Artificial Intelligence Texts

Prolog Programming: A Tutorial Introduction

Artificial Intelligence Texts

Prolog Programming: A Tutorial Introduction

CARLTON McDONALD
Department of Computer Science
Coventry Polytechnic

MASOUD YAZDANI
Department of Computer Science
University of Exeter

OXFORD
BLACKWELL SCIENTIFIC PUBLICATIONS
LONDON EDINBURGH BOSTON
MELBOURNE PARIS BERLIN VIENNA

Blackwell Scientific Publications
Editorial Offices:
Osney Mead, Oxford OX2 0EL
25 John Street, London WC1N 2BL
23 Ainslie Place, Edinburgh EH3 6AJ
3 Cambridge Center, Suite 208
Cambridge, Massachusetts 02142, USA
54 University Street, Carlton
Victoria 3053, Australia

First published 1990

Printed and bound in Great Britain by
Billing & Sons, Worcester

DISTRIBUTORS

Marston Book Services Ltd
PO Box 87
Oxford OX2 0DT
(*Orders:* Tel: 0865 791155
Fax: 0865 791927
Telex: 837515)

USA
Blackwell Scientific Publications, Inc.
3 Cambridge Center,
Cambridge, MA 02142
(*Orders:* Tel: (800) 759-6102)

Canada
Oxford University Press
70 Wynford Drive
Don Mills
Ontario M3C 1J9
(*Orders:* Tel: (416) 441-2941)

Australia
Blackwell Scientific Publications
(Australia) Pty Ltd
54 University Street
Carlton, Victoria 3053
(*Orders:* Tel: (03) 347-0300)

British Library
Cataloguing in Publication Data
McDonald, Carlton
Prolog programming: a tutorial
introduction – (Artificial intelligence texts)
1. Artificial intelligence. Application of
computer systems. Programming
languages; Prolog
I. Title II. Yazdani, Masoud III. Series
006.302855133

ISBN 0-632-01246-3

Library of Congress
Cataloging in Publication Data
McDonald, Carlton.
Prolog programming: a tutorial
introduction / Carlton McDonald,
Masoud Yazdani.
p. cm. — (Artificial intelligence texts)
Includes bibliographical references.
ISBN 0-632-01246-3
1. Prolog (Computer program language)
I. Yazdani, Masoud II. Title. III. Series.
QA76.73.P76M43 1990
006.3—dc20 90-504

Preface

This book is an introduction to Prolog intended for those who have not yet bought Prolog (either as a piece of software or as a new way of looking at computer programming) but want to find out how it works and also for those wondering whether they should use Prolog in an application. It assumes no knowledge of Prolog and very little knowledge of other computer languages.

We have deliberately tried to produce a short text. The idea is to give you a good feel for the language and leave it to you to assess the potential of Prolog for meeting your requirements. The book is divided into two parts. Part One explains the basic concepts incorporated into almost all implementations of Prolog. Part Two shows examples and areas in which Prolog shows itself in the best light (with the exception of the chapter on operators which is used to enable the meta-level expert system to be developed). We wish you a short but hopefully enjoyable read.

Acknowledgements

This project is one of the results of a period of industrial secondment which I spent at Expert Systems International Ltd. (ESI). As with software products we have produced, this book incorporates the work of the whole company. In particular, I am grateful to Tony Dodd, Carlton McDonald and Jocelyn Paine who helped me a great deal while I was with ESI.

M.Y.

I am extremely grateful to Masoud Yazdani who encouraged me to contribute to this book, despite the fact that I was inundated with many other things. The experience of co-authoring has enabled me to commit to paper ideas that I may never have made time to share. I would also like to thank the research teams at Oxford Polytechnic and the Open University for being both helpful and inspiring. Finally, I would like to thank my mother, whose example of diligence and perseverance I have tried to emulate.

C. M.

Introduction

Prolog is both a new programming language and a new way of looking at programming. Most other programming languages, such as BASIC and Pascal present the computer with the solution to a problem in the form of a series of instructions to the machine to be executed strictly in the order in which they are specified.

