Instructional Science 19: 257–282 (1990)

What's wrong? Understanding beginners' problems with Prolog

MAARTEN W. VAN SOMEREN
Department of Social Science Informatics, University of Amsterdam, Roetersstraat 15, 1018 WB Amsterdam, The Netherlands

Abstract. This paper reviews psychological research on programming and applies it to the problems of learning and teaching Prolog. We present a psychological model that explains how a certain class of errors in programs comes about. The model fits quite well with the results of a small sample of students and problems. The problems that underlie these and other errors seem to be (a) the complexity of the Prolog primitives (unification and backtracking) and (b) the misfit between students' naive solutions to a problem and the constructs that are available in Prolog (e.g. iterative solutions do not map easily to recursive programs). This suggests that learning Prolog could be helped by (1) coherent and detailed instruction about how Prolog works, (2) emphasis on finding recursive solutions that do not rely on primitives such as assignment and (3) instruction in programming techniques that allow students to implement procedural solutions.

1. Introduction

If we want to help Prolog programmers with their task and help students to learn Prolog as well and as quickly as possible, we need to understand the nature of their difficulties. Tools that support the design and debugging of programs and also the way in which Prolog is explained to students need to be directed at the actual difficulties and should avoid the introduction of new difficulties or the solution of marginal problems.

This paper addresses the problems that students have when they are learning Prolog. First we consider psychological research on programming to see what this tells us about novices problems with programming in general. We then try to apply the results to Prolog. In the second part of the paper we explore in depth one kind of problem that frequently occurs among novices, to obtain a better understanding of some of the problems that are particular to Prolog.

Computer programming is a term that covers a very wide range of tasks and activities. Each of these will have its own difficulties. Before we can discuss these difficulties, we need to draw a global map of the area. Dimensions of the domain are:

Programming language: there is considerable variety between programming languages. Two distinctions that are important in characterizing the position of Prolog are functional vs. imperative languages and high level vs. low level languages. Functional languages can be characterized as non-imperative, i.e. programs are not basically instructions to the computer to perform certain operations,

but they are functions that can be applied (as in LISP) or statements that can be verified (as in Prolog). High level languages provide complex and powerful primitives as an inherent part of the language. Prolog is a high level language in this sense, with unification and depth first search as primitives. If we do not take the programming environment into account and if we consider relatively simple, well specified programming tasks, the most important differences are the declarative meaning of Prolog programs and the major primitives of the language: depth first search and logical variables with unification.

Type of specifications: The nature of specifications of programming tasks can vary widely. They may be stated in a more or less formal terms, may range from very detailed to very global, they may be consistent or almost contradictory, they may require no special understanding of the subject or a lot of mathematical or domain specific knowledge. Since we concentrate on novices, the programming tasks that we consider are stated in informal terms, detailed, consistent and require little special knowledge of the domain.

The "software life cycle": Prescriptive literature such as textbooks on programming and software engineering, and to a lesser extent, empirical work, distinguish several stages in the process of programming. The stages can be characterized as shown in Figure 1. They can be traversed top down, but there are other possibilities. A problem may be split into subproblems that are treated more or less separately. A partial solution may exist in the form of a piece of code which can be used bottom up as a starting point for the rest of the design. We focus on the initial stages: problem analysis, design and implementation and on problems that novice programmers encounter (mostly simple exercises).

Programming environment: The tools that are available to the programmer are another aspect. These may range from structured editors, checking syntax and semantic properties to debugging tools, etc. (As we concentrate on the initial stage of the programming process, the environment plays no significant role.)

Sub-areas in this space of programming tasks will in general require different skills and pose different difficulties to the programmer. This implies that a programmer may be an expert in one sub-area, but a novice in another.

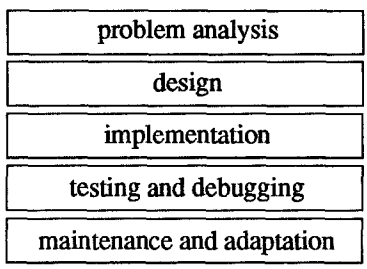

Figure 1. Stages of the programming process

Here we are primarily interested in the design and implementation of rather simple programs in Prolog. However, we first take a more general look at the psychological literature on programming to make an inventory of findings about this area and similar tasks. The studies in the literature tend to focus on relatively "pure" tasks and do not consider tasks where the programming environment plays an important role, or where a complex solution must be found before the actual design and implementation can begin.

1.1 Psychological research on programming

In this section we discuss psychological research on programming in terms of types of knowledge that novices and experts have about programming. We are interested in both correct knowledge and incorrect knowledge such as misconceptions. The next section discusses the nature of beginners' problems that result from using inappropriate or faulty knowledge. The implications of this for Prolog are considered and compared with the (few) empirical results. The literature review is organised by stages of the life cycle. (As said, these stages should be viewed as activities of the programmer, rather than chronological stages.)

In the *problem analysis* stage the specification of the program is completed, interpreted and structured. An abstract plan, method and description of the data that play an important role is produced here. In the next stage, a design for the program is produced in the form of an algorithm, abstracted datastructures, etc.

There is not much empirical work on problem analysis. An interesting study that indicates how much knowledge is required was done by Kahney (1982). His subjects took a very long time to understand what the program that they should write was supposed to do. They seemed to interpret the task of solving a programming problem as "find a method to perform a task" or even "describe how you would perform the task yourself". They felt that they had solved it if a method was found that would be adequate for *themselves*, not taking account of the fact that the *computer* would have to use the method. Therefore problems that are stated in terms of datastructures of the implementation languages can be solved, but isomorphic problems stated in natural language may not be solved at all by novices in the sense that they are unable to produce a program at all. Note that this makes it difficult for us to interpret the task. Some programming problems will appear to be non-problems to a novice, because it is just too obvious how it should be done. Consider the task to write a program that can find the bigger of two numbers. It is so obvious how this is done, that it seems impossible to take it apart and formulate a method for it.

Problems of the analysis stage will be similar to those encountered in other languages. For example, understanding what a computer program is and what it means to write a program for a particular problem, and analysing the knowledge

needed to solve a problem, are likely to cause similar problems in most languages. The possibility of analysing problems in the style of logic is a complication. It is not clear yet if this has (should have) consequences for the analysis stage.

In the *design stage* an abstract design of the program is produced, consisting of a detailed algorithm and selection of datastructures for different types of data. A number of studies show how important this stage is. From these we can derive the knowledge that is used here. The knowledge involved in program design has a task oriented aspect and a strategic aspect that we shall discuss in turn.

