Chapter 2.3

Language Semantics, Mental Models and Analogy

Jean-Michel Hoc and Anh Nguyen-Xuan

CNRS - Université de Paris 8, URA 1297: Psychologie Cognitive du Traitement de l'Information Symbolique, 2, Rue de la Liberté, F-93526 Saint-Denis Cedex 2, France

Abstract

The semantics of a number of programming languages is related to the operation of a computer device. Learning a programming language is considered here from the point of view of learning the operating rules of the processing device that underlies the language, as a complement to the learning of new notations, or a new means of expression to be compared to natural language. This acquisition leads beginners to elaborate a new representation and processing system (RPS) by analogy with other systems that are associated to well-known devices. During acquisition, beginners not only learn new basic operations but also the constraints of these operations upon program structures. Learning therefore concerns a basic problem space as well as abstract problem spaces within which planning takes place. The links between this approach to learning to program and a number of related works on learning to use software are underlined. Implications of these research findings in the programmer training are drawn.

Psychology of Programming
ISBN 0-12-350772-3

1 Introduction

At first glance, programming may be defined as a procedure specification task by means of a computer language. This conception of programming, as pointed out by Miller (1974), led a number of researchers to stress the importance of the acquisition of the means – the computer language – in learning to program. Green offers an overview of this perspective in this book with his phrase 'dimensions of notations' (Chapter 2.2). Such a view is also prevalent in the research done by the Soloway team on Pascal (e.g. Bonar and Soloway, 1985). These researchers have established a taxonomy of errors done by novices in situations where they have been able to collect verbal protocols during program design. Their studies reveal the fact that many novice errors are due to wrong transfers of natural language constructs to computer programs (e.g. 'then' interpreted as 'afterwards' instead of 'in these conditions'). Confusion between Prolog notations of logic expressions and natural language or other formalisms are also shown to be an important factor of novices difficulties in learning Prolog (Taylor and du Boulay, 1986).

In teaching programming, the emphasis stressed on the means of expression can lead to an overestimation of the learning of programming language syntax within the triad determining human-computer interaction: external task structure, language syntax, and language semantics (Moran, 1981). Certainly a number of difficulties for beginners can be drastically reduced by designing ergonomic programming language syntax. For example, some interesting properties of a two-level syntax have been demonstrated (task-action grammar: Payne and Green, 1986). This enables the user to learn a limited number of general rules that play the role of schemas and which can be instantiated so that several specific rules can be generated. Nevertheless these syntactic schemas have their semantic counterpart and the efficiency of this kind of syntax is probably determined by a certain compatibility with a pre-existing two-level structuration of the contents of the language.

Syntactic errors have been shown to be of limited importance even to beginners (Youngs, 1974). In a number of studies on learning to program, which stress the acquisition of a new communication means, semantic rather than syntactic difficulties are shown. This is especially true of Soloway's works which show that beginners introduce distortions into programming language syntax when their programming knowledge is lacking. These distortions are indicators of transfers from other knowledge domains that are not compatible with the programming language structure. Even with professional programmers, this kind of transfer, from a well-known programming language to a new one, has been shown to exist (Hoc, 1988b). Syntactic errors clearly revealed semantic difficulties when programmers did not succeed in transforming well-known contents into quite different contents expressible in the new language. This phenomenon often occurs in translation between natural languages. Taylor and du Boulay (1987) show the Prolog experts' ability to adopt a problem representation compatible with the language in the very beginning of the design process. Within the same study, programmers who are expert in programming but unfamiliar with Prolog initiate their design activity with representations that are not compatible with Prolog and very often fail to produce a program.

In this chapter, the relationship between task structure and programming language semantics will be discussed as a critical component of learning to program. Semantics and syntax are considered to be complementary components in the study

of user models. Although a major effort has been made to bridge the gap between natural language and programming languages (mainly for English-speaking programmers) the semantic problem remains. As Jackson (1980) has pointed out, it is almost impossible to design languages that are purely problem oriented. They remain largely machine oriented and their semantics consist of controlling machine operation (especially a sequential mode of operating in performing tasks where human operators may use parallel processing). Indeed, this machine is a formal one and varies with the language used: e.g. the Pascal machine is different from the Cobol machine. Research is only beginning on more 'declarative' languages such as Prolog but the need to learn the operating rules of the Prolog machine is already shown to be a necessary condition to designing complex programs in this kind of language (Taylor and du Boulay, 1987).

The purpose of this chapter is to show that whatever the programming language, beginners have to learn the operating rules of what is called the 'device' underlying the programming language. This learning develops along two directions:

(1) construction of new elementary operations (primitives), different from familiar ones, which are sometimes coded by the same wordings (especially the READ and WRITE statements, as has very often been shown);

(2) restructuring of well-known plans which are incompatible with the new primitives – beginners become aware of the constraints upon the structure of the new plans defined by these new primitives.

Beginners learn to program by building programs. Hence they learn by problem-solving. The important features of this problem solving situation will be presented here, giving reasons for the theoretical framework used in analysing the elaboration of user models of language and devices: learning by doing and by analogy. A basic construct will then be introduced (after Hoc, 1977, 1988a), in relation to mental models, to define the knowledge architecture within which these learning mechanisms take place: the representation and processing system (RPS). General problem solving by analogy strategies will be discussed in the context of human computer interaction studies aimed at describing the beginner's acquisition of computerized tasks. And finally, some implications on training will be stressed.

