Chapter 2.5

Programming Languages in Education: The Search for an Easy Start

Patrick Mendelsohn[1], **T.R.G. Green**[2] **and Paul Brna**[3]

[1] *Faculty of Psychology and Educational Science University of Geneva, Switzerland*
[2] *MRC Applied Psychology Unit, Cambridge, UK*
[3] *Department of Artificial Intelligence, University of Edinburgh, UK*

Abstract

(i) We first discuss educational objectives in teaching programming, using Logo research as a vehicle to report on versions of the 'transfer of competence' hypothesis. This hypothesis has received limited support in a detailed sense, but not in its original more grandiose conception of programming as a 'mental gymnasium'. (ii) Difficulties in learning quickly abnegate educational objectives, so we next turn to Prolog, which originally promised to be easy to learn since it reduces the amount of program control that the programmer needs to define, but which turned out to be very prone to serious misconceptions. Recent work suggests that Prolog difficulties may be caused by an inability to see the program working. (iii) Does the remedy therefore lie in starting learners on programmable devices that are low level, concrete and highly visible? Research on this line has brought out another problem: learners find the 'programming plans' hard to master. (iv) Finally, we sketch a project designed to

Psychology of Programming
ISBN 0-12-350772-3

teach standard procedural programming via 'natural plans'. Our conclusions stress pragmatic approaches with much attention to ease of use, avoiding taking 'economy' and 'elegance' as virtues in their own right.

1 Introduction

For all the efforts of the computing fraternity, computing remains inaccessible to many people, especially the old, the very young, the disabled, and those with low educational achievements. In this chapter we shall outline some of the attempts to open up computing for educational purposes, especially school children and 'distant learners' – pupils learning without face-to-face contact with a tutor. For these purposes one wants to avoid languages that enforce many new abstract ideas, or have too many 'programming tricks' to be learnt, or give too little immediate concrete feedback. So there has been much interest in new languages designed for children that are friendlier and more concrete, such as Logo; and in languages like Prolog, which appear to avoid some of the problems of abstractions and programming tricks. We shall describe work with both of these languages. We shall also look at attempts to break out of the 'programming language' mould and to create systems that are more immediate than either Logo or Prolog.

At the same time as new languages have been developed, there has been much discussion of whether the primary aim should be to teach children programming for its own sake, or to use programming in the service of some other end or discipline – 'programming to learn, or learning to program'. Different programming cultures have emphasized different balances between the two polar positions. These educational objectives are not, strictly speaking, within the scope of this volume, which takes as its starting point the view that unless the programming language is adequately matched to the abilities of its users, nothing else can be done and all objectives will fail. Yet, as we shall quickly observe, these aspects tend to get themselves mixed together.

Of course, choosing a language in school is not a free choice. The development of educational computing is directly dependent on different sources of material constraints, both institutional and human. Teachers who decide to use a computer in the classroom have, at least initially, only a limited influence on these constraints. They passively follow developments in hardware and software and adapt to political choices concerning computer equipment more than participating in decisions. They accompany human transformations more than determining them. Nevertheless, there are enough pedagogical alternatives remaining for any reflection on the adaptation of computer languages to the goals of teaching to be useful and fruitful.

The rapid and spectacular progress made in computer and software performance suggests that great care should be taken when deciding on which equipment to introduce into the classroom for teaching programming. How and with what goals in mind must programming languages be used at school such that this utilization and teaching are not completely outdated in five years time? It quickly becomes clear that even if organizational constraints can be put aside, language choices must be based on many different criteria.

First, the *language* level will affect the level at which the pupils will work, and will partly determine how much of their time is spent on learning to program and how much on using that knowledge to aid other kinds of learning.

Secondly, there is the *transfer* problem of making skills and knowledge learned in the context of programming available in a different context. Kurland *et al.* (1989) point out that the degree of transfer depends on how close the domains are. 'Near' transfer effects can be obtained without too much difficulty (Littlefield *et al.*, 1988), but 'far' transfer is difficult to establish unless the analogy between domains of knowledge is explicitly taught.

Finally, the teacher must choose a *teaching style*. The first applications of computing in education were marked by a debate, often lively (Solomon, 1986), between the defenders of programmed teaching and the partisans of learning through discovery and self-teaching. For the first clan, computing is essentially an effective tool for training and repeating teaching sequences (Suppes, 1979). For the others, computing is more like a medium, a support for elaborating environments in which the child is his or her own knowledge builder (Papert, 1980).

1.1 Language cultures and research questions

Given these complex interrelationships, it is hardly surprising that the choice of a classroom language brings with it a particular culture. 'If you weren't interested in the problems that Logo can deal with, you wouldn't have chosen Logo' – and thus, further research on Logo tends to be dominated by the same set of questions. In such a way does a culture perpetuate itself. And more: the Logo culture has always emphasized interaction, either with real devices ('turtles') or else with virtual devices ('screen turtles'); these crawling and drawing devices add amusement but also present powerful challenges. Logo systems without graphics are unthinkable. Not so Prolog, which has always emphasized reasoning about a database of assertions; many Prolog systems have no graphics, or limited graphics that tend to subvert the declarative style of Prolog with a procedural outlook.

Problems with the Logo language itself have not been a major discussion point. There are some predictable difficulties, such as learning to tell the turtle which way to go when it is facing south (when the turtle's left is the viewer's right), but, in general, turtle graphics has been fairly successful. It is not surprising, then, that the major research efforts in connection with Logo have dwelt on 'the cognitive effects of learning to program', in Pea and Kurland's phrase.

Prolog, on the other hand, has been a troublesome language, and fewer interesting research results have come out of teaching children Prolog. The reasons appear to be an emphasis on relational database querying, which is relatively unproblematic; emphasis on teaching some other subject using Prolog simply as a means to represent the knowledge; and the use of a 'toolkit' as a front end to give a simplified way in which children can add knowledge to a ready-programmed computational mechanism. These can all be seen as ways to get some benefits from Prolog while not having to teach children how to program in Prolog.

Our first two sections, therefore, will deal with research in each of these traditions. We then present some contrasting work in which the traditional, extremely abstract concept of a programming language has been rejected in favour of far more concrete devices, in one case a microprocessor with sensors and in the other, a model train. Certain themes will recur throughout these treatments, such as the problem of helping novice programmers to perceive programming plans, and we finish with a forward-looking research project which aims to present plans more directly than has been achieved before.

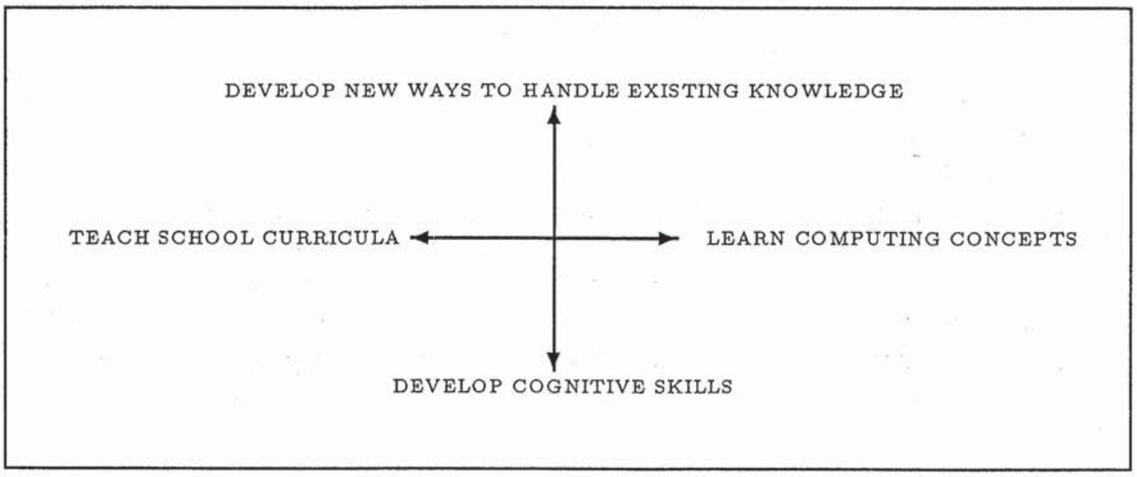


Figure 1: Schematic representation of different objectives for programming activities in school: vertical axis, transfer of competence hypothesis; horizontal axis, the learning of content.