A useful view of the design process is to see it as *planning* (e.g. Soloway and Woolf, 1980; Rist, 1986). A plan here is similar to an algorithm, but is organised around goals. Programming is viewed as a planning task: the programmer should construct a plan to get to the output, given the input. The plan should be executable by the computer. The knowledge required to construct plans consists of ready made plans, methods for dealing with interacting subgoals, methods for combining plans (to obtain a more economic program), knowledge of how to evaluate a plan (symbolic evaluation, estimating efficiency) and, quite important, heuristic knowledge of how to find the right plan for a goal. A plan may go through refinement cycles and at some point it is implemented.

Soloway and Woolf (1980) describe a set of schematic "plans" that can be selected using properties of the problem specification. A schematic plan can be refined and "filled" with the appropriate content, e.g. in LISP there could be a "predicate-plan" that has as parts the conditions under which the function should return "true" or "false". There may be subcategories for different kinds of conditions (e.g. conjunctive or disjunctive, conditions on all elements of a list or on "at least one", etc.). Soloway and his colleagues (Spohrer, Soloway and Pope, 1985) performed a number of experiments in which they showed that

(a) a correct plan is no guarantee of a correct program. Novices often make plans that are basically correct, but make severe errors in their implementation

(b) the source of plans is often a *natural* plan (e.g., a set of instructions to a human who is to perform the task of the program, or a description of how the programmer would do it). It is often difficult to see what a plan means, because much can be left implicit or is ambiguous

(c) parts of a plan can be merged during design, but this is a source of errors, because interference and side effects of the implementation are not taken into account.

The notion of *plans* may be less useful with Prolog than with imperative languages, because Prolog has no built-in primitives that correspond closely to the building blocks of plans, e.g., "repeat an operation until some condition holds" or "hold this object" have no direct counterparts in Prolog.

An important aspect of the design stage is the strategy that is to be followed. In general, a plan will be very complex, and to avoid inconsistencies and holes in the plan it will be necessary to deal with this complexity of the mapping in a systematic way, which requires some way of decomposing the task. The effect of a *plan* is that it decomposes the problem into subproblems, but apart from a way to recognise the appropriate plan for a problem, a strategy is required that organises plan construction. In the literature we find many ways to decompose the problem, using different abstractions (e.g. decomposing data, functions or complete systems). Part of the knowledge will consist of a *repertory of abstractions and decompositions*, plus a way to apply them. The complexity of the task will also require *systematic documentation* and *a systematic working method* (to maintain consistency). Computer science has developed a number of abstractions with corresponding notations and methods to do the decomposition. All these methods use hierarchical decomposition, where an initial design is constructed using a particular abstraction and this design is then decomposed and each component is refined. Some methods decompose into levels of abstraction, which makes it possible to verify the design at intermediate stages.

A related result was found by Jeffries, Turner, Polson and Atwood (1981), who compared the behaviour of novices and experts on a fairly complex programming task. The novices had followed an introductory programming course. One difference was that, roughly speaking, beginners followed a top-down depth-first strategy: they elaborated one part of the problem in great depth and then continued with a related subproblem that could now be elaborated. Experts, on the other hand, also followed a top-down strategy, but breadth first, elaborating the complete problem in more and more detail. The same was found by Anderson (1983; Anderson *et al.*, 1984) and Adelson *et al.* (1984), who uses the term *balanced development*. Anderson proposes the following explanations for this phenomenon. One is that the depth-first approach poses less load on working memory. The programmer can concentrate on a small problem and does not have to take the full complexity of the task into account. (Obviously, the price is, that he may have to backtrack and revise his work, which can be very expensive.) Another explanation is that to profit from the breadth first strategy, one (a) needs a language to express intermediate solutions and (b) must be able to verify and evaluate them. Novices may not have that kind of knowledge. Anderson also points out that benefits only become clear when rather complex programs are involved.

A more subtle effect was found by Hoc (1981) who was able to train students in one of two different design methods. The first was based on decomposition of the input data, working forward to the specification of the result. The second worked the other way round, starting from a decomposition of the (specification of the) result. When the students solved a data processing problem of intermediate complexity, in which the problem was in the complex specification of the input data, the "forward" method was more effective (producing fewer errors). This

indicates that decomposition can be done in different ways and that one may be more effective in reducing the complexity than another.

In the literature we also find some indirect empirical evidence that supports this analysis. Anderson (1984) and Spohrer and Soloway (1986) attribute a number of errors to items being lost from Working Memory. This supports the idea that a good strategy is indeed important for programming, because a good strategy provides a framework for documentation (on paper or in memory) for partial results and goals. Perkins and Martin (1986) tutored novices who were learning BASIC and found that much tutoring consisted of general advice about problem solving, suggesting that students be more careful, consider alternative solutions, plan ahead, etc. Their interpretation is, that novices simply lack the knowledge to take the right decision at certain points in the programming process and they should take this into account when they solve a problem. Thus they should remember dubious decisions, shaky assumptions, etc. in order to be able to revise their decisions and also, they should realise that there may be unexpected possible solutions to a subproblem. Therefore, they should look for other possibilities. Tromp (1989) found that teaching a *systematic working method* caused a considerable increase in performance of students learning Pascal. The main idea is to let students first produce an explicit design of the program, before they write the actual code. The role of the explicit design is presumably to verify if the solution will actually work when implemented and also it provides useful documentation that prevents errors during implementation.

The role of a systematic strategy and documentation is also unlikely to be different for Prolog, because it is a consequence of the complexity of the analysis. The nature of the design process could possibly be different. The fact that, at least to some extent, Prolog programs can be interpreted as statements rather than instructions to the machine, gives the programmer the possibility of constructing and debugging his program in two different ways. He can either design methods, algorithms and data structures or he can translate his knowledge into Horn clause logic and use that as an initial version of the program. The latter is called logic programming. Logic programming can be the basis of both design and debugging a program. A program can be verified by reading it and deciding if it is a correct and complete statement of the truth about a relation or property.

Prolog allows programmers to abstract from control issues to some extent (see below). This possibility is an extra compared with other languages. The price is, however, that the underlying machine that is required to achieve this effect is quite complex. The built-in unification and depth-first search procedures make Prolog very powerful, but also difficult to understand, especially for a novice.

The situation is even more complicated, because the declarative meaning is in fact limited. Prolog contains a number of constructs that affect the derivation procedure and therefore it may well happen that a fact that in principle could be

derived from a Prolog program in its declarative interpretation may not be derived in the execution, for reasons that can be explained only by taking account of the procedural meaning. (Other notable differences are the way in which Prolog treats negation and quantification.) Another reason is, that a natural language statement can be expressed in logic in many ways, but not all of these are equivalent in terms of program execution. A well-known example consists of the following two expressions, which are intuitively very similar, have computationally different effects. When used in combination with a series of assertions about particular persons being married (e.g. "married(john, mary)", "married(peter, jane)", etc.), the first program will cause infinite loops in many cases (e.g., for the query "married (jim, mary)", where the second will behave as intended, if the predicate "married_couple" is used in the query:

```
married(X, Y):-
        married(Y, X)
and
        married_couple(X, Y):-
        (married(X, Y) ;
        married(Y, X))
```

Finally, many programming problems more naturally have a procedural solution that must be transformed to a declarative statement before it can be expressed (and verified) in logic.

