STUDYING NOVICE PROGRAMMERS: WHY THEY MAY FIND LEARNING PROLOG HARD

Josie Taylor and
Benedict du Boulay

ognitive Studies Research Papers
erial No. CSRP.060

he University of Sussex
ognitive Studies Programme
chool of Social Sciences
almer, Brighton BN1 9QN

One of the aims of workers in Artificial Intelligence (hereafter AI) is to develop programming environments and programming languages that can be used to produce "intelligent" systems. Such environments (e.g. Interlisp, Poplog) and languages (e.g. Lisp, Prolog and Pop-11) are specially designed for this kind of task in the same way that other programming languages such as Fortran and Cobol are tuned to the needs of the scientific and commercial programmer. The success of this tool-building activity, whether within AI or some other field, can be judged on a number of criteria, one of which is the power of the tool and another of which is the usability of the tool. Investigating how the human user copes with the complex task of learning to use and exploit computing facilities is the concern of the field of Human/Machine Interaction (HMI).

There are many aspects to the 'interface' between humans and machines - HMI research addresses such psycho-physical, ergonomic issues as the layout of keyboards and terminal screens, as well as more cognitive issues to do with programming practices and programming languages. It is interested both in the qualities of a system that affect expert performance as well as in the learnability of the systems by both naive and experienced users (for general discussions of the relevant issues see e.g. Smith and Green, 1980; Coombs and Alty, 1981; Monk, 1985; Shneiderman, 1982).

Psychologists - in particular cognitive psychologists (e.g. Anderson, 1982; Norman and Draper, 1986) - have turned to the development of a user psychology, not only because of the necessity of identifying the needs of users in interaction with increasingly complex computer systems, but also because computing offers a microcosm of many of those perceptual and intellectual issues in which psychology has always been interested and, in many cases, a ready means of capturing useful data about human performance.

In this paper we focus on a particular, new type of 'high-level' programming language, Prolog. This is used mainly in AI although Education is showing a growing interest in it for teaching such diverse subjects as History, Logic and Mathematics (see e.g. Ramsden 1984). Our main objective here is to delineate some of the issues that Prolog beginners have to deal with, but first we explain the notion of a 'high-level' language, and the concept of a 'notional machine'.

The term high-level suggests two ideas. One is that the kinds of entities one can manipulate in the programming language are similar (in some way) to the kinds of entities in the problem situation. The other is that the means provided for such manipulations fit the task, are powerful and easy to use. Broadly speaking, the higher level a language is, the less it need be constrained by the particular hardware upon which it runs and the more it is designed for a specialised class of problems.

Because the design of high level languages tends to reflect the structure of problems rather than the structure of the computer on which they run, they allow beginners to write interesting programs when they may know very little about how a computer is physically constructed (i.e. its hardware), or about how its internal workings are organised on a global scale (i.e. the operating system software which allows different kinds of operations to be carried out).

For most programming languages, including Prolog, the crucial notions for the beginner are first, the 'workspace' area where activities are conducted, second, the 'building blocks' provided by the particular programming language for constructing 'mechanisms' and third, the means of making these mechanisms perform. This idealised view of the system, shorn of irrelevant detail, is described as the 'notional machine' (du Boulay et al., 1981) - where the notional machine is an 'idealized conceptual computer whose properties are implied by the constructs in the programming language employed'. An analogy here is to think of a programming language as being like a Meccano set or a Lego set with standard parts for standard tasks, such as a wheel and bearing. Building a program is not unlike building a model mechanism with one of these sets. As a beginner, one needs to master the accepted means of putting the parts together, ways of translating problem descriptions (e.g. a crane) into the 'language' of the parts provided and the expected uses for what at first may seem rather oddly shaped bits and pieces. Of course, one also has to master pragmatic issues, such as what to do when the model does not work as intended or how to divide up a complex mechanism sensibly into buildable constituent assemblies.

Until recently most programming languages for beginners stressed a procedural/command oriented approach to structured problem solving, and the language constructs supported this approach (e.g. Logo, Basic). A facet of this problem solving approach is that it fits well within the architecture and design of computing machinery, where the computer is directed by the programmer to execute step(a), step(b)...step(n). Analysis of beginner's difficulties has typically focussed on the development of goals and plans in the programming process with emphasis placed on the way that novices build mal-formed plans or join sub-plans together incorrectly (see e.g. Soloway and Ehrlich, 1984). In their analysis it is the plans which express the procedural knowledge of how to get things done in the particular programming language.

However, the development of Prolog (Colmerauer et al. 1973) has introduced a new type of language which is based on first-order predicate logic. Here the programmer specifies what is true of the eventual problem solution, leaving the computer to sort out the steps towards finding the solution. Ideally, in the logic programming ethos, the programmer can specify a problem solution in logic, and that specification will run as a program. This is the distinction made much of in the Prolog literature between telling the machine HOW to solve the problem (by issuing commands in a procedure) and telling the machine WHAT has to be true of the solution in the end.

It has been claimed, primarily by Kowalski (1979), that 'logic' programming languages like Prolog are easier to learn and use than other, high-level, AI languages because predicate logic is 'human-oriented', and is therefore more readily understood than other programming formalisms. This argument would not convince a psychologist, and, in fact, given the low level of performance on logical deductive reasoning tasks noted in the psychological literature one might predict a whole host of potential problems. Unsurprisingly, observations of students learning Prolog confirm that many of them do not find it easy or natural (Taylor, 1984).