Mainly research on the learning of procedural languages will be referred to, since it is available in books or journals. Similar components can be found in the learning of more declarative and recent languages as shown by studies that have not been widely published.

2 Problem solving by beginners in programming

Whatever the kind of learner and whatever the teaching method may be (e.g. top-down structured programming methods, or algorithmics), programming language acquisition remains the first necessary step to more advanced acquisitions.

As has been stressed above, learning to program is learning by problem solving. A number of studies have been devoted to beginner problem solving strategies in these learning-by-doing situations and it is now possible to define their principal features. After Hoc (1988a), the notion of problem is to be contrasted against the notion of task. A task is defined by a goal and conditions for reaching it. A problem

is a representation of a task a subject evokes or elaborates that cannot yet trigger an acceptable procedure to reach the goal (i.e. a procedure which is in conformity with the prescribed conditions).

Beginners are confronted with computer tasks, like stock updating, file sorting, etc., for which they have no available procedures. These tasks cannot be considered as problems for professional programmers who can activate well-known computer schemas ready to fulfil them (see Chapter 3.1). The notion of problem is therefore subjective: it is related to the interaction between subject and task and must be defined by both the task characteristics and the subject's knowledge. What then constitutes a programming problem for a beginner in contrast to an expert?

More often than not teachers do not ask beginners to invent algorithms. Beginners are required to produce programs for familiar tasks they can perform by hand. In other words the goals are familiar to them and they have procedures at their disposal so that these goals can be reached. These familiar tasks become problems because there are mainly two new conditions that must be satisfied:

(1) Programs have to be practicable by a computer through a definite means of communication: the programming language (this communication situation is very different from the human-human communication situations where interpretation takes place). Programmers thus have to restrict themselves to the use of elementary operations the computer can perform and of procedure structures compatible with these operations.

(2) New procedures have to be explicitly stated beforehand (in most of the cases new elaborations cannot take place at the execution time); hence a strong planning constraint is introduced.

In this chapter the first kind of condition will be discussed, the second one is examined elsewhere in this book (see Chapter 2.4). However, some short comments on the constraints on procedure expression must be given here. To be able to express procedures in a program, beginners must elaborate representations of these procedures, such representations may not be necessary in usual problem-solving situations. In ordinary situations the procedure can be elaborated at execution time, where concrete feedback is available after the execution of each operation: in programming, feedback is delayed. Usually the goal is specific (e.g. sorting of a particular file for which a particular procedure can be adopted) while in most of the programming situations the goal is a class of specific goals (e.g. the sorting procedure must be valid for any file defined by a set of characteristics). This creates two kinds of difficulty for beginners:

* shift from value to variable processing,

* elaboration of a representation of the procedure control structures of which beginners are not necessarily aware in usual problem solving situations.

In several experiments, we have adopted a methodology whereby the two aspects of problem solving in beginners – the design of a procedure practicable by the computer, and the expression of this procedure – are studied separately (Hoc, 1983; Nguyen-Xuan and Hoc, 1987). Beginners are first asked to perform a task by hand in

a situation with as few constraints as possible and the familiar procedure is observed. They are then asked to command a computer device to carry out the task step by step: the elementary commands available correspond to elementary instructions of a programming language. In this command situation all the data to be processed (e.g. an entire file) and the content of the computer memory cells are visible in a first stage and covered in a second stage. These two situations enable the experimenter to observe the adaptation of the familiar procedures to the operating rules of the device and force the beginners to adopt a general procedure (covered situation). Finally the beginners are asked to verbally state their general procedure after having elaborated it in the command situations where they had feedback information (step-by-step error messages in the covered situations). Figure 1 illustrates the application of this kind of methodology to the sorting of a list.

This methodology enables the observer to identify the diverse sources of difficulties that beginners encounter when designing a program in standard programming situations: acquisition of the operating rules of the new device and procedure expression.

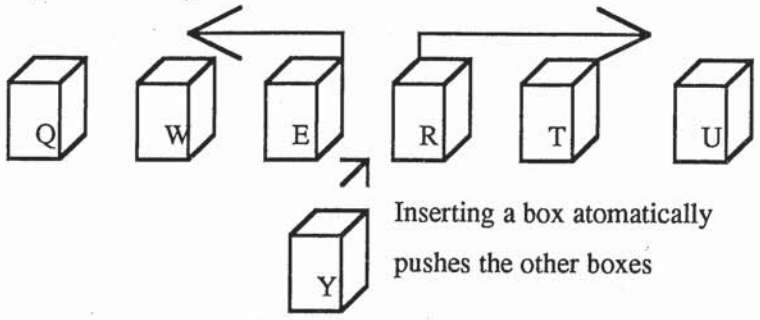
3 The concept of representation and processing system

In human-computer interaction as well as process control areas the notion of the mental model has been introduced to describe operator knowledge about machines (Rouse and Morris, 1986). Although the uses of this concept are various, the common idea is to describe a kind of operational knowledge that is specific to a limited class of situations. The application of mental models to operator behaviour raises methodological difficulties that will not be discussed here. In particular these mental models must be opposed to 'conceptualizations' as observer constructs (Norman, 1983). Although conceptualizations can only be defined by the observer as compatible with expert behaviour (instead of being actually used by the operator), explicit teaching of these conceptualizations in learning to use a device can provide the learner with a helpful representation of the device, and can be of interest (du Boulay *et al.*, 1981).

