Chapter 2.4

Acquisition of Programming Knowledge and Skills

Janine Rogalski and Renan Samurçay

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

Acquiring and developing knowledge about programming is a highly complex process. This chapter presents a framework for the analysis of programming. It serves as a backdrop for a discussion of findings on learning. Studies in the field and pedagogical work both indicate that the processing dimension involved in programming acquisition is mastered best. The representation dimension related to data structuring and problem modelling is the 'poor relation' of programming tasks. This reflects the current emphasis on the computational programming paradigm, linked to dynamic mental models.

1 Introduction

Learning to program in any language is not an easy task, and programming teachers are well aware of the myriad difficulties that beset beginners. Why is it so difficult to learn or teach programming? How can analysis of these difficulties serve to indicate

Psychology of Programming
ISBN 0-12-350772-3

ways of remedying them? The process of teaching and learning not only involve learners but a set of situations in which teachers stage knowledge about programming Teachers need to know more than the mechanisms of human learning in order to analyse and structure their teaching. They also must be aware of the fact that what learners learn is dependent on their own conception of the activity of programming and choices and decisions they as teachers make during instruction.

This chapter presents a framework for the analysis of programming. It serves as a backdrop for a discussion of findings on programming learning and is a basis for an understanding of why some programming concepts and procedures are difficult to teach and to learn. This framework is based on a conceptual view of programming Programming, like problem solving activities in other scientific fields such as physics or mathematics, can be analysed in terms of expert competence (expertise-oriented framework), or in terms of a constituted knowledge domain (content-oriented framework). Knowledge of this latter type is socially and historically constructed, and is formed of normative and symbolic representations used for communicative purposes

Studies in the field of programming learning can be categorized by their orientation. A number are expertise oriented, where a model of expert knowledge is used as a reference to analyse novices' errors and misconceptions (Bonar and Soloway 1985; Soloway and Ehrlich, 1984). Other studies are content oriented, in that greater attention is paid to an analysis of programming knowledge as a domain having its own concepts, procedures, notations and tools (Pea; Rogalski; Rouchier, Samuçay Taylor and du Boulay). A third category of studies takes no explicit epistemological position on the analysis of programming knowledge as a specific task domain. Programming is used here primarily as a paradigm which lends itself to the testing of general models of subjects' cognitive architectures and learning processes (Anderson *et al.*, 1984).

This chapter focuses on the acquisition of programming knowledge, as testified to by students' ability to solve 'programming problems' at various levels of complexity Acquisition of programming skills for general educational goals and its developmental aspects are discussed in Chapter 2.5. The students referred to here are adults or adolescents with general educational backgrounds although they may be complete novices with respect to programming knowledge. Thus we will be looking at issues of computer literacy and those related to pre-professional and professional training.

We will be arguing that the teaching process cannot reproduce the real process of construction of knowledge by scientific or professional people in the classroom Rather the teaching process creates new 'objects for teaching' which need to satisfy certain functional properties in order to create meaning in learners' minds. Pair's work (see Chapter 1.1) is used as a springboard for showing that the impact on the teaching process of teachers' conceptualizations of programming activity leads to the development of different types of knowledge and strategies in learners.

2 A framework for programming activity

2.1 General framework

Figure 1 presents a general theoretical framework for knowledge representation in the programming field. It is made up of four related 'spaces': the knowledge structure, problem solving, practice and cognitive tools. This framework is illustrative of the fact that knowledge is constructed and assessed by problem solving. The knowledge

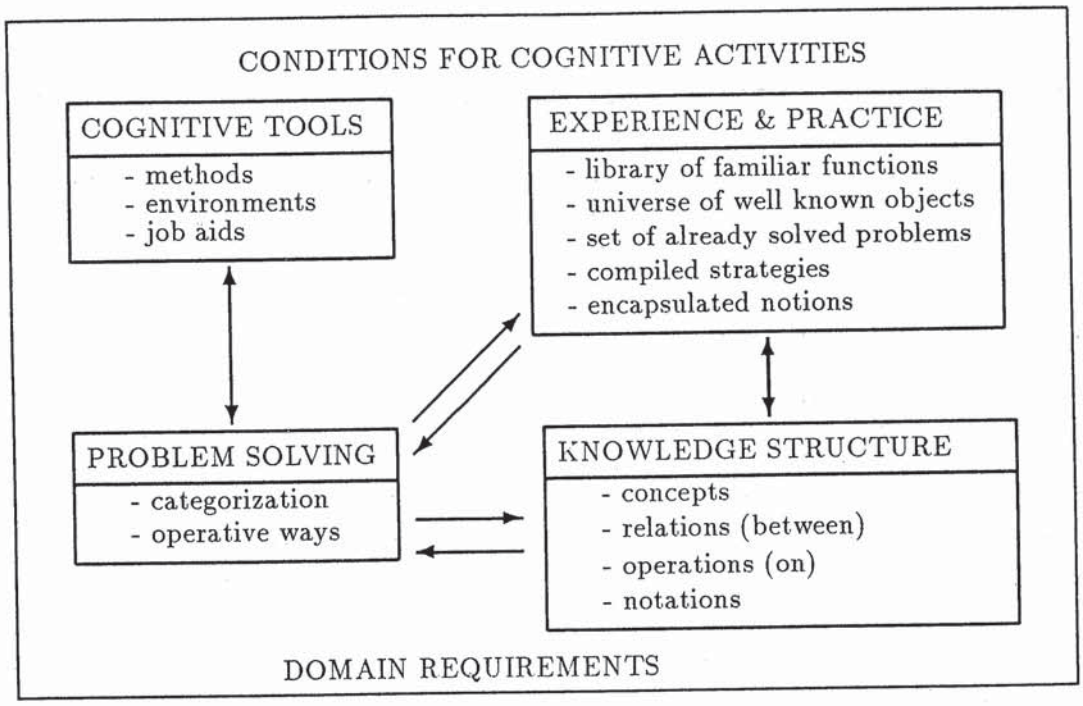

Figure 1: Frame for task analysis.

