PROLOG Programming

PROLOG Programming

Nigel Ford
University of Sheffield, UK

JOHN WILEY & SONS
Chichester · New York · Brisbane · Toronto · Singapore

Wiley Editorial Offices

John Wiley & Sons Ltd, Baffins Lane, Chichester,
West Sussex PO19 1UD, England

John Wiley & Sons, Inc., 605 Third Avenue,
New York, NY 10158-0012, USA

Jacaranda Wiley Ltd, G.P.O. Box 859, Brisbane,
Queensland 4001, Australia

John Wiley & Sons (Canada) Ltd, 22 Worcester Road,
Rexdale, Ontario M9W 1L1, Canada

John Wiley & Sons (SEA) Pte Ltd, 37 Jalan Pemimpin #05-04,
Block B, Union Industrial Building, Singapopre 2057

Library of Congress Cataloging-in-Publication Data:

Ford, Nigel.
PROLOG programming / Nigel Ford.
p. cm.
Bibliography: p.
Includes index.
ISBN 0 471 92141 6
1. Prolog (Computer program language) I. Title.
QA76.73.P76F66 1989
005.13'3—dc20 89-31491
CIP

British Library Cataloguing in Publication Data:

Ford, Nigel
PROLOG programming
1. Computer systems. Programming languages : Prolog
I. Title
005.13'3

ISBN 0 471 92141 6

Typeset by Photo-graphics, Honiton, Devon, England
Printed and bound in Great Britain by Courier International, Tiptree, Essex

Preface

WHY LEARN PROLOG?

PROLOG (PROgramming in LOGic) represents an exciting new approach to programming. It is a high-level language which takes much of the drudgery out of programming. Compared to other languages such as BASIC, PASCAL, FORTRAN, even LISP, the novice programmer in particular can spend much less time and effort going through the contortions of specifying in minute detail the steps the computer must go through to solve the problem he or she is working on. More time and effort can be spent thinking about the problem itself, and the logic of how it might be solved. The novice can arrive much sooner at the stage of being able to approach relatively complex and meaningful problems.

Programs written in PROLOG are often very much shorter than equivalent programs written in other languages. They are often also much more intelligible, particularly to the newcomer to programming. PROLOG is the first efficient and practical programming language to emerge from research into 'programming in logic' – the ideal of which is that computers should be able to translate the logical reasoning of the programmer into runnable, efficient programs. Unfortunately, we are far from this ideal. But PROLOG is the nearest thing we have to it at present!

Due to the availability of excellent implementations of PROLOG for personal as well as larger computers, PROLOG is increasing rapidly in popularity as a powerful programming language. PROLOG has been selected by the Japanese in their fifth-generation project as the basis for a new generation of computers which work by logical deduction rather than mathematical calculation.

THIS BOOK

This book is designed to teach you how to read and write PROLOG programs. Despite its power and user-friendliness, whilst it is easy to make a start in PROLOG, the 'learning curve' is relatively steep. Beyond a certain point, developing competence requires considerable effort. As one of the leading UK software suppliers and training organizations in the field of artificial intelligence note in their training brochure:

> Experience has shown that mastering the basics of the language . . . is often very difficult for newcomers, and especially so if they are already highly experienced in traditional 'procedural' languages

It is well worth this effort, though, since arguably the power of PROLOG means that for the same amount of effort you can go much further, in terms of the problems you can approach, compared to other languages.

TRAINING VERSION OF PROLOG

If you want to use this book, but haven't got your own copy of PROLOG, you can (at the time of writing) obtain an inexpensive copy of Chemical Designs' tutorial version of PROLOG-2. You can run most of the programs shown in this book on this tutorial version. It is powerful, and comes complete with built-in editor and help system. However, the main limitation is that you can't save your programs to a disk.

Appendix C in this book tells you how to get started using this tutorial version of PROLOG. It runs on IBM-compatible personal computers with at least 512K of RAM.

No such limitation apples to PROLOG-2 Personal, on which all the programs in this book were run, and which is also available.