2 Programming to learn, or learning to program? Educational objectives and Logo

Logo is the language that is predominantly used in general education in France and is also widely used in Britain and North America. It has immediate appeal to children because of its 'turtle graphics', which originally were instructions driving a small device that crawled around the floor. The device could lower a pen to mark its path and had sensors to record obstructions. Interest in the possibilities of Logo's graphics model gradually became one of the dominant themes of Logo work, and the floor-based turtle is now less important than the screen turtle which draws lines on the VDU. The same instructions are used, preserving the 'turtlocentric' view: the turtle knows how to turn left or right by a stated number of degrees, and how to go forwards or backwards by a given length. Logo is a procedural language with provision for procedures (using parameters called by value), variables, conditionals, recursion and graphics. Data structures include numbers, strings and lists.

Following the guiding principles described in our introduction we shall devote our attention to the central question raised by this choice of language, namely, the problem of deciding upon educational objectives. Four types of project will be presented; we believe that the reflections proposed here can readily be generalized to many different programming contexts.

The four approaches we shall describe can be represented on two axes (Figure 1). The options that consist of using Logo, with the 'transfer of competence' hypothesis as a guiding concept, are represented on the vertical axis. The research scientist is more interested here in programming as the exercising of competence that can be reinvested in other school situations, than programming as an activity of developing programs.

Represented on the horizontal axis, on the other hand, are the practices of teachers who use programming languages with the central idea that children learn, above all, specific information and know-how related to programming.

Before going further into the details of this analysis we can already note that what actually does happen in the classroom corresponds more to the horizontal axis, whereas the vertical axis (transfer hypothesis) is generally called upon to justify these practices!

2.1 The transfer of competence hypothesis

The general hypothesis of transfer covers, in fact, two quite different conceptions. On the one hand there is the most classical position at the origin of the creation of Logo. A programming language is considered as a medium that creates new ways of dealing with existing knowledge. This perspective is supported by several authors (Papert, 1980; O'Shea and Self, 1983; Solomon, 1986; Lawler, 1985). These authors stress the role of self-training and the child's discovery of his or her own problem-solving strategies as well as in co-operative problem solving by learners exploring a programming world that is shared between work stations. They also consider the introduction of programming to be a revolution in teaching practice. Knowledge is transformed and the classical competence taught at school is generally rendered obsolete. Problems of evaluation are generally circumvented in that it is considered that if one child can do something, other children can do the same if they are given the means.

On the other hand there are those authors who support a more cognitivist perspective. Through programming practice children develop cognitive skills that are identifiable and transferable to other situations. The most studied of these elementary aptitudes solicited by programming activity have been analogical and temporal reasoning, mathematical operations (Ross and Howe, 1981), the planning of action (Pea and Kurland, 1984; Lehrer *et al.*, 1988), error correction (Klahr and Carver, 1988), and the development of logical and spatial operations (Mendelsohn, 1986).

2.2 Transfer of competence as new approaches to knowledge and learning

Within Papert's perspective, Logo provides an easy approach to the art of heuristics. The principal force of structured programming is that procedures can be created, like separable blocks, to obtain a progressive construction of the solution or for solving larger problems (an example of this is provided in the classical procedure HOUSE given to beginners). It is therefore logically possible to know how each one of us breaks down a problem into simple units and then co-ordinates these units into macro-actions.

Howe and O'Shea (1978) provide an illustration of this approach. These authors attempted to test the hypothesis according to which the child learning to program in Logo uses a system of strong metaphors to describe reality. These metaphors can be linked to the body schema (using Logo to add meaning to and describe movements), to denomination (the fact that procedures have a name and can be recalled in another part of the program), to the breaking down of a problem into subproblems (structured programming). Why not therefore conclude that the child can then use these metaphors to transmit knowledge to someone else in a different context?

Howe and O'Shea designed the following experiment to test this hypothesis. One group of children having learnt to program in Logo and another group with no programming experience were placed in a situation similar to the game 'battleships' in front of a screen hiding a partner. The subjects have in front of them a figure composed of small geometrical shapes like in a jigsaw puzzle and their partners have a set of geometrical shapes, a subset of which is identical to those used in the figure. The subjects have to explain to their partners, without showing them any of the shapes, how to build the figure.

Howe and O'Shea hypothesized that if the children with programming experience in Logo have learnt anything relative to the metaphors of communication (modularization, denomination, sequentiality, etc.), then they should be more capable of explaining how to build the figure than the children with no programming experience. The authors expected the 'programming' children to break down the figure into relevant subsets, give names to these subsets using suitable labels and only to propose executable actions to their partner. The 'non-programming' children were expected to enumerate the different pieces of the puzzle giving ambiguous instructions.

The results show that some children do seem to have recognized the analogy between this problem and the programming situation. The results must be considered with caution, however, as the necessary methodological precautions were not taken to allow a non-subjective interpretation of the data. Other similar experiments have not shown such analogical transfer (Littlefield *et al.*, 1988). Nevertheless, these considerations and the pedagogical reflections associated with these experiments have a non-negligible impact on teachers and can encourage them to re-evaluate some of the pedagogical objectives of their classes.

2.3 Transfer of competence as the development of cognitive skills

The other perspective of the 'transfer hypothesis axis' is characterized by a more rigorous experimental approach and by its explicit reference to cognitive psychology. A 'cognitive skill' is a competence associated with the manipulation of identifiable operators, not specific to programming. Moreover, this competence must be sufficiently generalizable and exercised to be re-invested in other tasks. Pea and Kurland's (1984) study on the development of aptitude for action planning is the most representative of this type of research. This aptitude is evaluated by the capacity for optimizing the representation of a string of actions that is too large to be managed directly in working memory. These authors have experimented with seven- and eleven-year-old children some of whom had one year of Logo instruction at the rate of one hour a week. For Pea and Kurland, planning is only necessary if the situation imposes several constraints on the subject: (1) planning is the only means of solving the problem; (2) the task must be sufficiently complex so that memorizing the subgoals is impossible; and (3) the area of knowledge is familiar enough for the children to be able to identify the elementary actions to be performed. The situation-problem used by these authors was represented by a three-dimensional model of a classroom with objects and furniture (chairs, tables, plants, etc.). The starting point was the door of the classroom. The children had to carry out a number of concrete actions on the model: water the plants, clean the blackboard, put the chairs in front of the tables, wash the tables, move certain objects, etc. These different actions can be performed in any order but the children were instructed to find the route that minimized the number of movements necessary. Each subject had three trials at the beginning of the school year (pre-test) and three more trials at the end of the year (post-test). A measure of performance was calculated for each subject. The results provided little support for the transfer hypothesis. The expected effects of age and order of trial were observed but no significant difference was observed between children with Logo experience or not (neither in terms of the types of strategy used nor the objective measures of efficiency).

In a second experiment Pea and Kurland provided a further analysis of this problem. As the transfer task in the first experiment may have been too different from the type of task performed in programming situations, a computerized version of this task was used in the second experiment. Thus, instead of being performed on a scale model, the operations were simulated on a computer and the subjects had to provide a list of commands. Even if this situation penalizes the control group the results were no better than in the previous experiment. Similar results were obtained by Littlefield *et al.* (1988) using the same planning situation.

These results, disappointing for those who defend the transfer of competence hypothesis, lead to the formulation of at least three statements (for a detailed analysis see Crahay, 1987; Mendelsohn, 1988; Kurland *et al.*, 1989).

* It is unrealistic to think that one hour a week of programming over a one-year period will allow transfer of such a specialized competence as planning. Would anyone have attempted a similar experiment for a less attractive activity, such as the game of chess, where no programming language is involved?

* One should perhaps be more realistic and take more time analysing what children really do before hypothesizing about the expected form of transfer. Pea and Kurland (1984) also stress the methodological problems associated with inaccurate evaluation of the level of expertise that the 'programming' children attain. To surmount this difficulty Klahr and Carver (1988) suggest that a formal analysis be made of what is learnt and what is supposed to be transferable. However, this costly precaution is rarely taken into considerations by the researchers in this field.

* The principal interest of Logo in the classroom is perhaps not to be found in this direction. A programming language is above all a system of representation and the interest of such a system lies in its capacity to highlight new properties of the manipulated objects while allowing the automatic execution of complex processing. Learning arithmetic benefits the child more by proposing a formal language that facilitates the symbolic processing of complex numerical problems than by developing hypothetical aptitudes for reasoning. The same is perhaps true for learning to program. This point of view characterizes the second axis of Figure 1.