Most languages tend to be adopted by a band of people eager to claim that their language is natural, easy to learn and just right for this or that class of problem. Despite the hyperbole, it is clear that learning to program in any language is not necessarily easy. The interesting question that Prolog poses because of its structure (which we discuss below) is whether problems in learning to program centre around learning to cope with the machine and its idjosyncracies, or whether problems are to do with thinking in a particularly constrained and sometimes counter-intuitive manner (i.e. in a formal logical way). An issue of particular concern to developmental psychologists is the question of how a student's general cognitive competence, especially the ability to handle logic and inference, interacts with the ability to learn Prolog successfully.

Prolog does offer some automatic facilities which, in theory, ought to relieve the beginner of certain chores associated with understanding the machinery in learning to program. Prolog can, for example, make inferences from statements. Given the information that 'Firemen wear red braces' and that 'John is a fireman', Prolog can automatically infer that John wears red braces if it needed to. Furthermore, by a process called backtracking, Prolog can find as many solutions to some query as there are in the database. After finding the first successful answer to a query, the user can ask Prolog to look for any others, and so obtain, upon occasion, multiple solutions to one question.

This process, in turn, is facilitated by the use of 'variables'. The way in which Prolog uses variables is interesting, and differs from the way that most other programming languages use them. Discussion of these subtleties is not essential for our purposes here, but we occasionally need to refer to the concept of variables so we provide a somewhat simplified example of their use.

The easiest way to think of a variable is as an empty slot which can be filled by anything that 'matches' and is of the correct form. The most straightforward case of filling a variable is when the variable is empty at the outset (i.e in the query, say) and the database contains information of the correct form which allows the variable to be filled. The following example illustrates the basic process. The database in the computer has an entry regarding the colour of elephants:

```
      DATABASE ENTRY:          grey(elephant).
```

The user types in a query asking 'what do we know about that is grey?':

```
      USER'S QUERY:            grey(X).
```

'X' here representing the variable, or empty slot that will be filled. Prolog will match the query with the database item, and X will represent, or be linked to 'elephant' in the query. Prolog will answer:

```
      PROLOG'S RESPONSE:       X = elephant
```

This example illustrates the use of a variable in its most elementary form, and there are many ways in which this basic idea can be elaborated and extended. The relationship of variables to backtracking is that if the user wishes to search further to see if there is anything else in the database which is grey, he or she can say, "OK, but now look for any other instances of database entries that match my original query". At this point, X (the variable) goes back to being empty (the link between 'X' and 'elephant' being broken), and Prolog continues looking down the database for any other legal match. If the database contains the statement that 'skies are grey', e.g.

```
      DATABASE ENTRY:          grey(skies).
```

then X in the query will stand for 'skies' and Prolog will respond:

```
      PROLOG'S RESPONSE:       X = skies
```

So at first sight Prolog looks as though it should be fairly easy to learn. However, the language's apparent simplicity is beguiling, beginners occasionally being lulled into a false sense of security and then finding themselves quite at sea with no idea how to proceed and some uncertainty about whether they have correctly understood those topics they thought were secure. The automatic facilities, whilst simplifying certain sorts of tasks, introduce their own subtle difficulties for the unwary beginner, and a large part of a Prolog programmer's skill lies in knowing how to constrain them.

Our objective is to see what, if anything, is special about Prolog in this respect, and to discuss some of the more misleading claims made on its behalf. One of the issues that will emerge is that we do not yet have a convincing picture of what the ideal Prolog notional machine should look like, nor do we yet have good strategies for teaching the language (see Bundy and Pain, 1985).

Difficulties for Novice Programmers

Difficulties associated with learning most languages, including Prolog, can be separated into six overlapping classes. These classes should not be thought of as general stages in learning to program or particular stages in the process of producing a working program. Rather they represent views of the programming process at different levels. Each of these views needs to be elaborated and assimilated by the the student in order to become expert.

(i) There is the general PROBLEM OF ORIENTATION, finding out what programming can be used for, what general kinds of problem can be ackled and what the eventual advantages might be of expending effort in earning the skill. For example, at Sussex University we run short ntroductory courses for Arts undergraduates who want to acquaint nemselves with AI approaches and programming techniques (du Boulay, 986). Such students sometimes find the high-level theoretical iscussions of AI interesting and stimulating, but find the programming omponent of the course difficult because it is not clear to them what ney could ever use it for - the problems computers can solve are not art of these students' conceptual framework.

ii) There are the difficulties of INTERPRETING PROBLEM DESCRIPTIONS. nis includes getting a precise enough understanding of what the problem s in order to determine what might count as a solution. This will ormally involve identifying what elements, relations or objects in the iven situation need to be taken into account and what links there are etween the solution and important factors in the problem. Even if the eginner does not know what the solution to the problem will eventually e, he or she does have to know the kind of shape a solution would take n order to progress to the next stage.

iii) There are difficulties associated with the mapping from an nderstanding of the problem to an understanding of the general roperties of the THE NOTIONAL MACHINE (i.e. the simplified conceptual omputing system with which the beginner is working, see above). urthermore, the beginner is often not only dealing with one notional achine, that is the one associated with the language they are using. ne beginner will usually also have to master an editor, and an perating system, and both of these have notional machines associated ith them - i.e. at no point are we talking about the actual physical achine built of metal and silicon with wires and disks, but idealised mental models of part of the computer's functioning. If the beginner is lucky, the three notional machines of the programming language, the editor and the operating system may be seamlessly interwoven, or at least not conflict with one another. Whether it is one notional machine or many, this will entail realising how the behaviour of parts of the tangible machine (the keyboard and screen) relate to these notional machines - for example whether the information on the terminal screen is a record of prior interactions between the user and the computer, or is instead a window onto some part of the machine's innards.