structure consists of a hierarchical system of concepts, operations on, and relations between concepts having a set of notations. In any problem in the field, concepts need to be co-ordinated. For instance a list problem involves variables, iteration or recursion, list functions, test, inputs and outputs, whatever is the programming language, and whatever are its semantical and syntactical specificities. Appropriate notation needs to be selected (even within a programming language, choices must be made between different control structures). In addition there is a relationship between types of problems and operative pathways. Sum problems, sort problems, list or graph processing are related to different classical algorithms. Cognitive activities in programming tap both prior experience and individuals' cognitive 'tool bag'.

During the process of knowledge acquisition, all four 'spaces' evolve. New notions are acquired through new interactions. For instance, the parameter passing in procedures or functions involves the notion of procedure and a redefinition of variables as global or local, and thus generates a new level of understanding of the relationship between variable, name and value.

Complex data structures gradually become familiar objects (such as tables for beginners in procedural languages for example). The co-ordination of functions or procedures can be conceptualized in terms of the role they play in the problem, without reference to mental running with specific values (encapsulated notion of functions). The set of previously solved problems increases. Certain strategies become directly available, relating types of problems with operative pathways (compiled strategies). The complexity of problems that can be tackled increases.

At the start of the acquisition process students necessarily refer to previous everyday experiences, the acquisition of the first RPS (representation and processing

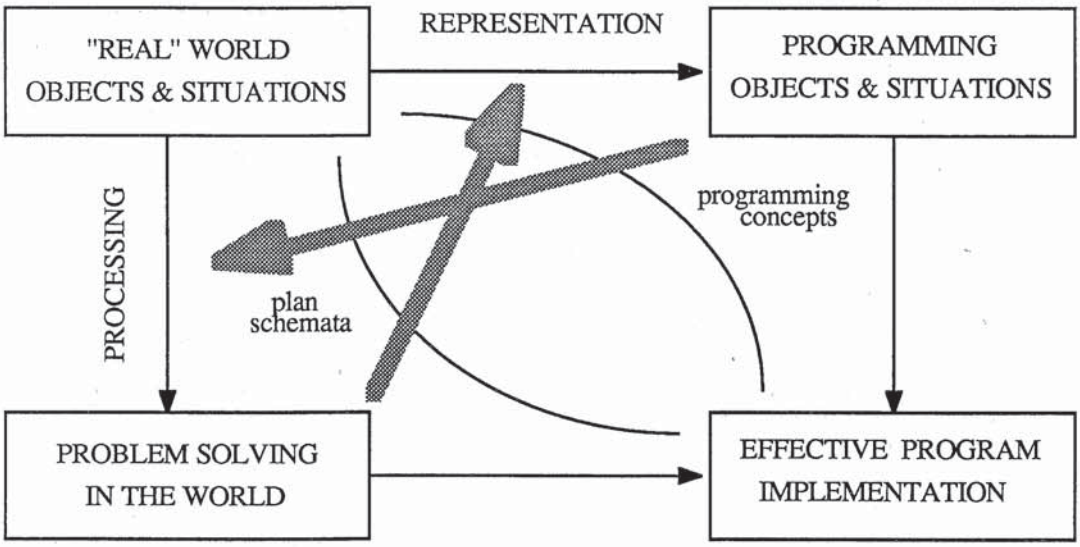

Figure 2: The task of programming.

system, see definition in Chapter 2.3) can be built up by such mechanisms as learning by doing, or by analogy, or from other fields of knowledge such as mathematics (the prime source), physics, or from exposure to other programming languages. Precursors of this type can generate both negative as well as positive effects because of similarities and differences in knowledge structures (Rogalski and Vergnaud, 1987). These effects may account for two types of empirical results. Preprogramming knowledge is known to be a source of misconceptions (Bonar and Soloway, 1985), and previous programming knowledge can even constitute an obstacle, requiring an 'unlearning process' (Taylor and du Boulay, 1987; Chapter 4.2). Also, students with mathematical backgrounds do not need the same amount of teaching (van der Veer and van der Wolde, 1983; van der Veer *et al.*, 1986).

2.2 What is programming

Figure 2 presents a schema for analysing the cognitive activities involved in programming tasks 'from a real-world problem to a runnable program text'. In certain respects this schema is the psychological counterpart of the epistemological analysis developed in Chapter 1.1.

The crucial dimensions in the activity of programming are processing and representation. There are two ways individuals can move from a real-world problem to program text implementable on a given device. A real-world problem can first be solved in the domain and then translated into program text. Or, alternatively, it can be approached in the programming language and applied to the real-world object. When problem solving takes place on a real-world object, processing pre-

cedes representation, even if the properties of the objects in the target programming language intervene in the choice of solution. This approach is closely related to the 'computational' programming paradigm where a program is defined as a succession of computations. The 'functional' perspective where the program is seen as a function that needs to be decomposed is more infrequent. When problem solving initiates in the programming text, the structuring of data and relationships between pieces of data is the core of programming activity.