In a study of the learning of programming, Hoc (1977) has introduced the notion of the representation and processing system (RPS) which is similar to a 'mental model'. A RPS is a part of the semantic memory network that can be activated in executing tasks belonging to a common task domain (see Hoc, 1988a, for further developments). Such a network is elaborated by instruction and by doing, and has the double status of a social and individual construct. As different task domains are separately taught – algebra, French, physics, etc. – they are separately internalized in the form of different RPSs. Connections between pieces of knowledge belonging to the same RPS are stronger than connections between different RPSs.

In a RPS, declarative (representation) and procedural (processing) knowledge are strongly connected as dual aspects of knowledge. This accounts for the fact that formally isomorphic tasks may not be processed in the same way, because they are represented differently by the subject. A particular kind of task representation opens access to certain procedural skills that cannot be so easily triggered by another representation. It may happen that the same problem statement triggers different RPSs for solving various sub-problems, and the communication between different RPSs may be difficult to manage. In an experiment conducted by Hoc (1977), subjects at different levels of programming expertise had to solve a problem of ticket

Step a: Sorting of boxes



Step b: sorting the row of an array (decreasing order)

step b1: visible device

1	2	3	4	5	6	7
Q	W	E	R	T	Y	U

Command keys:

COMPARE
COPY
DELETE

1	2	3
4	5	6
7	8	9

Syntax:

COMPARE CELL(A,B) WITH CELL(C,D) Result: e.g.: CELL(C,D)>CELL(A,B)

COPY CELL(A,B) IN CELL(C,D) Result: as an ordinary assignment

DELETE CELL(A,B) Result: CELL(A,B) empty

step b2: covered device

While processing, the content of the cells are covered.

	1	2	3	4	5	6	7
1							
2							

3 extra commands are available:

ANYTHING IN CELL...?
DISPLACE '*' ON CELL...
DISPLACE '^' ON CELL...

In trying to transfer the box insertion procedure,

a shifting sub-procedure has to be applied before

inserting a letter.

Figure 1: Sorting of a list (after Nguyen-Xuan and Hoc, 1987). (a) Familiar situation: sorting of boxes. (b) Sorting of a row in an array: (b1) visible contents, (b2) covered contents.

machine control simulation in a Metro station. In solving this kind of problem, a number of difficulties encountered by subjects were representation and processing translations from four separate RPSs into a common 'computer' RPS:

* 'traveller', in which traveller goals (e.g. ticket asked) and actions (coins inserted) were represented;
* 'ticket machine', describing the machine reactions to the traveller and computer messages;
* 'numerical code', the transfer of information and commands between the computer and the ticket-machine;
* 'accounting', the knowledge domain in which the relation between information and commands took its meaning (e.g. computation of the change to be returned).

Given the multiplicity of task domains that have been internalized by an individual, a number of RPSs are available and are triggered by problem statements or perceptual cues. These knowledge structures play a central role in learning new domains and dealing with novel tasks. They are the source of problem solving by analogy; for example in learning computerized tasks – triggering of a RPS related to the typewriter domain in learning to use a text editor (Wærn, 1989), or to the calculator in learning to use a turtle command language (Shrager and Klahr, 1986: calculator 'view application').

RPS can be defined at several levels of abstraction; from a basic definition of elementary operations (e.g. the operation meant by a read statement), and data properties or relations (e.g. an integer variable), to abstract operations (e.g. iteration to go through a file) and representations (e.g. file structure). This hierarchy enables individuals to plan their actions (see Hoc, 1988a, for further developments about this hierarchy of abstract spaces). The basic definition is called the (basic) device associated to the RPS, the precision of which depends upon the individual's expertise and goals. For example the Pascal machine can be considered as the basic device in designing programs, but a more elementary device would be considered if compiler or execution error messages had to be understood.

Research into use of computer devices offers a comparison between two types of knowledge that are embedded in the same RPS (Hoc, 1978; Young, 1981; Richard, 1983):

* operating rules that describe the conditions of validity of the operations and their effects, and can only be 'surrogates' (Young, 1981) without causal semantics or based on deeper knowledge;
* utilization rules that describe operations to be used in relation to goals, and make task-action mapping easier.

Each of these types of knowledge can be defined at different levels of abstraction. Operating rules can not only refer to very elementary operations of the device but also to macro-operations. Utilization rules may reflect different levels of analysis of situations, as has been shown in different contexts (Card *et al.*, 1983; Wærn, 1989; Richard, 1986):

* the goal (or task) level describing the intentional or functional aspects of the activity;
* the means (or method) level at which procedural aspects of goal attainment are processed;
* and the prerequisite (or condition) level giving access to details such as those that are necessary to process interactions between elementary operations.

As far as operating rules are concerned, it has been shown that the levels of abstraction correspond to processing priorities, from the analysis of goals down to the analysis of prerequisites (Richard, 1986; Morais and Visser, 1987). So, if a goal representation can trigger a familiar procedure, this procedure is applied to a novel device before detecting mismatches which lead to a deeper analysis of the situation.

4 Problem solving by analogy in programming

4.1 Rationale for an analogical transfer

We argue above that people learn to program through practice, i.e. they learn by problem solving. But unlike learning a knowledge domain (e.g. physics), the learner knows a procedure in a RPS associated to a well known device (e.g. a box-sorting procedure by hand, although the task has to be accomplished by a computer with an array: see Figure 1). That is, a means is available for reaching an analogue goal state from an analogue initial state. The problem consists in making the computer reach the goal.