Second, there are difficulties in transforming an understanding of the problem into the terms of reference of the programming language. If the concepts embodied in a language are entirely new to the student, this can sometimes be a slow and difficult task. One of the subtle changes that occurs in the programmer is that he or she comes to view new problems in terms of the potentialities (and limitations) of the available tool. Those who argue that a particular language is 'natural' are usually right: they have been changed by their own exposure to the language and now see problems in its terms, which may prevent them from perceiving the difficulties that a new learner is having.

The beginner has to recognise and then integrate these two components of the notional machines to gain an understanding of the sometimes complex execution of a program.

(iv) There are problems associated with the notation of the various FORMAL LANGUAGES that have to be learned, including mastering both the syntax and their underlying semantics. Understanding syntax usually means knowing what kinds of symbols are legal, and how they may be strung together to produce certain kinds of effects. The semantics may

be viewed as an elaboration of the properties and behaviour of th
programming language's notional machine, crudely sketched above
Relating syntax to semantics involves realising which strings of symbol
will make the computer produce what particular types of 'actions'.

In practice this would entail understanding how the notional machin
works in its own terms, mostly independent of any particular problei
(which is what distinguishes this level from that described in (iii
above).

(v) Associated with notation are the difficulties of acquiring STANDAR
STRUCTURES, cliches or plans that can be used to achieve small seal
goals, such as working down a list of items performing some operation
or transforming one structure into another. For example, Ehrlich an
SoJoway (1982) have investigated the 'tacit plans' that expert
construct from experience for dealing with standard situations. Th
plans are tacit because the expert may not be consciously aware of thei
presence. So, an expert can look at some problem description, decid
that a certain type of construct is appropriate, and employ th
corresponding plan of action without devoting special thought to th
matter, whereas beginners have to generate a plan afresh every time.

The hypothesis is that experts assign a functional role to programmin
structures and eventually disregard the execution details (i.e. how th
computer actually does it) thus enabling them to take a higher leve
view of the programming process. Beginners, on the other hand
understand programs by examining how the computer does something, whic
is a correspondingly lower level view. So, given a program to read an
understand, experts will be concerned with WHAT it does, what are th
inputs, what are the outputs, and so on, whereas novices interpret HO

; does it, and may not retain information about what the overall goal
' the program was.

nee there often are standard ways of accomplishing certain effects in
ty computing language, it is useful for these to be taught directly to
udents, rather than have them laboriously work them out for
temselves.

i) Finally there is the issue of mastering the PRAGMATICS of
ogramming, that is learning the skill of how to specify, develop, test
td correct ('debug') a program using whatever tools are available.

me of these six issues are entirely separable from the others and much
 the shock of the first few encounters between the learner and the
astem are compounded by the student's attempt to deal with all these
fferent levels of difficulty at once.

lis paper concentrates on four of the above issues:

* interpreting problem descriptions
* moving from problem descriptions to Prolog programs
* the semantics of Prolog
* pragmatics.

; various points we will mention experimental work that we have
idertaken with Prolog novices (mostly undergraduates and postgraduates)
id Prolog experts (mostly research fellows and faculty).

JTERPRETING PROBLEM DESCRIPTIONS

itities and relations

le of the skills that the beginning programmer has to master is that of reading a piece of text expressing a problem and deciding what the problem is. This requires an analysis of the major entities involvec of their relationships and how a solution may be obtained in principle For some programming languages the kinds of entity about which problen can be stated are numerical or at least similarly well delineated. Thes quantities stand out from the surrounding description and are thi automatically highlighted. As Prolog allows statements to be made aboi any relationships and implications there is no clear boundary betwee things that can be described in Prolog and those that cannot. ThJ means that it is much harder for the beginner to use the kind c landmarks that might serve, say, in a numerical problem to see what *t* major entities are. One way to reduce this difficulty is to stress tf notions of relationships and individuals and give the students lots c practice in using a given restricted vocabulary to express limit€ aspects of English sentences (see Ennals, 1984 for examples of thj approach).

Even if the major entities and reJationships are clear there is *tl* problem of deciding how these should be represented. Should a proble that involves 'John liking Mary[1] be thought of as problem involving relationship, 'liking', and two individuals, 'John' and [f]Mary', whic in Prolog might look like this:

EXAMPLE 1: `likes(John, mary).`

Perhaps it should be thought of as a single event with three parameters 'John', 'liking' and 'Mary'.

