LEARNING MICRO-PROLOG A PROBLEM-SOLVING APPROACH

Tom Conlon

LEARNING MICRO-PROLOG A PROBLEM-SOLVING APPROACH

Tom Conlon

Computer Education Department
Moray House College of Education
Edinburgh, Scotland

Addison-Wesley Publishing Co, Inc.
Reading, Massachusetts · Menlo Park, California ·
Wokingham, England · Don Mills, Ontario ·
Amsterdam · Sydney · Singapore · Tokyo · Mexico City ·
Bogotá · Santiago · San Juan

Library of Congress Cataloging-in-Publication Data
Conlon, Tom, 1954-
Learning Micro-PROLOG.

Bibliography p
Includes index
1. Micro-PROLOG (Computer program language)
I. Title.
QA76.73.P76C66 1985 005 13'3 85-15051
ISBN 0-201 11241-8

Illustrations by Michael Davidson.
Cover design by Marshall Henrichs.
Typeset by Computerset (MFK) Ltd, Ely, Cambridgeshire.

ABCDEFGHIJ-AL-898765
First printing, September 1985

Preface

This book is based on a course of work which has been written, taught, and extensively re-written over an approximately two-year period. It centers on the programming language PROLOG, and on the use of PROLOG to solve problems. The version used throughout is micro-PROLOG (release 3.1 or later), which is now available for a wide range of hardware such as MS DOS and PC DOS systems including the IBM PC, and most CP/M systems. Micro-PROLOG for various other computers, including the Apple II and the Commodore 64, will be released in the very near future.

My original students were drawn variously from the upper years of secondary education. Naturally, then, I hope that the material between these covers will interest this group. But computers respect neither age nor position, and really there is no reason why anybody should not learn from the book, whether they happen to be following a course of study or not. (The majority of present-day computing professionals, for instance, will find that most of the ideas contained here are new to them.) I have taken for granted no particular knowledge of mathematics or computing, although the book does assume that the reader has access to a PROLOG computer system. This is strongly recommended, although not absolutely necessary.

I believe that the content is worthwhile for two main reasons. The first is that it provides an introduction to a radically different approach to computer programming, one which is powerful, relatively easy to understand and – especially in view of fifth generation computing developments – likely to be of sharply increasing importance. The second reason is that a PROLOG computer system happens to be a marvellous tool for problem-solving, an activity which most human beings (given half a chance) find compelling. Problem-solving is both useful and educational too, but I think the reason why people choose to spend so much time solving problems (with Rubik cubes and adventure games, bridge matches and crosswords, dominoes and chess, crime stories and logic puzzles . . .) is that problem-solving is great fun in its own right. Hopefully, after reading this book you will think so too.

Acknowledgements

Many people have contributed to the ideas which I have tried to present in this book. The influence of both Robert Kowalski and George Polya is especially acknowledged.

Keith Clark of Logic Programming Associates (which has done tremendous work in making PROLOG available on low-cost microcomputers) is to be thanked for his help, as is Paul Fellows of Acornsoft.

Many colleagues and friends have read and commented upon various parts of the manuscript at different stages, and I would like to thank George Connell, Brian Higgins, John McCarney, John McGee, Shiona McDonald, Douglas MacKenzie, Jeremy Nicoll, Margaret Somerville, Robb Sutherland and Tony van der Kuyl.

Peter Barker, my head of department, is to be thanked for his support and unfailing good humour.

Credit is due to the pupils of Jordanhill College School, Glasgow, and Beath High School, Fife, who were the main guinea-pigs in the development of this material, and to the teachers who made it possible.

A large number of teachers and lecturers have been the victims of my attempts to explain PROLOG, and for suffering me without silence I am grateful to them all.

Jean Casey is to be thanked again.