2.4 The acquisition of new knowledge

With respect to the axis that represents the learning of content hypothesis (the horizontal axis of Figure 1), it is also possible to oppose two approaches with very different goals. On the one hand there is the idea that programming helps children learn, above all, concepts of computing: programming operations, data structures, variables (reference to the term learn). On the other hand, there are those who stress the teaching of specific courses (programming is always applied to particular content). Thus programming is used as a back-up for teaching geometry, arithmetic, or even grammar.

2.5 Knowledge acquisition as mastering computer concepts

The teaching of programming at school is not aimed at producing computer programmers but rather computer users. One can nevertheless think that basic computer concepts are an integral part of computer culture. Because of this, the learning of these concepts has received much attention from psychologists who have considered this to be a particularly rich field for fundamental research. The development of these concepts in children, the cognitive difficulties involved in their acquisition, and their origin, are the main themes of research that interest computer educationalists (Rogalski and Vergnaud, 1987). Some examples of the operations studied here are iteration (Soloway *et al.*, 1983; Kessler and Anderson, 1986), conditional branching (Rogalski, 1987) and recursion (Anzai and Uesato, 1982; Rouchier, 1987; Mendelsohn, 1985; Pirolli, 1986; Kahney, 1983). Some more-simple concepts can be added to this list: sequentiality, modularity, and the notion of computer variable. These have been studied less since they are implicitly linked to the former.

Recursion interests many researchers because of its particular status in programming. It is a powerful operation which is trivial in its definition yet raises many problems when it comes to teaching it in its most complex forms. Hofstadter (1979) describes recursion as a mode of reasoning characterized by self-reference and the nesting of processing ('This sentence has five words'). Roberts (1986) stresses the algorithmic point of view. Recursion is thus a means of solving a problem by reducing it into one or several subproblems which are (1) structurally identical to the original problem, and (2) more simple to solve. Finally, strictly from a computing point of view, recursion is a program control structure (its written form depending on the language used). Kurland *et al.* (1989) stress the fact that it is important for the learner to have a good representation of what is going on inside the 'black box' when running recursive procedures in order to be able to make good use of them.

With a similar aim, we wanted to combat children's difficulties in understanding recursive procedures: we developed (Mendelsohn, 1985; Rouchier, 1987) a teaching technique that involves recognizing the specific figural characteristics that allow a graph to be described by recursion. The model of reference chosen for this is central recursion, which can be described formally as follows:

```
TO PROCEDURE-NAME :VAR
IF predicate [ACTION3 STOP]
ACTION1
PROCEDURE-NAME :VAR
ACTION2
END
```

(with `ACTION2` non-null in order to rule out tail recursion).
The prototypical example used to help children understand the functioning of this recursive procedure is the example of the ski tow. It is possible to imagine a series of ski tows that take the skier to the top of the slopes. Each time the skier takes a ski tow (ACTION1) a ski down (ACTION2) is potentially accumulated, the parameters of which are related to ACTION1 (difference in height, for example). Performing ACTION2 is suspended and only carried out once the skier decides to stop climbing. At the top of the slopes the skier can perform any other kind of action such as rest or eat (ACTION3) and then ski down the slopes in the opposite order in which they

were climbed. Using this model one can vary during learning both the existence, content and structure of the procedures ACTION x, and the number and nature of the transformation functions associated with the variables. The teaching is therefore centred on the conceptualizing of regularities observed during these variations (Figure 2).

In this way the child learns to use recursion as a means of describing objects with very precise characteristics. In the examples provided one can underline the initial symmetry and the order in which the figure is constructed. The teacher can progressively introduce sources of variation that slowly add to the diagram until it is formally complete. This teaching technique is the same as used with other operations such as addition and multiplication in arithmetic. Teaching of computer programming can be seen in this way as a laboratory that allows testing relative effects of various instruction methods on student's mental models. This topic is now starting to be explored in the direction of guided discovery learning (Lehrer *et al.*, 1988).

2.6 Knowledge acquisition as teaching school curricula

This last theme deals with the use of Logo for teaching school curricula. It is the course content that is stressed here, the programming language being considered as a support for the representation of original properties concerning contents and transformations (Hoyles and Noss, 1987). For many teachers this is often the only project that incites them to use Logo in the classroom. Several original examples of this have been provided by Bourbion (1986). These are educational applications in which the control of the subject's performance is centred on the course being taught (mostly arithmetic and geometry). Programming thus becomes an implicit activity, mastery of which is often considered as an introduction to the main work.

The problem that has been chosen as an example here consists, first of all, in editing a small graphics program. With this program, two walls the same height can be built with bricks of different thickness in two places P1 and P2 (Figure 3, first phase). The algorithm places a brick at P1 then checks if the height of the wall is lower at P2. If this is not the case another brick must be added at P1 and the relative height checked again. If this is the case then a brick must be added to P2 and then a check made to see if the height at P1 is lower, and so on.

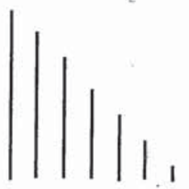
The second step (Figure 3, second phase) involves subtly transforming this program by removing all graphic aspects from the procedures. In this way we obtain an isomorphic program which by generalization becomes a procedure for calculating the lowest common denominator.

With this technique, arithmetic conceptualization re-adopts its principal aim, that is the progressive pruning of the actions that one is led to perform in reality in order to conserve only the essential transformations. The programming language thus acquires the status of a formal language and can even become, as such, an object of teaching (see also Vitale, 1987). Many other examples involving geometry concepts, physics or language could be mentioned. The recurrent idea remains that of using a programming language to represent specific components of the subject discipline and throw light on new links between different domains (mediated learning).

```
TO PROCEDURE1 :LENGTH
IF :LENGTH < 0 [STOP]
ACTION1 :LENGTH
PROCEDURE1 :LENGTH - 10
END

TO ACTION1 :LENGTH
FD :LENGTH BK :LENGTH
PU RT 90 FD 20 LT 90 PD
END
```

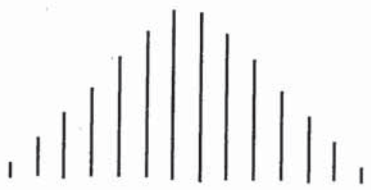
PROCEDURE1 70



This is a standard procedure with terminal (tail) recursion.

```
TO PROCEDURE2 :LENGTH
IF :LENGTH > 70 [STOP]
ACTION1 :LENGTH
PROCEDURE2 :LENGTH + 10
ACTION1 :LENGTH
END
```

PROCEDURE2 0

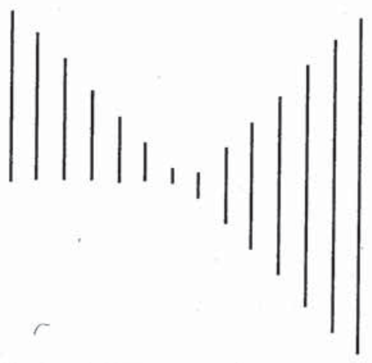


This procedure, which uses ACTION1 before and after the recursion, introduces the idea of an apparent symmetry with opposite orders of evaluation for the two parts.

```
TO PROCEDURE3 :LENGTH
IF :LENGTH < 0 [STOP]
ACTION1 :LENGTH
PROCEDURE3 :LENGTH - 10
ACTION2 :LENGTH
END

TO ACTION2 :LENGTH
FD :LENGTH BK :LENGTH * 2 FD :LENGTH
PU RT 90 FD 20 LT 90 PD
END
```

PROCEDURE3 70

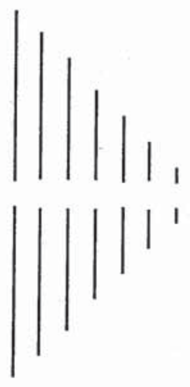


This procedure adds the idea that the two halves can relate to different forms (ACTION1 and ACTION2 are different).

```
TO PROCEDURE4 :LENGTH
IF :LENGTH < 0 [ACTION3 STOP]
ACTION1 :LENGTH
PROCEDURE4 :LENGTH - 10
ACTION1 :LENGTH
END

TO ACTION3
RT 180
PU RT 90 FD 20 LT 90 FT 20 PD
END
```

PROCEDURE4 70



In this last procedure the two parts of the figure can be disassociated. The procedure now controls two traces in parallel at any position on the screen.