PROgramming in LOGic (PROLOG) should be declarative, a program should simply be the statement of the problem. The way the problem is solved and the sequences of instructions that the computer must go through to solve it, are decided by the system.

We use the word 'should' because there is no way one could stop people misusing Prolog and using it as if it was Pascal, or any other programming language that they may be familiar with. If they do this, they lose the benefits which declarative programming offers.

On the face of it, there is no reason why one would want to use a programming language other than Prolog! It is easier to say what we want done and leave it to the computer to do it for us. Computers would then be humanity's corporate slaves !

The problem with Prolog, and any other mechanical slave, is that unless one is absolutely clear as to what one wants, one is going to get piles of rubbish. In many cases people find it easier to show someone how a job is to be done and leave the slave to imitate them. This is why most professional programmers (Kay, 1984) suffer a culture shock when they first use Prolog. They have been brain washed into expecting to show the computer what to do in a great deal of detail. With Prolog they see magic things happening during the first half hour and then spend the rest of their time undoing the power of Prolog in order to make it

similar to one of the programming languages they know and love. On the other hand total novices spend a great deal longer learning everything about Prolog and piggy backing on its power for a long time to come.

Prolog was born within the realm of Artificial Intelligence (AI). AI is concerned with the design and the study of the properties of intelligent systems. Human behaviour (understanding language, perception, learning, reasoning and so on) is something with which we associate intelligence. AI involves manipulating symbolic representations instead of number crunching. LISt Processing (LISP) is the oldest programming language for AI. LISP is similar in its philosophy to Pascal and other procedural languages, in as much as a program is built out of a series of instructions on how to perform a task.

Prolog originally became popular with AI researchers, as they seemed to know more about what intelligent behaviour is than how it is achieved. Prolog has therefore become a serious competitor to LISP. However, it soon became clear that there are many other problem areas in which we know more about the what than the how.

The philosophy behind Prolog (i.e. both the logical and declarative aspects) is for academics the real power. At present, the commercial world looks at speed of performance as a major criterion. The Prolog concept will come into its own with the use of parallel architectures, as Prolog solves problems by searching a knowledge base (or more correctly a database) the search will be greatly improved if several processors are made to search different parts of the database.

Prolog is an ideal prototyping language because of the speed with which a system can be developed (partly because of the interpretive nature of the language, other factors include the declarative nature, the compactness and inherent modularity of Prolog programs). Once the developer has satisfied the customer

that the system is feasible, a conventional procedural language is often used. Even though Prolog compilers do exist their use is still not widespread outside of academia.

A Prolog program consists of a collection of two types of entity:

a) facts
and
b) rules.

This collection is known as the ***database***. When the database has been set up, it is then possible to ask if certain things are true, given the facts and rules of the database.

Facts are statements that are known unconditionally to be true, such as:

Socrates is a man.

Rules are conditional facts. Such as:

If someone is a man then he is mortal.

Assume that we have given Prolog this information and we are happy with its correctness. We can then ask Prolog the question:

Is Socrates mortal ?

Prolog will answer

yes.

This **yes** is an indication that in the context of the given facts the query is true. Suppose we now ask Prolog another question:

Is Tony mortal?

Prolog will answer

no.

Prolog is only able to answer yes or no. In this case, it lacks certain information on Tony and so is unable to prove that Tony is mortal. As far as Prolog is concerned, this is untrue.

In Prolog, no should be interpreted as meaning; 'Given the information I have, I am unable to prove this to be true'.

Given, this simple way of writing a program and a set of basic primitive facilities, which the user could incorporate into his program, Prolog can be used for a diversity of applications.

Obviously, the application needs to lend itself to Prolog's strengths. If we wish to do a great deal of calculation in order to get an answer (say in calculating income from stock market shares) then we would use a more 'traditional' language. If however, we wish to deal with less tangible, problem solving issues (such as forecasting share price fluctuations) then we might use Prolog.