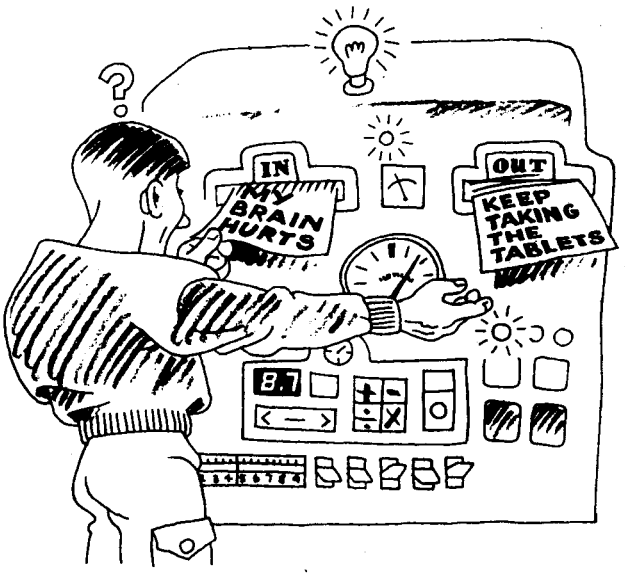
None of the people mentioned above are to blame for any weaknesses or mistakes, whether real or imaginary, which are discovered in this book. These, alas, must be down to me.

Edinburgh
September 1984

Tom Conlon

Contents

1 The problem-solving machine



The history of the modern electronic computer spans less than half a century, but it is a history of dynamic change. We begin with a brief consideration of the place of PROLOG in that history.

TWO STYLES OF PROGRAMMING

A computer is a machine for solving information problems. It is a tool which expands the power of the mind, just as older tools — the lever, the steam engine and the aeroplane, for example — expanded the power of the human body.

To use a computer for problem-solving requires that we communicate with it. At root, this communication is through the activity of **programming**.

One way of programming a computer is to tell the machine exactly what to do. That is, we present it with a **sequence of instructions**

which we know will lead to a solution to the problem in which we are interested. Such a program might look like this:

```
1 FOR N = 3 TO 100
2 LET I = 2
3 IF N/I = INT(N/I) THEN 6
4 IF I < SQR(N) THEN LET I = I+1 : GO TO 3
5 PRINT N
6 NEXT N
```

The computer obeys the instructions more or less blindly, like a slave which has been given its commands.

This is the **imperative**[1] style of programming. It is the style which has been employed with the first four generations of computers, from the pioneering vacuum-valve machines of the 1940s through to computers which were based in turn on transistors, on the first silicon chips and on the microprocessors of today. A variety of imperative programming languages have been developed, for example BASIC, LOGO, COMAL and Pascal: these languages reflect different ideas about the best ways in which to write the instructions.

The imperative style has been established for so long that many people (especially people who have programmed computers) find it hard to imagine any alternative to it. But an alternative becomes obvious when you ask yourself a simple question. Suppose you have a problem, and you are lucky enough to have access to a really powerful problem-solving machine. How would you actually **like** to be able to communicate with the machine? Most people agree that they would wish to be able to **describe** their problem to the machine in general terms, probably by talking to it in English. The computer should then supply the answer. That, after all, is how we communicate with human problem-solvers such as, say, doctors, lawyers, architects or whomever.

For the moment, let us leave aside the desirability of talking to a computer in English, and concentrate on the idea of communicating any kind of a **description of the problem**. Is the idea feasible? Could a computer solve a problem just on the basis of a description? In theory, yes: providing the description contains all the information which is needed to solve the problem, then a smart enough computer should be able to use it to work out the answer. With such a computer, we would not need to provide instructions telling it **how** to use the information: it would be enough to declare all the relevant aspects of the problem to be solved, and let the machine take over from there.

That is the essence of the **declarative** style of programming.

THE DECLARATIVE APPROACH

What will a program written in the declarative style be like? It should not be too different from the kind of descriptions which are supplied to human problem-solvers. For instance, in communicating a problem to a doctor, a patient will give descriptions like:

The pain is on the right side of my chest.

and:

My head spins if I climb the stairs quickly.

These are statements communicating a **fact** and and a **rule** respectively. A declarative program written for a computer is precisely like this: it is a description which takes the form of **a set of facts and rules**. Let us look at an example of one of these programs. The following is part of a declarative computer program designed to solve medical problems:

```
1  Asthma sufferers should avoid smoky atmospheres.
2  The atmosphere in Ed's Bar and Grille is smoky.
3  Henry is an asthma sufferer.
```

If the problem is:

```
What should Henry avoid?
```

then a computer using this program should be able to answer `Ed's Bar and Grille`. It's quite reasonable to expect the computer to be able to find this answer, since it can be arrived at **by applying logical deduction to the description.** And logical deduction is something which a smart computer should be good at.

