人工智能程序设计 第二版

ARTIFICIAL INTELLIGENCE PROGRAMMING

SECOND EDITION

EUGENE CHARNIAK
Brown University
CHRISTOPHER K. RIESBECK
Yale University
DREW V. MCDERMOTT
Yale University
JAMES R. MEEHAN
Cognitive Systems, Inc.

人工智能资料中心

ARTIFICIAL INTELLIGENCE PROGRAMMING

SECOND EDITION

EUGENE CHARNIAK
Brown University
CHRISTOPHER K. RIESBECK
Yale University
DREW V. McDERMOTT
Yale University
JAMES R. MEEHAN
Cognitive Systems, Inc.

LEA LAWRENCE ERLBAUM ASSOCIATES, PUBLISHERS
1987 Hillsdale, New Jersey Hove and London

Lawrence Erlbaum Associates, Inc., Publishers
365 Broadway
Hillsdale, New Jersey 07642

Library of Congress Cataloging in Publication Data

Artificial intelligence programming.

Rev. ed. of: Artificial intelligence programming / Eugene Charniak, Christopher K. Riesbeck, Drew V. McDermott. 1980

Bibliography: p.

Includes index.

1. Artificial intelligence—Data processing
2. Programming (Electronic computers) 3. LISP (Computer program language) I. Charniak, Eugene.
Q336 A78 1987 006.3 87-8981
ISBN 0-89859-609-2

Printed in the United States of America

PREFACE TO THE SECOND EDITION

Since the first edition of this book appeared, some things in the AI programming world have changed a great deal, and some things are almost exactly the way they were five years ago. Perhaps the most significant development has been the appearance of COMMON LISP, documented in abundant detail by Guy Steele [102]. All of the LISP code in this new edition has been rewritten in COMMON LISP. COMMON LISP is a pleasant surprise, given the normal result of compromise solutions designed by committees. It manages to be a synthesis of many of the best ideas present in modern LISP dialects, rather than a fossilization of the worst. While there are other dialects that have a more coherent semantics, such as SCHEME [82] and T [81], COMMON LISP is more than adequate for our needs.

The choice of COMMON LISP has affected the contents of this book in several ways. First, many features that we spent some time developing and adding to our earlier dialect of LISP are already available in COMMON LISP. In some cases, we have therefore just described the COMMON LISP feature. In other cases, we have retained the developmental material in order to explain the underlying principles.

COMMON LISP is a "large" language, and we cover only part of it; some of the best-designed features allowed us to remove material from the first edition that dealt with the friendly but limited dialect we used then, UCI LISP. Gone are the sections on FEXPRs and LEXPRs; while we still discuss the issue of extending the language by adding new data types, COMMON LISP's DEFSTRUCT is an example of a tool that we had to build from scratch in UCI LISP.

We considered both SCHEME and T for this edition. SCHEME has the right essential semantics for programming, such as lexical scoping and closures, and

T extends that to include the right primitives for object-oriented programming and language-extension via macros, as well as a host of well-designed support features. (The next edition (!) will probably be in T.) Happily, many of these ideas are also present in COMMON LISP, and given its greater visibility, we chose it instead. In particular, the availability of lexical closures has allowed us to re-implement a number of disparate ideas in a uniform manner.

One problem with COMMON LISP is that it is not yet widely available. Most implementations exist only on very large machines. While this situation will certainly change in the next few years, we have tried to ameliorate the problem in two ways. First, we have used only a subset of COMMON LISP in our code. Second, we have provided a glossary describing the subset we used.

In the LISP chapters in Part 1 and in the AI topics in Part 2, five years' additional experience has led us to provide completely new explanations, examples, and implementations. The original chapter on alternative control structures, which described the implementation of a variation of SCHEME, served two purposes: to introduce some of the power behind lexically scoped languages with procedures as first-class objects, and to give an example of how other languages with very different control structures could be implemented in LISP. Since COMMON LISP is lexically scoped, there is no longer any need to treat that topic separately. The topic of control structure in SCHEME is now described briefly in a new chapter on higher-order functions, continuations, and coroutines. Some of the flavor of implementing interpreters can be gleaned from a new chapter on production systems. (For more information on SCHEME and implementing a SCHEME interpreter, the interested reader is referred to Abelson and Sussman [2].)