Further details may be obtained from

Chemical Designs Ltd,
7 West Way,
Oxford OX2 0JB
Tel. (0865) 251483

Contents

1 Introduction

PROGRAMMING IN LOGIC

PROLOG stands for PROgramming in LOGic. It represents a fundamentally new approach to computing compared to more traditional languages like BASIC, FORTRAN, PASCAL and even LISP (a popular and well-established language for artificial intelligence programming).

Such other languages require the programmer to think very much in terms of computer processes. He or she must tell the computer step by step exactly *how* to tackle each task.

PROLOG adopts a fundamentally different approach. It allows the programmer to think much less in terms of what processes the computer must go through (in other words, *how*) to solve a particular problem – and much more in terms of the logic of the problem itself and of its possible solution (in other words, *what* the problem is). In this way, the programmer can leave it to PROLOG to work out many of the details of the exact processes required to go about solving the problem.

PROLOG is not designed primarily around our understanding of what a computer requires in order to function efficiently. Rather, it is based on a model of logic which is independent of any particular machine and relates more to how human beings think and solve problems.

PROLOG falls short of the ideal of programming in 'pure' logic. Logic is not hampered by considerations of procedural control – for example, the precise ordering of sequences of operations. Although much less so than with other languages, the PROLOG programmer *does* have to take account of procedural details. In fact, one of the main traps that novice programmers can fall into is attributing to PROLOG too much ability to translate his or her intentions into an error-free program – only to find that the program does not behave as expected.

This is the reason why this book stresses a thorough procedural mastery of PROLOG. PROLOG is often taught with a strong 'declarative' emphasis – that is, an emphasis on thinking of the logical relationships between the objects or entities relevant to a given problem, rather than on the procedural steps necessary to solve it. This 'declarative' understanding of PROLOG programs is fine – indeed essential – so long as it is rooted in a sound procedural understanding of what is going on. This book deals with declarative understanding, but within the context of procedural mastery.

You can do in PROLOG what you can do in any other programming language. However, PROLOG has distinct advantages as well as disadvantages. Numbercrunching (large and fast numerical calculation) is not PROLOG's strong point. PROLOG *can* handle numbers, but not as fast or efficiently as languages designed with this in mind. PROLOG scores when it comes to symbol manipulation (problem-solving and decision-making not based primarily on numeric calculation).

Symbol manipulation is the core of what has come to be known as 'artificial intelligence'. You can write artificial intelligence programs in any language – but not as quickly, effectively, and enjoyably, as in a language designed with this type of application in mind. In the opinion of many, PROLOG is the supreme language of this type.

PROLOG makes it particularly easy for us to use many of the useful problem-solving techniques provided by research into artificial intelligence. In particular, it is well geared to the creation of

- intelligent systems (programs which perform useful tasks by utilizing artificial intelligence techniques);
- expert systems (intelligent systems which reproduce decision-making at the level of a human *expert*);
- natural language systems (which can analyse and respond to statements made in ordinary language as opposed to approved keywords or menu selections);
- relational database systems.

Because it frees the programmer from concern with many of the 'nuts-and-bolts' minutiae traditionally associated with writing computer programs, PROLOG allows us to think of problems at a high level. We can pay more attention to thinking about the characteristics of the problem we are trying to solve, or the task we want the computer to perform.

For this reason, PROLOG is an excellent language in which to build rapid prototype systems, and to explore a variety of different approaches to solving a problem or approaching a task. We can try things out and produce scaled-down test programs without incurring the enormous amount of effort required with many other languages.

This applies to both experienced and novice programmers. The novice can make progress more rapidly and approach tasks of real complexity early, and the experienced programmer can increase his or her productivity in terms of the balance of effort between problem-solving and coding.