EXAMPLE 2: `event(John, likes, mary).`

Another possibility is to think of it involving a single individual

'Mary', who happens to be in the state of being 'liked-by-John'.

EXAMPLE 3:   liked_by_john(mary).

There are further possibilities as well. Each of these looks rather different when translated into Prolog and has repercussions on how the rest of the program should be expressed and on what kinds of inferences can be drawn by the program. Questions of how best to solve this kind of issue depend very much on what kind of solution is being sought. In general changes of representation can have a profound effect on how easily a problem can be tackled (and on how efficient a program may eventually be). For example, although at some levels each of these representations refer to the same state of affairs, the ease with which they can be manipulated and changed in the context of a database differs. Example 1 could probably be thought of as the standard way of representing relationships in Prolog, and allows for the creation of a database in which we can represent the 'likes' relationship holding between multiple 'partners' or 'objects' - i.e. the representation is easily extensible:

```
likes(john, jane).
likes(jane, mike).
likes(sally, susan).....etc.
```

Example 2 has a structure which is more general - 'events' might be of many different types:

```
event(likes, john, reading).
event(likes, susan, sally).
event(likes, judith, pasta)
event(helps, judith, john).......etc.
```

But this very flexibility may be a trap, because the minimal amount o structure is imposed on the database. We could represent other things:

```
event(world, war, two).
event(fido, fights, cats).
event(anything, you, like)....etc.
```

Example 3 could be seen as an overreaction to Example 2. Th datastructure is somewhat overconstrained so that we can only tal either about things which are liked by John:

```
liked_by_john(apples).
liked_by_john(school).
```

or about a particular individual (Mary):

```
liked_by_peter(mary).
liked_by_mother(mary).
```

Of course for some purposes this constraint may be exactly what i needed, but the point is that Prolog allows the learner to represen information in a variety of ways, and the learner must acquire a sens of what is a 'good' way for any given problem.

Generality of solution

A widespread problem that beginners face when interpreting proble descriptions is deciding how general a solution should be. This proble may occur in Prolog to a larger degree than in other languages for th reasons given above. This issue was thrown into sharp relief when w asked a selection of Prolog experts to provide solutions for th following problem:

> "Write a program for designing an architectural unit obeying the following specifications: Two rectangular rooms, each room has a window and interior door, rooms are connected by interior door, one room also has an exterior door, a wall can have only one door or window, no window can face north, windows cannot be on opposite sides of a unit." (Coelho, Cotta and Pereira, 1982, p.63)

Although this is a very loosely expressed specification, we were rather surprised by the very wide range of interpretations that were placed upon it. Some saw it as a problem to design a general purpose architectural planner which could solve the problem automatically, using the given data as an example of the class of data that such a planner should handle. Others saw it more specifically as a puzzle about the stated unit. Others again couched their solutions as essentially constraint-checkers and as a memory aid to help someone who would do the actual problem-solving for him or herself.

FROM PROBLEMS TO PROGRAMS

High-level languages

Certain forms of logic have a long history of being used as problem solving tools, and in computing such forms are frequently used as specification languages (i.e. 'intermediary' languages which express the logical correctness of some problem solution prior to its being couched in any particular programming formalism). What Prolog has to offer is that it is both a form of predicate logic and a runnable programming language with machine independent syntax and semantics. So if a programmer specifies the logic of some solution, he or she can then simply run it (it may run slowly) instead of having to translate it into a more conventional programming language with idiosyncracies due to its dependency on hardware. This streamlines the process of program development by disposing of the need for major translations from one representation to another, and allows for more effective deployment of programmer time and effort, though minor changes within the representation may be needed for the purposes of efficiency.

At both a practical and a theoretical level we can see the advantages that logic programming languages have for the programmer (indeed one of us uses it for research). But the programmer here is often a professional who can appreciate the freedom of programming in such a machine-independent language. The situation is less clear-cut for beginners. Note that we are NOT arguing that beginners should not be taught Prolog or that it is unteachable. What we ARE arguing is that teaching it to beginners poses special problems, above and beyond those posed by other more conventional languages, precisely because it is higher level. It presents the beginner with the double edged blade of freedom to create interesting programs quickly, and the freedom to make serious errors equally as quickly.

Claims for Prolog

In common with other languages, Prolog has suffered from enthusiastic claims for its ease of use. The particular line of reasoning to support this claim for Prolog is that since Prolog is based on logic it will be simple to learn (implying that everybody is good at logic) and that it will easier for novices to express themselves in the language because logic can be understood in terms of its natural language equivalents (Kowalski, 1979).

We believe this argument is a liberal interpretation of the more formal discussion of Prolog's high-level status described above and is based on assumptions about people's natural reasoning abilities. Apart from leading novice programmers into false expectations of what the process of programming is about, and what they can expect of their own performance, this kind of interpretation runs counter to what we know of people's logical deductive reasoning abilities.

Logic, reasoning and natural language

There are three assumptions which will be questioned here: first, that predicate logic captures the logic underlying human reasoning and language; second, that logic is natural; and third, that logic has natural language equivalents. We shall take each in turn.