To achieve the declarative meaning, Prolog uses very complex built in mechanisms (notably unification and depth first search) that are both powerful and difficult to understand. For these reasons, Taylor and du Boulay (1986) argue that logic programming and Prolog programming are subtle and difficult skills and will require both an understanding of Prolog's virtual machine (i.e., the procedural meaning) and Prolog programming and design techniques. We may add that these design techniques may differ from standard design techniques.

In the *implementation* process, the design is translated into the implementation language. This process is similar to the design stage. In fact we saw that in some studies design and implementation were not separate stages, but two ends of a continuum. What knowledge is required for the implementation stage? Both a design (a plan, an algorithm, etc.) and the resulting program will in general be complex objects. Their elements are strongly interrelated and the relation between the parts and structures of a design and the resulting (correct and optimal) pieces of code is very complex. Several studies suggest that expert programmers have organised their knowledge of the implementation language according to the function that constructs of the language can have with respect to the design. This enables them to find possible implementations quickly for pieces of a design. The studies described below support this idea and indirectly lend support to the notion of plans, as described earlier.

Adelson (1981) used a free recall paradigm to show that experts organise a set of statements in a programming language by their functions in a program, where novices use syntactic similarity as a criterion. She claims that the functions used by the experts have a hierarchical structure. McKeithen *et al.* (1981) found the same results and also showed that experts can memorise structured programs much more readily than novices (analogous to De Groot's [1965] experiments with chess experts). The difference between experts and novices disappeared when meaningless statements were used. This can be explained by the effect of the structure in memory. The memory structures work as chunks and allow faster recognition and storage. These chunks may not correspond to language structures that are explicitly defined in the programming language. They may be constructed from experience and have no standard name. In principle, they can be added in the form of conventions, or idiomatic forms of expression. Shapiro (1981) showed that *clichés* correspond to units of meaning. A cliché may correspond for example to a *filter*, a *generator* or a *guard* on an input stream. (Shapiro exploits the existence of these clichés to recognise bugs.) The notion of cliché is applied to Prolog by Plummer (1990) and Gegg-Harrison (1989).

Maintaining an overview of intermediate versions of a program during implementation is easier for experts, because they have high level concepts that can be used as chunks to represent the intermediate design efficiently. Jeffries *et al.* (1981) found this to be an important difference between experts and novices. Indirect evidence for the importance of this difficulty comes from the beneficial effect of explicit intermediate representations such as diagrams of data structures, data flow and control flow. See, for example, Hassel and Law (1982).

To create a program that is correct, the programmer must know the *syntax and the semantics of the programming language*. The syntax of most programming languages (including Prolog) is very limited and, in the initial stages only, a source of errors. Knowledge of the semantics may take the form of a *notional machine* that is used as a model of how the computer executes a program. This notional machine provides a language that can be used to express (and thereby simulate, explain, debug, etc.) the operational meaning of a program (du Boulay, O'Shea and Monk, 1981). Empirical evidence for this aspect comes from a variety of sources. Mayer (1981) describes an experiment that shows that novices who were taught about a notional machine for BASIC performed better than controls taught BASIC without a virtual machine in writing programs for small problems that differed substantially from the examples given in the textbook. Another experiment by Mayer showed that first understanding the virtual machine facilitates studying a textbook with explanation about a programming language (as compared with presenting the virtual machine afterwards).

A notional machine can also be misleading, as illustrated by several authors. Burstein (1983) showed that representing a BASIC variable as a box, a value as an object and then representing BASIC '=' as putting an object in a box or taking

it from the box can be quite misleading. It implies that after executing the program lines

```
A = 1
B = A
```

box A would now be empty, because the second statement means that the 1 will be taken from box A to box B. Douglas and Moran (1980) give a similar account of the effects of explaining a text editor with a typewriter as the virtual machine. Anderson (1983; Anderson *et al.* 1984) studied LISP programmers and found that novices frequently use an existing program as *analogous model* for a new program. The analogous program goes through a series of modifications that are similar to a debugging process. The programs involved were rather simple, which made this approach feasible. Analogy can be used at any stage of the process, from analysis to implementation.

Understanding and using recursion is a notoriously difficult issue for novices. Kahney (1982) studied students' understanding of recursive programs. To understand the behaviour of recursive programs, an understanding of the recursive nature of the machine that executes a program is required. Kahney found that novices usually imagine an iterative machine executing a recursive program. Recursive calls and mechanisms to pass parameter values between procedures are interpreted as some kind of iteration, which results in very predictable errors. The recursive call is supposed to pass control back to the original procedure and students have varying ideas about what happens to the values of variables.

Prolog's notional machine is more complex than other languages, in the sense that primitives like unification and backtracking do a lot in a single step. A major benefit is that programs can be very short, reducing the complexity of the programming process. However this makes Prolog more difficult to learn, and could cause many errors in the initial stages. Coombs and Stell (1984) modelled students' misconceptions about Prolog backtracking as variations of the standard interpreter. They can diagnose six errors, for example, *try once and pass* (no backtracking within the body of a rule; thus if one subgoal fails on the initial substitutions, control is passed to the next rule) and *redo body from left* (failure of a subgoal causes the first goal to be retried rather than the chronologically last subgoal). For a detailed discussion of novices' misconceptions about unification, see van Someren (1990). A good understanding of the notional machine, supported by tools that give the student access to the details, will be even more important for learning Prolog than for other languages (see Pain and Bundy, 1987).

1.2 Novices' problems

Novices lack the knowledge that is required to deal will the complexities of the programming task, so they try to get as far as they can with the knowledge that they do have. Not only their knowledge of the new programming language may

be limited, but also general knowledge and knowledge in other relevant areas. The novice/expert distinction is relative: a person can be an expert on one aspect, but not on another. Compare someone who knows a lot about methods and objects that play an important role in a programming problem, an expert programmer and an expert programmer who is using a new programming environment.

Our analysis above is in line with Spohrer and Soloway (1986) present a list of problems that students meet that is in line with our analysis above. They distinguish *plan composition problems* from problems related to the *semantics of the programming language*. The former correspond mostly to design problems and the latter to a class of problems in the implementation stage.

(a) Semantics of the programming language

Because they have a partial understanding of the semantics of the programming language, novices use knowledge about natural language to design and understand programs. This can cause several types of errors, for example, *Natural language problems*, where a term's natural language meaning is confused with its programming language meaning. This applies to datastructures and procedures. In the latter case, Spohrer and Soloway call it the *human interpreter problem*: the programmer believes that the computer works as a human reading the program as an instruction. We mentioned a counterpart of this problem in the analysis stage, where novices don't even understand the task as it applies to a computer.