Figure 2: An example of progression based on a starting diagram for the writing of centrally recursive procedures in Logo.

First phase: Developing a procedure for building two walls of equal height from bricks of different thicknesses

```
TO CONSTRUCT
IFELSE :H1 < :H2 [WALL1] [WALL2]
IF :H1 = :H2 [STOP]
CONSTRUCT
END
```

```
TO WALL1                      TO WALL2
SETPOS [0 0]  FD :H1          SETPOS [0 0]  FD :H2
MAKE "THICKNESS :E1           MAKE "THICKNESS :E2
PLACE.BRICK                   PLACE.BRICK ................add 1 brick of given thickness (code not shown)
MAKE "H1 :H1 + :E1            MAKE "H2 :H2 + :E2
END                           END
```

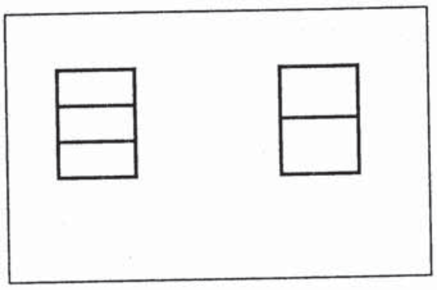
```
TO WALLS :A :B
INIT .........................MAKE "H1 0   MAKE "H2 0   MAKE "E1 :A   MAKE "E2 :B
CONSTRUCT
RESULT....................PRINT SENTENCE "Height: :H1
END
```

Input

```
? WALLS 12 18
Height: 36
```

Output

Graphics output →



Second phase: The graphics instructions and semantic references are removed from the program, leaving only the calculations.

```
TO PART1                   TO PART2                   TO RESULT
MAKE "H1 :H1 + :E1         MAKE "H2 :H2 + :E2         PRINT :H1
END                        END                        END
```

The procedure for building two walls becomes a procedure for least common multiple:

```
TO LCM..................................(Lowest Common Multiple)
INIT
CONSTRUCT
RESULT
END
```

Figure 3: An example of teaching school curricula through Logo: 'Procedure for building two walls' (after Bourbion, 1986).

3 The misconception problem: Prolog

The unfortunate novice is prey to misconceptions when meeting any programming language. There have been excellent studies of misconceptions about even so 'small' a topic as assignment (Mayer, 1979; Putnam *et al.*, 1986; du Boulay, 1986). Intensive studies of misconceptions were made by the group working with Solo, a language designed for distance teaching of adult learners in the Open University (Eisenstadt, 1979): some of their work was put into analysing error messages and how these were misunderstood (Eisenstadt and Lewis, 1985), and other work was devoted to analysing faulty mental models of recursion and conditionality (Kahney, 1983).

All programming languages found in the classroom are prone to a wide range of misconceptions – but arguably, the language that has been found to engender the greatest number of problems is Prolog. The idea behind Prolog is that to build a program, the programmer just writes down true statements that capture some logical relations, and the Prolog interpreter can draw on these as needed to discover whether they imply the truth of some arbitrary proposition. The original shining hope was that there would be no need to understand what the Prolog interpreter did – or even that it did anything at all: the programmer would just write down the conditions that defined the required answer, and out it would pop.

For example, the logic of the `ancestor` relation can easily be described in words as 'somebody is your ancestor either (1) if that somebody is one of your parents, or (2) if that same person is the ancestor of one of your parents'. It can be written in Prolog as two distinct propositions, or *clauses*, as:

```
1.      ancestor(X,Y):- parent(X,Y).
2.      ancestor(X,Y):- parent(X,Z), ancestor(Z,Y).
```

where `ancestor(X,Y)` is read as 'Y is the ancestor of X', the symbol `:-` is read as 'if the following is true', and `parent(X,Y)` is read as 'Y is the parent of X'. The comma symbol has the logical reading of 'and the following is true' so that the second clause means 'there is a third person Z who is a parent of X, and Y is an ancestor of Z'.

This definition of `ancestor` would be used in conjunction with a database of facts about who is the parent of whom:

```
parent(abel, adam).
parent(abel, eve).
parent(cain, adam).
parent(cain, eve).
parent(enoch, cain).
...  etc...
```

Prolog is then able to deduce whether or not one person is the ancestor of the other. The programmer needs to know no 'programming tricks' about searching data bases: those are all built into the interpreter. By setting the *query*

```
?ancestor(enoch, eve)
```

Prolog will correctly state that 'eve' is an ancestor of 'enoch'.

Of course, the Prolog interpreter is not gifted with foresight or special knowledge: it simply performs conventional depth-first search over the database. Ideally, that should mean that the programmer need only worry about the 'logical' meaning of a program and not about the way that Prolog evaluates a program. This is to say that *the logical reading of a program is supposed to coincide with the behaviour of the Prolog system*. Unfortunately this is not the case – and the discrepancy between the logical reading and Prolog's behaviour in executing the program is one fairly well-known source of problems. (Example: if the subgoals of clause 2 are reversed, so that `ancestor` is tried before `parent`, the logical meaning may well appear unchanged – but Prolog will eventually enter an unending recursion.)

Until about 1980, the use of Prolog was restricted to a small number of research groups around the world. Prolog first became known to the primary and secondary educational communities through the work of Kowalski and Ennals (see Nichol *et al.*, 1988, for a recent account of this group's work). For the next few years, the advocates of Prolog made much of the possibilities inherent in a language in which it was fairly easy to describe logical relationships without having to worry quite so much about how to control computation. But, although the language is very powerful and can be used with great effect by some students, other students seem to become tremendously confused. One problem was that there were inconsistencies present in the teaching materials and some major conceptual problems with the language (Bundy, 1984). In fact, Pain and Bundy (1987) outlined over six different, partial accounts which were then in use!

Evidence relating to misconceptions associated with Prolog began to accumulate. For example, students had problems with *backtracking*, which is what Prolog does when a query fails. Students were also known to have problems with the way Prolog tries to match two data structures. This form of pattern matching is special to Prolog and is known as *unification*. For example, matching `ancestor(enoch, X)` with `ancestor(Y, eve)` would result in the variable `X` being bound to `eve` and the variable `Y` being bound to `enoch`. Coombs and Alty (1984) were quick to note these difficulties, and more-detailed work demonstrating particular misconceptions was reported by a variety of researchers (Ormerod *et al.*, 1986; Fung *et al.*, 1987; Taylor, 1987; van Someren, 1985).

In due course a lengthy taxonomy based on collected reports was produced by Fung *et al.* (1987). Many misconceptions were particular to Prolog and its control structure; others were more general, such as the 'meta-analysis' documented by many researchers – i.e. the belief that Prolog is able to 'foresee' which goals will and will not succeed, and can choose its behaviour accordingly. This was especially noted in the context of extremely familiar relationships, such as kinship relations (parent, cousin, aunt/uncle, etc.). Such relationships are frequently used in introducing Prolog – as we have done here, in fact – but their very familiarity can make trouble for some students.

We need to consider three questions: Where do misconceptions come from? Should they be avoided at all costs, or are they part of learning? Can they be avoided, or at least reduced?

3.1 Where do misconceptions come from?

Taylor (1987) has provided a detailed analysis of constructing a Prolog program in terms of a three-level discourse model, distinguishing between the general, logical and mechanistic levels. Each type of discourse has its own appropriate components to deal with inputting data, reasoning about the data, outputting the results, and doing 'reflective evaluation' (the process of reconsidering what one did and whether it could be different). This model allows her to categorize various pathways through the framework in terms of the discourse level adopted at each stage. She considers the activity of learning to program as beginning with the general problem-solving discourse level. The student then has to come to terms with the formal problem-solving discourse level. This formal level requires students to come to terms with both the level of logical discourse and also the level of mechanistic discourse. Not surprisingly, when an alternative discourse path offers itself, by performing reasoning at the 'real-world' level, students will be tempted to adopt it, leading themselves to ignore the actual behaviour of the program. One of the attractive features of Taylor's model is that it emphasizes that certain problems arise only because Prolog is so close to the general discourse level. If it were further removed, such 'meta-analytic' problems would arise less often.

Negative transfer from previous programming experience has been frequently put forward as an explanation for learners' difficulties. Van Someren (1988) found that many students seemed to have a simple algorithm for converting a Pascal-based program into a Prolog form. Their algorithm, which was faulty, appeared to account for several errors, Pascal assignment being a particular problem. White (1988) also followed this claim up in an attempt to find statistical evidence that some such effect exists. His results were strongly suggestive but fell short of statistical significance.