LEARNING PROLOG

Getting started in PROLOG is relatively easy. We can often translate between normal English and PROLOG with little effort. For example, as soon as we know that :- means **if** and , means **and**, we can readily understand the following program:

PROLOG	*English*
`is_situated_in( london, england ).`	London is in England.
`born_in( sarah, london ).`	Sarah was born in London.

nationality(Person, english):-	A person is English if
born_in(Person, City),	that person was born in a city
is_situated_in(City, england).	and that city is in England.

However, beyond a certain point, a lot of effort is required, and the 'learning curve' rises steeply. For example, to solve the 'Towers of Hanoi' problem shown below, you must transfer the three discs from pole A to pole B. But you can only move one disc at a time, and you must never put a disc on top of a smaller one.

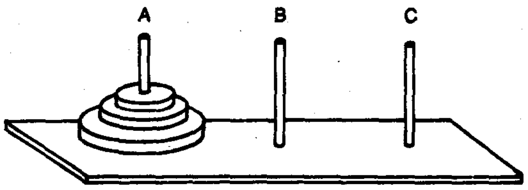

Figure 1

The PROLOG program below solves this problem very elegantly – and the same program can be used for any number of discs.

```
towers(Number):-
  transfer(Number, left, middle, right).

transfer(0, _, _, _).

transfer(Number, Source, Destn, Spare):-
  N is Number -1,
  transfer(N, Source, Spare, Destn),
  write(['Move a disc from ',Source,' to ',Destn]),
  nl,
  transfer(N, Spare, Destn, Source).
```

All you have to do is type:

```
?- towers(3).
```

for three discs (or more or less if you wish), and PROLOG will reply:

```
[Move a disc from ,left, to ,middle]
[Move a disc from ,left, to ,right]
[Move a disc from ,middle, to ,right]
[Move a disc from ,left, to ,middle]
[Move a disc from ,right, to ,left]
[Move a disc from ,right, to ,middle]
[Move a disc from ,left, to ,middle]

yes
```

All well and good – but try understanding the program! For the newcomer to PROLOG, coming to understand the detailed workings poses considerable difficulties. The same can be said even for the three-line program below. And all it does is add two lists together to form a third!

```
add( [], List, List ).

add([Head|Tail], List_2, [Head|List_3]):-
     add(Tail, List_2, List_3).
```

Understanding PROLOG programs can be difficult – particularly trying to read elegant ones written by other people. Yet we need to be able to read fluently in order to learn – to see how others have approached particular problems, and to add new techniques to our repertoire.

Some people take to PROLOG programming like ducks to water. But most people find that whilst floating may be easy, paddling with any speed (not to mention actually steering!) requires considerable effort. PROLOG may be easy up to a point. Beyond that point the going can get tough.

Yet many texts introduce PROLOG techniques at a rapid pace – often with minimal explanation of precise procedural details. The basics of PROLOG may be explained in an early chapter, then taken for granted as techniques are introduced in subsequent chapters. The reader is left with a lot of work to do – having to extract for him or herself a detailed understanding of exactly how particular techniques actually work, having to infer for him or herself underlying themes, patterns and principles. This book is designed to make explicit much of what is only implicit in many other texts, so that you can speed your learning of PROLOG.

Most books concentrate on PROLOG very much from the point of view of *writing* programs. This is fine – but they do not also offer specific help in coping with the difficulties of *reading* programs written by other people. But this is one of the most important, yet difficult, aspects of learning to program.

As with any foreign language, it is easy, once presented with some rudimentary grammar and vocabulary, to begin to write coherent sentences. But to develop competence and extend our linguistic command we must be able to read text written by others more fluent in the language. This skill has its own particular problems. It is not just a question of being given more and more grammar and vocabulary. We need to develop strategies for approaching and deciphering difficult text.

Good programs abound in the increasing number of texts on PROLOG. They are a rich potential source of learning. But for this potential to be tapped, the person learning PROLOG must develop program-reading skills. Yet reading programs is often much more difficult than writing them. One of the objectives of this book is to help you to do this.