The chapter on production systems illustrates several aspects of AI programming. A production system is much easier to implement than a deductive retriever, and correspondingly more limited in power. Production systems, however, have proved to be very useful in the development of expert systems, which are the basis for most of the commercial AI work at the moment.

As before, the intended audience of *Artificial Intelligence Programming* remains the advanced undergraduate or early graduate student in AI. Although the material involved requires only modest knowledge of programming, the student who has had the most experience in creating AI programs already will understand best the benefits of the techniques described. In addition, in response to the changing nature of AI in industry, we have changed the text to be a bit more like a cookbook. It has turned out that many people learning AI programming are doing so on their own, either at home or at work. Many of them prefer to begin with working pieces of code that they can then extend, rather than inadequate pieces of code that are then corrected in the text and exercises. This latter technique works in classrooms, but not in a self-study situation. Therefore, we have eliminated almost all figures with deliberately incorrect code, and added an appendix with the answers to nearly all the exercises in the book. We have also removed the chapters on the sample course project.

ACKNOWLEDGMENTS

We would like to thank the teachers, students, and other programmers who suggested changes (and supplied corrections!) to the first edition, as well as reviewers who gave us criticisms prior to this revision. Thanks to Paul Hudak and Bill Ferguson for comments on Chapter 7, and to Robert Farrell for comments on Chapter 12. We are indebted to Bob Strong at Cognitive Systems Inc. for reviewing the new edition, recommending improvements, and helping us test the code. Answers to some of the exercises were written by Gregory Parkinson and Rika Yoshii when they were students at UCI. The Yale Artificial Intelligence Project and Cognitive Systems Inc. provided computer support for the preparation of the new manuscript. Finally, Chris Riesbeck would like to dedicate this book to Maxine.

Christopher K. Riesbeck
Jim Meehan

PREFACE TO THE FIRST EDITION

Artificial Intelligence (henceforth AI) is still a field where disagreement is more common than solid theory, and interesting ideas more common than polished programs. Yet there is slowly coming into being a small core of accepted (though not universally accepted) theory and practice. This book is an attempt to gather together the "practice" aspect of this "core" AI.

The "practice" of AI is, of course, the writing of programs. AI problems are usually ill-defined and the theories proposed are often too complex and complicated to be verified by intuitive or formal arguments. Sometimes the only way to understand and evaluate a theory is to see what comes next. To find this out, and to check for obvious inconsistencies and contradictions, we write programs that are intended to reflect our theories. If these programs work, our theories are not proved, of course, but at least we gain some understanding of how they behave. When the programs don't work (or we find ourselves unable to program the theories at all), then we learn what we have yet to define or re-define.

With this emphasis on programming, it becomes important that an AI researcher have a wide library of programming tools available. This is particularly true because of the "level" problem. That is, your theory describes what to do at a fairly high level, but you need to tell the machine what to do at a low level. So a theory of, say, coherency in conversation, will in all probability say nothing about pattern matching or efficient data retrieval. It is not that these topics are not worthy of their own theory. How people manage to retrieve knowledge efficiently under a wide variety of circumstances is a fascinating question. But if you are worried about conversation, it is simply not your department.

The intent of this book is to give you a wide variety of commonly used tools for programming Artificial Intelligence theories: discrimination nets, agendas, deduction, data dependencies, backtracking, etc. By having these tools, we hope you will find that your programs better reflect your intentions.

Almost all of the ideas that are described here are in common use, particularly at the larger Artificial Intelligence centers. But very few of them have ever been written down in one place. There are a number of books that introduce you to LISP (although none of them are completely satisfactory), and there are a number of books on theories and algorithms in Artificial Intelligence. Until now, however, there have been no books that fill in the middle ground and present the methods that all the old-timers know for getting from theory to practice. That is what this book is all about.

The major problem in writing a book such as this is that of selection. In some cases it is easy. It seems unlikely that anyone would seriously contest our inclusion of discrimination nets, pattern matching, or agendas. These techniques have been used by many researchers in the field and in a variety of problem areas—from natural language comprehension to problem solving to medical diagnosis. However once one moves beyond this handful of topics, or even starts getting specific about the type of pattern matching, or agenda, then consensus is not so easy. So to some degree the selections made in this book are personal ones. Of course, to say they are personal is not to say they cannot be defended on scientific grounds, but rather that the defense would take the form of an extended debate on the nature of AI and where it is going. For example, data dependencies receive a chapter to themselves here in spite of the fact that they are fairly new on the scene and hence relatively untested, at least compared to something like unification pattern matching. Naturally we try to show the usefulness of these ideas, but only to show how the ideas are motivated, not to defend particular approaches against competitors. Such a defense would be well worth having, but it would be out of place in a text such as this.