Yet a third possible origin for misconceptions is to be found in the unification process. Van Someren (1990) suggests that many of the errors made by students in connection with the unification process can be explained by a 'damaged' version of the correct algorithm together with a 'repair' mechanism for handling impasses, an explanation similar in style to certain accounts of children's problems with subtraction (Brown and Burton, 1978).

It is possible, of course, that misconceptions are not always harmful to learning. Indeed, Hook *et al.* (1990) ask whether gaining (and losing) misconceptions actually plays a vital part in the learning process. Making use of Taylor's discourse model, they were able to group the subjects of a small empirical study into three classes: those who produced an account for Prolog's behaviour without straying from this mechanistic level; those who solved an impasse in their understanding of the mechanistic level via a temporary excursion to some other domain in which to reason; and those who abandoned an attempt to provide a mechanistic explanation in favour of some other discourse levels. Learners in the first group often seemed unaware when they had a misconception. In the other groups, there was some evidence that students used their 'misconceptions' to help them, but it is not clear how.

3.2 Reducing misconceptions by exhibiting behaviours

The fact that Prolog's execution-time behaviour is not 'on show' is clearly likely to create misconceptions about what it does. There have therefore been several attempts to construct tools to make Prolog's computational mechanisms more visible to novices. One approach has been to make data flow more visible (Dichev and du Boulay, 1988), but in general attention has been focused on the problem of exhibiting control flow.

Rajan (1986) has outlined a principled design for dynamic tracing environments and created the APT (animated program tracer) system. Among his principles are the following. The code seen in the tracer is to be a direct copy of the code prepared by the novice. (This is fiercer than it sounds: Prolog tracers generally replace variable names with internal representations, making the trace very hard to match up to the original code.) No extraneous symbols are to be inserted, and the code is to be evaluated in the debugging environment in exactly the same order that it would be evaluated in the normal environment. Side-effects are to be made visible, and novices are not to be compelled to learn new command sets and display formats, nor to understand unnecessary multiple viewpoints. In general, Rajan's principles are built round 'What You See Is What Happens', and a respect for the cognitive load experienced by the learner.

In a slightly different approach to the same end, Brayshaw and Eisenstadt (1989) have provided a fine-grained account of Prolog execution that is useful both for teaching and debugging, supported by a graphical tracer/debugger, TPM (the transparent Prolog machine), that allows the execution of a program to be displayed as an 'AORTA' structure (AND-OR tree, augmented). One of the most important aspects of their work is that their goal is a system that is simple enough to be useful to novices, yet powerful enough to be used by professional programmers working with big programs. This means that graphical techniques have to be designed with great care – otherwise they will run out of screen space. At the same time, they wanted to tell the user far more than is available from conventional Prolog tracers, which do little beyond reporting success, failure, and trying again to satisfy a goal. Part of the secret in the AORTA notation is to differentiate between no less than nineteen different types of behaviour by the Prolog interpreter – 'about to attempt new goal', 'system primitive', three types of success, six types of failure, and so on. Each of these has its own compact marking on the tree. Even so, screen space is at a premium. To cope with this, the output has an overall view, with detail suppressed, and a window that can be used to zoom in on any part of the trace that is of interest.

3.3 Supporting plan-level program construction

Evidence on 'programming plans' or schemas is presented elsewhere in this volume, showing that the knowledge of experienced programmers includes knowing sets of instructions to achieve certain familiar small tasks, such as forming a sum over a list. For an excellent account of the programming plans approach to learning programming, see Rist (1989). Although little is yet known about the repertoire of Prolog plans (see Brna *et al.*, 1990, for a review of progress), much can be achieved in this area by armchair consideration of plan structures. Gegg-Harrison (1990) has detailed an approach to teaching students how to build list processing programs from schemas. To have some idea of the complexity of the Prolog world, it is instructive to consider his classification: six simple schemas and eight complex schemas grouped into two different parallel hierarchies according to different viewpoints, with secondary dimensions as well, such as whether the program is a function or a predicate.

The would-be Prolog programmer evidently has a good deal to learn. Learning through a schema-based system may enable much better learning, or it may overwhelm the learner in fine distinctions which are not yet comprehensible at an early stage of learning. This will be an interesting area of development.

4 Programming as the control of real devices: DESMOND and HyperTechnic

All the languages mentioned so far, Logo, Prolog and Solo, enforce new and difficult abstract ideas, such as procedure, parameter and recursion. In response to the resulting problems one school of thought is to move away from the abstract towards the programming of concrete devices. This, indeed, was part of the original Logo platform, the physical turtle that crawled slowly around the floor and could be made to draw lines as it crawled; but Logo still has a high content of syntactic abstractions. The two projects we shall mention have very clearly tried to avoid syntactic load, and have equally clearly tried to supply a very concrete picture so that the user's mental model would be very likely to be highly accurate. Both projects use genuine, physical devices. Unfortunately we have no space to describe explorations in the direction of using highly realistic simulated worlds, such as the 'alternate reality kit' (Smith, 1986) or the 'interactive book' where physics experiments can be programmed by an author and reprogrammed by a user (separately proposed in the 'Boxer' language by diSessa and Abelson, 1986, and in 'Thinglab' by Borning, 1985).

The first project, Desmond, is only one step removed from a good electronics kit – 'Desmond' stands for 'digital electronic system made of nifty devices', and the project was developed at the Open University in association with the 'Microelectronics in Schools' initiative. The kit comprised a small microprocessor, a small touch-sensitive keypad, several sensors (for light, temperature, tilt, etc.) and several output devices (coloured LEDs, a buzzer, a small stepping motor, and a small LCD screen). The microprocessor was programmed in a very conventional single-address assembly code to read sensors and respond to them. Although the maximum program length was quite short, at about 100 steps, a good deal could be achieved. A simple example of a Desmond program fragment is:

```
LDA   91              load accumulator from address 91
ADI   01              add 1 to accumulator
STA   91              store accumulator in address 91
```

This small fragment adds 1 to the value held in location 91.

It can be seen that Desmond code is very much lower level than the other languages considered. There are no special abstractions, such as 'formal parameter', to be mastered; no unexpected syntactic complexities, such as Logo's use of colons and square brackets; no semantic complexities, such as unification; no hidden states of the machine – the entire machine state can be inspected via the LCD screen, which shows the contents of any address in binary, denary and instruction code format. Every instruction has a clear and simple meaning which is unambiguously described in concrete terms of changes to the machine state. How could it fail?

Jones (1989) studied a group of adults learning to use Desmond. They reported many different types of difficulty, some with the teaching materials and some with the device (e.g. expectations that memory-mapped devices, such as the lights, would have the least-significant bit at the left-hand end, whereas it was actually at the right)

Outstanding among their problems were confusions between contents and location – especially in the context of indirect addressing – and difficulties in comprehending the behaviour of a computer address, as something containing a value that is readily overwritten but less-easily put aside. This problem, of course, has afflicted many generations of computer novices. Evidently semantic complexities have *not* been banished, after all!