Studies on the acquisition of programming skills are mainly centred on the processing dimension. There are at least three reasons for this: (1) the historical role of procedural languages, (2) the importance of planning in programming design (Hoc, 1988a) and (3) the productive role of organization of actions as a first programming model (Samurçay and Rouchier, 1985). In addition, problems given to novices are often directly defined in terms of programming entities, such as numerical data and variables in the well-known sum problems (Spohrer *et al.*, 1985; Soloway *et al.*, 1982; Soloway and Erhlich, 1984; Samurçay, 1985) or explicitly in terms of a given programming language, such as: 'define a function called list-sum. Given a list of numbers, list-sum returns the sum of these numbers. For example, list-sum′ (5 10 – 4 27) returns 38. (List-sum′ ()) returns 0' (Katz and Anderson, 1988).

Few studies have dealt with the representation task related to data structuring even though a number have reported findings on the effects of problem content or semantics (Ormerod *et al.*, 1988; Widowski and Eyfert, 1986). In procedural languages, complex data structures are often related to complex problems and the representation task may only appear crucial for advanced students. However, there is evidence that program design aids may cause difficulties when the language is insufficiently rich in data structures flexible enough to represent the variety of data (Hoc, 1988b). In a relational language like Prolog, beginners may run up against two difficulties: choosing how to represent objects and the relationship between them, and deciding how general the solution should be (Taylor and du Boulay, 1986). From the cognitive analysis developed by Rosson and Alpert (1988), it can be hypothesized that the same difficulties could be encountered with object-oriented programming languages.

The acquisition of programming knowledge and skills can be characterized in a number of ways. Problem solving in programming can be centred on problems in the real world (research on planning) or on the program as text (research on programming language and use). Here, schemas are defined as sets of organized knowledge used in information processing, and plans are defined as organized sets of dynamic procedures related to static schemas. For a given problem, plans and schemas can be defined at several levels (strategy, tactics, implementation) (Samurçay, 1987).

Studies on the construction and instantiation by experts and novices of schemas (the program-as-text perspective) or plans (the programming-as-an-activity perspective) can be categorized as a function of these perspectives. Program text studies are centred on the cognitive activities involved in the understanding or debugging of written programs (see Chapter 3.1). In these experiments schemas are defined as standard structures that can be used to achieve small-scale goals such as those involved in sum problems using variable plans and loop plans. Training research indicates that schemas can be induced by creating analogies between training problems and programs, but are most efficient in understanding and debugging tasks, and less efficient in program design.

We take a slightly different point of view which is more oriented towards the acquisition of fundamental programming concepts that can be implemented in various programming tasks (design, execute, modify or complete programs). One reason for this emphasis is that bugs in novice programming are mainly conceptual errors (Spohrer and Soloway, 1986). In the framework presented above (Figure 1) the development of strategies is seen as that facet of acquisition related to problem solving, and is more general than either plan or schema acquisition (see also Chapter 3.2). There are advantages in seeing plans and schemas as special cases of 'compiled' strategies and 'encapsulated' notions, related to the set of previously solved problems: these concepts are valid whatever programming language is concerned (Rosson and Alpert, 1988). Methods (related to task analysis) may enhance the cognitive activities involved in strategy research and management and help programmers in problem solving.

3 Cognitive difficulties in learning programming

Programming as a knowledge domain differs from other neighbour domains such as mathematics or physics in two ways. First, there are no everyday intellectual activities that can form the basis for spontaneous construction of mental models of programming concepts such as recursion or variables, in contrast to such notions as number or velocity. These concepts, however, are the basis for concept acquisition in programming. Secondly, programming activity operates on a physical machine which may not be transparent in its functioning for learners.

Which familiar RPS can be activated in learners for the construction of a new RPS that is operational for programming? Which kinds of transformations are needed for the construction of this new RPS? How do beginners construct the conceptual invariants related to programming?

The difficulties encountered by novice programmers have been subdivided below into four areas corresponding to the main conceptual fields novices must acquire during the learning process. This subdivision has been made purely for readability's sake since it is obvious that novices suffer from syndromes rather than from single misconceptions. All errors arise in conjunction with others and may have multiple causes as a function of the problem context.

3.1 Conceptual representations about the computer device

When designing, understanding, or debugging a program, expert programmers or software developers refer necessarily to their knowledge of the systems underlying programming languages, and the programming tools available on these systems. They are able to move from one system to another and change their representation of the problem to adapt to new constraints (see current practices of professional programmers in Chapter 4.2). This knowledge extends beyond operating rules and involves the representation of the whole system (editor, operating system, input/output devices, etc.) This is what Taylor and du Boulay have termed the 'notional machine'.

There are two basic difficulties encountered by novices as concerns conceptual representations. The first involves the effects of the relative complexity of the command device in the acquisition process. The second is related to the construction of the notional machine.