(b) Plan composition problems

These are problems that are related to the complexity of keeping an overview of the program under development and to making the right composition. Some examples are as follows (see Spohrer and Soloway [1986] for a complete list):

- *Summarisation problem*: Novices may summarise complex plans in terms of one primary function and overlook side effects of the plan in a later stage of the programming process.
- *Optimisation problem*: Novices may be so eager to optimise their plans, that they do not adequately check if the optimisation can really be carried out.
- *Previous experience problem*: Plans from previous problems may be used for a new program, but they bring inappropriate aspects along.
- *Natural language problem*: Novices may use or construct plans in natural language form. Mapping this to a programming language may produce bugs.

The *summarisation problem* and the *previous experience problem* seem to be caused by the design task's administrative complexity. The *optimisation problem* seems to be caused by bad strategy (no evaluation) and the *natural language problem* by poor understanding of the programming language's semantics.

Bonar (1985) presents a detailed analysis of programming errors that result from a two stage process: the novice first constructs a plan that consists of instructions in natural language and then tries to transform this into Pascal. Even if the plan itself is correct (which is usually the case), then problems can still arise from several sources (see Bonar (1985) for a complete description):

- a Pascal construct is selected to implement part of the natural language plan on the basis of superficial similarity. Though this works in some cases, the Pascal statement may have a different effect.
- a variable is assumed to play several roles at the same time (where in fact it implements only one role).
- two Pascal statements that have similar functions with respect to a plan are confused.

The scope of the work that will be described in the rest of this paper is restricted to the context of an introductory Prolog course. Therefore, we shall not look at the design and implementation of complex programs, but concentrate on textbook examples and exercises.

2. How do novices design a solution to a (simple) problem?

The first step of the programming process is to find a solution to the problem that can serve as an initial design for the program. It is important to know what these solutions look like, because they are likely to structure the rest of the process: once a design has been constructed, a novice will translate it into an implementation part by part. It is known from earlier work (Bonar [1985] and Kahney [1984]) that the initial solutions that students formulate are stated in informal natural language, vary in level of detail, terminology and structure and result from different interpretations of the problem (novices have no clear idea what a solution should look like and therefore, they may come up with unusual solutions that not only use a different terminology and structure, but are quite unlike a design for a program). A similar study was conducted by Miller (1975). It is particularly important if Prolog novices produce recursive designs. Although after training it may well become natural to interpret "write a program that can find the biggest number in a list" as "take the first number, find the biggest number in the rest and take the bigger of the two", this seems not an intuitive solution.

We conducted a small informal study to verify these intuitions. Five simple programming problems were selected as representative of elementary exercises. An example of the problems used is (translated from Dutch to English):

> *Imagine the following situation: in front of you is a box with cards in it. Each card has a number written on it. The cards are in no particular order. The problem is: get the card that has the biggest number on it. How would you do this?*

The formulation was adjusted to remove any reference to datastructures and the problems were presented to 4 subjects who had no experience with programming and very little with computers in general. Answers were classified as *iterative* if they were described as "and continue until..." (or a similar phrase) and as *recursive* if they were described as "and apply the same procedure to...".

All subjects gave a solution quickly and without much thought. All solutions were clearly iterative procedures: no recursive solutions appeared. Because the subjects' answers were the same (at least in this respect), we felt no need for more subjects. Most solutions were incomplete as specifications. (e.g., "and so I would work my way through the box" implies a stopcondition "the box is empty").

The terminology employed by our subjects varied considerably. In all protocols there was a concept of iteration or at least quantification over all cards in the box. Boundary conditions were implicit in most protocols.

3. How do novices implement an iterative design in Prolog?

A set of buggy programs that were produced by students of an introductory course in Prolog were analysed. About two thirds of these bugs can be explained by assuming that students worked from an algorithmic design (that is, an algorithm) and tried to translate this into Prolog. We propose a model that can explain the errors by assuming that students performed these translations by looking for Prolog constructs that are both similar to parts of the algorithm and to well known programs. The model is supported by analyses of think aloud protocols taken from novice programmers.

This study concentrates on a detailed analysis of the errors and does not provide a quantitative analysis of the types of errors. To obtain a detailed view of the knowledge that is required to construct a Prolog program from an iterative algorithm, we constructed a computer program that can perform this task: *Iterator*. The second purpose of building Iterator was to use it as a basis for constructing models of implementation errors, by perturbation of the correct Iterator. We first describe how Iterator solves a programming problem. The description is in the form of *implementation rules*. These are rules that translate part of an algorithm (or Pascal program) into Prolog code.

3.1 Iterator: a model of the implementation of an iterative algorithm in Prolog

Iterator starts by selecting an iterative algorithm for the problem at hand and then translates that into Prolog code. This translation is achieved by translating parts of the program into *similar* Prolog constructs. As a sample problem we take "maximum of a list of numbers". The *natural* algorithm for maximum uses an accumulator and enumerates the list forward. When the end of the list is reached,

the iteration stops and the accumulator then holds the maximum. There are several possible formulations of the algorithm. For our purposes they amount to the same effect. We can formulate the algorithm as follows:

1. Initialize the accumulator *current maximum* with the first element of the list. (Note: This is not an explicit part of the naive algorithm. Subjects who solve the problem using the iterative algorithm often become aware of the need for an initialisation at a later stage.)
2. Compare the first element of the rest with the current maximum and keep the bigger.
3. Go through the list and compare each element with the current maximum. If the new element is bigger, then it becomes the current maximum, else keep the one you have.
4. If the end of the list is reached (i.e., the list becomes empty), the current maximum is the maximum.

An algorithm is specified in a simple Pascal-based algorithmic language (Pascalese). The algorithm is translated into Prolog by translating pieces of the algorithm into pieces of Prolog code. The translation rules are then the basis of "malrules": faulty implementation rules that produce buggy programs.The constructs that Iterator recognises as input are:

Pascalese construct	*Example*
variable	first element of the rest, rest, current maximum
accumulator	current maximum
assignment	current max := first element of rest list := tail of list first := first of list
initialisation	initialize accumulator with first element
(while) loop	(consists of body and stopcondition)
stopcondition	list is empty
body	see conditional
conditional	IF first is bigger than current maximum THEN replace ELSE do nothing

We now discuss the correct implementation rules that can be used to translate these algorithms into Prolog. In the next section we shall present the faulty versions of these rules that our subjects seem to follow.

Rule 1

A *variable* is translated to an argument in the head of the clauses of a procedure and a logical variable, unless it is an accumulator (see Rule 2). The argument position characterizes the variable, but the logical variable may not be the same at any position (see assignment).