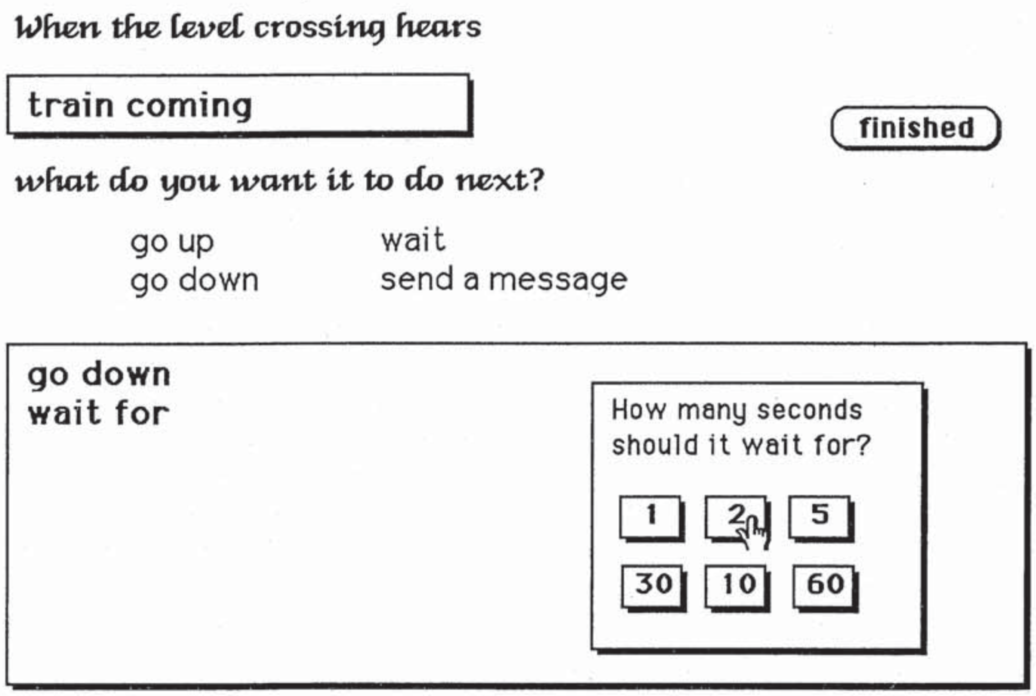
Jones's studies also pointed to another type of problem: users could see no plans. The role of 'programming plans' in novice learning has been mentioned above; here, the importance of comprehending programs at the functional level of goals and plans was again demonstrated, but this time at the lowest possible language level. Her analysis picks out a number of plans used *by implication* in the teaching materials. The fragment above adds 1 to an address. Other plans that she identified as being used by implication, but not explicitly described, included plans to operate one of the on-board devices, using one device (e.g. switches) to control another (e.g. lamps), adding two numbers, delay loop, test for non-zero numbers (this gave the learners great trouble), bit masking, etc. Readers familiar with assembly code working will recognize many of these from their own experience. But these higher-level plans were not taught as such, and of course the Desmond device contained no tools to identify plans. Learners were left to discover them for themselves, which in some cases was far from easy. Failing to do so meant that program construction became very arduous.

Finally, the Desmond work also points up the importance of visibility and 'viscosity' in programming. The LCD screen gave information about only *one* program location at a time: minimal visibility, making it hard to gain an overview of the program. Similarly, the editor allowed one program location to be changed at a time, and the means for doing so somewhat long-winded: finding the location to be changed and making the change could require many keystrokes. Thus program modification was labour intensive. In the terms used by Green (Chapter 2.2), the poor visibility and high viscosity did not make opportunistic programming possible. Desmond was designed for the 'errorless transcription' view of programming.

As technology improves, so do the opportunities for solving the problems of viscosity and visibility. In an interesting new approach, the resources of HyperCard on the Apple Macintosh are being exploited. Whalley (1990) has adapted the Lego building kits for children, which can be used to make up concrete microworlds such as train lines with level crossings. In his HyperTechnic system, described as 'a graphic object-orientated control environment', a sensor can detect the approach of a train towards the level crossing. The children can tell the sensor what to do when it hears a train coming, such as send a message 'Train coming' to the level crossing. The children can also tell the level crossing what to do when it hears the message 'Train coming': instructions available include go up, go down, wait (which leads to a prompt asking how many seconds), and send a message (see Figure 4).

Whalley reports that 'Trials have shown that children as young as seven can learn to use the train system, and are then able to teach others. An interesting, and as yet unresolved issue, is whether this success is due to the intuitively accessible graphic interface, or to the easy comprehension of the underlying actor language'.

Work of this sort is well ahead of what can be put into the average school. (The trains system requires an Apple Macintosh *and* an Apple II; and even so the microworld is still very limited.) But as a foretaste of the future possibilities, it is startling to see how responsive a system can be built. And yet, one has to apply the


When the level crossing hears

train coming

finished

what do you want it to do next?

go up
go down
wait
send a message

go down
wait for

How many seconds should it wait for?

Figure 4: The HyperTechnic programming interface (redrawn from Whalley, 1990).

same criteria. Are global structures visible? No. As HyperTechnic programs grow they will require some better system to make visible the structures of the program Moreover, although programming any individual device has been made unbelievabl easy, reprogramming the whole structure will remain difficult. But perhaps that i what learning to program is partly about: learning to manage increasing amounts o complexity in a partly abstract system, anticipating as many future contingencies a possible.

5 Matching 'natural plans': BridgeTalk

In this section we present the latest reported version of 'Bridge' and its successo 'BridgeTalk', developed by Bonar and associates (Bonar *et al.*, 1987; Bonar an Liffick, 1990). First we describe the rationale behind their novel and possibly ver fruitful approach.

Bonar's thinking starts from programming plans, which have been introduce elsewhere in this volume. Programming plans are exactly what novice programmer lack, according to some authors, and therefore what they need to learn. It is no that novice programmers – even children – are planless; they have perfectly goo informal plans for counting sheep, adding up totals, etc. But they do not kno how to translate those into programming languages. Conventional programmin languages hide programming plans by dispersing them throughout the text, and so novices, instead of learning how to move from informal plans to formalized plans, instead spend time on purely surface features. 'In our video protocols of novice programmers, we see novices working linearly through a program, choosing each statement based on *syntactic features* of the problem text or program code' (Bonar and Liffick, 1990, p. 330, our italics).

On the other hand, Bonar and Liffick argue, the target of teaching programming should be the ability to understand and use a conventional programming language. So we should not seek to evade this responsibility by teaching a simplified programming language which merely postpones the problem of getting to grips with real programming. Instead, the solution is to use an *intermediate representation*, one which minimizes initial difficulty but which leads into real programming. What should that intermediate representation be?

BridgeTalk boldly sets out to encourage *plan-level programming in Pascal*. It is intended to encourage novices to recognize how their informal plans fit into a programming environment, to support them in learning a vocabulary of programming plans, to teach them how to implement those plans in a standard programming language, and to support plan-like composition of programs.

The version of BridgeTalk that we shall describe is no less than the sixth generation. (Readers interested in seeing how the ideas developed should consult Bonar and Liffick, 1990.) Each particular plan has its own icon, which fit together like a jigsaw. Slots in the icons can hold smaller icons for values and constants. Figure 5 shows a program for finding the average of a data set using plan icons; the flow of control is handled by an icon for a loop 'that repeatedly gets a new value until that new value equals a specified sentinel value', a new-value controlled loop plan. 'The key idea with [this plan] is to hide all the syntactic and control flow complexity that a student would need to confront [in order to] implement such a loop in a standard language' (Bonar and Liffick, 1990, p. 338).

One of the most attractive features of BridgeTalk is that it uses graphical representations and mouse dragging to provide a radically new information display, in marked contrast to many ventures into graphics programming (see Chapter 1.2). This representation not only makes both data flow and control flow information available in a well-balanced way, it *also* appears to present plan-level information in a usable manner. Moreover, the environment allows program development to take place in a natural, unforced way. Whereas many tutorial environments use a syntax-based editor to impose a rigid, top-down development on the novice, in order to teach him or her how to 'think properly' (see, for instance, the 'Struedi' Lisp editor for novices described in Chapter 1.2), BridgeTalk eschews such authoritarian pedagogy and allows free construction:

> We have become increasingly suspicious of the glib discussion of 'top-down design' found in most programming textbooks. In particular, the design of BridgeTalk shows that programming design involves many different mappings, including informal to formal, declarative to procedural, goals to plans and processes, natural language to Pascal, linear structure to tree structure, and weakly constrained to strongly constrained. We believe that programming texts do their students a disservice by presenting a design model that at best ignores the differences between novices and experts and at worst is completely unrelated to actual programming practice. (Bonar and Liffick, 1990, p. 363).

We agree.

