AI Algorithms, Data Structures, and Idioms in Prolog, Lisp, and Java

George F. Luger
William A. Stubblefield

Addison-Wesley 2009

Contents

Preface

What we have to learn to do
We learn by doing…

- Aristotle, *Ethics*

Why Another Programming Language Book?

Writing a book about designing and implementing representations and search algorithms in Prolog, Lisp, and Java presents the authors with a number of exciting opportunities.

The first opportunity is the chance to compare three languages that give very different expression to the many ideas that have shaped the evolution of programming languages as a whole. These core ideas, which also support modern AI technology, include functional programming, list processing, predicate logic, declarative representation, dynamic binding, meta-linguistic abstraction, strong-typing, meta-circular definition, and object-oriented design and programming. Lisp and Prolog are, of course, widely recognized for their contributions to the evolution, theory, and practice of programming language design. Java, the youngest of this trio, is both an example of how the ideas pioneered in these earlier languages have shaped modern applicative programming, as well as a powerful tool for delivering AI applications on personal computers, local networks, and the world wide web.

The second opportunity this book affords is a chance to look at Artificial Intelligence from the point of view of the craft of programming. Although we sometimes are tempted to think of AI as a theoretical position on the nature of intelligent activity, the complexity of the problems AI addresses has made it a primary driver of progress in programming languages, development environments, and software engineering methods. Both Lisp and Prolog originated expressly as tools to address the demands of symbolic computing. Java draws on object-orientation and other ideas that can trace their roots back to AI programming. What is more important, AI has done much to shape our thinking about program organization, data structures, knowledge representation, and other elements of the software craft. Anyone who understands how to give a simple, elegant formulation to unification-based pattern matching, logical inference, machine learning theories, and the other algorithms discussed in this book has taken a large step toward becoming a master programmer.

The book's third, and in a sense, unifying focus lies at the intersection of these points of view: how does a programming language's formal structure interact with the demands of the art and practice of programming to

create the idioms that define its accepted use. By idiom, we mean a set of conventionally accepted patterns for using the language in practice. Although not the only way of using a language, an idiom defines patterns of use that have proven effective, and constitute a common understanding among programmers of how to use the language. Programming language idioms do much to both enable, as well as support, ongoing communication and collaboration between programmers.

These, then, are the three points of view that shape our discussion of AI programming. It is our hope that they will help to make this book more than a practical guide to advanced programming techniques (although it is certainly that). We hope that they will communicate the intellectual depth and pleasure that we have found in mastering a programming language and using it to create elegant and powerful computer programs.

The Design of this Book

There are five sections of this book. The first, made up of a single chapter, lays the conceptual groundwork for the sections that follow. This first chapter provides a general introduction to programming languages and style, and asks questions such as "What is a *master programmer*?" What is a programming language *idiom*?", and "How are identical *design patterns* implemented in different languages?". Next, we introduce a number of design patterns specific to supporting data structures and search strategies for complex problem solving. These patterns are discussed in a "language neutral" context, with pointers to the specifics of the individual programming paradigms presented in the subsequent sections of our book. The first chapter ends with a short historical overview of the evolution of the logic-based, functional, and object-oriented approaches to computer programming languages.

Part II of this book presents Prolog. For readers that know the rudiments of first-order predicate logic, the chapters of Part II can be seen as a tutorial introduction to Prolog, the language for **pro**gramming in **log**ic. For readers lacking any knowledge of the propositional and predicate calculi we recommend reviewing an introductory textbook on logic. Alternatively, Luger (2005, Chapter 2) presents a full introduction to both the propositional and predicate logics. The Luger introduction includes a discussion, as well as a pseudo code implementation, of unification, the pattern-matching algorithm at the heart of the Prolog engine.

The design patterns that make up Part II begin with the "flat" logic-based representation for facts, rules, and goals that one might expect in any relational data base formalism. We next show how recursion, supported by unification-based pattern matching, provides a natural design pattern for tree and graph search algorithms. We then build a series of abstract data types, including sets, stacks, queues, and priority queues that support patterns for search. These are, of course, abstract structures, crafted for the specifics of the logic-programming environment that can search across state spaces of arbitrary content and complexity. We then build and demonstrate the "production system" design pattern that supports rule based programming, planning, and a large number of other AI technologies. Next, we present structured representations, including

semantic networks and frame systems in Prolog and demonstrate techniques for implementing single and multiple inheritance representation and search. Finally, we show how the Prolog design patterns presented in Part II can support the tasks of machine learning and natural language understanding.

Lisp and functional programming make up Part III. Again, we present the material on Lisp in the form of a tutorial introduction. Thus, a programmer with little or no experience in Lisp is gradually introduced to the critical data structures and search algorithms of Lisp that support symbolic computing. We begin with the (recursive) definition of symbol-expressions, the basic components of the Lisp language. Next we present the "assembly instructions" for symbol expressions, including car, cdr, and cons. We then assemble new patterns for Lisp with cond and defun. Finally, we demonstrate the creation and/or evaluation of symbol expressions with quote and eval. Of course, the ongoing discussion of variables, binding, scope, and closures is critical to building more complex design patterns in Lisp.

