Use of Program Schemata in Lisp Programming: an Evaluation of its Impact on Learning*

Mária Bieliková, Pavol Návrat
Slovak University of Technology, Dept. of Comp. Sci. and Eng.,
Ilkovičova 3, 812 19 Bratislava, Slovakia
E-mail: {bielik, navrat}@elf.stuba.sk
WWW: http://www.elf.stuba.sk/~{bielik, navrat}

Abstract. In the paper, we describe our approach and experience with teaching fundamentals of functional programming by program schemata construction and explanation. Program schemata for processing of lists are presented. Our approach reflects our ultimate goal – to support the learning process. As the main result, we report on experiments that allow to judge quite favourably the approach to teaching functional (Lisp) programming when the student learns a set of program schemata and how to apply them.
Keywords: program schema, functional programming, Lisp

1 Introduction

Software engineering as a discipline and a field of study is going through a phase of development that promises to elevate the level of its maturity. There are various efforts aiming at developing standards that would describe both software processes and products. Within the field of software engineering, there are emerging disciplines such as Architecture of Software Systems that generalize, describe, and classify the standard patterns of architectural style [4, 18], design [10, 17], etc. Besides the very active area of design patterns, research in frameworks and programming idioms contributes to the overall effort. The motivation is for software engineering to proceed in gradually becoming a "true" engineering in the more traditional and established sense.

The above outlined efforts overlap in some sense with efforts within programming to identify parts of its knowledge in a more standardized way in form of program plans [21, 5, 6, 14], program schemata [1, 7, 9, 23], program patterns [24, 20], program skeletons, or programming techniques [3, 22, 19]. The motivation of investigating the nature of programming by identifying the relevant knowledge parallels and complements similar efforts at the level of

* The work reported here was partially supported by Slovak Science Grant Agency project No. G1/4289/97.

software systems mentioned above.

In both cases, one possible effect of the achievements in codification of analysis, design, implementation (including programming) experience results in formulating the body of relevant knowledge in a more systematic way. This opens room for teaching the discipline in new, possibly more effective ways [15]. How can teaching programming benefit from these efforts?

One fundamental aspect that determines organizing a programming knowledge hierarchy is the programming paradigm. It is a well known fact that knowledge of one programming language usually greatly facilitates learning another one, provided they both fall into the same paradigm. If they do not i.e., we have the case of an interparadigmal transition, the above need not hold. On the contrary, one can observe a temptation to transfer the programming habits from one paradigm to another. As a consequence, learners may incline e.g., to forming procedural style programs in a functional language.

We have outlined only one out of several issues connected with teaching functional programming. We report on our experience in teaching the subject *Functional and Logic Programming*. We use the programming language Lisp which is a choice that deserves perhaps a word of explanation. The reasons for choosing Lisp were rather pragmatic and we are fully aware of the restrictions that Lisp bears with it when used as a functional language. However, we employ as a rule only those features which allow programming functionally. The learners form programs using standard functions only from a pre-selected set.

Our approach is best characterized as a development, explanation and use of program schemata. This is in full correspondence with the main trend towards finding ways of representing standardised programming knowledge. Our contribution is directed to teaching functional programming using program schemata that had been devised by us.

In the following section an approach to teaching functional programming is outlined. In the third section program schemata for functional processing of lists are explained. We give a hierarchy of schemata which we use in teaching and briefly comment on techniques for schemata combining. The most detailed is the fourth section which reports on our experiments aimed at verifying our hypotheses about suitability of this approach with respect to learning functional programming by evaluating the effectiveness of the learning process. Finally, some conclusions are drawn.

2 The approach

To be able to form relatively large and practically interesting programs in Lisp requires to master only "a few" syntactical constructs. However, it would be most naive to expect that based on that fact, programming in Lisp can be learnt very swiftly and simply. More likely, the opposite is true. To be able to form both elegant and efficient functional (Lisp) programs requires as a rule a considerable programming experience. Additional difficulty is inter-paradigmal transition, as manifested by the problems that experience many our students who had mastered quite extensively the procedural paradigm [13]. Some of them seem to be constrained to the extent that their understanding of the alternative paradigm effectively is blocked (cf. similar experience reported by Joosten et al. [12]).