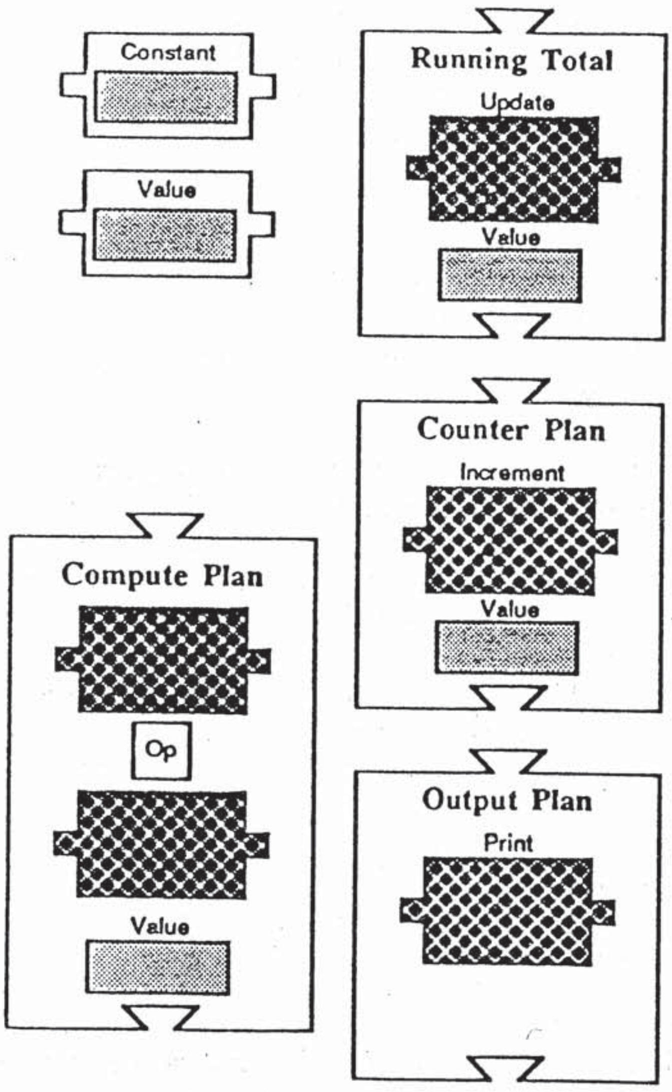


Figure 5: BridgeTalk icons that express a sequence of values, compute new values, and operate on values.

6 Conclusions

This chapter on how to make computing more accessible for educational purposes started with a view of programming as the 'great white hope' of education. Klahr and Carver (1988, p. 363) speak of 'the possibility that it [computer programming] may be the long-sought mental exercise that enables its practitioners to increase their general thinking abilities'. Perhaps there really is such a mental exercise, and just possibly programming is it. But neither anecdotal nor laboratory evidence gives strong support to this hope.

A lot of the learner's time is spent learning the repertoire of programming tricks and techniques, and so our next topic was the comparative lack of success of Prolog, a language that seemingly promised to free the learner of much of that baggage. Experience now suggests that while removing the syntactic apparatus of control flow and data declaration is perhaps a benefit, it also takes away the information about how the program works and thereby mystifies the learner. Contemporary research in Prolog is therefore aiming to create environments that will supply the information about execution which comes freely available with traditional languages.

Also, we glanced briefly at two very different projects which both aim to provide a more concrete learning world for programming, thereby overcoming the difficulties of abstraction and high-level conceptualization in other programming environments. Research on the first of these two, the Desmond system, has shown that despite the concreteness, learners still have problems, and that many of their problems are caused by inability to discover the high-level plans and how to put them together.

The last topic, therefore, was a plan-based system, which consciously attempted to help learners understand the target language, Pascal, in terms of 'everyday plans', via a bridging representation, BridgeTalk. It is far too early to evaluate this system, but the goal is impressive, and so is the care in designing the interface for maximum visibility of important components and for supporting opportunistic planning.

So, where does all this leave us? We present our conclusions in terms of slogans.

Forget about the transfer of competence dispute ...

Disputes about educational objectives are unnecessary and (in the current state of language design) premature. We contrasted 'learning to program' and 'programming to learn'; within the 'learning to program' choice, a programming language is above all a new language to be learnt. Eventually it can serve in the teaching of conventional curricula. Within the alternative choice, 'programming to learn', a language like Logo acts as a medium for the practising of specific skills, such as geometric planning; and it can also be a source of self-knowledge, a mirror for watching oneself solve problems. A language like Prolog can also successfully be used as a 'knowledge bank' in teaching history or ecology – topics where many detailed facts need to be available (e.g. Rasmussen, 1988).

... and go for programming in context.

Alternatively, programming can be used as a step in understanding the world that is coming into being around us, the world of computer applications and computer methods. Soon we shall be living in a world where knowing how to choose, learn and use a computer application will be of the first importance – a world where many of the tasks will be either quite new, or else greatly affected by the existence of computer systems. By programming even the most elementary versions of some of these applications, whether a word processor, a database management system, or a circuit design package, an inside view will have been obtained.

Make it easy to comprehend ...

Not many educational objectives can be accomplished if would-be learners are frustrated by unnecessary difficulties with the programming language and environment. It seems to the authors that far too much attention has been given to the debate on transfer of competence, and far too little to attacking the problems of ease of learning and ease of use.

Back in 1981, du Boulay *et al.* described the difficulties in presenting computing concepts to novices. They introduce the concept of the 'notional machine', '... an idealized, conceptual computer whose properties are implied by the constructs in the programming language employed'. This notional machine should be kept *functionally* simple by giving it a small set of what they call 'transactions' (after Mayer, 1979); it should be kept *logically* simple, so that problems of interest to the novice can be tackled by short, simple programs; and it should be *syntactically* simple, i.e. the rules for writing instructions should be uniform and have well-chosen names. But simplicity is not enough. The notional machine must also be *visible in its own terms*. This is the crux of the problem in many cases – novices cannot see what the notional machine is doing.

Today it is clear that comprehensibility requires more than simplicity and visibility. Jones (1990), in her study of Desmond, Logo, Solo, and a microprocessor language not described here, showed that even in systems meeting those criteria, problems still arose. She demonstrated the existence of unexpected semantic complexities and the need to help learners understand at the functional level, the level of plans and overviews.

... and easy to use ...

It must be obvious to anybody that systems for novices should be easy to use. Yet novices are still struggling with inadequate systems! Too much typing, with no spelling correction; too much syntax, with poor error messages and no error correction or 'Do What I Mean'; poor visibility of different parts of the program, and poor display of execution behaviour – these are still typical. Perhaps one reason has been the habit of treating the programming language and its operating environment in very separate ways. Basic, for all its faults, supplied a seamless environment for learners; one wishes every innovator had done the same. DiSessa and Abelson (1986) observe that 'User interfaces are often considered to be separable from programming language semantics and almost an afterthought in language design. Worse, most present languages assume only character-stream input and output. A useful medium must be much more flexibly interactive'.

This persistent separation of language and environment reflects slow acceptance of the ideas of opportunistic planning and exploratory programming. Exploratory programming requires a suitable support environment, of course. Chapter 2.2 ('Programming languages as information structures') shows how recently this idea has displaced previous views, notably the view of programming as 'errorless transcription'.

... and avoid the dogmas of economy.

Finally, it must be remembered at all times that the users are *learners*. Their needs, their knowledge and their abilities are different from those of experienced programmers, who may well have been self-selected for some rather special attributes. Bonar and Liffick (1990) say in some detail why they think current programming languages are unsuitable for novices. They point out that their 'emphasis in economy of expression ... is misplaced in designing languages for novice programmers. By looking for ever more abstracted ways to express similar procedural behavior, modern languages have excised most clues to goal and purpose that are essential to novice understanding of a program'. They suggest that students would prefer a language in which each plan was expressed by a different construct. Conversely, modern languages emphasize tools for abstraction, which novices are not yet ready to use.

Once again, we agree with Bonar and Liffick. We look forward to seeing the next generation of languages and environments for easy access by learners.

References

Anzai, Y. and Uesato, Y. (1982). Learning recursive procedures by middleschool children. *Proceedings of the Fourth Annual Conference of the Cognitive Science Society.* Ann Arbor.

Borning, A. (1985). A prototype electronic encyclopaedia. *ÈACM Transactions on Office Information Systems,* **3**, 63-88.

Bonar, J. and Liffick, B. W. (1990). A visual programming language for novices. *In* S.-K. Chang (Ed.), *Principles of Visual Programming Systems.* Englewood Cliffs: Prentice-Hall.

Bonar, J., Cunningham, R., Beatty, P. and Riggs, P. (1987). Bridge: intelligent tutoring with intermediate representations. Technical Report, Learning Research and Development Center, University of Pittsburgh.

Bourbion, M. (1986). *Le choix Logo.* Paris: Armand Colin Editeur.

Brayshaw, M. and Eisenstadt, M. (1989). A practical tracer for Prolog. Technical Report no 42, Human Cognition Research Laboratory, Open University, Milton Keynes. *International Journal of Man-Machine Studies*, in press.

Brown, J.S. and Burton, R. (1978). Diagnostic models for procedural bugs in basic mathematical skills. *Cognitive Science,* **2**, 155-192.

