Output),
    transistor(Input2,ground,X),
    resistor(power,Output).
```

and_gate(Input1,Input2,Output) ←
    Output is the logical and of *Input1* and *Input2*.

```
and_gate(Input1,Input2,Output) ←
    nand_gate(Input1,Input2,X),
    inverter(X,Output).
```

Program 2.2 A circuit for a logical and-gate

The program demonstrates the style of commenting of logic programs we will follow throughout the book. Each interesting procedure is preceded by a relation scheme for the procedure, shown in italic font, and by English text defining the relation. We recommend this style of commenting, which emphasizes the declarative reading of programs, for Prolog programs as well.

Particular configurations of resistors and transistors fulfill roles captured via rules defining the functional components of the circuit. The circuit describes an and-gate, which takes two input signals and produces as output the logical *and* of these signals. One way of building an and-gate, and how this circuit is composed, is to connect a nand-gate with an inverter. Relation schemes for these three components are `and_gate(Input1,Input2,Output)`, `nand_gate(Input1,Input2,Output)`, and `inverter(Input,Output)`.

To appreciate Program 2.2, let us read the inverter rule. This states that an inverter is built up from a transistor with the source connected to the ground, and a resistor with one end connected to the power source. The gate of the transistor is the input to the inverter, while the free end of the resistor must be connected to the drain of the transistor, which forms the output of the inverter. Sharing of variables is used to insist on the common connection.

Consider the query `and_gate(In1,In2,Out)?` to Program 2.2. It has the solution `{In1=n3,In2=n5,Out=n1}`. This solution confirms that the circuit described by the facts is an and-gate, and indicates the inputs and output.

2.1.1 Exercises for Section 2.1

(i) Modify the rule for `brother` on page 21 to give a rule for `sister`, the rule for `uncle` in Program 2.1 to give a rule for `niece`, and the rule for `sibling` in Program 2.1 so that it only recognizes full siblings, i.e., those that have the same mother and father.

(ii) Using a predicate `married_couple(Wife,Husband)`, define the relationships `mother_in_law`, `brother_in_law`, and `son_in_law`.

(iii) Describe the layout of objects in Figure 2.3 with facts using the predicates `left_of(Object1,Object2)` and `above(Object1,Object2)`. Define predicates `right_of(Object1,Object2)` and `below(Object1,Object2)` in terms of `left_of` and `above`, respectively.

Figure 2.3 Still-life objects

2.2 Structured Data and Data Abstraction

A limitation of Program 2.2 for describing the and-gate is the treatment of the circuit as a black box. There is no indication of the structure of the circuit in the answer to the `and_gate` query, even though the structure has been implicitly used in finding the answer. The rules tell us that the circuit represents an and-gate, but the structure of the and-gate is present only implicitly. We remedy this by adding an extra argument to each of the goals in the database. For uniformity, the extra argument becomes the first argument. The base facts simply acquire an identifier. Proceeding from left to right in the diagram of Figure 2.2, we label the resistors `r1` and `r2`, and the transistors `t1`, `t2`, and `t3`.

Names of the functional components should reflect their structure. An inverter is composed of a transistor and a resistor. To represent this, we need structured data. The technique is to use a compound term, `inv(T,R)`, where `T` and `R` are the respective names of the inverter's component transistor and resistor. Analogously, the name of a nand-gate will be `nand(T1,T2,R)`, where `T1`, `T2`, and `R` name the two transistors and resistor that comprise a nand-gate. Finally, an and-gate can be named in terms of an inverter and a nand-gate. The modified code containing the names appears in Program 2.3.

The query `and_gate(G,In1,In2,Out)?` has solution {`G=and(nand(t2,t3,r2),inv(t1,r1)),In1=n3,In2=n5,Out=n1`}. `In1`, `In2`, and `Out` have their previous values. The complicated structure for `G` reflects accurately the functional composition of the and-gate.

Structuring data is important in programming in general and in logic programming in particular. It is used to organize data in a meaningful way. Rules can be written more abstractly, ignoring irrelevant details. More modular programs can be achieved this way, because a change of data representation need not mean a change in the whole program, as shown by the following example.

Consider the following two ways of representing a fact about a lecture course on complexity given on Monday from 9 to 11 by David Harel in the Feinberg building, room A:

```
course(complexity,monday,9,11,david,harel,feinberg,a).
```

and

```
course(complexity,time(monday,9,11),lecturer(david,harel),
    location(feinberg,a)).
```

The first fact represents `course` as a relation between eight items — a course name, a day, a starting hour, a finishing hour, a lecturer's first name, a lecturer's surname, a building, and a room. The second fact makes `course` a relation between four items — a name, a time, a lecturer, and a location with further qualification. The time is composed of a day, a starting time, and a finishing time; lecturers have a first name and a surname; and locations are specified by a building and a room. The second fact reflects more elegantly the relations that hold.

The four-argument version of `course` enables more concise rules to be written by abstracting the details that are irrelevant to the query. Program 2.4 contains examples. The `occupied` rule assumes a predicate less than or equal, represented as a binary infix operator $\leq$.

Rules not using the particular values of a structured argument need not "know" how the argument is structured. For example, the rules for `duration` and `teaches` represent time explicitly as `time(Day,Start,Finish)` because the `Day` or `Start` or `Finish` times of the course are desired. In contrast, the rule for `lecturer` does not. This leads to greater modularity, because the representation of time can be changed without affecting the rules that do not inspect it.

We offer no definitive advice on when to use structured data. Not using structured data allows a uniform representation where all the data are simple. The advantages of structured data are compactness of representation, which more accurately reflects our perspective of a situation, and

```
lecturer(Lecturer,Course) ←
    course(Course,Time,Lecturer,Location).

duration(Course,Length) ←
    course(Course,time(Day,Start,Finish),Lecturer,Location),
    plus(Start,Length,Finish).

teaches(Lecturer,Day) ←
    course(Course,time(Day,Start,Finish),Lecturer,Location).

occupied(Room,Day,Time) ←
    course(Course,time(Day,Start,Finish),Lecturer,Room),
    Start ≤ Time, Time ≤ Finish.
```

Program 2.4 Course rules

2.3 Recursive Rules

The rules described so far define new relationships in terms of existing ones. An interesting extension is recursive definitions of relationships that define relationships in terms of themselves. One way of viewing recursive rules is as generalization of a set of nonrecursive rules.

Consider a series of rules defining ancestors — grandparents, great-grandparents, etc:

```
grandparent(Ancestor,Descendant) ←
    parent(Ancestor,Person), parent(Person,Descendant).
greatgrandparent(Ancestor,Descendant) ←
    parent(Ancestor,Person), grandparent(Person,Descendant).
greatgreatgrandparent(Ancestor,Descendant) ←
    parent(Ancestor,Person), greatgrandparent(Person,
                                              Descendant).
```

A clear pattern can be seen, which can be expressed in a rule defining the relationship `ancestor(Ancestor,Descendant)`:

```
ancestor(Ancestor,Descendant) ←
    parent(Ancestor,Person), ancestor(Person,Descendant).
```

This rule is a generalization of the previous rules.

A logic program for `ancestor` also requires a nonrecursive rule, the choice of which affects the meaning of the program. If the fact `ancestor(X,X)` is used, defining the `ancestor` relationship to be reflexive, people will be considered to be their own ancestors. This is not the intuitive meaning of ancestor. Program 2.5 is a logic program defining the `ancestor` relationship, where parents are considered ancestors.

ancestor(*Ancestor,Descendant*) ←
    Ancestor is an ancestor of *Descendant*.