The crux of the argument is this: although most adults are capable of formal reasoning, the suggestion that such reasoning either conforms to, or is captured in, the precise rules of predicate logic is unwarranted. Formalisms such as classical logics, and programming languages attempt to be unequivocal. In the study of deductive reasoning - and of programming - we see people struggling to solve problems within the framework of a formalism which, in the interests of maintaining its formality, embodies certain necessary, but counter-intuitive, constraints. Such tasks are difficult and require practice, so the argument that Prolog somehow short circuits this learning pattern simply because it is based on predicate logic is meaningless.

The significant difference to grasp is between the formalisms of classical logic as devised by logicians, and the hypothesised 'logical' thinking processes used by people in their everyday lives, and it is this difference which is frequently blurred in the Prolog literature.

Psychologists are uncertain about whether a 'natural' logic exists, let alone what kind of classical logic it might resemble, if any (see Johnson-Laird, 1983). It may be true that first-order predicate logic usefully formalises some aspects of the human reasoning process, but so do many other formalisms (e.g. various branches of mathematics). Predicate logic is not special in this respect, nor should we expect untutored people to have a natural aptitude for using it any more than we would for a more mathematically oriented logic. Evidence produced by psychologists in fact supports the view that, in general, people have no natural aptitude for formal logic (see Evans, 1982 for a review of this literature).

Finally, the claim that logic has natural language equivalents throws our previous remarks into relief because part of the explanation in the psychological literature for poor performance at deductive reasoning is attributed to people failing to understand the difference between expressions couched in natural language and expressions in the formalism. The extraction of 'meaning' from natural language expressions often depends upon knowledge of causal or temporal factors known by the listener, who may, in fact end up misinterpreting the expression by importing too much knowledge into the discussion domain. Logic, on the other hand, conforms to quite a different set of rules, often dealing simply with truth functional relations between elements of expressions, regardless of the meaning. In propositional calculus, for example, propositions have one of two truth values - they are either true or false - which is not the case in natural language. Connectives are defined solely as functions of the truth values of the propositions they interrelate (see Johnson-Laird and Wason, 1977). Furthermore, standard logic does not deal with temporal or causal events, and basing interpretations on the semantics of natural language may lead to all kinds of difficulties. For example, the two English sentences: (taken from Johnson-Laird and Wason, p.79)

She inherited a fortune and he married her

and

He married her and she inherited a fortune

could be attributed different meanings by someone untutored in logic, but in the predicate calculus with its restricted meaning for 'and', they are identical.

People are quite naturally inclined to misinterpret logical expressions if these differences are not spelled out to them clearly. Furthermore, misinterpretations can arise from the content and form of the material to be analysed. Students may be led astray by 'atmosphere effects' (Begg and Denny, 1969) i.e. assertions couched in natural language create an atmosphere which, in the absence of rigorous logical analysis, may seduce people into deriving conclusions which favour that atmosphere (for example, people are disinclined to draw a 'negative' conclusion from 'positive' assertions). Lastly, untutored people are led into fallacious reasoning by not considering all the relevant information. Some diagrammatic representations of logic (e.g. Venn diagrams) try to overcome this well-known problem. Interestingly, this kind of difficulty also surfaces in the programming environment, where people tend to underspecify algorithms - e.g. telling the computer what to do if some situation arises, but forgetting to tell it what to do if it doesn't.

In theory, therefore, rather than appearing to be an easier programming language to learn than any other, one could argue that Prolog might be disastrous as a first language, since the combination of logic and programming at the same time, each with their independent constraints and their relationship in programming, would make the novice's task extremely arduous. For example, we have seen that failure to consider the full scope of the problem/solution domain in programming will result in underspecification of algorithms; in logic, it will lead to fallacious reasoning. In Prolog it may lead to both.

For practical purposes, however, the dilemma is this: if the problem statement refers to abstract notions then the task is often difficult; if concrete material is substituted for abstract (e.g. natural language expressions are substituted for symbolic expressions) then task difficulty diminishes greatly, but the reasoning process becomes error-prone because people feel free to use their natural language comprehension skills where it is inappropriate. Their interpretations of formal expressions are based on what they know of the real world, which, in certain cases, will lead them astray in the logical domain.

From a teaching point of view, we would emphasise the need to encourage learners to view Prolog programming as a problem-solving exercise, and discourage the notion of 'translation' from English to Prolog and back again, since the semantics of each is quite different.

We have some evidence of this tendency to confuse the rules governing natural language and the rules governing a formalism. At Sussex University we teach Prolog to first year Arts undergraduates who frequently have no strong science or mathematical background. One course lasts 9 weeks and is designed to introduce them to Artificial Intelligence ideas and techniques. It is not a programming course as such, but they learn how to use a high-level AI programming language, in this case Prolog.

We asked a group of these students to perform two straightforward tasks: the first was to give us an English rendering of some Prolog clauses, the second was to create a Prolog database from information expressed in English. Whilst the first task presented no difficulties at all, the second task was problematic. The English sentences were a mixture of facts and conjunctions, such as

John works hard and likes music

which are easily represented in Prolog:

```
works_hard(John), likes(John, music).
```

and some causally dependent sentences such as

John is a good student because he works hard

which are not easily represented, because Prolog, like logic, doesn't deal with causal relationships.