Notice however that we should not expect more than is reasonable. The patient in the doctor's surgery can dither, and get confused over the symptoms, and generally mess up the description of the problem; the doctor can usually be relied on to make sense of it all. It would be foolish to put such obstacles in the way of a computer. The declarative approach is more likely to be successful if programs are very precise, ideally written in some language which is tailored for exact logical description. Notice, too, that in solving a problem the doctor

adds a large quantity of additional knowledge to the patient's description; logical deduction is applied to the combination of the two. A declarative program for a computer on the other hand will be required to supply **every scrap** of information needed to solve the problem. We can expect that the problem-solving machine is capable of logical deduction, but we mustn't assume that it has access to any more knowledge than that which is contained within its program.

Yet the attractiveness of the declarative approach is clear from a comparison of the two programs shown above. Declarative programs should always be easier to understand. Since they are simply descriptions of problems, and not recipes for solving them, they should be easier to write. The meaning of a declarative program is self-evident, whereas, to understand the meaning of an imperative program, you are forced to think in terms of the behavior which the program will produce on a machine. And the connection between the machine's behavior and the problem which is being solved can be very obscure.

The end result of the declarative approach, of course, should be computers which are more effective problem-solving machines at the service of humankind.

PROLOG AND THE FIFTH GENERATION

Why have the first four generations of computers been programmed imperatively? The short answer is that computers have mainly been too small, too slow and above all too stupid to support the declarative style. But the fifth generation, which is being developed now, promises to overcome these limitations. Exactly what fifth generation computers will be like, nobody yet knows: but it is a very sure bet that, while imperative programming will not disappear, the declarative approach will become increasingly important.

The programming language which is at the center of some of the most important fifth generation developments is called PROLOG. PROLOG — the name stands for 'PROgramming in LOGic' — is the most successful declarative language which has been developed to date. Already it has been used to build expert systems, to analyze natural languages, to prove theorems in mathematics, to construct translators for computers, and to solve problems in a host of other areas. It must be said that PROLOG, as it exists today, falls short of the declarative ideal which has been described above. Its limitations principally mean that programmers cannot altogether ignore the

ways in which their descriptions will be used by the computer. In spite of this, a PROLOG computer system is still an enormously powerful problem-solving tool. This we shall discover for ourselves in what follows.

We end this introduction with some PROLOG terms which we shall require immediately. A set of facts and rules which makes up a problem description in PROLOG is called a **program** or **database** (the two words are used interchangeably). Each fact and rule is known as a **sentence**; sentences must be written in a special precise form known as **sentence form**. A question to the computer is called a **query**. A query is really a request that the computer should solve a problem. This it will try to do by applying logical deduction to the sentences in the database.

SUMMARY

(1) Computers are tools for solving problems. The imperative style of programming is concerned with communicating with computers by giving them instructions. The declarative style is concerned with giving descriptions.

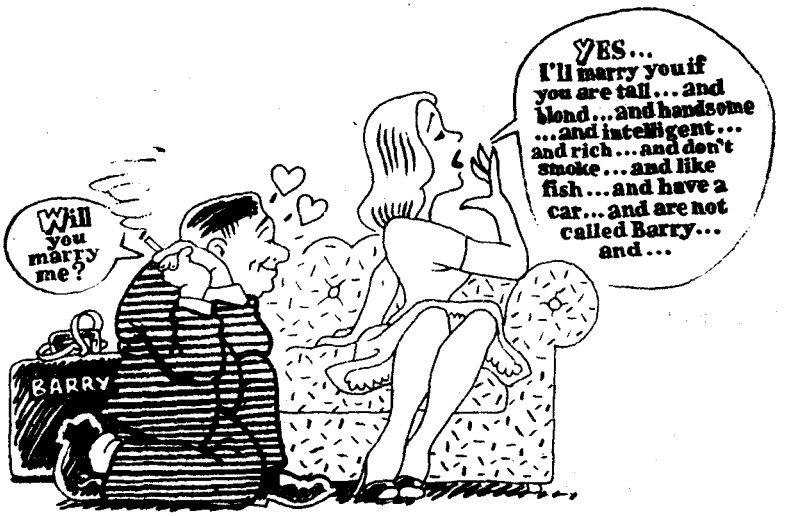
(2) A declarative program consists of a set of facts and rules, written in a precise form, which contains the information necessary to solve the problem. The computer solves problems by applying logical deduction to the facts and rules.