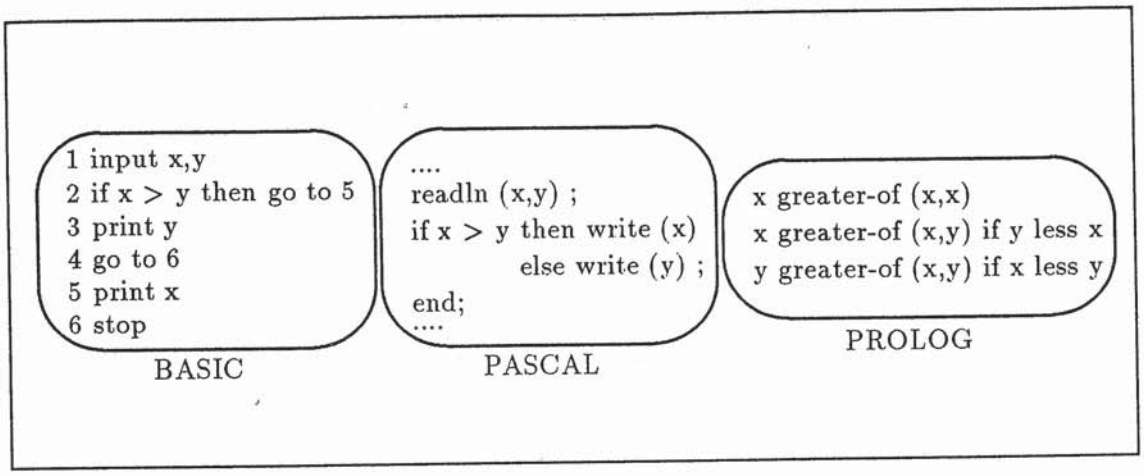

Figure 3: Examples of programs in Basic, Pascal and Prolog.

The very elementary level of construction of a RPS by beginners for simple command devices has been described in Chapter 2.3. This first RPS constructed 'by doing' and by analogy is, however, not sufficient for full conceptualization of the notional machine with respect to the command device. The notion of 'complexity' of a command device is itself a highly relative concept. Learning a text editor, how to use a pocket calculator, or a programming language are not analogous tasks because of the features inherent to the problems each device can be used to solve. Even in programming languages, the conceptualization of a 'Basic machine' a 'Pascal machine' or a 'Prolog machine' will not involve the same cognitive operations. For instance, compare the programs in Figure 3 written in these three languages that will print the larger of two input numbers.

Basic and Pascal programs describe *how* a result is computed; they specify actions to be performed by the machine. The Prolog program describes *what* is true of the problem solution, leaving it up to the computer to sort out the steps involved in finding the solution. In the case of imperative languages, a first, although extremely limited RPS can be built on the representation of *actions* and their sequential control. In languages like Prolog this representation requires the concept of *logical truth* and a representation of how Prolog *evaluates* this truth. Thus the RPS that beginners must construct with Prolog involves objects such as truth values and relations that are conceptually far removed from the objects involved in imperative languages. Moreover, as Taylor and du Boulay (1987) point out, mastery of these objects in the pure logic domain is not enough. The way Prolog processes to evaluate the truth of predicates (backtracking) has no counterpart in 'manual' logic.

There are two dimensions to the relative complexity of the command device: the proximity of the control, and the existence of virtual entities (virtual memories, variables, files, etc.) simulating entities which have no physical identity. It is believed that the command device increases in complexity for learners as a function of the distance of the control and the variety of virtual entities. For example, assembler language and Prolog constitute extremes in this case. Prolog is used here for purposes of illustration of a language that differs from imperative ones and identical analysis could be made of object-oriented languages such as Smalltalk. It would be worth-

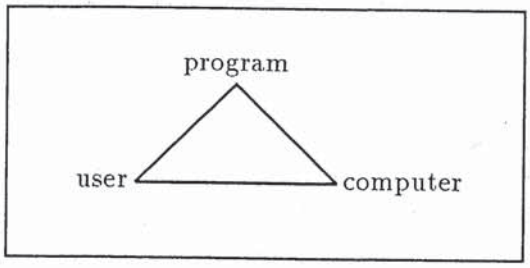

Figure 4: Relationships involved in program running.

while to investigate how experts deal with virtual entities, which are an important component of 'high-level' programming languages.

In program design, beginners also need to construct mental models of the functional relationships between program, user (who will enter the data) and the computer device the program is run on (Figure 4).

Misrepresentations of these relationships interfere with planning activity in beginners. They also affect the meaning beginners assign to the results of execution and to error messages. Novices' erroneous representations of the notional machine can be analysed from two points of view.

The first concerns language-independent conceptual bugs which disrupt the way in which novices program and understand programs. Several terms have been coined to describe these misconceptions: 'superbugs' (Pea, 1986), 'meta-analysis' (Hook *et al.*, 1988), 'preprogramming knowledge' (Bonar and Soloway, 1985), 'wrong RPS transfer' (Hoc and Nguyen-Xuan, Chapter 2.3). These mainly characterize errors in communication rules involving computer jumps into the real-world domain, or false expectations that the program will flow from top to bottom and left to right.

A number of studies have focused on these errors. Rogalski and Hé (1989) present a model ('PRES') implemented by novices when dealing with conditional problems in a Basic-like language. In this model novices assume that 'in a conditional part of a program when condition A has been processed the other instruction is applied in the notA situation (it is unnecessary to specify the latter condition notA in the program text); conversely as long as the processing of situation A is not terminated, the instruction should be applied to case A '. This model illustrates an inadequate representation of sequentiality and is based on natural communication rules. It, however, applies successfully to CASE-OF problems (where there is no sequentiality problem) but fails with NEST-style conditional problems.

Effects of misconceptions have also been observed in simple iterative programs designed by novices (Laborde *et al.*, 1985; Samurçay and Rouchier, 1985; Bonar and Soloway, 1985). A common error consists of writing the operations in the order 'description of actions/repeat mark' which again reflects the fact that communication with the computer is considered to be an instance of natural language communication. Even when beginners acknowledge that there is no mind inside the computer they still credit it with semantic abilities including presupposition and interpretation of the content of communication with the user.