We suggest that in such a situation the learner is prone to relying on a familiar RPS to build the goal structure needed, instead of elaborating a new goal structure compatible with the computer from scratch. In other words, the learner will consider that the computer solution is analogous to the familiar one. But goal structures available within this RPS are compatible with the device associated to the RPS without being necessarily compatible with the device underlying the programming language. So some more-or-less profound adaptation will take place before reaching an acceptable program.

4.2 Empirical evidence

Hoc (1977) has shown that, in learning procedural programming, beginners who are dealing with tasks they can execute by hand may evoke procedural plans that are available in familiar RPS. They then try to refine these plans until reaching the level of the programming language statements: in most cases, this refinement leads either to a detection of incompatibilities between the plans and the device, or to reach programs which are not optimal. This phenomenon is reinforced in training beginners to use top-down programming methodology before learning the operating rules of the computer device (Hoc, 1983). The incompatibilities between these plans and the device involve mainly sequential data access and result production modes different from those used by the human cognitive system which is capable of parallel processing.

Most of the beginner programming errors can be interpreted as wrong analogical transfers from other RPS to use of the computer device (Bonar and Soloway, 1985).

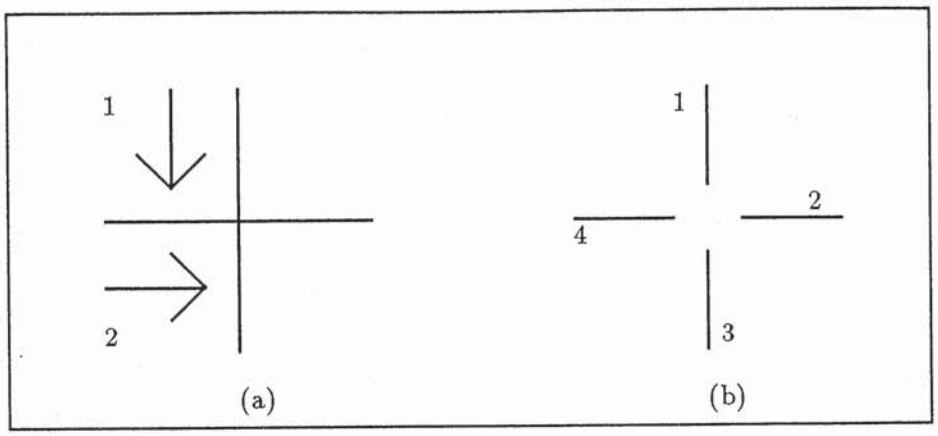


Figure 2: The two representations of the cross (after Mendelsohn, 1986): (a) two orthogonal lines (before LOGO learning), (b) four orthogonal lines (after LOGO learning).

Moreover, in executing a task where several methods are available, the program is designed from the method most frequently executed by hand. This has been shown in an experiment on sorting: beginners tried at first to adapt the sorting-by-insertion method they readily used by hand, even if it was more difficult to program than the extremum method (Nguyen-Xuan and Hoc, 1987).

In learning a new programming language, the RPS elaborated to program in an already familiar programming language can be used as an analogue. Van Someren (1984) points out that when starting to program in Prolog beginners who already know an instruction-oriented language just try to implement an algorithm and then translate it into Prolog rules.

Indeed, analogical transfer is a paramount phenomenon. Beginners have been shown to borrow plans from their execution-by-hand RPS. But experienced programmers hand-manipulate files in a different way from novices: they use the plans they would have used in a computer program (Eisenstadt *et al.*, 1985). This 'reverse' analogical transfer may concern the structure of the representation of well-known objects which has been modified in order to be adapted to computer programming. Mendelsohn (1986) has found such transformations in elementary school pupils who learn Logo. For example, the cross which is genuinely seen as composed of two orthogonal lines is seen after the Logo course as composed of four orthogonal lines (Figure 2).

The importance of analogical transfers at the outset of new device learning has been stressed by a number of authors in very different contexts: text editing (Wærn, 1985; Allwood and Eliasson, 1987), pocket calculator use (Bayman and Mayer, 1984; Friemel and Richard, 1987). When analogical transfer is impossible from well-known RPSs this transfer may come from previously acquired constructs in the same language as has been shown in learning Lisp (Anderson *et al.*, 1984) or in operating a robot (Klahr and Dunbar, 1988).

4.3 Effects of analogy

More often than not authors stress the negative effects of the analogical transfer when trying to interpret novice errors. However, two kinds of analogical transfer

must be distinguished: a transfer through an abstract schema, the effect of which can be positive, and a transfer by direct mapping, which has been frequently shown to be confusing.

An experiment on learning to use electronic devices – such as notepad, clock, or chequebook – conducted by Kamouri *et al.* (1986) clearly demonstrates the superiority of an exploration-based training over an instruction-based training, thanks to the inducement of analogical reasoning in the exploration situation. This experiment, however, uses a well-suited frame to obtain positive effects from analogy, and follows Gick and Holyoak's works (1980, 1983) which reveal in particular that the condition for an efficient analogical transfer to occur is a prior elaboration of an abstract schema. This elaboration of an abstract schema has been shown to be improved when subjects are required to solve several analogous problems. So Kamouri *et al.* had their subjects solve problems with three analogous devices before discovering a new device either analogous to the previously acquired ones or not. The authors insist that this kind of situation is difficult to find in real work settings. But it may be a possible explanation of the well-known fact that learning a new programming language or software is improved by previous knowledge of other similar programming languages or software.