Rule 2

An *accumulator* (i.e., an output variable that is accumulated through the iteration) is translated into two arguments. (One argument will be used to pass the current value of the accumulator down and the other one to pass the final value back up to the original call. See below.) In the clause that implements the stopcondition of the iteration, these arguments must have identical (variable) names. (Note that the term *accumulator* refers to the complete mechanism, not just to the variable.)

Rule 3

Assignment is implemented as unification plus the introduction of a new logical variable, e.g., to assign the first element of a list to H and the rest to T, a list is unified with the term [H|T]. To re-assign the value of the first element to the current maximum, a new variable is used for the new value of the maximum (e.g., NewMax) and explicit unification is used:

NewMax = H

The new variable (NewMax) is now used to operate on the value of the current maximum.

Rule 4

To initialize an *accumulator* with the first element of the list, write a new procedure which calls a new predicate with the initial values at the argument positions of the variables that must be initialized. Thus, the new procedure will have 3 arguments (for the list, the input of the current maximum and the output of the current maximum). The initial values are specified as assignments to the new variables. The initialisation becomes:

```
max([Head|Tail], Max):-
        max2(Tail, CurrentMaxIn, CurrentMaxOut),
        CurrentMaxIn = Head,
        Max = CurrentMaxOut.
```

Rule 5

To code a *while loop*, code the stopcondition and the body.

Rule 6

To code the *stopcondition*, make a clause that has the name of the program as functor, has one argument for each input variable and has two arguments with identical names for each output/accumulator variable. The condition under which the loop must stop is added as the body of the clause (e.g. the list is empty or has only one element, a counter reaches 0):

```
max2(L, Max, Max):- L = [].
```

The effect of this clause is, that when the empty list is reached, the value of the accumulator is passed back up through the recursion.

Rule 7

To code a loop *body*, write the body of a recursive clause and end with a recursive call:

```
max2([Head|Tail], Accu, Max):-
        < body >
        max2([Tail, NewAccu, Max).
```

Rule 8

To code each *"case" of a conditional*, construct a separate clause, which has the condition of the case as the first term of the body and the operation as the rest of the body. Adjust the ordering of the clauses such that the clause with condition is tried first. (This could be generalised to more complex conditionals, using CASE or nested IF-THEN-ELSE.)

Application of these rules leads to the following program:

```
max(L, Max, Max):- L = [].          Base case - accumulator value
                                    passed back up
max([H|T], CurrIn, Out):-           Assign H and T
        CurrIn < H,                 Condition
        NewCurrIn = H,              Assign H to current maximum
        max(T, NewCurrIn, Out).     Iterate
max([H|T], CurrIn, Out):-
        max(T, CurrIn, Out).
```

This program can be modified by small optimisations that remove the explicit unifications ("="), but this was not implemented. The resulting program finds the maximum of a list by keeping and updating an accumulator that is passed on in the second argument. When the list is empty, the first clause ("[]" as the first argument) is used to bind the value of the result to the third argument. This is automatically substituted into the third argument of all previous recursive calls.

The method described covers problems like manipulation of lists and simple arithmetic and enumeration for which an algorithmic solution can be specified in Pascalese. This covers a large part of most introductory textbook problems. For many problems Iterator produces a solution that is correct, but not efficient.

3.2 Modelling students' errors

Following Brown and VanLehn (1980) we try to explain students' errors by hypothesizing malrules: modifications of correct rules that produce performance errors under certain conditions. Brown and VanLehn explain the occurrence of

malrules from a process of repairing rules that were acquired from explicit teaching or experience, but that are incorrect. In particular an incomplete ruleset may encounter an "impasse", resolved by introducing a "repair" that is *similar* (in novices' eyes) to the incomplete rule. In our case, we will therefore expect malrules that may either be acquired directly from experience or that may have been generated as repairs to impasses. The malrules will in general be similar to correct rules (although interference with another domain may affect the rules). It is important to note that many Prolog constructs that are similar to expressions in algorithmic programming languages, or to expressions in natural language are in fact correct implementations, e.g., several clauses belonging to a single Prolog procedure are often direct implementations of conditional statements. Therefore analogy can easily become an important principle during implementation.

The malrules are presented in the context of an example. Consider the following program for maximum that was written by several students.

```
maxa([],_).
maxa([H|T],M):-
        M>=H,
        maxa(T,M).
maxa([H|T],M):-
        maxa(T,H).
```

From a declarative point of view, this program is odd and must be wrong. It is so odd that it is hard to understand that it was ever written. However, it becomes quite understandable if we take the iterative algorithm for maximum as a starting point and hypothesize faulty variations of the implementation rules on which Iterator was built.

Malrule 1

Implement a variable as an argument (in the head of a clause) and a *single* Prolog variable. This rule causes errors if a value is to be assigned to a variable, e.g., the clause for maximum in which the first element of a list is assigned to the current maximum, would become:

```
max([H|T], CurrMaxIn, Out):-
        H > CurrMaxIn,
        CurrMaxIn = H,
        max(T, CurrMaxIn, Out).
```

This malrule can be explained by the obvious semantic and (e.g. ,in Pascal-like programming languages) syntactic similarity between a variable in an algorithm and a Prolog variable. In Iterator the Prolog construct that corresponds to a variable in an algorithmic language may be: a Prolog variable, a Prolog argument or a pair of arguments (e.g., as in Iterator's implementation of an accumulator, see malrule 2).

Malrule 2

To implement an accumulator variable, use a single argument plus Prolog variable. (The correct technique is to have separate arguments for passing data down and up the recursive calls. Assigning both functions to a single argument with variable can be seen as "overloading" this argument *cum* variable with two functions: accumulating and outputting.) This would produce, for example, the following clause for the body of the iteration:

```
max([H|T], Max):-
    H > Max,
    max(T, H).
```

The base case of the recursive program will become:

```
max(L, Max):- L = [].
```

Malrule 3a

Assignment can be implemented by Prolog "=" or "is". This does not automatically lead to bugs, but it does if it is combined with Malrule 1. The use of "is" is inappropriate, e.g.:

```
max([H|T], M):-
        H > M,
        M is H,
        max(T, M).
```

This again can be explained by obvious similarity between Prolog's unification operator and assignment operators in Pascal-like algorithmic languages or intuitive notions expressed as *becomes, is now*.

Malrule 3b

Assignment combined with function evaluation (as in x := x + 1) is implemented by unifying a term with a variable. This appears in several variations. Since we have found no examples of this malrule applied to max, we use the program for grandparent as an example:

```
grandparent(X, Y):-
        parent(X, parent(Y)).
grandparent(X, Y):-
        parent(X, parent(X, Y)).
grandparent(X, Y):-
        parent(X, parent(Z, Y)).
```

This malrule may well be due to similarity between Prolog constructs (compound terms) and functions from other languages (LISP, Pascal, general mathematical notations).