Selection also implies that some things are omitted, and there are at least two notable omissions from these chapters. One of these is inadvertent. The techniques discussed here all come from what might be thought of as "abstract" AI. That is, if we think of AI programs on a spectrum from "concrete" programs which must deal with the real world in terms of sound and light input (or sound and muscle output) to "abstract" programs which only deal with abstractions, the techniques described here fall most naturally towards the abstract end. This book does not have the space and the authors do not have the expertise to do justice to the concrete end of things.

A second omission is quite deliberate. We have made no attempt to survey, much less teach, the many AI languages (CONNIVER, QA-4, KRL, etc.). This stems from our conviction that at present there is no commonly agreed-upon set of functions above the level of list processing which everyone would agree is useful in a wide variety of AI settings. Experience has shown that each major project has found it necessary to build up its own tools, starting, typically, from LISP. We do not see this situation as likely to change in the foreseeable future. Hence rather than covering the basics of the various languages we have tried instead to explain the techniques which typically lie behind these languages.

The book is divided into two parts. Since almost all serious programming in Artificial Intelligence is done in the language LISP, Part 1 tells you how to improve your general abilities as a LISP programmer. [The first chapter covers] most of the basic LISP concepts needed for the rest of the book. We intend the introductory material to cover all the concepts of LISP needed later, but if you have never programmed in LISP before, we recommend that you spend some time writing simple LISP programs until you get a feel for the language.

The [second through seventh] chapters are concerned with the many features found (or implementable) in LISP that make the language an attractive one to use. Many of the ideas that pass under the rubric of "structured programming" will be found here. Although LISP is almost as old as FORTRAN, it is surprisingly amenable to things like top-down programming and data types.

Part 2 contains more advanced and complex techniques. Since this book is intended not just to be a description of ideas, but also to give you a chance to learn the craft of Artificial Intelligence, we present actual LISP implementations of all the ideas discussed, along with exercises which modify and extend the code. These exercises are intended to make you familiar, in a practical hands on way, with the techniques involved. We hope that the exercises will inspire you to experiment and learn on your own.

This book is intended mainly for use as a textbook for an AI course in which programming is emphasized. This could be either an advanced or a fast elementary course. The book might also be used as an auxiliary text for a systems course; for this purpose, the chapters on macros, structured programming, and alternative control structures would be most useful.

ACKNOWLEDGMENTS

The first edition of this book was the outgrowth of a graduate course given by the authors in the spring of 1978 at Yale University for students in the Artificial Intelligence Project of the Yale Computer Science Department. We would like to acknowledge Roger Schank who originated the idea of the course. We'd also like to thank Dave Barstow and Walter Stutzman for their detailed comments on the complete manuscript, Laury Miller and Glenn Edelson for their comments on the introductory chapters, and Jon Doyle for his comments on the chapter on data dependencies.

The Yale Artificial Intelligence Project is funded by the Advanced Research Projects Agency of the Department of Defense and the Office of Naval Research.

During the writing of [the first edition of] this book, Eugene Charniak and Christopher Riesbeck were supported by the Advanced Research Projects Agency monitored under the Office of Naval Research under contract N00014–75–C–1111.

Eugene Charniak
Christopher K. Riesbeck
Drew V. McDermott

Contents

1 Lisp REVIEW

> LISP has jokingly been called "the most intelligent way to misuse a computer." I think that description is a great compliment because it transmits the full flavor of liberation: it has assisted a number of our most gifted fellow humans in thinking previously impossible thoughts.
>
> — Edsger Dijkstra

LISP was the world's first elegant language, in the sense that it provided a parsimonious base with rich possibilities for extension. LISP has been applied mainly to problems of symbolic manipulation and artificial intelligence, partly because manipulating symbols is so easy in LISP, and partly because AI programmers tend to be lazy and undisciplined, like pilots who refuse to file a flight plan before taking off, and LISP's interactive structure allows them to get away with this.

1.1 Data Structures

LISP data structures are called "S-expressions." The S stands for "symbolic." In this text, the terms "S-expression" and "expression" are used interchangeably. An S-expression is

1. a *number*, e.g., `15`, written as an optional plus or minus sign, followed by one or more digits.
2. a *symbol*, e.g., `FOO`, written as a letter followed by zero or more letters or digits.