In the human-computer interaction area, analogical transfer by direct mapping is very often observed in beginners: two analogous objects are confused instead of being considered as different instantiations of the same schema. For example, the text editor space bar is considered to have exactly the same function as the typewriter space bar. These confusions are reinforced by the choice of familiar command names or programming language statements similar to natural language descriptions of actions. If feedback is available, this kind of direct transfer can stimulate active learning, for it can trigger accommodation processes, and differences are as useful as similarities (Carroll and Mack, 1985).

4.4 Mechanisms of learning by analogy

Several research studies shed light on the mechanisms of acquisition of operating rules in diverse learning-by-doing situations and on the conditions of efficiency of these situations. In experimental game situations where subjects are mostly unfamiliar with the games (Anzai and Simon, 1979; Nguyen-Xuan and Grumbach, 1985), four main mechanisms have been described:

* progressive elaboration of the problem space;
* identification of wrong actions which lead to errors, and generation of procedures in order to avoid them;
* identification of correct actions which lead to the goal, and creation of subgoals to which these actions can be applied;
* structuring of the subgoals by processing goal interactions.

But the story could be somewhat different in learning to program, and, more generally, in learning to use a command device.

4.4.1 Borrowing problem space and goal structure

In learning to use software or to program, the problem space is not generated from scratch, but rather from familiar problem spaces relating to familiar devices (e.g. typewriter when using and editor) or RPSs that are related to the problem domains (e.g. algebra, management, accountancy, etc., when programming). In the case of programming, familiar goal structures are transferred from familiar RPS as a number of studies have shown (Hoc, 1977, 1983; Nguyen-Xuan and Hoc, 1987). Hence, in contrast to usual learning by solving problems, the novice programmer already has a goal structure at the outset, although it may not be quite relevant. Sometimes, this goal structure comes from other programs (examples used in programming courses or handbooks, programs written by anybody else, etc.).

In Hoc's experiment (1983) on stock updating by the mean of a computer device, novices tried to use the '+' command corresponding to the ordinary binary addition as a summation operation after having entered a list of numbers to be added into the same memory cell. In our experiment on sorting (*op. cit.*) we observed a clear transfer of the insertion method used by all the subjects in the execution-by-hand situation. The goal structure of this method consists in decomposing the problem into iterative subgoals. Each one corresponds to the insertion of a new element into a well-sorted sub series. This goal structure was transferred into the command device situation, although it was very costly in comparison to the constraints of the device. Most of the subjects tried to locally modify subgoals, instead of building a totally new goal structure.

4.4.2 Repairing goal structure

When a goal structure is available, attempts are made to reach the subgoals by means of the device operators. A number of difficulties have been pointed out, which lead the learner to modifying the borrowed goal structure (Carbonell, 1983; Nguyen-Xuan, 1987). Some of them are particularly relevant to the use of a new computer device:

* The necessity to satisfy preconditions in the target situation, which are automatically satisfied in the source situation. For example, in our experiment on sorting (*op. cit.*) the subjects transferred the insertion method. In the source situation where boxes are sorted, the insertion action automatically pushes the adjacent boxes to make room. In the target situation, where an array is used, 'room made' is a precondition that must be satisfied by a quite complex shifting procedure (see Figure 1).

* The necessity to decompose elementary source actions into even more elementary ones. The already cited example of assimilating the '+' operator to a summation is illustrative of this decomposition.

* The discovery of unexpected goal interactions that are not present in the source procedure. An example can be found in Burstein's work on learning Basic (1986). Some novices assimilate the assignment statement to stacking in a box. Interaction occurs when they want to store a series of values: each new value deletes the previous one, which does not happen in the case of stacking in a box.