Contents

Part One:

Prolog Programming

CHAPTER 1

A Simple Model for Prolog Execution

This chapter seeks to introduce the way in which Prolog programs are executed. The philosophy behind the graphical approach is that the esoteric parts of Prolog may be easier to understand if one is able to visualise what is happening. It also means that a graphical debugger could be developed in order to facilitate the development of Prolog programs. There already exists a graphical debugger produced at the Open University by Mike Brayshaw and Marc Eisenstadt, their product is called the Transparent Prolog Machine (TPM). They use a different representation to the one used here, but they have found that a graphical representation enables Prolog systems to be understood and developed a lot easier than conventional textual approaches.

Before you try the examples and exercises you will need to know how to get information into the database. The database will be all the things that the system knows. The easiest way to tell the system something is by typing:

```
?- consult(user).
```

Most systems allow you to shorten this by typing:

```
?- [user].
```

Any information which you now type in will go straight into the database. The termination of the entering of information (consultation) into the database is machine dependent, but is often

the end of file key sequence for the machine (control-D for Unix systems, control-Z for IBM etc.). Unfortunately the information in the database will be lost when you exit Prolog unless you save it to a file. This will be covered later. The other alternative at this stage is to type the information into a file and consult the file rather than the user. For example suppose we had a file called 'cars' then

```
?- [cars].
```

would load the information in the file 'cars' into the database. The system uses the information to create a problem representation. The system solves all problems by traversing this representation of the Prolog database of facts and rules in order to determine the validity of any queries you make. All Prolog programs are expressed as facts and rules. With the exception of input (reading) and output (writing) the system does nothing except answer queries.

Facts and questions

The Prolog interpreter answers questions, but you must provide it with enough information to answer the question. For example you can ask the system if it knows about some fact. For example, suppose you told the system three facts (representing three animals cat, rat and bat). The Prolog representation is simply:

```
cat.
rat.
bat.
```

These three facts are represented as three boxes in Figure 1.1. If

we want to ask the system if it knows about a rat we type:

```
?- rat.
```

The system then looks among the boxes to see if rat is among them. If it finds it, it responds 'yes' otherwise it responds 'no'. In this case searching for **rat** it responds:

```
yes.
```

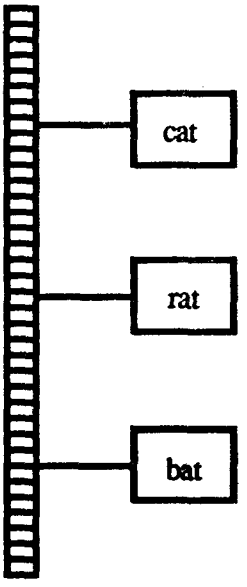

Fig. 1.1 Representation of three simple facts

The system does not know that the facts are animals and therefore cannot answer questions about the size or number of legs that the animals have. All it knows is that rat, cat and bat exist in its memory.

Rules

Sometimes it may not be possible to give the system everything

in terms of facts. The system can look for other facts in order to determine whether or not a query is true. You can specify rules for the system to follow, for example if looking for the box (the fact) a, and the box is not found, then if you can find boxes b, c, and d then fact a is true. To inform the system of this rule we write in Prolog:

```
a :- b, c, d.
```