The subject Functional and Logic Programming is covered in one semester (13 weeks). The relative modesty of its extent forces to concentrate to the fundamentals of functional programming (which constitutes only a part of the subject's contents; besides it, logic programming is treated, too). We aim to help learners understand the essence of this programming paradigm. We also aim to achieve this by a learning process that involves practical problem solving. Again, the subject's extent constraints this aspect as only relatively short programs can be explained or devised. "Larger" programs with several dozens or hundreds of functions are written by students when completing their assignments for subjects *Artificial Intelligence* or *Knowledge Based Systems*.

Our objective is that the students learn to recognize the kinds of problems and their relevance to the structure of the solution, and this not only at the level of the original problem, but at the levels of subproblems as well. When solving problems, basic schemata should be combined according to various programming techniques. The crucial point is to recognize a schema and then to specialize appropriately its generic parts. This is a particular know-how that students should learn. To achieve that, they learn about the kinds of problems that one can encounter most frequently, and they learn about the schemata that correspond most closely to the kinds of problems.

The approach to learning programming as described above is based on proceeding from specific examples to schemata (generalization). After the students master this phase to the extent of being able to formulate useful schemata, they have achieved a level of competence which allows them to make a choice

of a schema and to specialize it accordingly. As Návrat [16] has pointed out, the interplay between generalization and specialization is critically important for the method of programming.

As soon as the students learn to recognize properly the schema which is to be applied in the given situation, the task of forming the function's definition is already quite straightforward. We guide the students to organize their knowledge in form of a mental library to allow a more effective use of schemata when solving more complex problems. This can be achieved by gaining more practice through solving a bigger number of small problems. One of the last assignments is to design and implement an abstract data type (e.g., set, table, array, graph) from its specification. Here, strict adherence to functional style in programming is expected.

This approach to teaching functional programming is very promising. It seems to be especially effective for learning a paradigm which does not have a central role in the curriculum nor in the industrial practice, but which nevertheless is considered vital for deepening the learner's understanding of the fundamental processes of programming. Main advantages of it can be summarized as follows:

- use of program schemata facilitates understanding the basic principles;
- process of acquiring the basic programming skills is speeded up (this fact is very important in our specific case when not one, but two different paradigms are treated in one subject);
- already novices write programs which are better in the sense that they tend to be truly functional; students in fact do not have much chance to try to apply their previously acquired procedural habits as these soon came into conflict with the schemata;
- schemata are a means not only to learn programming knowledge by the student, but also to assess by the teacher the level to which the student has mastered it.

3 Schemata for functional processing of lists

One of the sources of difficulty in learning functional programming is recursion, cf. [11]. To understand it and to be able to apply it appropriately causes problems to many learners. Another difficulty is lack of variables, and consequently avoiding assignments. We therefore put much stress on explaining the

role of recursion thoroughly. We have found the problems related to processing of lists very suitable in this respect.

Our students can be considered novices in programming, especially with respect to the functional paradigm. Therefore we restrict ourselves to problems in which: (i) the only kinds of data are lists including atoms (numbers and symbols); later, we consider also dotted-pairs; (ii) no indirect recursion is involved; (iii) no operations with side effects are involved (such as input/output). Only gradually, when proceeding to solving more complex problems, other schemata are presented (e.g., assignment, input/output, etc.).

3.1 Schema representation

With regard to schema representation, our utmost objective has been simplicity. We deliberately avoid introducing any unnecessary complexity, such as a new, different formalism when new concepts can be satisfactorily represented using a known language, perhaps with only a minor enhancement.

One feature that makes the programming language Lisp different from many other languages is a relatively close distance from syntax to its interpretation. Assuming that this property holds one can make conclusions on the semantical similarity of two functions by comparing their syntactic structure.