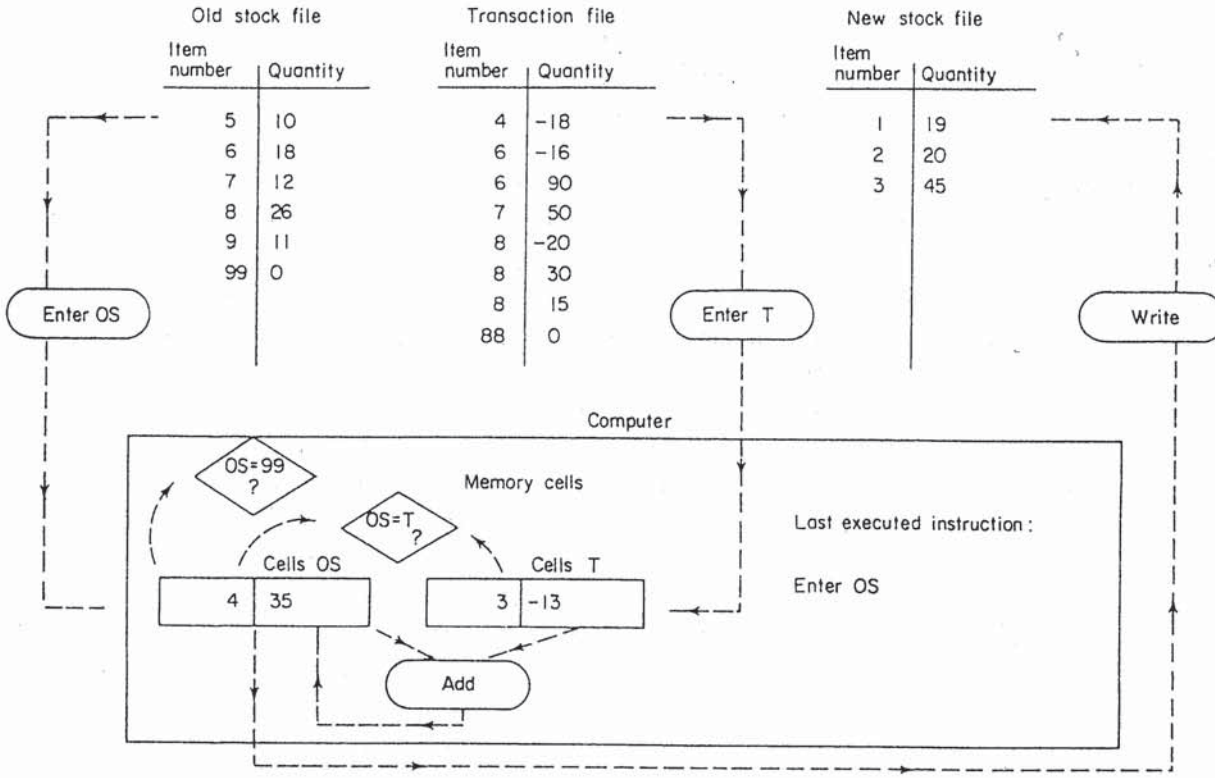


Figure 3: Updating device (after Hoc, 1983): (a) visible situation, subjects can see the contents of the files and they visually compute the number of transactions before entering an item from the old stock file; (b) covered situation, they have only access to the contents of the memory cells and cannot identify the number of transactions before processing an item.

* The incompatibility between the source goal structure and the data identification means. In a procedure, not only transformation operations but also identification operations must be considered. Identifications concern data properties that are relevant for choosing the appropriate action to perform (e.g. in an updating task, the identification of the number of transaction per item). The available means to perform the identifications may deeply affect the goal structure. This is the reason for difficulties encountered by novices when they had to transform a 'read-process' iterative schema into a 'process-read' one (Soloway *et al.*, 1982; Samurçay, 1985) as is shown by an experiment conducted by Hoc (1983).

In the above-mentioned experiment by Hoc, novices had, successively, to update a stock file either with visual access to the files or with access to only one item and one transaction at a time in computer memory cells (Figure 3). In the former situation the identification of the type of item (number of transactions) was performed before processing the item, so the 'read transaction - process it' schema was used. In the

latter this identification was no longer possible, so the 'process transaction – read the next one' schema had to be used to identify the end of the processing of the current item. The subjects showed very strong reluctance to abandon the 'read-process' goal structure: they tried to stick to it by attempting to perform actions that were not allowed. When they noticed that the transaction that had just been entered did not correspond to the item being processed, some subjects tried to return the transaction to the input file so they could enter it when they processed the item to which it corresponded.

5 Implications for training

From the results presented above, some implications on training design can be drawn, which could facilitate the elaboration of an operative device model and lead to the generation of optimal procedures. In most of the cases, the learner relies too heavily on available goal structures which may be inadequate, following the processing priority of the goal analysis level and neglecting the other levels: means and prerequisite analysis (see above).

(a) Learning situations have to be designed to prevent the learners from only being oriented towards the attainment of the task goal.

Friemel and Richard (1987) noticed that the presentation of a visual simulation of the internal operation of a pocket calculator was not efficient enough to enable the acquisition of operating rules. Novices devoted more attention to attaining of the goal than to analysing the display. This led the authors to ask novices to predict the effects of written procedures before learning by doing. This method has proved to be efficient when different procedures leading to the same result are presented: the novices are then encouraged to turn from a goal-level analysis to a means-level analysis. Certain recommendations proposed by Wærn (1989) lead in the same direction: encouraging the learner to use various methods of performing the same task, to process unexpected results, and to reflect on observations.

(b) Goal structures which could be transferred have to be known by the teacher so that learning situations can be designed, which force the learner to abandon these structures or which facilitate the adaptation.

In the first case constraints can be imposed which reveal the inadequacy of the source goal structure. In our experiment on sorting (*op. cit.*) the subjects had a two-dimensional array at their disposal (see Figure 1). The elements to be sorted were displayed on the first line and the well sorted series could be constructed on the second line. In this situation most of the subjects adhered to their insertion method without being able to discover an optimal procedure. However, when the subjects were given a single line and an extra cell (transfer) they rapidly turned from the insertion procedure to an optimal extremum procedure. In the second case the question concerns the effectiveness of the error identification and the possibility of recovering from errors. This principle underlies the recommendation (Carroll and Carrither, 1984) to design successive devices which from the onset restrict degrees of freedom in behaviour so that the feedback should be easy to manage.

(c) Even analogical transfer by direct mapping may be helpful, but with the con dition that a sufficient amount of feedback is provided to the learner.

Immediate feedback has been shown to be more effective than delayed feed back. In an experiment on learning to use a pocket calculator Friemel an Richard (1987) found that an exploration-based situation is more efficient tha a situation where subjects are encouraged to program their procedures befor executing them. Immediate feedback is more effective, probably because i enables subjects to causally connect elementary actions to their direct effects This has been shown in an experiment where naive subjects had to elaborat a sorting procedure in a stepwise command situation (Nguyen-Xuan and Hoc 1987). At the beginning of the experiment subjects were given access to th content of the cells of a table they had to sort (see Figure 1). They then had t do the same task without access to the content of cells. No subject succeede in changing his/her procedure to an optimal one in the covered situation: th change, when it occurred, was made in the visible situation.