Similar errors have also been reported for Prolog novices on tracing problems, or analysis of the result of trace functions. Novices tend to jump into the real-world

domain in order to solve the problem; this model works in restricted circumstances (meaningful variable names, simple and familiar problem domain) but fails when the problem cannot be solved without a minimal understanding of the backtracking mechanism (Hook *et al.*, 1988). Novices may also introduce certain features of their reading skills, for instance by assuming that control flow through a clause or through a program always goes from left to right (Taylor and du Boulay, 1987).

The second type of error related to representations of the notional machine has to do with the system as a whole. Novices not only have problems with the representation of the machine underlying programming languages but also encounter difficulties with the computing system they are working with. Beginners need to differentiate which elements of this system belong to the language, and which are system entities. Prolog novices frequently alter their programs and forget to reload or reconsult the new file; alternatively they may inadvertently assert what are meant to be queries, thereby accidentally altering the program/database (Taylor and du Boulay, 1987). They have comparable difficulties maintaining the distinction between the Prolog program/database they can see on the terminal screen and the version of the program/database Prolog will process. Even in imperative languages where the distinction is clearer between program and data, novices do not discriminate clearly 'who' is controlling the input/output commands when the program is running, or the functional role of outputs on the screen with respect to memory management (Rogalski and Samurçay, 1986).

3.2 Control structures

One of the major points made by Pair (Chapter 1.1) is that programming can be conceptualized in a variety of ways including describing calculations (computational paradigm), describing functions (functional paradigm), or defining and processing relations between objects (relational paradigm). Regardless of the type of conceptualization or type of language, programs must be able to describe: (1) an undefined number of processings which remain to be described in a finite manner; (2) a general solution for a set of infinite data. Conditional and iterative or recursive structures are built as responses to these requirements in various forms depending on the language. The prime characteristic of control structures is the fact that they disrupt the linearity of the program text.

Control structures can be analysed and taught in different ways as a function of conceptualization. They can be seen (and taught) as *computational models* (oriented towards the description of calculations in the program), or as *functional models* (oriented towards analysis of the function of the program with respect to the problem). This distinction does not entirely mirror the declarative versus the procedural distinction since objects as well as processings are involved in both.

In the majority of studies describing early acquisition processes, as well as in programming tutoring systems, the computational approach dominates. Few studies have examined the acquisition of programming from the functional point of view. Research on the acquisition of control structures by novices has mainly centred on conditionals, iterative and recursive statements, and procedural languages. There is no explicit conditional statement in Prolog, and conditional treatments have to be handled via logical conditions (such as conjunctions, negation, etc.), and backtracking controlling (such as 'cut'). The problems novices encounter in the conceptualization

of backtracking is illustrative of the level of difficulty they face in the conceptualization of control structures (Taylor and du Boulay, 1986, 1987).

3.2.1 Conditional statements

The earliest studies in this field examined the elaboration of programming languages and compared programming styles such as JUMP style using the GOTO statement, NEST style using the embedded alternatives IF .. THEN .. ELSE, CASE-OF style using a succession of positive conditions (IF...THEN). Overall these studies indicate that the JUMP style, although seemingly easier in terms of expression, is more difficult to control than the NEST style (Green, 1980) and that styles using positive alternatives present fewer difficulties for acquisition. In addition, syntactic markers such as BEGIN..END defining the scope of the THENs and ELSEs and spatial organization of the text make conditional structures easier to learn and use (Sime *et al.*, 1977). However, difficulties in understanding conditional statements are also related to the type of question, which can be either conditional or sequential (Green, 1980), and the organization of logical cases. Depth of nesting of conditionals also increases difficulty. This set of studies also shows that there is no uniform 'mental language' used by novices in acquiring conditional structures (Gilmore and Green, 1984).

In terms of the acquisition of notions, logical and mathematical precursors have been shown to play an important role. At a global level, students with more grounding in mathematics learn new structures more rapidly (van der Veer *et al.*, 1986; van der Veer and van der Wolde, 1983). The ability to use logical connectors (AND, OR, and NOT) and represent structured cases are necessary for the acquisition of conditionals, but they are not sufficient in themselves because of the role played by an inappropriate model of sequentiality of execution by the 'computer device' (Rogalski and Hé, 1989).

3.2.2 Iteration

The construction of an iterative plan involves the identification of elementary actions/rules which must be repeated, and the condition governing end or continuation of the repetition. There are three operations entering into the construction of an iterative plan: construction and expression of the loop invariant (updating), identification of the end control and its place in the loop plan (test), and identification of the initial state of the variables (initialization).

There are two types of iterative problems. In the first, the end control is a constant, whereas in the second the end control is a value of a variable computed in the loop. In the latter instance, two plans are possible with respect to goal attainment

process variable/test variable

or

test variable/process variable

These two forms are not equally accessible in a the novice's existing plan catalogue (Soloway *et al.*, 1982; Samurçay, 1985). Novices' 'spontaneous' iterative models have the following structure: description of actions, repetition counter, repetition mark

and expression of the end control. Anticipation and explicitation of the end control are not spontaneously available (Laborde *et al.*, 1985; Samurçay and Rouchier, 1985), probably because this control is implicit in ordinary action plans. These features make it more difficult for novices to deal with 'test/process' plans (e.g. the WHILE loop in Pascal). Difficulty stems from representing and expressing a condition about an object on which they have not yet operated (Soloway *et al.*, 1982).