To keep the matters simple, we decided to write schemata in a language similar to Lisp. We shall use two kinds of variables: first order variables (Lisp function arguments) and schemata variables (variable function symbols). Variables begin with a capital letter. Schemata that express classes of functions with various numbers of arguments or forms are represented using an additional optional symbol. This symbol allows to mark place for an argument or a form that may but need not occur in a definition of a particular function of the class expressed by the schema. The optionality is represented by brackets. Optional argument or form can be generalized by prefixing a schema variable with &. For example, in the schema

```
(defun Map (List  &V1)
       (cond ((null List) nil)
             ( t   (cons (Transform (car List)  &V2)
                         (Map (cdr List)  &V3)))  ))
```

the expression $\&V_1$ can be replaced by a sequence of arbitrary number of arguments (including none).

3.2 Hierarchy of schemata

The basis for a particular schema is a typical structure of functions to solve some class of similar problems. Moreover, we have defined a catalogue which includes several similar schemata. The schemata can be organized into hierarchies. We have identified two important groups of schemata, according to two general kinds of processing of lists: (i) list processing at their top level only; (ii) list processing at all levels.

One of the important issues in teaching how to process lists is the order in which the learners become acquainted with the different schemata. In both groups, we define pairs of similar problems e.g., substituing a symbol in a list by another one at the top level of the list, and similarly, substituing a symbol in a list by another one at all levels of the list. Our experience for several years has been that the students can cope better with the kind of problems from the former class. In other words, to program processing of lists at all levels seems to be more difficult for our students. That is why we start teaching the former class and proceed to processing at all levels only later.

In Fig. 1, a hierarchy of schemata for processing of lists at the top level only is outlined. The schemata identified in Fig. 1 can be found in Appendix. Thicker lines are used to depict schemata that were actually taught during the semester that we report about here.

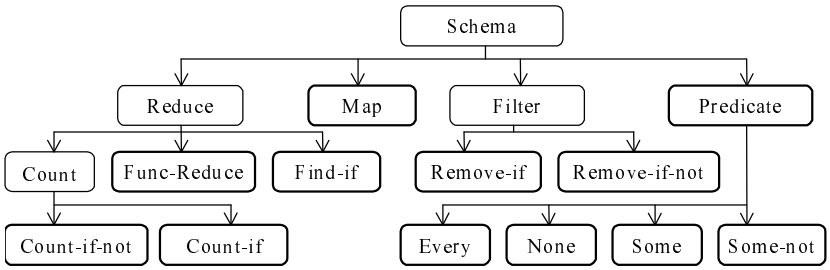

Fig. 1: The hierarchy of schemata for processing of lists.

In the case of processing of lists at all levels, the classes of problems are the same. However, in many cases of processing of lists at all levels it is advantageous to consider data not to be lists, but arbitrary s-expressions. In such a case the argument could be considered to be a binary tree rather than a list. Each node in the binary tree is either an atom or a dotted-pair. A node that is a dot-

ted-pair has two successors (*car* and *cdr*). Atoms are leaves of the tree. Each function must either process an atom (making use of the fact that the empty list *nil* is an atom, too) or it must recursively call itself to *car* part of the dotted-pair and the *cdr* part of the dotted-pair (and consequently, of the list).

We grouped such schemata into one subgroup headed by the node *DS-exp*. It is a parent for *Reduce, Map* and *Predicate* nodes in the hierarchy. A general schema for processing of a dotted-pair can be written:

```
(defun DS-exp (S-exp  &V1)
        (cond     &((Test S-exp  &V2) Value)
                  [ ((atom S-exp)   End-value) ]
                    ( t    (Combiner  (DS-exp (car S-exp)  &V3)
                                      (DS-exp (cdr S-exp)  &V4)))  ))
```

In all the schemata we use a generalized optional argument (in form of $\&V_i$). It expresses an option that the schema instance (i.e., a function definition) and similarly also expressions which are applications of that function can have additional arguments (cf. the schemata *Map* or *DS-exp* above). We assume that having in mind the main purpose of the schema (i.e., processing of lists), these additional arguments usually play an auxiliary role. Their incorporation can thus be reduced to a syntactical manipulation. Therefore, we shall not include them in schemata.