(3) PROLOG is the first successful language for declarative programming. It is the core language of important fifth generation computing developments.

NOTES

(1) Literally, 'imperative' means 'commanding'.

2 Writing and using descriptions



The aim of this chapter is to provide an informal introduction to the PROLOG way of writing descriptions. We practice writing facts and rules in sentence form, we find out how to enter a database into the computer, and we make up queries to solve some simple problems.

FACTS AND RULES

Let us pretend that we have arrived at a party. The music is loud, the lights are low and we are among friends. The scene cries out for description — so we shall wander round, gathering facts and rules as we come across them, and writing them down first in English and then in PROLOG.

We quickly notice that Bill is sitting next to Jean, gazing at her admiringly. Clearly Bill likes Jean. That gives us our first fact. In the form of a PROLOG sentence, we write it as:

```
likes(Bill Jean)                                    (1)
```

The PROLOG and the English versions of the sentence are not all that much different.

Sam, the acting barman, has started to tell us a long gossip story. The details are too complicated to relate, but it can be summed up by writing down two more facts about people who like other people. In PROLOG, the facts are:

```
likes(Diane Colin)                                    (2)
likes(Janet Ian)                                      (3)
```

PROLOG facts have a special structure. They comprise a term by itself called the **predicate** followed by a bracketed list of terms called the **arguments**. Predicates roughly correspond to verbs in English sentences, and arguments correspond to nouns. So far, our facts have all had `likes` as the predicate and the arguments have been the names of individuals. Note that the order of the arguments matters: saying that Diane likes Colin is not the same thing as saying that Colin likes Diane, neither in English nor in PROLOG.

Speaking of arguments, loud voices are coming from one corner of the room. A disagreement has begun over the kind of music which is to be played. Diane is an avid rock fan, while Jean enjoys reggae and Ian favors heavy metal bands. You decide to set down the facts in PROLOG:

```
enjoys(Diane rock)                                    (4)
enjoys(Jean reggae)                                   (5)
enjoys(Ian heavy-metal)                               (6)
```

The predicate here is `enjoys`. The first argument is the name of an individual; the second is the type of music which is enjoyed. Sometimes we use the word **relation** instead of predicate, and say that the above are 'facts for the `enjoys` relation'. Notice the hyphen in `heavy-metal`, making it one term instead of two. Spaces and brackets are used to tell PROLOG where one term ends and another one begins. Putting a hyphen in `heavy-metal` makes it a single term, like `rock` and `reggae`.

Suddenly your eyes start to water and you are coughing. The air has grown thick with tobacco smoke, and you look round for the culprits. Just as you expected: Diane and Ian — the two notorious puffers — are smoking away. The facts are simply described in PROLOG:

```
smokes(Diane)                                    (7)
smokes(Ian)                                      (8)
```

This time the predicate is `smokes` and the single argument is the name of the smoker.

But smoking is not Diane's only vice. At this very moment, she is being given a large gin and tonic by Sam. It looks as though Sam may have a busy night: he has just given a cola to Colin, and now he is about to fill a glass of white wine for Jean. You note down the facts in PROLOG:

```
gives(Sam Diane gin-and-tonic)                   (9)
gives(Sam Colin cola)                            (10)
gives(Sam Jean white-wine)                       (11)
```

Notice again the use of hyphens to ensure that each fact has the same pattern as the other facts for the `gives` relation. It is important to be consistent about the number of arguments and the position of each argument, as will become clear later.