Unfortunately, in a programming situation, one has to write an entire progran before being able to get feedback from the execution. In addition, error messages ar not as easily understood as they should be. The aim of these messages is to protec the compiler or the operating system rather than operative evaluation of the error (d Boulay and Matthew, 1984). Nevertheless, the absence of feedback for the beginner may be somewhat repaired by the presentation of a concrete model of the compute device. Several experiments conducted by Mayer (1975, 1976, 1981) have shown tha the presentation of a device model prior to learning did have a positive effect o program interpretation, iteration management, and transfer to novel situations, i contrast to the sole presentation of the language rules. However, these effects are no found neither in program generation nor when subjects are good at mathematics These results are interpreted by the author in the context of meaningful learning the model enables subjects to assimilate new pieces of knowledge with previously acquired knowledge.

6 Conclusion

Research into the acquisition of mental models of computer devices shows how im portant it is to consider the programming language as a semiotic tool, the conten of which corresponds to the operation of a device. This acquisition implies learning by-analogy mechanisms which cannot be successful without analysis of the feedback obtained from execution.

Learning-by-doing situations, however, have been shown to present limitations i they are not designed to improve spontaneous mechanisms. The crucial improvement consists in discouraging the learner to concentrate on goal-level analysis of the task and encouraging him to access to means level and prerequisite level. In addition the learner has to be assisted in identifying and recovering from errors. Investigation into learning-by-doing mechanisms points towards design of this kind of support.

Most of the research works concerning the very start of learning to program have mainly been concerned with procedural languages. Research on more recent programming languages such as Prolog or object-oriented languages (Smalltalk) are now in progress. Results concerning Prolog (Bundy *et al.*, 1986; Taylor and

du Boulay, 1986; White, 1988) suggest that the story could be somewhat different. Prolog has some special features (backtracking, unification, pattern-directed control, etc.) that cannot be found in other procedural languages or everyday situations. But although the Prolog user has to deal with logic specification, the learning of the Prolog machine is necessary to the implementation of the specification. Procedural aspects of Prolog programs are shown to play an important role even at an expert level, where tracing tools are very often used (Taylor and du Boulay, 1987). Results on object-oriented languages are not yet available.

This construction of a mental model of device operating rules mainly applies to the start of the learning process. It results in the acquisition of elementary rules as well as goal structures compatible with the novel device. Macro-operators and programming plans, which contain the operation constraints of the device as built-in constructs, are learnt. Complex procedures can then be designed without analysing the details of the device operation. Certainly these learning-by-doing situations have to be restricted to the elaboration of simple procedures. The stepwise command of complex procedures (e.g. with deep iteration embedding) would lead to a mental load too heavy to be practicable.

The subsequent stages of learning to program, through the acquisition of adequate problem representations, programming plans, and schemas, are examined in the next chapter of this book.

References

Allwood, C.M. and Eliasson, M. (1987). Analogy and other sources of difficulty in novices' very first text editing. *International Journal of Man-Machine Studies,* **27,** 1-22.

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

Anzai, Y. and Simon, H.A. (1979). The theory of learning by doing. *Psychological Review,* **86,** 124-140.

Bayman, P. and Mayer, R.E. (1984). Instructional manipulation of user's mental models for electronic calculators. *International Journal of Man-Machine Studies,* **20,** 189-199.

Bonar, J. and Soloway, E. (1985). Preprogramming knowledge: a major source of misconceptions in novice programmers. *Human-Computer Interaction,* **1,** 133-161.

Bundy, A., Pain, H., Brna, P. and Lynch, L. (1986). *A proposed PROLOG story.* University of Edinburgh, Department of Artificial Intelligence, Research paper no 283.

Burstein, M.H. (1986). Concept formation by incremental analogical reasoning and debugging. *In* R.S. Michalski, J.G. Carbonell and T. Mitchell (Eds), *Machine Learning: An Artificial Intelligence Approach,* vol. 2. Los Altos, CA: Kaufmann.

Carbonell, J.G. (1983). Learning by analogy: formulating and generalizing plans from past experience. *In* R.S. Michalski, J.G. Carbonell and T.M. Mitchell (Eds), *Machine Learning.* Palo Alto, CA: Tioga, pp. 137-161.

Card, S.K., Moran, T.P. and Newell, A. (1983). *The Psychology of Human-Computer Interaction.* Hillsdale, NJ: Erlbaum.

Carroll, J.M and Carrithers, C. (1984). Training wheels in a user interface. *Communications of the ACM,* **27,** 800-806.

Carroll, J.M. and Mack, R.L. (1985). Metaphor, computing systems, and active learning. *International Journal of Man-Machine Studies,* **22,** 39-57.

du Boulay, B. and Matthew, I. (1984). Fatal errors in pass zero: how not to confuse novices. *In* G. van der Veer, M.J. Tauber, T.R.G. Green and P. Gorny (Eds), *Readings on Cognitive Ergonomics – Mind and Computers.* Berlin: Springer-Verlag, pp. 132-143.

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

Eisenstadt, M., Breuker, J. and Evertsz, R. (1985). A cognitive account of 'natural' looping constructs. *In* B. Schackel (Ed.), *Human-Computer Interaction – INTERACT 84.* Amsterdam: North-Holland, pp. 455-459.

Friemel, E. and Richard, J.F. (1987). Apprentissage de l'utilisation d'une calculette. *Psychologie Française,* **32,** 227-236.

Gick, M.L. and Holyoak, K.J. (1980). Analogical problem solving. *Cognitive Psychology,* **12,** 306-355.

Gick, M.L. and Holyoak, K.J. (1983). Schema induction and analogical transfer. *Cognitive Psychology,* **15,** 1-38.