A didactic study shows that in a set of teaching situations where a variety of sum problems involving different types of constraints were introduced, novices could successfully be guided in the construction of an adequate representation of different loop plans (Samurçay, 1986). However, these representations are unstable and fail on new problems; novices return to step-by-step construction. In other words, the novice model is based on a representation of a succession of actions (dynamic model) rather than on a representation of the invariant relationship between the different states of variables (static model).

A typical loop plan implemented by novice programmers is given below:

```
.................
sum := 0 + number
counter := 1
sum := number1 + number2
counter := 2
repeat
.................
```

This fragment of protocol reveals two types of difficulties. The first is the construction of the loop invariant (`sum := sum + number` and `counter := counter + 1`) and the second is related to the designation of variables. The novice tends to use different names at each step in execution to label the same functional variable.

As indicated above, most research work, as well as tutoring systems such as Bridge (Bonar, 1984) or Proust (Johnson and Soloway, 1983), on the acquisition of the iterative control structure has been conducted from a computational point of view. It is likely that computational teaching reinforces the novices' dynamic model. This model fails when novices encounter new problems because they have not acquired how to search and construct the 'loop invariant' (relations must be conserved between variables during execution). The representation of the succession of variables and specific training on search for relationships between them could have positive impact on the learning process. Even with object-oriented language, it was observed difficulties on 'iterative methods used for scheduling objects, especially those in which iteration counter is used as a variable in the computation' (Goldberg and Kay, 1977).

3.2.3 Recursion

Recursive functions are defined in terms of themselves. The well-known factorial function for example is defined recursively by:

```
fact (0) = 1
fact (n) = fact (n − 1)*n
```

Whereas iterative programming is based on the descriptions of actions modifying the situation, recursive programming describes the relationships between successive states, without expressing the computation strategy. The most important functionality of recursive procedures intervenes in program decomposition, i.e. in the representation of the problem domain as a part-whole relationship. This decomposition calls for two kinds of conceptual constructs: autoreference (declarative aspect) and nesting (procedural aspect) (Rouchier, 1987).

Recursive programming is very hard to learn and to teach. Studies have consistently shown that even on very simple problems students have enormous difficulty learning recursion. Beginners tend to use an iterative model of recursion (Kurland and Pea, 1983; Kessler and Anderson, 1986; Pirolli, 1986; Rouchier, 1987; Wiedenbeck, 1989). This type of model is compatible with tail recursion (the recursive calls are not nested when executed) but fails for full recursion (the recursive calls are nested when executed). The reasons for these difficulties are two-fold. First, there is no natural everyday activity that can serve as a precursor for recursion. Second, the dynamic model of iteration is (overly) salient and constitutes an obstacle for students in moving to this new program schema (Kessler and Anderson, 1986). A dynamic model may at times be useful for program design (Pirolli, 1986). The learning mechanisms such as 'by doing' or 'by example' are not sufficient for thorough understanding of recursion. This kind of situation reinforces novices' existing dynamic models.

Data is scarce or virtually non-existent on how expert programmers design and understand complex recursive programs. However, a crucial factor in programming recursion appears to be the use of static representations of situations in a functional or relational point of view of programming. It seems difficult to start a training process with novices directly with this type of representation. There is some empirical evidence, however, suggesting that more appropriate teaching situations can be designed. In particular these should not employ an iterative model but rather should start with full recursion problems (in contrast to the Kessler and Anderson (1986) suggestion to use iteration as a source of analogy. These would make the relationship between the structure of the programs and the structure of the problem explicit (for instance in a Logo procedure the places of recursive calls in the text of the program are related to the order in which the objects are produced). In these situations the emphasis is on the static properties of the procedure. In this case students can succeed in constructing more-effective strategies in program design (Rouchier, 1987).

3.3 Variables, data structures and data representation

Variables are the (proverbial) tip of the iceberg of the various entities that can be defined and managed in the programming process. Whatever the programming language, programmers must deal with the questions of available primitive entities (objects and operations on them), and how such entities should be modified to represent the elements of the domain problem. A key factor in the acquisition of programming knowledge is mastery of the relationship between the functional role of entities and their successive values during program running. This involves the acquisition of notions organized in a hierarchical fashion with strong connections between the notions of functions and variables (or predicates and variables in logical languages such as Prolog, objects and messages in object-oriented languages).

Numerous errors found in novice control structure protocols can be accounted for by the conception novices have of variables. Defining a variable as a name (related

to the role played in the domain problem) or as an address (related to the role played in the program) is the first step towards understanding (Samurçay, 1989). When the value of a variable is modified, its naming and the functional relationship with other program elements remain invariant. At this level some of the errors made by novices involve the use of local rather than functional properties.

Variables also differ with regard to their functional meaning in student planning. 'External' variables correspond to the values controlled by the program user, i.e. they are explicit inputs (data) or outputs (results) of the domain problem. 'Internal' variables are controlled by the programmer, and produce intermediary results. Internal variables may be irrelevant for 'hand solutions', which is a source of error in novice programming (Samurçay, 1989). When a variable plays more than one role in the problem, errors are observed on the program fragment it appears in (Spohrer *et al.*, 1985).

A higher-level concept is required when variables occur in iterative or recursive programs in imperative languages. In this case, the variable is no longer an address (with a value) but needs to be seen as a function of execution, or a sequence of values. At this stage, the algebraic model of variable and equality novices may transfer from physics or mathematics fails as a mental model. A similar interaction has been observed in Prolog between 'logical' definitions and effective values of variables and functions. Novices encounter difficulties in understanding that the specific function defined in a given clause is determined by the instantiation of the inputs (Taylor and du Boulay, 1987).