The second aspect that must be taken into account when explaining schemata is the function definition's syntactical structure itself. We have formed a set of schemata which cover three important points of a recursive function definition:

1. a stopping condition (single test recursion, multiple test condition);
2. a method of performing the recursive step (tail recursion, fat recursion);
3. a way of reducing the original task to a simpler task (monotonic recursion, nonmonotonic recursion, single parameter reduction, multiple parameter reduction, single recursive call, multiple recursive call).

The schemata are described in full detail in [2]. We note that when explaining the processing of lists at the top level, we mention to students also schemata with the tail recursion.

3.3 Schemata combining and programming techniques

The crucial thing in applying schemata in programming is the ability to recognize a situation when a combination of schemata is desirable. Let us consider an example when those elements of a list which fulfil a condition are to be transformed and the other ones simply deleted. The solution is to combine a schema for a list mapping and a schema for a filter. The students should be explained a proper way of schemata combining. The process is not always a simple one. One solution is to introduce a new schema that incorporates both the schemata in question. We are firmly convinced (cf. also Sollohub [20]) that the programming experts in fact use such schemata. However, an introductory course in functional programming should concentrate on those basic schemata from which the other more complicated ones are formed. Number of schemata that a student is presented and expected to learn within the limited period of time must be also quite limited.

When devising schemata, one important methodological tool that has proven invaluable is generalization, and specialization as its inverse [1]. Known examples of typical specific solutions are generalized. Several such generalized skeletons are merged into one schema after possibly abstracting away some aspects in some of them so that a common structure can be identified. As a result, we have a schema that is rather general and abstract. To apply such a schema, a specialization (defining a special value for a feature – inverse of generalisation) and a concretization (adding a feature – inverse of abstraction) frequently takes place. Moreover, the schemata are combined.

4 Experiments

It has been our assumption that teaching functional programming with schemata is more efficient and that the students improve much faster (especially during the early stages of the programming process). In order to either prove or refute this assumption, we proposed a set of experiments. The experiments build on our earlier experience with teaching functional programming with schemata (limited to a few syntactically oriented ones such as structural variants of a recursive function definition) which nevertheless improved the speed with which the students mastered the basic practice in functional programming.

4.1 The method of experimentation

There are several issues that should be considered: what the students know, or what can be assumed about their knowledge; experimental procedure; kinds of problems to be solved and their assignment to students; which attributes to evaluate; the hypotheses.

The students who took part in the experiment can be divided according to two criteria. First, whether they were given an explanation about the schemata. Second, whether they had a catalogue of schemata available during the test. The first criterion is a consequence of our University's statutes, according to which attending lectures is not obligatory for the students. Some students were present at the lectures when schemata were introduced and explained to a considerable detail. The complement of the class can be assumed to be practically ignorant (the version of our textbook [2] available at that time did not include program schemata for processing of lists).

A subgroup of students was allowed to use the catalogue of schemata (elaborated by the lecturer). The catalogue comprised of several schemata (six to eight). By applying them, some assignments could be solved straightforwardly, some could be solved after schemata were combined, and some after schemata specialization. The catalogue included the schemata for processing of lists using both the fat recursion and tail recursion. In order to eliminate possible influence of other factors, schemata are named in such a way that the names do not reveal the essence of the processing they express. For example, a schema for selectively counting those elements of a given list at the top level which posses a given property was presented to the students in two forms:

```
(defun Schema-1 (List)
    (cond    ((null List) 0)
             ((Test (car List)) (+ 1 (Schema-1 (cdr List))))
             ( t  (Schema-1 (cdr List)))   ))

(defun Schema-2 (List)   (Schema-2-aux List 0))
(defun Schema-2-aux (List Aux)
    (cond    ((null List) Aux)
             ((Test (car List))  (Schema-2-aux (cdr List) (+ 1 Aux)))
             ( t  (Schema-2-aux (cdr List) Aux))   ))
```