and the facts:

```
a.
b.
c.
d.
```

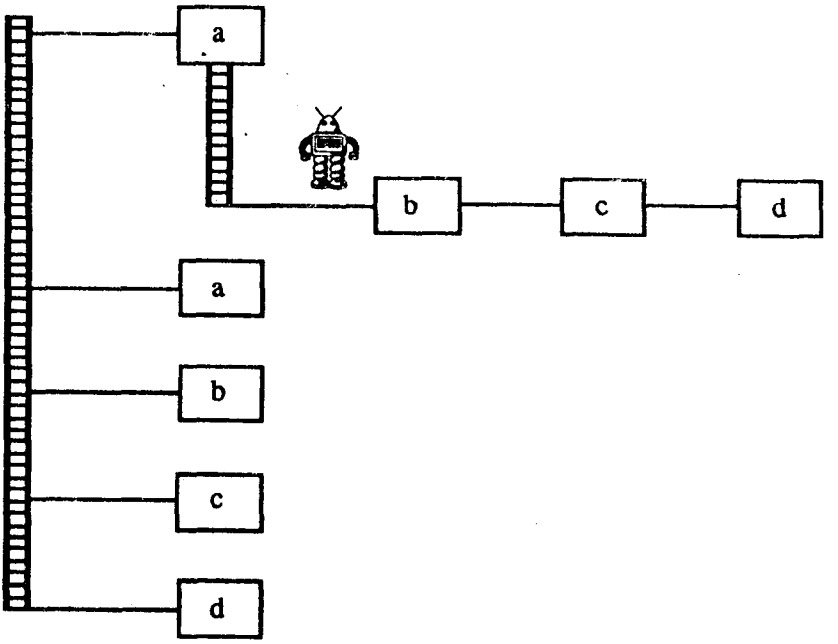

Fig. 1.2 A rule and four facts

In the rule 'a' is termed the ***head goal*** and the goals b, c, and d are termed the ***tail goals***. This single rule and the list of facts are

represented by the system diagrammatically in Figure 1.2.

The system likes climbing down stairs whenever it can, sometimes doing more work than is necessary. It needs a little tempering, so in order to stop it going off on a wild goose chase, care should be taken to give it facts before rules, as in Figure 1.3 instead of the ordering in Figure 1.4.

```
a.
a :- b, c, d.
```

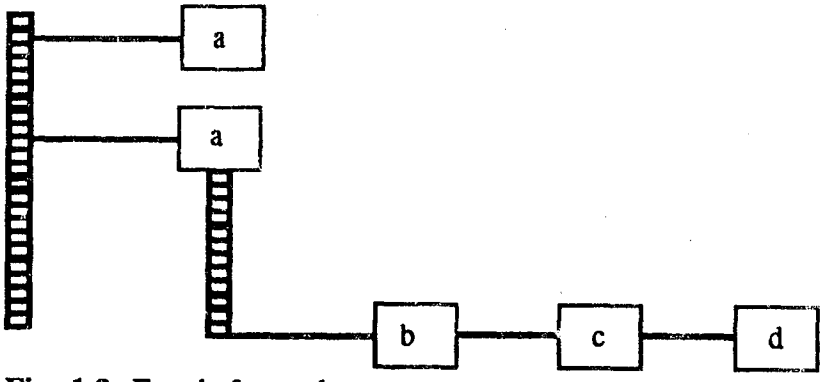

Fig. 1.3 Fact before rule.

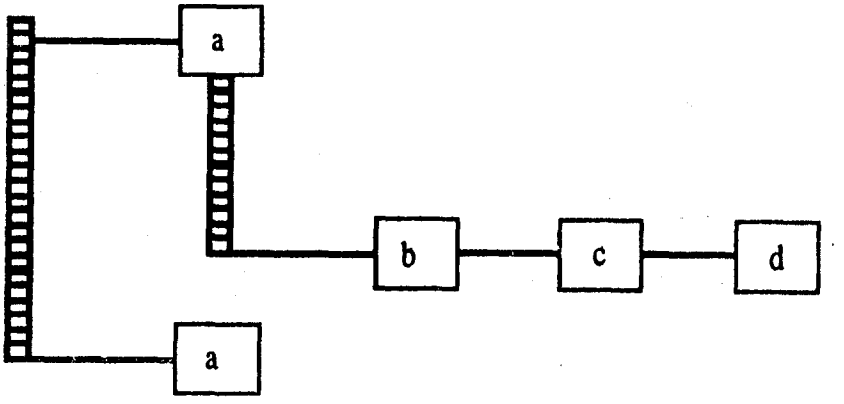

Fig. 1.4 Rule before fact