Experts do not generally achieve their expertise on account of having superior mental hardware compared to that of less gifted individuals. To a large extent, we are all working with similar equipment. It's how we use it that's important. This book is geared to helping you optimize this use. It is based on a number of principles derived from research into human learning. If you are interested in the details, read on. But if you would prefer to get straight into PROLOG, skip straight to Section A (page 9).

LESSONS FROM LEARNING RESEARCH

Here are some findings from learning research upon which this book is based. For the interested reader, references to papers describing the research, referred to by numbers in the text, are given in the 'References' section (page 249).

But first, we need to know something of the nature and limitations of a central component in all our learning processes – our memory. The diagram shown in Figure 2 depicts how many psychology researchers view the human memory system.[1,2]

Grouping pieces of information into more economical chunks reduces memory load, which may otherwise prevent effective learning. Our short-term or 'working' memory has a limited capacity. We can hold only a certain number of discrete units of information in working memory for processing at any one time. But if we can 'chunk' information so that each individual unit contains a lot of items, we can handle more complex processing.[1,2] Reading PROLOG programs often involves keeping track of a good many variables and their individual values, often through more than one recursive cycle (recursion will be thoroughly explained in Section B). To try to keep track of each of these values as isolated items is too much for our working memory. We lose track, and confusion sets in.

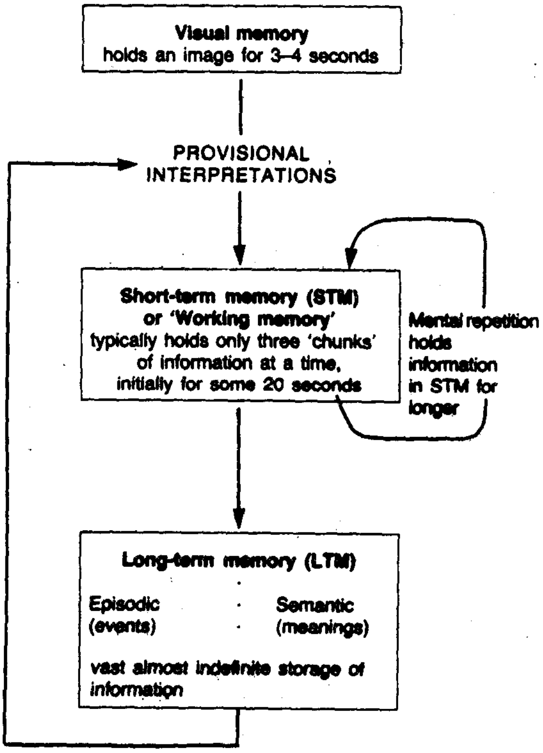
Visual memory
holds an image for 3–4 seconds
PROVISIONAL INTERPRETATIONS
Short-term memory (STM) or 'Working memory' typically holds only three 'chunks' of information at a time, initially for some 20 seconds
Mental repetition holds information in STM for longer
Long-term memory (LTM)
Episodic (events)
Semantic (meanings)
vast almost indefinite storage of information

Figure 2

Retention of information in long-term memory is also much more efficient if we identify 'deep', superordinate themes which bring together and organize otherwise discrete items of information (again, a form of 'chunking'). Without such deep organization, surface learning may occur, in which attempts at memorization replace deep understanding.[3,4] Unless we can develop such deep understanding of concepts in long-term memory, we cannot approach more complex programs which incorporate them within more complex structures.

In the development of any skilled performance, we can further reduce load on working memory by 'compiling' knowledge. That is to say, we can move away from the use of very general-purpose processes which rely on moving a lot of data into working memory for processing, to the development of specialized procedures which have 'built-in' data.[5,6] In this way, we can save working memory space for other tasks. This involves amalgamating discrete processes into more complex units. This point is similar to the first one – we need to be able to call up procedures, each of which contains a lot of knowledge, yet does not tie up working memory. This is important in reading complex PROLOG programs, when we need to assign one unit of working memory to a whole complex of procedures, which represent only one of a number of themes we are trying to keep track of. Without such self-contained units of information, our processing capacity is taken up with what in the context of the whole program may be just one component detail.