So how do our logically unsophisticated students cope with such a problem? We had a variety of responses, but none of our students explicitly said that this sentence could not be represented because of the limitations of the basic formalism. Some students argued that the solution to this problem is to create a rule which says: if I can prove that John works hard then I can infer that he is a good student. In Prolog form this can be read: infer that John is a good student if you can prove that he works hard.

```
good_student(John) :- works_hard(John).
```

Note that ':-' in Prolog means 'if^{f} and the inclusion of this rule in a program would enable Prolog to infer that 'John is a good student^{1} by

using another fact in the database, i.e. 'John works hard'. This is
interesting solution because it highlights the sometimes sub
differences between causal reasoning and logical inference in natu
language. Our students, in effect, are asking 'Will this happen?' rat
than 'Does this logically follow?' more clearly illustrated in
sentence: If I tip my cup then my tea will fall out. This kind of cau
reasoning requires an intuitive temporal dimension (one thing m
happen before another). Prolog viewed as logic has no such dimensi
but Prolog viewed as an executable programming language does - the
Prolog actually works through a program allows for a non-linear (beca
of the backtracking), but plausible sense of time. Provided that
students do not make the mistake of thinking they are being logica
correct, and provided they are consistent in this somewhat unortho
interpretation of the ':-' operator, they will probably progress qu
happily.

Another response was for students to declare that the sentence
impossible to represent. For them, 'if' and 'because' "don't mean
same and so it can't be done". In fact, although none of these stude
were logically trained, this response is closest to the logical stan
However, we suspect that rather than being correct for this reas
these students were actually trying to translate English into Prol
and expected Prolog to have corresponding operators to the Engl
connectives 'because', 'so', 'therefore'. Interestingly, though, whi
fretting about the causal links, they were quite happy to c
quantifiers such as 'all' and 'every' from statements like "All Mish
students work hard", and "Every student of Misha's works har
Quantifiers are another feature of English with which Prolog does
deal directly, but which are usually dealt with by careful use

ariables. It may be that the techniques for dealing with quantifiers re less obviously altering the meaning of a sentence than the if/because' substitution. But, either way, if students are inclined to onstruct the meaning of a program from the meaning of English and not rom the semantics of the programming language, they may find life ifficult.

he situation is problematic both at the level of predicate logic and at he level of computer programming. From the point of view of logic, tudents who do not realise that logical expressions are subject to the ules of logic, and not the rules of natural language may find hemselves in very deep water later on in trying to interpret the meaning[1] of sentences such as 'Either some dogs are animals or no nimals are not dogs' which doesn't have a very sensible English meaning ut which is logically sound. From the point of view of computing, there re two issues.

irst, data or information in computer programs, whether in Prolog or ot, has to be structured 'sensibly' for the current purpose, whatever hat may be, or else programs will produce meaningless or incorrect nswers. Learning how to sensibly represent knowledge is arguably the ost important issue to be confronted by the beginner in learning to rogram. So long as students think that they merely have to embark on a urface translation from English into the hieroglyphs of Prolog, they void the crucial issue of knowledge representation which is relevant to 11 aspects of computing and AI. Such students will confuse themselves iy writing disorganised programs which will either take a long time to un, or which will not run at all. Students may then have to waste time lebugging badly conceived programs.

Second, from the beginner's point of view, programming involves the notion of problem solving, which usually means devising an algorith which the computer will use to work out the answer to some problem. The difficulty in adopting the 'translation' view is that simply re-representing the English text in Prolog is not going to solve the problem - the student must work out how the algorithm will proceed, an what information it needs to succeed. Students who lose sight of thi goal, are likely to be severely hampered in their learning.

Another facet of the natural language issue is when students try t understand a program in terms of natural language, and not in terms o the functional relations between different parts of the program. The may think that Prolog 'understands' the meaning of words, or that som words are significant to Prolog. This becomes particularly apparent i database manipulation tasks. Ross (1982) points out that beginners ar usually presented the declarative viewpoint of Prolog (see next section angled towards database creation and search programs which often use a lot of English words. In his opinion this approach promotes complacenc because Prolog looks very easy at this level. Beginners can foo themselves that they are learning to program, when in fact they ar understanding the program by understanding the English, i.e. b attributing meaning to the words used. This can allow students to ignor various important aspects of the Prolog notional machine, and it behaviour, which will cause a great deal of confusion later.

Debugging will be difficult if there is a disparity between wha students think their program is saying (described at a linguisti level), and what it is actually doing (at the notional machine level) For example, we have noted that, when asked to describe what thei Prolog programs were supposed to do, some students at Sussex were abl to give quite competent English descriptions. But the code they had written either simply did not reflect this English description, or did not work in the desired way because the logical structure of the problem solution had not been extracted from the English. The solution, as expressed in these programs, lay in understanding the meaning of the English words, which of course Prolog cannot do.

So encouraging beginners to understand logic/Prolog by the use of natural language may not be the great advantage it at first appears. Kowalski (1979) makes the caveat that natural language will only provide an 'informal guide' to understanding logic. This is fine for someone who understands either logic, or programming, or both. He or she will have a fundamental grasp of the limitations of the 'informal guide' as an aid to understanding. But the warning is not adequate for the optimistic novice.