Once the preliminary tools and techniques for Lisp are presented, we describe and construct many of the design patterns seen earlier in the Prolog section. These include patterns supporting breadth-first, depth-first, and best-first search as well as meta-interpreters for rule-based systems and planning. We build and demonstrate a recursion-based unification algorithm that supports a logic interpreter in Lisp as well as a stream processor with delayed evaluation for handling potentially infinite structures. We next present data structures for building semantic networks and object systems. We then present the Common Lisp Object system (CLOS) libraries for building object and inheritance based design patterns. We close Part III by building design patterns that support decision-tree based machine learning.

Java and its idioms are presented in Part IV. Because of the complexities of the Java language, Part IV is not presented as a tutorial introduction to the language itself. It is expected that the reader has completed at least an introductory course in Java programming, or at the very least, has seen object-oriented programming in another applicative language such as C++, C#, or Objective C. But once we can assume a basic understanding of Java tools, we do provide a tutorial introduction to many of the design patterns of the language.

The first chapter of Part IV, after a brief overview of the origins of Java, goes through many of the features of an object-oriented language that will support the creation of design patterns in that environment. These features include the fundamental data structuring philosophy of encapsulation, polymorphism, and inheritance. Based on these concepts we briefly address the analysis, iterative design, programming and test phases for engineering programs. After the introductory chapter we begin pattern building in Java, first considering the

representation issue and how to represent predicate calculus structures in Java. This leads to building patterns that support breadth-first, depth-first, and best-first search. Based on patterns for search, we build a production system, a pattern that supports the rule-based expert system. Our further design patterns support the application areas of natural language processing and machine learning. An important strength that Java offers, again because of its object-orientation and modularity is the use of public domain (and other) libraries available on the web. We include in the Java section a number of web-supported AI algorithms, including tools supporting work in natural language, genetic and evolutionary programming (a-life), natural language understanding, and machine learning (WEKA).

The final component of the book, Part V, brings together many of the design patterns introduced in the earlier sections. It also allows the authors to reinforce many of the common themes that are, of necessity, distributed across the various components of the presentation, We conclude with general comments supporting the craft of programming.

Using this Book

This book is designed for three primary purposes. The first is as a programming language component of a general class in Artificial Intelligence. From this viewpoint, the authors see as essential that the AI student build the significant algorithms that support the practice of AI. This book is designed to present exactly these algorithms. However, in the normal lecture/lab approach taken to teaching Artificial Intelligence at the University level, we have often found that it is difficult to cover more than one language per quarter or semester course. Therefore we expect that the various parts of this material, those dedicated to either Lisp, Prolog, or Java, would be used individually to support programming the data structures and algorithms presented in the AI course itself. In a more advanced course in AI it would be expected that the class cover more than one of these programming paradigms.

The second use of this book is for university classes exploring programming paradigms themselves. Many modern computer science departments offer a final year course in comparative programming environments. The three languages covered in our book offer excellent examples on these paradigms. We also feel that a paradigms course should not be based on a rapid survey of a large number of languages while doing a few "finger exercises" in each. Our philosophy for a paradigms course is to get the student more deeply involved in fewer languages, and these typically representing the declarative, functional, and object-oriented approaches to programming. We also feel that the study of idiom and design patterns in different environments can greatly expand the skill set of the graduating student. Thus, our philosophy of programming is built around the language idioms and design patterns presented in Part I and summarized in Part V. We see these as an exciting opportunity for students to appreciate the wealth and diversity of modern computing

environments. We feel this book offers exactly this opportunity.

The third intent of this book is to offer the professional programmer the chance to continue their education through the exploration of multiple programming idioms, patterns, and paradigms. For these readers we also feel the discussion of programming idioms and design patterns presented throughout our book is important. We are all struggling to achieve the status of the *master programmer*.

We have built each chapter in this book to reflect the materials that would be covered in either one or two classroom lectures, or about an hour's effort, if the reader is going through this material by herself. There are a small number of exercises at the end of most chapters that may be used to reinforce the main concepts of that chapter. There is also, near the end of each chapter, a summary statement of the core concepts covered in that chapter.

Acknowledgments

First, we must thank several decades of students and colleagues at the University of New Mexico. These friends have not only suggested, helped design, and tested our algorithms but have also challenged us to make them better. In the acknowledgments for each chapter we have mentioned students and colleagues that have helped develop key aspects of that chapter.

Second, there are several professional colleagues that we owe particular debts. These include David MacQueen of the University of Chicago, one of the creators of SML, Manuel Hermenegildo, The Prince of Asturias Endowed Chair of Computer Science at UNM and a designer of Ciao Prolog, Paul De Palma, Professor of Computer Science at Gonzaga University, and Alejandro Cdebaca, our friend and former student, who helped design many of the algorithms of the Java chapters.

Third, we thank our friends at Pearson Education who have supported our various creative writing activities over the past two decades. Especially important our editors Alan Apt, Karen Mossman, Keith Mansfield, Owen Knight, and Simon Plumtree, and Matt Goldstein, along with their various associate editors, proof readers, and web support personnel.

We also acknowledge wives, children, family, and friends; all those that have made our lives not just survivable, but intellectually stimulating and enjoyable.

Finally, to our readers; we salute you: the art, science, and practice of programming is great fun, enjoy it!

GL

BS

July 2008

Albuquerque