This feature is closely related to difficulties teachers frequently notice when trying to clarify the difference between local and global variables, although one of the requirements for computer literacy is the learning of the use of procedures and functions to write modular programs. What is masked behind the distinction between local and global variables is the notion of context. Its acquisition calls for a thorough understanding of the 'notional machine'. Understanding backtracking as well as full recursion are related to the acquisition of this complex notion.

Dupuis and Guin (1989) have analysed how students use coding variables in a complex programming task that introduces recursive procedures in Logo. Two modes of coding were observed: a 'descriptive' code (different objects correspond to independent different variables) and an 'analytic' one (expressions with a single variable can be used to represent various related objects). The choice of mode is dependent on the mental model students generate of the computer device. In the descriptive mode coding variables are managed by the programmer, and are linked to an executive model. In the analytic mode, computing the values of variables is under computer control. This mode is related to a more static model of programming and is the only mode compatible with recursivity.

3.4 Programming methods

A method can be defined as an explicit aid for strategy research and management in solving problems of a given class. Implementing a method is dependent upon the relationship between type of problem and programming paradigm, and on the programmer's knowledge. Methods can provide help on various subtasks in program design (from the initial problem analysis to program documentation) and at various levels of problem decomposition (see Chapter 4.2). Up to now programming methods have only been used in training professional programmers, although there are needs

for programming methods even in early acquisition. Van Someren has stressed this point for languages like Prolog which are not process oriented (van Someren, 1985). Existing methods share a number of points. All prescribe a top-down strategy which necessitates working on high-level before low-level structures. In terms of problem solving they are either prospective (analysis oriented by input data) or retrospective (analysis oriented by results). A recent overview of programming methods (Rogalski *et al.*, 1988) indicates that this particular area is characterized by its disproportionately low empirical data with respect to the high number of unanswered questions.

Studies on the use of methods by beginner programmers (Hoc, 1983; Morais and Visser, 1987) show that 'top-down' structured methods are difficult to use because beginners' spontaneous strategies are based on mental execution (from the input data). Rist (1986) reports that in the early stages of learning programming, the plan structures used by novices are action oriented (rather than object oriented as in expert plan structures). Even programmers familiarized with structured programming resort to decomposition oriented by processing and linked to mental execution in cases of increased problem difficulty (Hoc, 1981; Ratcliff and Siddigi, 1985). Professional programmers may also experience difficulty in using aids for program design because they lack sufficiently rich and flexible data structures to handle the requirements of a retrospective strategy (Hoc, 1988b).

These two points suggest that: (1) the representation and processing dimensions involved in the programming task (Figure 2) both enter into the issues surrounding teaching and learning methods; (2) an overly dominant dynamic model of programming can be an obstacle to the use of existing aids including models. Introducing models early on in training may help prevent negative reinforcement of the spontaneous computational programming paradigm.

Data from fields other than programming (mathematics and emergency management) indicate that methods can be taught to novice students in certain conditions. With respect to programming, the most critical condition is the existence of sufficient prior knowledge in the student (computer literacy) and use of teaching of situations containing relatively complex problems which call for making choices in data structuring, task organization and task distribution among students. This might involve a transposition of certain properties of 'programming at large' which would encompass the programming task as a whole and would lend meaning to programming methods and tools.

4 Conclusion

Acquiring and developing knowledge about programming is a highly complex process. It involves a variety of cognitive activities, and mental representations related to program design, program understanding, modifying, debugging (and documenting). Even at the level of computer literacy, it requires construction of conceptual knowledge, and the structuring of basic operations (such as loops, conditional statements, etc.) into schemas and plans. It requires developing strategies flexible enough to derive benefits from programming aids (programming environment, programming methods).

Taking the effective device into account calls for the construction of new systems of representation and processing in order to conceptualize the properties of the 'notional machine' and lead to appropriate human-computer interaction in program-

ming. The familiar RPS related to action execution is a precursor that plays both a positive 'productive' role and a negative 'reductionist' one, assigning meaning to initial programming notions and becoming an obstacle when more static representations are needed.

Studies in the field and pedagogical work both indicate that the processing dimension involved in programming acquisition is mastered best. The representation dimension related to data structuring and problem modelling is the 'poor relation' of programming tasks. This reflects the current emphasis on the computational programming paradigm, linked to dynamic mental models, where programs are considered to be dynamic data processing, and procedures and functions are analysed as operations modifying data instead of defining new entities.

More research is needed on the acquisition of adequate 'static' representations, clearly required for the acquisition of functional or relational languages, but also necessary for efficient use of the powerful recursive tool. The recent stress on the necessity of some 'unlearning' by professional programmers testifies to the fact that focusing on dynamic aspects of programming may constitute an obstacle to further acquisitions. The major challenge today lies in the design of new paradigms in casual and professional programmers' training that will enable them to become more aware of the existence of various programming paradigms, more familiar with them, and more flexible in their representations, in order to open up the field of possible programming choices.

References

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

Bonar, J. (1984). *Bridge: an intelligent programming tutor/assistant.* Technical Report. LRDC. University of Pittsburgh.

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

Dupuis, C. and Guin, D. (1989). Gestion des relations entre variables dans un environnement de programmation LOGO. *Educational Studies in Mathematics,* **20,** 293-316.

Gilmore, D. and Green, T.R.G. (1984). Comprehension and recall of miniature programs. *International Journal of Man-Machine Studies,* **21,** 31-48.