3. a *string*, e.g., "This is a string", written as a double quote, followed by zero or more characters, followed by another double quote.
4. a *character*, e.g., #\a, written as a sharp sign, followed by a backslash, followed by a character. (Numbers, symbols, strings, and characters are called *atoms*. There are other kinds of atoms. We will see them in later chapters.)
5. a *list* of S-expressions, e.g., (A B) or (IS TALL (FATHER BILL)), written as a left parenthesis, followed by zero or more S-expressions, followed by a right parenthesis.

Parentheses are more significant in LISP than they are in most other programming languages. Parentheses are virtually the only punctuation marks available in LISP programs. They are used to indicate the structure of S-expressions. For example, (A) is a list of one element, the symbol A. ((A)) is also a list of one element, which is in turn a list of one element, which is the symbol A. Notice also that the left and right parentheses must *balance*. That is, a well-formed S-expression has a right parenthesis to close off each left parenthesis.

1.2 Program Structures

Syntax: The syntax of a LISP program is simple: *Every* S-expression is a *syntactically* legal program! That is, any given data structure could be executed as a program. Most of them, however, fail on semantic grounds.

Semantics: The function that executes S-expressions (and hence defines the semantics of LISP) is called EVAL. EVAL takes one S-expression and returns another S-expression. The second expression is called the *value* of the first expression. We notate this as *expression* ⇒ *value*.

The rules for evaluation are fairly simple.

Rule 1: If the expression is a *number*, a *string*, a *character*, the symbol T, or the symbol NIL, then its value is itself. So 5 ⇒ 5.

Rule 2: If the expression is a *list* of the form

(*function* arg_1 . . . arg_k)

then the value is found by first evaluating each argument (arg_1 to arg_k), and then calling *function* with these values. For the moment, all our functions are named by symbols. For example, the symbols + (for addition) and * (for multiplication) name functions that are defined in LISP for doing arithmetic.

```
(+ 15 2)          ⇒ 17
(* 3 5)           ⇒ 15
(+ (* 3 5) 2)     ⇒ 17
```

Note that in order to evaluate the last expression, each argument has to be evaluated first. Since the first argument is itself a list, `(* 3 5)`, **Rule 2** is applied again, and the arguments, 3 and 5, are evaluated. By **Rule 1**, they evaluate to themselves. They are passed to the multiplication procedure, *, which returns 15. Similarly, 2 evaluates to itself, and 15 and 2 are passed to +, which returns the number 17

Rule 3: If the expression is a *list* of the form

(*reserved-word* arg_1 . . . arg_k)

then the value depends completely on the definition of *reserved-word*. The arguments may or may not be evaluated. Reserved words are named by symbols, just as functions are

One such reserved word is `SETQ`. `SETQ` is used for assigning values to symbols. (`SETQ` *symbol expression*) causes *symbol* to be "bound to" (or "set to" or "assigned") the value of *expression*. The value of *expression* is returned as the value of the `SETQ` (i.e., the value that the `SETQ`-expression produces). For example, `(SETQ X (+ 15 1))` sets `X` to 16.

Rule 4: If the expression is a *symbol*, then its value is the last value that has been assigned to it. If no value has been assigned, then an error occurs. So if `X` is bound to 16, then `X` ⇒ 16. These are the four rules employed by the `EVAL`, which is part of the LISP interpreter. The interpreter is a program that you run. You type an S-expression and it prints back another S-expression. This second S-expression is the value of the one you typed in. (If something goes wrong, an error message is printed instead of a value).

LISP programs don't always need to be compiled like ALGOL or FORTRAN programs. That is, you do not have to take a file of LISP text, convert it into internal machine code, and then run the machine code. Instead, you can type LISP text to the interpreter, which evaluates it and types the result back at you as more LISP text.

While interpreting means that programs run slower, it also usually means that you always have your expressions available during execution for inspection and modification. LISP can support very powerful debugging and editing facilities for this reason.

The "top-level" loop of the LISP interpreter can be written in a ALGOL-like language as

```
BEGIN
LOOP: EXP := READ (INPUT);
      VAL := EVAL (EXP);
      PRINT (VAL, OUTPUT);
      GO TO LOOP
END;
```

This is usually referred to as the `READ-EVAL-PRINT` loop. All three functions are available to the user — that is, when you write a function that uses