Malrule 4a

For *initialisation* there are several malrules. The first is to add an extra clause to the program that results in a new call with the initial value. This usually occurs in combination with a single argument for the accumulator, e.g.:

```
max([H|T], Max):-                    % initialisation
        max(T, H).
max([H|T], Max);-
        H > Max,
        max(T, H).
```

Malrule 4b

A slightly more sophisticated version uses the system predicate "var" to test if the initialisation clause should be used. This clause would then become:

```
max([H|T], Max):-
        var(Max),
        max(T, H).
```

Malrule 4c

A final possibility is not to build the initialisation into the program, but to require that the user call the program with an initial value. The user is instructed to enter a call like "maximum([2, 7, 10, 3], 0, Max)", where 0 is the initial value.

The correct version of Iterator implements initialisation in a rather unintuitive way, by introducing a special new procedure. Its malrules can be interpreted as attempts to stay close to the idea of an initialisation step within a single procedure.

Malrule 5

To implement a *(while) loop*, write a clause with a body that starts with the stop-condition, followed by the implementation of the body and ending with "fail". "Fail" is thought to bring control back to the beginning of the body. This malrule can be explained by similarity between the effect of "fail" (e.g. in the context of a failure-driven loop) and transferring control back to the beginning of a loop.

Malrule 6

The *stopcondition* of an iterative loop is often the only guide to writing the base case of a recursive program. This results in the kinds of errors of Malrule 2. A different idea is to cause the iteration to stop by forcing the program to fail, e.g.:

```
max(L, Max):-
        L = [],
        fail.
```

More sophisticated versions of the same idea are

```
max([], Max):- fail.
```

and leaving the base case out altogether. This fails automatically, because then the program will finally be called on the empty list and there are no clauses that would match.

3.5 The scope of the model

Iterator is not a complete model of the programming process in novice Prolog programmers. It models only the actual coding actions. There are several other processes that play an important role and that are not part of the family of models defined by the rules and malrules of Iterator. Beside the process of designing an algorithmic solution, two important processes are *understanding* programs and *repairing bugs*. Below are some observations from protocols about these issues.

Evaluation takes place in general by comparing the behaviour of the program with the behaviour of the algorithm. Both algorithm and program are "mentally" executed and at each step the programmer checks if (a) the correct piece of the program is used and (b) the output of the procedure is correct. If an error is discovered, then the corresponding part of the program is modified, e.g. by adding, deleting or changing clauses or subgoals, changing the order of clauses or adding "fail". When an error is repaired, the programmer leaves the rest of the program as it was. In effect, the result of applying a (mal)rule is replaced by the result of a different (mal)rule, such that the perceived error can no longer occur.

Novices tend to evaluate pieces of code *locally*, as separate modules, so do not take into account how, for example, clauses collaborate. A piece of program code can be locally correct, but may fail to work in a particular context. Take for example the following clause that was written for the maximum program:

```
max([H|T], M):-
        H > M,
        max(T, H).
```

This clause implements part of the iteration. It looks correct as an implementation of that part of the program. However, as discussed before, there is no provision for the accumulated value of the current maximum to be passed back to the original call and the value of M should be initialized on the first use of the clause. Depending on the test, the programmer may decide that:

(a) This piece of code is correct

(b) Some initialisation should be provided in a different part of the program

(c) A way to pass the result back to the original call is needed. This should be outside the loop, and therefore in a different piece of the program.

As we saw before, the conclusion that this piece of code is correct but that the program should be augmented, is false. At a later stage, the programmer should realise that this decision must be revised.

3.6 Evaluating the model

The model can be summarized as follows:

(a) an *iterative algorithm* is used as a design. For most programming tasks that are used in introductory courses, there is a rather simple algorithm. This is generally used as the basis of the program. It appears that constructing declarative designs has to be taught before students will do it.

(b) an algorithm is implemented by translating it into Prolog constructs that are similar in appearance or similar in meaning. This similarity can be semantic or syntactic and internal (i.e., to constructs within the language) or external (i.e., to another language, like an algorithmic language or natural language). The malrules can be partially explained as modifications of correct rules, which they resemble fairly closely. In particular similarities between Prolog constructs on one hand and constructs in imperative languages such as Pascal seem to be responsible for the malrules.

The models were evaluated on two types of data: a collection of buggy programs and two sets of think aloud protocols. The buggy programs were collected from students who were doing Prolog exercises during tutoring sessions of an introductory course in Prolog programming for AI applications. The group of students consisted of (first year) computer science students, (third year) psychology students and some others from other disciplines. The computer science students and some of the others had a 3 month background in Pascal. About one quarter had no previous programming experience. To stimulate free "submission" of buggy programs, these were collected anonymously, but as a result the programming experience of the author of an error is unknown. Few students contributed more than one bug, so the collection covers a rather wide range of subjects. The problems were all simple problems in the style of introductory textbooks, such as "find the maximum of a list of numbers". Although most of the subjects knew Pascal before they began to learn Prolog, I found that algorithmic solutions were almost universal. There were no differences in this respect between subjects with and without experience with Pascal (or other languages) and between novices without programming experience. It should be noted that in the initial stage of the course the emphasis was on understanding the virtual machine and though *logic programming* was explained, it was not consequently enforced as the programming method.

The model is evaluated as follows: an algorithm is specified in Pascalese and then an attempt is made to reproduce the program written by the student, using rules and malrules from Iterator. This was done by hand, because Iterator cannot

Table 1. Results of bug analysis

Number of programs:	54	
Unexplained by malrules:	32	
Explained by malrule(s):	22	
Frequency of malrules:		
Malrule 1		0
Malrule 2		16
Malrule 3a		3
Malrule 3b		7
Malrule 4a		1
Malrule 4b		3
Malrule 4c		0
Malrule 6		1

automatically recognize the wide range of syntactic variations (e.g., order of arguments and procedures, different notations for equality, etc.). The second kind of data are think aloud protocols, taken from students from the same population, who were writing an initial version of a program.

3.6.1 Modelling a collection of bugs

I shall first discuss the results related of the bug collection. For each buggy program it was determined if it could be produced by implementation rules and malrules from an iterative algorithm. Programs for which this was impossible were categorized as "different issue". Examples were compound bugs, as the effect of malrules was combined with misunderstanding list syntax. Some bugs could not be explained at all. Table 1 lists the frequency of hypothesized malrules. This suggests that about half of the errors in a more or less random sample of solutions to simple programming problems can be explained by an iterative algorithm and the malrules from the previous section.