```
ancestor(Ancestor,Descendant) ←
    parent(Ancestor,Descendant).
ancestor(Ancestor,Descendant) ←
    parent(Ancestor,Person), ancestor(Person,Descendant).
```

Program 2.5 The `ancestor` relationship

The ancestor relationship is the transitive closure of the parent relationship. In general, finding the transitive closure of a relationship is easily done in a logic program by using a recursive rule.

Program 2.5 defining ancestor is an example of a linear recursive program. A program is *linear recursive* if there is only one recursive goal in the body of the recursive clause. The linearity can be easily seen from considering the complexity of proof trees solving ancestor queries. A proof tree establishing that two individuals are n generations apart given Program 2.5 and a collection of parent facts has $2 \cdot n$ nodes.

There are many alternative ways of defining ancestors. The declarative content of the recursive rule in Program 2.5 is that Ancestor is an ancestor of Descendant if Ancestor is a parent of an ancestor of Descendant. Another way of expressing the recursion is by observing that Ancestor would be an ancestor of Descendant if Ancestor is an ancestor of a parent of Descendant. The relevant rule is

```
ancestor(Ancestor,Descendant) ←
    ancestor(Ancestor,Person), parent(Person,Descendant).
```

Another version of defining ancestors is not linear recursive. A program identical in meaning to Program 2.5 but with two recursive goals in the recursive clause is

```
ancestor(Ancestor,Descendant) ←
    parent(Ancestor,Descendant).
ancestor(Ancestor,Descendant) ←
    ancestor(Ancestor,Person), ancestor(Person,Descendant).
```

Consider the problem of testing connectivity in a directed graph. A directed graph can be represented as a logic program by a collection of facts. A fact edge(Node1,Node2) is present in the program if there is an edge from Node1 to Node2 in the graph. Figure 2.4 shows a graph; Program 2.6 is its description as a logic program.

Two nodes are connected if there is a series of edges that can be traversed to get from the first node to the second. That is, the relation connected(Node1,Node2), which is true if Node1 and Node2 are connected, is the transitive closure of the edge relation. For example, a and e are connected in the graph in Figure 2.4, but b and f are not. Program 2.7 defines the relation. The meaning of the program is the set of goals con-

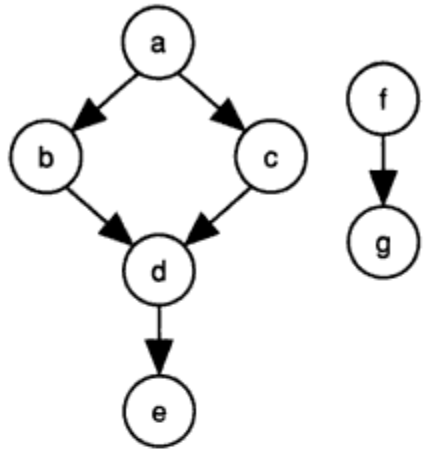


Figure 2.4 A simple graph

```
edge(a,b).    edge(a,c).    edge(b,d).
edge(c,d).    edge(d,e).    edge(f,g).
```

Program 2.6 A directed graph

connected(*Node1*,*Node2*) ←
    Node1 is connected to *Node2* in the
    graph defined by the *edge/2* relation.

```
connected(Node,Node).
connected(Node1,Node2) ← edge(Node1,Link), connected(Link,Node2).
```

Program 2.7 The transitive closure of the edge relation

nected(X,Y), where X and Y are connected. Note that connected is a transitive reflexive relation because of the choice of base fact.

2.3.1 Exercises for Section 2.3

(i) A stack of blocks can be described by a collection of facts on (Block1,Block2), which is true if Block1 is on Block2. Define a predicate above(Block1,Block2) that is true if Block1 is above Block2 in the stack. (Hint: above is the transitive closure of on.)

Cover: Ichiryusai Hiroshige,
Shinagawa: Daimyos' Departure
(detail). John Chandler Bancroft
Collection, Worcester Art Museum,
Worcester, Massachusetts.

The MIT Press
Massachusetts Institute
of Technology
Cambridge,
Massachusetts 02142

0-262-19338-8