Hoc, J.-M. (1977). Role of mental representation in learning a programming language. *International Journal of Man-Machine Studies,* **9,** 87-105.

Hoc, J.-M. (1978). La programmation comme situation de résolution de problème. Paris, Université René-Descartes, Doctoral Dissertation.

Hoc, J.-M. (1983). Analysis of beginner's problem-solving strategies in programming. *In* T.R.G. Green, S.J. Payne and G. van der Veer (Eds), *The Psychology of Computer Use.* London: Academic Press, pp. 143-158.

Hoc, J.-M. (1988a). *Cognitive Psychology of Planning.* London: Academic Press.

Hoc, J.-M. (1988b). Towards effective computer aids to planning in computer programming. *In* G. van der Veer, T.R.G. Green, J.M. Hoc and D. Murray (Eds), *Working with Computers: Theory Versus Outcomes.* London: Academic Press.

Jackson, M. (1980). The design of conventional programming languages. *In* H.T. Smith and T.R.G. Green (Eds), *Human Interaction with Computers.* London: Academic Press, pp. 321-347.

Kamouri, A.L., Kamouri, J. and Smith, K.H. (1986). Training by exploration: facilitating the transfer of procedural knowledge through analogical reasoning. *International Journal of Man-Machine Studies,* **24,** 171-192.

Klahr, D. and Dunbar, K. (1988). Dual space search during scientific reasoning. *Cognitive Science,* **12,** 1-48.

Mayer, R.E. (1975). Different problem-solving competencies established in learning computer programming with and without meaningful models. *Journal of Educational Psychology,* **67,** 725-734.

Mayer, R.E. (1976). Some conditions of meaningful learning for computer programming: advanced organizers and subject control of frame order. *Journal of Educational Psychology,* **68,** 143-150.

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

Mendelsohn, P. (1986). Activation de schèmes de programmation et mémorisation de figures géométriques. *European Journal of Psychology of Education,* **1,** 126-138.

Miller, L.A. (1974). Programming by non programmers. *International Journal of Man-Machine Studies,* **6,** 237-260.

Morais, A. and Visser, W. (1987). Programmation d'automates industriels: adaptation par des débutants d'une méthode de spécification de procédures automatisées. *Psychologie Française,* **32,** 253-259.

Moran, T.P. (1981). The command language grammar: a representation for the user interface of interactive computer systems. *International Journal of Man-Machine Studies,* **15,** 3-50.

Nguyen-Xuan, A. (1987). Apprentissage par l'action d'un domaine de connaissance et apprentissage par l'action du fonctionnement d'un dispositif de commande. *Psychologie Française,* **32,** 237-246.

Nguyen-Xuan, A. and Grumbach, A. (1985). A model of learning by solving problems with elementary reasoning abilities. *In* G. d'Ydewalle (Ed.), *Cognition, Information Processing, and Motivation.* Amsterdam: North-Holland.

Nguyen-Xuan, A. and Hoc, J.-M. (1987). Learning to use a command device. *European Bulletin of Cognitive Psychology,* **7,** 5-31.

Norman, D. (1983). Some observations on mental models. *In* D. Gentner and A.L. Stevens (Eds), *Mental Models.* Hillsdale, NJ: Erlbaum, pp. 7-14.

Payne, S.J. and Green, T.R.G. (1986). Task-Action Grammars: a model of mental representation of task languages. *Human-Computer Interaction,* **2,** 93-133.

Richard, J.F. (1983). Logique du fonctionnement et logique de l'utilisation. Le Chesnay (F), INRIA, Research Report No. 202.

Richard, J.F. (1986). The semantics of action: its processing as a function of the task. Le Chesnay (F), INRIA, Research Report No. 542.

Rouse, W.B. and Morris, N.M. (1986). On looking into the black box: prospects and limits in the search for mental models. *Psychologiçal Bulletin,* **100,** 349-363.

Samurçay, R. (1985). Learning programming: an analysis of looping strategies used by beginning students. *For the Learning of Mathematics,* **5,** 37-43.

Shrager, J. and Klahr, D. (1986). Instructionless learning about a complex device: the paradigm and observations. *International Journal of Man-Machine Studies,* **25,** 153-189.

Soloway, E., Ehrlich, K., Bonar, J. and Greenspan, J. (1982). What do novices know about programming? *In* A.Badre and B. Shneiderman (Eds), *Directions in Human-Computer Interactions.* Norwood, NJ: Ablex.

Taylor, J. and du Boulay, B. (1986). Why novices may find programming in Prolog hard? University of Sussex, Cognitive Studies Research Paper No. 60.

Taylor, J. and du Boulay, B. (1987). Learning and using Prolog: an empirical investigation. University of Sussex, Cognitive Studies Research Paper No. 90.

Van Someren, M. (1984). Misconceptions of beginning Prolog programmers. University of Amsterdam, Department of Experimental Psychology, Memorandum 30.

Wærn, Y. (1985). Learning computerized tasks as related to prior task knowledge. *International Journal of Man-Machine Studies*, **22**, 441-455.

Wærn, Y. (1989). *Cognitive Aspects of Computer Supported Tasks.* Chichester: Wiley.

White, R. (1988). Effects of Pascal upon the learning of Prolog: an initial study. University of Edinburgh, Working paper.

Young, R.M. (1981). The machine inside the machine users' models of pocket calculators. *International Journal of Man-Machine Studies*, **15**, 51-85.

Youngs, E.A. (1974). Human errors in programming. *International Journal of Man-Machine Studies*, **6**, 361-376.