Goldberg, A. and Kay, A. (1977). Methods for teaching the programming language Smalltalk. Technical Report, SSL-77-2. Xerox Palo Alto Research Center.

Green, T.G.R. (1980). Ifs and Thens: is nesting just for birds? *Software Practice and Experience,* **10,** 371-381.

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

Hoc, J.-M. (1983). Analysis of beginner's problem-solving strategies in programming. *In* T.R.G. Green, S.J. Payne and D. 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). L'aide aux activités de planification dans la conception des programmes. *Le Travail Humain,* **51,** 323-333.

Hook, K., Taylor, J. and du Boulay, B. (1988). Redo "try once pass": the influence of complexity and graphical notation on novices' understanding of Prolog. Cognitive Science Research Reports No. 112. Brighton: University of Sussex.

Johnson, L.W., Soloway, E. (1983). Proust: Knowledge-based programming understanding. Technical Report. New Haven: Yale University.

Katz, I.R. and Anderson, J.R. (1988). Debugging: an analysis of bug-location strategies. *Human-Computer Interaction,* **3,** 351-399.

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

Kurland, D.M. and Pea, R. (1983). Children's mental models of recursive Logo programs. *Proceedings of the 5th Annual Conference of the Cognitive Science Society.* Rochester, NY: University of Rochester.

Laborde, C., Balacheff, N. and Mejias, B. (1985). Genèse du concept d'itération: une approche expérimentale. *Enfance,* **2-3,** 223-239.

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

Ormerod, T.C., Manketelow K.I., Steward, A.P. and Robson E.H. (1988). The effects of content and representation on the transfer of PROLOG reasoning skills. *Proceedings of the International Conference on Thinking.* Aberdeen.

Pea, R.D. (1986). Language independent conceptual "bugs" in novice programmers. *Journal of Educational Computing Research,* **2,** 25-36.

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

Ratcliff, B. and Siddigi, J.I.A. (1985). An empirical investigation into problem decomposition strategies in program design. *International Journal of Man-Machine Studies,* **22,** 77-90.

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

Rogalski, J. and Hé, Y. (1989). Logic abilities and mental representations of the informatical device in acquisition of conditional structures by 15-16 year old students. *European Journal of Psychology of Education,* **4,** 71-82.

Rogalski, J. and Samurçay, R. (1986). Les problèmes cognitifs rencontrés par des élèves de l'enseignement secondaire dans l'apprentissage de l'informatique. *European Journal of Psychology of Education,* **1,** 97-110.

Rogalski, J. and Vergnaud, G. (1987). Didactique de l'informatique et acquisitions cognitives en programmation. *Psychologie Française,* **32,** 275-280.

Rogalski, J., Samurçay, R. and Hoc J.M. (1988). L'apprentissage des méthodes de programmation comme méthodes de résolution de problème. *Le travail Humain,* **51,** 309-320.

Rosson, M.B. and Alpert, S.R. (1988). The cognitive consequences of object-oriented design. IBM Research Report RC 14191.

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

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

Samurçay, R. (1986). Understanding the cognitive difficulties of novice programmers: a didactical approach. *3rd. European Conference on Cognitive Ergonomics.* Paris.

Samurçay, R. (1987). Plans et schémas de programmes. *Psychologie Française,* **32,** 261-266.

Samurçay, R. (1989). The concept of variable in programming: its meaning and use in problem solving by novice programmers. *In* E. Soloway and J.C. Spohrer (Eds), *Studying the Novice Programmer.* NJ: Lawrence Erlbaum.

Samurçay, R. and Rouchier, A. (1985). De "faire" à "faire-faire": la planification de l'action dans une situation de programmation. *Enfance,* **2-3,** 241-254.

Sime, M., Arblaster, A. and Green, T.R.G. (1977). Reducing programming errors in nested conditionals by prescribing a writing procedure. *International Journal of Man-Machine Studies,* **9,** 119-126.

Soloway, E. and Ehrlich, K. (1984). Empirical studies of programming knowledge. *IEEE Transactions on Software Engineering,* **5,** 595-609.

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

Spohrer, J.C. and Soloway, E. (1986). Analyzing the high frequency bugs in novice programs. *In* E. Soloway and S. Iyengar (Eds), *Empirical Studies of Programmers.* Norwood, NJ: Ablex.

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

Taylor, J. and du Boulay, B. (1986). Studying novice programmers: why they may find learning Prolog hard. Cognitive Science Research Reports No. 60. Brighton: University of Sussex.

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

van der Veer, G. and van der Wolde, J. (1983). Individual differences and aspects of control flow notations. *In* T.R.G. Green, S.J. Payne and D. van der Veer (Eds), *The Psychology of Computer Use.* London: Academic Press.

van der Veer, G., van Beek, J. and Gruts, G.A.N. (1986). Learning structured diagrams. Effects of mathematical background, instruction and problem semantics. *Visual Aids in Programming,* Passau.

van Someren, M. (1985). Beginners problems in learning PROLOG. Technical Report. Department of Social Science Informatics, University of Amsterdam.

Widowski, D. and Eyfert, K. (1986). Representation of computer programs in memory: comprehending and recalling computer programs of different structural and semantic complexity by experts and novices. *3rd European Conference on Cognitive Ergonomics.* Paris, pp. 93-103.

Wiedenbeck, S. (1989). Learning iteration and recursion from examples. *International Journal of Man-Machine Studies,* **30**, 1-22.