Bundy, A. (1984). What stories should we tell Prolog students? Working Paper 156, Department of Artificial Intelligence, Edinburgh.

Coombs, M. J. and Alty, J. (1984). Expert systems: an alternative paradigm. *In* M. J. Coombs (Ed.), *Developments in Expert Systems.* London: Academic Press.

Crahay, M. (1987). Logo, un environnement propice à la pensée procédurale. *Revue Française de Pédagogie,* **80**, 37-56.

Dichev, C. and du Boulay, B. (1988). A data tracing system for Prolog novices. *In* T. O'Shea and V. Sgurev (Eds), *Artificial Intelligence III: Methodology, Systems, Applications.* Amsterdam: North-Holland.

diSessa, A. A. and Abelson, H. (1986). Boxer: a reconstructible computational medium. *Communications of the ACM,* **29**, 859-868.

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

Eisenstadt, M. (1979). A friendly software environment for psychology students. *AISB Quarterly,* **34**.

Eisenstadt, M. and Lewis, M. (1985). Errors in an interactive programming environment: causes and cures. Technical Report No. 4, Human Cognition Research Laboratory, The Open University, Milron Keynes.

Fung, P., du Boulay, B. and Elsom-Cook, M. (1987). An initial taxonomy of novices' misconceptions of the Prolog interpreter. CITE Report 27, Centre for Information Technology in Education, Institute for Educational Technology, The Open University.

Gegg-Harrison, T. S. (1990). Learning Prolog in a schema-based environment. *Instructional Science,* in press.

Hofstadter, D. (1979). *Gödel, Escher, Bach: an Eternal Golden Braid.* New York: Basic Books.

Hook, K., Taylor, J. and du Boulay, B. (1990). Redo 'try once and pass': the influence of complexity and graphical notation on novices' understanding of Prolog. *Instructional Science,* in press.

Howe, J.A.M. and O'Shea, T. (1978). Computational metaphors for children. *In* F. Klix (Ed.), *Human and Artificial Intelligence.* Berlin: Deutscher Verlag.

Hoyles, C. and Noss R. (1987). Synthesising mathematical conceptions and their formalisation through the construction of a Logo based school mathematics curriculum. *International Journal of Mathematics Education in Science and Technology,* **18.**

Jones, A. (1989). Empirical studies of novices learning programming. Ph.D. thesis, Institute of Educational Technology, The Open University, Milton Keynes.

Kahney, H. (1983). Problem solving by novice programmers. *In* T.R.G. Green, S.J. Payne and G.C. van der Veer (Eds), *The Psychology of Computer Use.* London: Academic Press. Reprinted in E. Soloway, and J. C. Spohrer, (Eds), *Studying the Novice Programmer.* Hillsdale, NJ: Erlbaum.

Kessler, C.M. and Anderson, J.R. (1986). Learning flow of control: recursive and iterative procedures. *Human Computer Interaction,* **2,** 135-166.

Klahr, D. and Carver, S. M. (1988). Cognitive objectives in a Logo debugging curriculum: instruction, learning, and transfer. *Cognitive Psychology,* **20,** 362-404.

Kurland, D.M., Pea, R.D., Clement, C. and Mawby, R. (1989). A study of the development of programming ability and thinking skills in high school students. *In* E. Soloway and J.C. Spohrer (Eds), *Studying the Novice Programmer.* Hillsdale, NJ: Erlbaum.

Lawler, R.W. (1985). *Computer Experience and Cognitive Development: A Child's Learning in a Computer Culture.* Chichester: Ellis Horwood.

Lehrer, R., Guckenberg, T. and Sancilio, L. (1988). Influences of Logo on children's intellectual development. *In* R.E. Mayer (Ed.), *Teaching and Learning Computer Programming.* Hillsdale, NJ: Erlbaum.

Littlefield, J., Delclos, V.R., Lever, S., Clayton, K.N., Brandsford, J.D. and Franks J.J. (1988). Learning Logo: method of teaching, transfer of general skills, and attitudes toward school and computers. *In* R.E. Mayer (Ed.), *Teaching and Learning Computer Programming.* Hillsdale, NJ: Erlbaum.

Mayer, R.E. (1979). A psychology of learning Basic. *Communications of the ACM,* **22**, 589-593.

Mendelsohn, P. (1985). Learning recursive procedures through Logo programming. *Proceedings of the Second Logo and Mathematics Education Conference.* University of London.

Mendelsohn, P. (1988). Les activités de programmation chez l'enfant: le point de vue de la psychologie cognitive. *Technique et Science Informatiques,* **7**, 47-58.

Nichol, J., Briggs, J. and Dean, J. (Eds) (1988). *Prolog, Children and Students.* London: Kogan Page.

Ormerod, T. C., Manktelow, K. I., Robson, E. H. and Steward, A. P. (1986). Content and representation effects in reasoning tasks in Prolog form. *Behaviour and Information Technology,* **5**, 157-168.

O'Shea, T. and Self, J. (1983). *Learning and Teaching with Computers: Artificial Intelligence in Education.* Brighton: The Harvester Press.

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.

Papert, S. (1980). *Mindstorms: Children, Computers and Powerful Ideas.* New York: Basic Books.

Pea, R.D. and Kurland, D.M. (1984). On the cognitive effects of learning computer programming. *New Ideas in Psychology,* **2**, 137-168.

Pirolli, P. (1986). A cognitive model and computer tutor for programming recursion. *Human-Computer Interaction,* **2**, 319-355.

Putnam, R.T., Sleeman, D., Baxter, J.A. and Kuspa, L.K. (1986). A summary of misconceptions of high school Basic programmers. *Journal of Educational Computing Research,* **2**, 57-73.

Rajan, T. (1986). APT: A principled design for an animated view of program execution for novice programmers. Technical Report 19, Human Cognition Research Laboratory, The Open University.

Rasmussen, J. (1988). Using Prolog in the teaching of ecology. *In* J. Nichol, J. Briggs and J. Dean (Eds), *Prolog, Children and Students.* London: Kogan Page.

Rist, R. (1989). Schema creation in programming. *Cognitive Science,* **13**, 389-414.

Roberts, E.S. (1986). *Thinking Recursively.* New York: Wiley.

Rogalski, J. (1987). Acquisition et didactique des structures conditionnelles en programmation informatique. *Psychologie Française,* **32**, 275-280.

Rogalski, J. and Vergnaud, G. (1987). Didactique de l'informatique et acquisitions cognitives en programmation. *Psychologie Française,* **32**, 267-274.

Ross, P. and Howe, J. (1981). Teaching mathematics through programming: ten years on. *In* R. Lewis and D. Tagg (Eds), *Computers in Education.* Amsterdam: North-Holland.

Rouchier, A. (1987). L'écriture et l'interprétation de procédures récursives en Logo. *Psychologie Française,* **32**, 281-285.

Smith, R. (1986). The alternate reality kit: an animated environment for creating interactive simulations. *Proceedings of the 1986 IEEE Computer Society Workshop on Visual Languages, Dallas.* IEEE.

Solomon, C. (1986). *Computer Environments for Children: A Reflection on Theories of Learning and Education.* MIT Press.

Soloway, E., Bonar, J. and Ehrlich, K. (1983). Cognitive strategies and looping constructs: an empirical study. *Communications of the ACM,* **26**, 853-860.

Suppes, P. (1979). Current trends in computer-assisted instruction. *In* M.C. Yovits (Ed.), *Advances in Computers,* vol. 18, New York: Academic Press.

Taylor, J. (1987). Programming in Prolog: an in-depth study of problems for beginners learning to program in Prolog. Unpublished Ph.D. thesis, School of Cognitive Studies, University of Sussex.

van Someren, M. W. (1985). Beginners' problems in learning Prolog. Memorandum 54, Department of Experimental Psychology, University of Amsterdam.

van Someren, M. W. (1988). What's wrong? Understanding beginners' problems with Prolog. VF Memo 89, Department of Social Science Informatics, University of Amsterdam.

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

Vitale, B. (1987). Epistemology and pedagogy of children's approach to informatics. *Proceedings of the International Conference on Education.* Bilbao.

Whalley, P. (1990). HyperTechnic – a graphic object-orientated control language. *Proceedings of the 7th Conference on Technology and Education.* Brussels, 1990.

White, R.H. (1988). Effects of Pascal knowledge on novice Prolog programmers. Research Paper 399, Department of Artificial Intelligence, Edinburgh.