SYNTAX AND SEMANTICS

The general problem for beginners with Prolog is that the underlying notional machine is both powerful and complex with a surface behaviour that is hard to predict accurately. Most of the preceding section concentrated on what is called the declarative semantics of Prolog (and the difficulties that novices have with it). The declarative semantics describe Prolog programs in terms of the high-level logical specification of what it will do (i.e. with no emphasis on 'how' it will be done). There is however another view of Prolog that links back to the remarks about mechanisms in the introduction.

Prolog is not just a means of recording and then interrogating static observations about a world. It is also a programming language in which things can be made to happen (inferences) and these will occur in a particular order. The user needs to understand this in order to predict accurately what a Prolog program will do. The procedural semantics is a formal account of what Prolog programs do and how they do it. A competent Prolog programmer must reconcile the declarative and procedural view of the language and recognise the circumstances when each is the best way to interpret what is presented. These two views are not in conflict, they are complementary.

Unfortunately, Prolog syntax does not offer any clear pointers to what is happening 'behind the scenes', i.e. what changes are taking place in the internal state of the machine. Much of the machine's activities have to be interpreted from dense and compact syntax, and because there are few indications of flow of control (i.e. the order of events) through the program, students sometimes find it difficult to interpret how the code will run. Prolog presents a particular difficulty for beginners in this respect because although it is true that in other languages identifying and sustaining the correct flow of control is important, part of Prolog's power is in its automatic inferencing and backtracking mechanisms. Backtracking is the mechanism that enables Prolog to try further rules, if available, when it fails in its attempt to use a particular rule to establish the truth of some inference. This mechanism ensures that Prolog automatically attempts every possible way of trying to prove an inference before giving up and reporting that it cannot.

The novice must learn how to control these processes by learning how to order sentences in the program, recognising circularities within a rule, ordering sub-goals within a goal and employing certain special controlling primitives (such as the the 'cut') to achieve a solution.

Prolog provides a facility, called a 'trace package' with which the programmer can observe the internal chain of events, such as inferencing and backtracking, while a program runs. These packages are not without their own drawbacks for the novice. Sometimes they present a view of the system which is at odds with the notional machine that the beginner thinks is there, or they provide either too much or too little detail, or indeed are just hard to control in their own right.

Beginners tend to write programs which Prolog can run away with by drawing an endless chain of useless but true inferences. This happens because they include rules that are in some sense circular e.g. that 'John is a good student if John is a good student'. The system can end up fruitlessly trying to find the end of the chain of identical inferences that such a rule would generate. The actual problems are often more subtle than this, of course, and involve the complex use of variables.

We asked students to write a small program on paper, and then describe what the machine would do with it. Most of them were capable of outlining one possible solution (the one they were expecting) but they gave incomplete descriptions of ALL the work the machine would have to do to get there. When we ran the program and switched on the tracing package they were surprised to see how much variable 'matching' and backtracking was involved during the solution process. This lack of knowledge about how much work Prolog may have to do behind the scenes even to produce an apparently simple - and to the human user often obvious - answer can make it difficult for students to debug faulty programs. In other words, if the solution is not correct first time, students may have great difficulty identifying where the error lies.

Backtracking confuses beginners in other ways. There have been a numbe of studies to identify common misconceptions about what Prolog does whe it backtracks whose results are confirmed by our own work with novice (see e.g. Coombs and Stell, 1984, Van Someren, 1984). One of the issue is that it can only be understood properly in dynamic terms and th static text of the written program is only a partial guide and may even according to Coombs (1985) be a positive hindrance!

PRAGMATICS

One of the findings from our work with experts was their much greate reliance on the Prolog tracing facilities to debug their programs. I did not merely seem a matter that they were more skilled in using thes facilities (which of course they were). There seemed to be a differenc in what they perceived as a reasonable course of action when developin a program. The experts appeared to be much readier to admit th difficulty of predicting exactly what a Prolog program would do unde various circumstances and used the system itself to help with th prediction and hence the debugging. The novices, on the other hand seemed to want to undertake this hard predictive task for themselve without help. It was as if they perceived their task as putativ programmers to be doing this job unaided.

It has to be admitted that learning to use the tracer and interpret it output can itself be hard. The messages from the tracer and th dynamically changing layout of information on the screen require tha the student understands the procedural semantics of Prolog if they ar to be fully appreciated. In particular Prolog's backtracking behaviou is much in evidence and we have already discussed above some of th problems that students have with this. Students who only partiall understand this aspect of Prolog will probably be helped by trying t interpret the trace output, but they may be reluctant to turn to this a a source of help for debugging.

CONCLUSIONS

Prolog provides beginners with the means to write interesting, non trivial programs at an early stage in the learning process, usin powerful computational mechanisms, so capturing the interest of th learner from the outset. The language allows for a various types o computing-related activities - aside from writing programs, it can b used as a database creation and search language, as a representatio language for problem solving, and because the principles on which Prolo operates are machine independent, the learner can get a global view o computers and their operations without getting unnecessarily bogged dow in highly specific machine-related details.

A problem common to all computing languages is that if a beginne suffers from woolly thinking, then the computer will highligh deficiences with unpitying relentlessness. A specific problem wit Prolog is that up to a point it will allow the user to write some fairl silly, usually contradictory, things without complaining. But clearly at some point, things will go awry, leaving the beginner in a state o confusion.