We learn the unknown in terms of the known. The central importance to effective learning of bridging the gap between learners' existing knowledge, and new information which is to be learned, has long been recognized.[3] More recently, the importance of learning by *analogy* has been stressed.[7] This is one way of allowing the processing of more complex information than would otherwise be the case. We all have well-developed compiled procedures covering certain areas of knowledge. We can use them at an intermediary stage in the development of new procedures – as structures or 'scaffolding', helping us cope with greater levels of complexity than we could otherwise cope with. For example, we can handle information about a new game at a higher level of complexity if we can use knowledge we already have about other games. It won't be exactly the same, but we can borrow the structure to some extent to help us more quickly assimilate knowledge about the new game. Once we have done so, this mental scaffolding can be jettisoned to reveal the edifice of our new understanding.

Effective learning requires an effective blend and balance of relatively narrow 'procedural' and broader 'strategic' knowledge. Taking a football example, procedural knowledge relates to the individual techniques used by a player – controlling the ball, shooting at goal, etc. Strategic knowledge is a more holistic understanding of when to apply these techniques (and combinations of them) within the context of a particular football match. This type of understanding is much less concerned with exact procedural details of the specific techniques – how they work and how they can be improved – and more concerned with how they can be applied in particular circumstances, and how combinations of them can form particular match strategies.[5,7,8]

Similarly, we can understand PROLOG techniques in a relatively narrow procedural way – in terms of how particular techniques actually work. We can also understand PROLOG techniques in a broader strategic way – in terms of what types of problems they are particularly suited to and how they may be combined to form different problem-solving strategies. We need this relatively holistic level of thinking about PROLOG programs. But without also paying attention to procedural details, we can find that our programs go sadly wrong – and that we may have overestimated our understanding of what is really going on.

This book is organized around these learning principles.

Particularly in the case of recursion (Section B) – one of the most important and difficult aspects of PROLOG programming – the book concentrates on identifying and presenting a number of recurring patterns and themes in PROLOG which underlie both simple and complex programs. These patterns are used to enable unfamiliar programs to be approached in terms of familiar, known concepts. They form chunks which, once fully understood and compiled, can be taken for granted and applied without using up working memory store. We need to be able to do this in order to see how more complex programs are built up.

The reader is encouraged to develop deep understanding rather than surface memorization, by being given practice applying new concepts to new situations. A given theme or pattern may crop up in a variety of programs from simple to complex, and may be disguised by surrounding detail and use in different contexts. The reader is encouraged to develop skills in recognizing underlying patterns and themes despite such surface differences. He or she must understand a given theme at a deep level, rather than memorizing its particular manifestations in specific contexts. Once this is done, complexity is reduced, making it easier to approach unfamiliar programs.

Where appropriate, extensive use is made of *analogies*. New concepts are presented explicitly in terms of very familiar concepts. This allows the relatively quick handling of complexity. The unfamiliar becomes understood in terms of the familiar. To supplement the use of analogies (and to provide help where analogies are less appropriate – for example, Section C) a bridge can still be laid between the learner's existing knowledge and new ideas. In this case, a more tangible form of mental 'scaffolding' is erected. This takes the form of diagrams and annotated program listings. These show explicitly the changing values of variables at different stages in the running of a program. They make visible simultaneously complex data structures and the larger chunks of which they are a part, and they show the complex flow of control as the programs run.

In Section E, the relationship between detailed procedural approaches and more holistic strategic approaches to PROLOG programming are discussed. The reader is given practice in developing his or her strategic thinking in relation to PROLOG programs. Both procedural and strategic approaches are necessary for effective learning and application.

Section A
First gear

This section is meant for the newcomer – it explains the basics of PROLOG. The reader with some familiarity with PROLOG may prefer to skip to Section B (page 59).

Section A deals with

- Basic input and output
- Facts, questions and variables
- 'and', 'or' and the 'anonymous variable'
- Rules
- Backtracking
- The 'cut'
- Lists
- Strings