3.6.2 Modelling think aloud protocols

In general a (buggy) program is the result of a series of implementation steps, some of which undo the effect of previous steps. Therefore most protocols show more than one malrule being applied, or show how a correct program is written after a series of malrule applications that were detected and repaired when the program was evaluated. Below we briefly summarise the implementation malrules that appeared in the protocols.

In an initial evaluation of the model, four think-aloud protocols of a single subject were analysed. This subject solved four exercises involving lists, using only paper and pencil. He was asked to give an initial solution, that he believed to be

completely or almost correct. The protocols were coded in the following categories: application of a rule or malrule as defined in Iterator, plus some extra categories (stating algorithm, detecting problem/impasse, discover new algorithm, evaluate by mental execution, other). One protocol may contain several rule applications, when a subject withdraws a partial implementation. Table 2a lists the frequency of each category. Table 2b lists results of the same analysis, applied to 5 protocols of other novices solving list problems. Like the first subject they had several weeks of introductory Prolog education from a standard textbook.

3.7 Discussion

With the small samples of subjects and problems, no claims for generality can be made here, but the data do show that the model explains a fairly large part of (some) novices' reasoning steps when solving simple Prolog problems that involve lists. The data support the model's validity for at least a class of novices, in the initial stage of one course. More data is needed to estimate its generality.

Another limitation of the methodology used here, is that the malrules approach suggests that novices' behaviour is consistent. In practice, the malrules are an approximation that characterize a certain stage. The think-aloud protocols show

Table 2a. Protocol analyses of a single subject

Rule 1	0	*Malrule 1*	1
Rule 2	0	*Malrule 2*	4
Rule 3	1	*Malrule 3a*	3
Rule 4	1	*Malrule 4a*	4
		Malrule 4b	1
		Malrule 4c	1
Rule 5	4	*Malrule 5*	0
Rule 6	3	*Malrule 6a*	0
Rule 7	2		
Rule 8	2		
Other	22		

Table 2b. Results of protocol analysis

Rule 1	0	*Malrule 1*	1
Rule 2	0	*Malrule 2*	3
Rule 3	0	*Malrule 3a*	0
		Malrule 3b	3
Rule 4	1	*Malrule 4a*	0
		Malrule 4b	0
		Malrule 4c	0
Rule 5	1	*Malrule 5*	1
Rule 6	2	*Malrule 6a*	4
Other	15		

that subjects often realize that an implementation may well be wrong, but they simply don't really know how to do it and just try their best guess. This means that under slightly different conditions they may try something else and that they are willing to use another implementation if they find one that looks more promising. A full model should take these changes into account.

4. General discussion

Only one aspect of novices behaviour was considered in detail: the design and implementation of small programs. The effects of teaching, tools and individual differences have not been considered. I shall first discuss the nature of the problems that (some) novices seem to have with Prolog and next the implications for teaching Prolog and for tools.

4.1 Novices' problems

The results of the empirical studies can be summarised in Figure 2, which shows the path to the class of buggy programs discussed here, together with its branches that could have avoided the bugs. This is in line with other psychological work on programming. Students seem to approach the programming problem initially as a planning task. Because we only considered small problems, difficulties did not originate from the complexity of the design, but from the lack of correspondence between students' designs and the Prolog constructs that they knew.

Figure 2 shows many paths that lead away from our subjects' buggy implementations. In the design stage a *declarative design* would have avoided most of their bugs, simply because the kind of errors implied by the malrules would have been absent. A declarative design would be a definition of a predicate in some

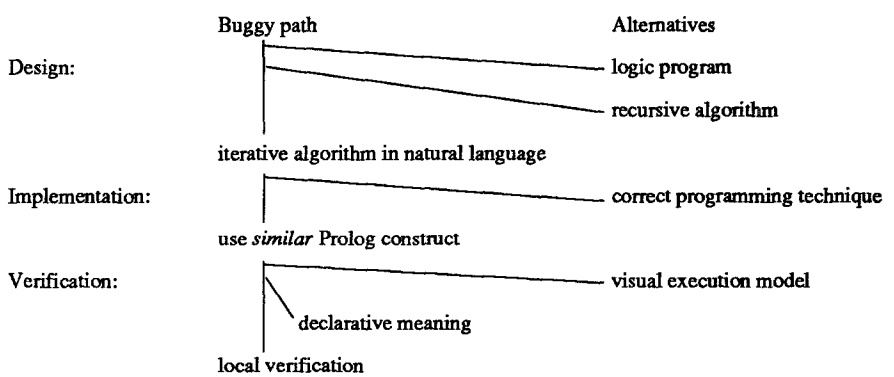

Figure 2. Process structure of novice programming in Prolog

structured natural language. In that case, the malrules in our model would not apply. However, as shown by Taylor and du Boulay (1986), this will by no means result in faultless programs. Students who follow a declarative approach may still follow the "buggy" path in the diagram, but the result will be different.

For several problems, working from a recursive algorithm instead of an iterative one might have avoided some of the errors, because for a recursive algorithm, the corresponding Prolog constructs are more similar to those in the algorithm. In particular, the dataflow in a recursive program can be identified more easily in the corresponding algorithm.

In the implementation stage, many problems might have been avoided if the novice had known the right programming techniques, rather than using Prolog constructs that are similar to a part of the algorithm. However, this requires these techniques to be made explicit and taught to novices.

Finally, because of the paper and pencil situation of the studies, the subjects had to verify the program by executing it mentally on examples. This requires a good understanding of how Prolog works and also great care to cope with the complexity. Note that this study shows that the student has to do two things. He must locate the source of failure of the program (the bug) and in addition to that, he should detect his own malrule, that was responsible for creating the bug. The results of this study show, that a buggy (initial version of a) program is the result of a number of interacting factors:

- *Iterative, procedural design:* If a student worked from a *recursive design*, he would not have met some of the implementation problems that are responsible for many errors, e.g., the relation between variables and their use in a recursive algorithm is much more similar to the use of Prolog variables and arguments than the implementation of an accumulator in an iterative design. Therefore errors are more likely in the latter case.
- *Inadequate programming techniques:* If the student had explicitly known the correct techniques for implementing a loop with accumulator, then he would not have made these mistakes.
- *Inadequate Prolog story:* If the student had had a complete and detailed understanding of the workings of the Prolog machine, the difference between a part of an algorithm and its mal-implementation would have been obvious.

4.2 Prolog or Pascal(ese)

The results of this study could also be interpreted as an illustration of the weakness of Prolog as a programming language, rather than a weakness of novices. If tricky techniques are necessary to implement a very simple design, then why use (or teach) Prolog in the first place? This issue cannot be discussed here in any depth. It just seems that Prolog is not primarily designed for the implementation

of algorithms and therefore this task will require some unintuitive techniques. This is compensated by the ease of implementing factual descriptions in Prolog, as compared to other languages.