By now the structure of a predicate followed by a list of arguments will be quite familiar. This structure is so fundamental in PROLOG that it is called **atomic**. Anything which has the structure is an **atom**. An atom describes a **relationship** between individuals or objects: a fact is a sentence which asserts that a relationship is true.(1)

Meanwhile, Janet has told you that whenever she smokes, she becomes ill. For her, it is an absolute rule. That gives you a chance to write down your first ever PROLOG rule. A rule has two parts known as the **consequence** part and the **condition** part. It is important to identify each part. In our rule, the consequence that `Janet is ill` holds true on fulfillment of the condition that `Janet smokes`. Turning the consequence and the condition into atoms, we can write `ill(Janet)` and `smokes(Janet)` respectively. Then a PROLOG version of the rule is the sentence:

```
ill(Janet) if smokes(Janet)                      (12)
```

Every PROLOG rule takes the form of an atom which describes the consequence part of the rule, followed by the word '`if`', followed by the atoms which make up the condition part (joined by the word '**and**' or '**&**' if there is more than one). Sometimes we call the consequence part the **head** and the condition part the **tail** of the rule. As you can see, the correct form (or **syntax**) of a PROLOG rule is very simple, but

for the computer to understand the rule the syntax must be followed exactly.

We get our second rule from Jean. She has not yet met Ian, but if he turns out to be a heavy metal fan then she intends to partner him in a dance. You note the consequence part of the rule: it is that `Jean partners Ian`. You note the one condition: that `Ian enjoys heavy metal`. To obtain a PROLOG translation of the rule, we turn these two into atoms and connect them with an `'if'`. So we write:

```
partners(Jean Ian) if enjoys(Ian heavy-metal)                    (13)
```

Jean seems to have started something here. Janet says that she too will partner Ian if he is a heavy metal fan, but she adds the extra condition that Ian must like Bill. You note the consequence: it is that `Janet partners Ian`. You note the two conditions: that `Ian enjoys heavy metal` is one; and that `Ian likes Bill` is the other. Re-writing each of these in atomic form leads to a PROLOG version of the rule:

```
partners(Janet Ian) if                                           (14)
                        enjoys(Ian heavy-metal)  &
                        likes(Ian Bill)
```

Setting it down over several lines like this makes the rule a little easier to read. Note that where a rule has more than one condition, as here, **all** the conditions must be true in order to prove that the consequence is true. So Janet might not partner Ian if he is a heavy metal fan but does not like Bill, for instance.

Diane isn't one to be left out of this game. She proclaims that she will partner Sam in a dance if Sam gives her a gin and tonic, on condition also that he does not smoke. You identify the consequence and the two conditions, and you set down Diane's rule as a PROLOG sentence:

```
partners(Diane Sam) if                                           (15)
                        gives(Sam Diane gin-and-tonic)  &
                        not smokes(Sam)
```

A set of two or more conditions joined with '&' or 'and' is sometimes called a **conjunction**. The second condition in the conjunction above is an example of a **negation**. A negation is formed by putting the word 'not' in front of an atom. Negations can appear anywhere in the tail of a rule; but it is one of the restrictions of the PROLOG language that

```
likes(Bill Jean)                                              (1)
likes(Diane Colin)                                            (2)
likes(Janet Ian)                                              (3)

enjoys(Diane rock)                                            (4)
enjoys(Jean reggae)                                           (5)
enjoys(Ian heavy-metal)                                       (6)

smokes(Diane)                                                 (7)
smokes(Ian)                                                   (8)

gives(Sam Diane gin-and-tonic)                                (9)
gives(Sam Colin cola)                                        (10)
gives(Sam Jean white-wine)                                   (11)

ill(Janet) if smokes(Janet)                                  (12)

partners(Jean Ian) if enjoys(Ian heavy-metal)                (13)
partners(Janet Ian) if                                       (14)
                       enjoys(Ian heavy-metal)  &
                       likes(Ian Bill)
partners(Diane Sam) if                                       (15)
                       gives(Sam Diane gin-and-tonic)  &
                       not smokes(Sam)
```

Fig. 2.1 PARTY.

the head of a rule must not be a negation. That is, the consequence of a PROLOG rule must always be an atom.

At this juncture we shall leave the party. The cool night air and the quiet comes as a welcome change after all that smoke and noise. In any case we now have quite an interesting description of the evening in the form of a database which we shall name PARTY. It contains fifteen sentences — eleven facts and four rules — which for convenience are gathered together in Figure 2.1. For now, it's time to find out how we can transfer PARTY from our notepad on to a computer so that PROLOG can use it to solve some problems.[2]