There is much that technology can do to ease the load on the user Improved debugging tools and friendly environments for Prolog are unde development. However, learners have to be made aware of the fact tha learning to program, just like acquiring any complex skill require effort and practice. It is unlikely that any language will be able to d away with hard work and application. Similarly, it may be tempting fc the tutor to think that because Prolog is a high level language the less effort needs to be put into the teaching materials - it will t easier to teach as well as to learn. This does not seem to be the case We have seen that some difficulties are related to Prolog's interne structure (i.e. the backtracking mechanism) which require the carefi development of clear and adequate models of the language to present t students (see Pain and Bundy 1985) Other problems are to do mor generally with students' expectations of themselves - that they shoul be able to learn programming effortlessly - and of computers (i.e thinking they can understand the meaning of English words). This clas of problem can sometimes combine with a student's limited proble solving skills (or abilities) to make the task of learning to progra appear an impossible task.

This paper scratches the surface of many complex problems which we ar continuing to investigate. There are a number of further interestir issues thrown up by our current experiments which need further analysis One example is the apparent unwillingness of novices to use the close world assumption and their wish to represent negative informatic explicitly rather than leaving it to be 'discovered[1] by Prolog as *a* absence of positive information.

ACKNOWLEDGEMENTS

We thank Julie Rutkowska and Maarten van Someren for helpful comments c early drafts of this chapter, and Jon Cunningham for discussions. ThJ work is supported by a grant from SERC/Alvey.

REFERENCES

ANDERSON, J. R., (1982) 'Acquisition of Cognitive Skill', Psychological Review Vol. 89, No.4, 369-406

BADRE, A., and SHNEIDERMAN, B., (1982) _Directions in Human Computer Interaction_ Norwood, N.J.: Ablex Publishing Corporation

BEGG, I., and DENNY, J. P., (1969) 'Empirical reconsideration of atmosphere and conversion interpretations of syllogistic reasoning errors', Journal of Exp. Psychology, 81, 351-354

COELHO, H., COTTA J. C., and PEREIRA, L. M., (1982) _How to Solve it with Prolog_, 3rd ed., Laboratorio Nacional de Engenharia Civil, (Obra 03/53.752), Lisbon, Portugal

COOMBS, M. J., and ALTY, J. L., (1981) _Computing Skills and the User Interface_ London: Academic Press Inc.

COOMBS, M. J., (1985) Alvey Conference, Edinburgh

COOMBS, M. J., and STELL, J. G., (1985) 'A Model for Debugging Prolog by Symbolic Execution: The Separation of Specification and Procedure', Dept. of Computer Science, University of Strathclyde

DU BOULAY, J.B.H., O'SHEA, T., MONK, J. (1981) 'The Black Box inside the Glass Box: presenting computing concepts to novices', _Int. J. Man-Machine Studies_, 14, 237-249.

DU BOULAY, J.B.H., (1986) 'POPLOG for beginners: A powerful environment for learning programming', in _Artificial Intelligence Programming Environments_, R. Hawley (Ed.), forthcoming, Chichester: Ellis Horwood

EHRLICH, K., and SOLOWAY, E., (1982) 'An Empirical Investigation of the it Plan Knowledge in Programming', Department of Computer Science earch Report No. 236, Yale University

ALS, R., (1984), Beginning M^cro-Prolog, Chichester: Ellis Horwood

NS, J. St. B. T., (1982) The Ps^chodogY of Deductive Reasoning don: Routledge and Kegan Paul

NSON-LAIRD, P. N., (1983) Mental Models Cambridge: Cambridge versity Press

NSON-LAIRD, P. N., and WASON, P. C. (eds), (1977) Thinking - Readings Cjognrtive Science Cambridge: Cambridge University Press

ALSKI, R. (1973) 'Predicate logic as programming language', Memo No. Department of Computational Logic, School of AI, University of nburgh

ALSKI, R., (1979) Logic for Problem Solving, New York: North Holland
.

K, A., (1985) Fundamentals of Human-Computer Intjeracjtion London: demic Press Inc.

MAN, Donald A., and DRAPER, Stephen W. , (1986) User Centered System ign - New Perspectives on HuiQ9J2""£2IILE!i££I iHl§L§£iion Hillsdale, N. Lawrence Erlbaum

N, H., and BUNDY, A., (1985), 'What Stories Should We Tell Novice log Programmers?', work in progress report, University of Edinburgh

SDEN, E., (1984) ed. Microcomputers in Education Chichester: Ellis wood

OSS, Peter (1982) "Teaching Prolog to Undergraduates" in AISBQ, Autumn 982

HNEIDERMAN, B., (1980) <u>Software Psychology</u> Cambridge, Mass.: Winthrop ublishers

MITH, H.T., GREEN, T.R.G. (1980), 'Human Interaction with Computers', ondon: Academic Press Inc.

OLOWAY, E., and EHRLICH, K., (1984), 'Empirical Studies of Programming nowledge', in IEEE Transactions on Software Engineering, Sept. 1984

AYLOR, J. (1984) 'Why novices will find learning Prolog ard', Proceedings ECAI, 1984

AN SOMEREN, M. W., (1984), 'Misconceptions of Beginning Prolog rogrammers' Memorandum 30, Dept. Of Experimental Psychology, University f Amsterdam