4.3 Implications for teaching and tool development

The psychological studies suggest some tentative conclusions on how Prolog should be taught and what kind of tools would make programming easier.

- To avoid the use of similar constructs, programmers need a complete and detailed execution model of Prolog – a Prolog story (see Pain and Bundy, 1987). That will allow them to see if a proposed implementation is indeed correct. Tools that support tracing the execution are likely to be essential.
- Some problems can be prevented by teaching students explicit programming techniques. Textbooks do not present these techniques. The correct Iterator model shows a few small examples of these programming techniques.
- The novices who were studied hardly ever used a recursive design. This can probably be repaired by explicitly teaching the design of recursive programs.
- It is not clear what the role of the declarative meaning of a Prolog program can be in the programming task. Our novices do not use it spontaneously, but other work (e.g. Kowalski [1982]) showed that it can be taught to them. However, that produces a different type of errors. Perhaps logic programming could be a first pass in designing and verifying programs.

Acknowledgements

Several colleagues and students contributed to the studies of Prolog novices reported here: Bob Wielinga, Yvonne Barnard, Anton Eliëns, Jeanette Quast, Kees Buisman, Gerard Wagenaar and Dick van der Vlugt. Radboud Winkels collected some of the data and implemented ITERATOR. Discussions with Josie Taylor, Helen Pain and Paul Brna have also been helpful in developing the ideas presented in this paper. Comments from the anonymous reviewers contributed much to the presentation.

References

Adelson, B. (1981). Problem solving and the development of abstract categories in programming languages. *Memory and Cognition, 9,* 422–433.

Adelson, B., Littman, D. and Soloway, E. (1984). A model of expert design. In Proceedings of the Cognitive Science Conference, Boulder.

Anderson, J. R. (1983). Learning to program. In Proceedings of the Eighth International Joint Conference on Artificial Intelligence, Karlsruhe, Germany.

Anderson, J. R., Farrell, R. and Sauers, R. (1984). Learning to program in LISP. *Cognitive Science, 8,* 87–129.

Bonar, J. and Soloway, E. M. (1985). Pre-programming knowledge: a major source of misconceptions in novice programmers. *Human-Computer Interaction, 1,* 133–162.

Brown, J. S. and VanLehn, K. (1980). Repair theory: a generative theory of bugs in procedural skills. *Cognitive Science, 4,* 379–426.

Burstein, M. (1983). Concept formation by incremental analogical reasoning and debugging. In Proceedings of the Machine Learning Workshop 1983, Illinois.

Clocksin, W. and Mellish, C. (1981). *Programming in Prolog.* Berlin: Springer Verlag.

Coombs, M. J. and Stell, J. G. (1984). A model for debugging PROLOG by symbolic execution: the separation of specification and procedure. Department of Computer Science, University of Strathclyde.

Douglas, S. A. and Moran, T. P. (1980). Learning operator semantics by analogy. In Proceedings AAAI-80, 100–103.

du Boulay, B., O'Shea, T. and Monk, J. (1981). The black box inside the glass box. *International Journal of Man-Machine Studies, 14,* 237–249.

Gegg-Harrison, T. S. (1989). Basic Prolog schemata. Report CS-1989-20, Department of Computer Science, Duke University, Durham, North Carolina.

de Groot, A. D. (1965). *Thought and choice in chess.* The Hague: Mouton.

Hoc, J.-M. (1981). Planning and direction of problem solving in structured programming: an empirical comparison between two methods. *International Journal of Man-Machine Studies, 15,* 363–383.

Jeffries, R., Turner, A., Polson, P. and Atwood, M. (1981). The processes involved in designing software. In J. R. Anderson (Ed.), *Cognitive skills and their acquisition.* Hillsdale, NJ: Lawrence Erlbaum Associates.

Johnson, W. L. and Soloway, E. M. (1984). Intention-based diagnosis of programming errors. Proceedings AAAI-84, 162–168.

Kahney, H., (1982). An in-depth study of the cognitive behaviour of novice programmers. Human Cognition Research Laboratory, Tech. Report 5, The Open University, Milton Keynes.

Kowalski, R. (1982). Logic as a programming language for children. In Proceedings of the European Conference on AI 1982, 2–10.

Mayer, R. E. (1981). The psychology of how novices learn computer programming. *Computing Surveys, 13,* 121–141.

McKeithen, K. B., Reitman, J. S., Rueter, H. H. and Hirtle, S. C. (1981). Knowledge organisation and skill differences in computer programmers. *Cognitive Psychology, 13,* 307–325.

Miller, L. A. (1975). Naive programmer problems with specification of transfer-of-control. In Proceedings of AFIPS National Computer Conference, 44, 657–663.

Pain, H. and Bundy, A. (1987). What stories should we tell novice Prolog programmers? In R. Hawley (Ed.), *Artificial intelligence programming environments.* Chichester: Ellis Horwood.

Perkins, D. N. and Martin, F. (1986). Fragile knowledge and neglected strategies in novice programmers. In E. M. Soloway and S. Iyengar (Eds.), *Empirical studies of programmers.* Norwood: Ablex.

Pirolli, P. L., Anderson, J. R. and Farrell, R. (1983). Learning to program recursion. In Proceedings Cognitive Science Conference, Boulder.

Plummer, D. (1990). Cliché programming in Prolog. In M. Bruynooghe (Ed.), Proceedings of the 2nd Workshop on Meta-Programming in Logic, 247–256, Leuven.

Rist, R. S. (1986). Plans in programming: definition, demonstration and development. In E. M. Soloway and S. Iyengar (Eds.), *Empirical studies of programmers.* Norwood: Ablex.

Shapiro, D. G. (1981). Sniffer: a system that understands bugs. AI Memo 638, MIT AI Lab.

Soloway, E. M. and Woolf, B. (1980). Problems, plans and programs. *SIGSCE Bulletin, 12,* 16–24.

Soloway, E. M. and Iyengar, S. (Eds.) (1986). *Empirical studies of programmers.* Norwood: Ablex.

van Someren, M. W. (1990). Understanding students' errors with Prolog unification. *Instructional Science,* this issue.

Spohrer, J. C., Soloway, E. and Pope, E. (1985). A goal/plan analysis of buggy Pascal programs, *Human-Computer Interaction, 1,* 163–207.

Spohrer, J. C. and Soloway, E. M. (1986). Novice mistakes: are the folk wisdoms correct? *Communications of the ACM, 29,* 624–632.

Taylor, J. and du Boulay, B. (1986). Studying novice programmers: why they may find learning Prolog hard. In J. Rutkowska (Ed.), *Issues for developmental psychology.* New York: Wiley.

Tromp, D. (1989). *The acquisition of expertise in computer programming.* Ph.D. thesis, University of Amsterdam.