Each student was solving two tasks. Solving them was a two-phase process. First, the student was expected to attempt to solve them with pencil and paper

only. The time alloted for this phase was 50 minutes. Next, the student was expected to implement the solution (in the remaining time to 100 minutes). The number of assignments was chosen with respect to the level of their difficulty.

We divided the assigned problems into two groups: (i) processing of lists at the top level, and (ii) processing of lists at all levels. Solutions to assigned problems from both groups could be formulated using the schemata from the hierarchy shown above. Each student was assigned two problems from one group.

In the experiment, the following characteristics were evaluated:

1. correctness of the solution, taking into account also how application of a schema contributed to it, expressed in marks out of two (maximum) for each problem;
2. role of computer in solving the problem (ideal interpreter in student's mind vs. real interpreter implemented in computer);
3. the kind of recursion used (fat, tail).

4.2 Experimental results

There were 105 students who took part in the experiments, all of them level 3 students of our baccalaureate Informatics course in the software engineering track. Distribution of the whole set of students into the four categories is shown in Tab. 1.

		Schemata available?	
		yes	no
Explanation received?	Yes	**31**	**21**
	no	**19**	**34**

Tab. 1. Distribution of students into groups.

Perhaps the most important question to answer when trying to evaluate the approach to learning functional programming based on schemata is concerned with a positive identification of their influence on students' learning process. In our formulation, does the use of schemata in learning functional programming influence results of novice learners? Can it speed up the process of learning the paradigm? Our first hypothesis is: *The results when using schemata are better.*

In Tab. 2a and 2b we display the overall results of tests of students from all the four subgroups (cf. Tab. 1) either based on their marks out of 4, i.e., marks from the range <0,4>, or expressed as a relative portion (per cent) of correct solutions within each subgroup.

Schemata available?

Explanation received?		Yes	no
	yes	**3.52**	**3.26**
	no	**2.61**	**2.27**

a

	yes	no
yes	**83%**	**58%**
no	**58%**	**38%**

b

Tab. 2. Overall results of tests.

The results in Tab. 2 allow some conclusions. Best results were achieved by the students who had a prior information on schemata and at the same time they had the catalogue of schemata at hand during the test. 83% out of them produced correct solutions. From the correctness point of view, accidentally the subgroup of students who heard the presentation and explanation but did not have the catalogue of schemata at hand during the test performed precisely equally as those students who had the catalogue of schemata at hand during the test but did not hear their presentation and explanation (both scored 58%). In the overall ranking, however, the students who learned about the schemata beforehand, even if they did not have the catalogue of them at hand during the test, performed better. Moreover, most of them were able to identify the schemata which they had applied. We feel endorsed in drawing a conclusion that their knowledge of the schemata was instrumental in achieving better results. The difference between them and those students who had not heard the explanation before the test once again supports the hypothesis about the positive influence of schemata on learning functional programming.

There is one problem which is relevant for most of our students. In their programming career, at the moment when they are first confronted with the functional paradigm, they have already gathered some experience with the procedural paradigm. Not only their first programming course was in C and they were also using some assembly language and Pascal later, but also most of them work in part time jobs for the local software industry where the use of the procedural paradigm is still quite frequent. Applying the *tail recursion* when

programming the processing of a list by way of introducing an auxiliary parameter which serves as a local variable (cf. the *Schema-2* above) is a technique close in spirit to the procedural paradigm (it amounts in fact to an assignment). It can be used also in a functional programming language, but it is not considered to be conform with the functional style of programming. However, based on their procedural past, we hypothesized that the students, when having available both options, choose the tail recursion. Similar assumption was made by Gegg-Harrison, who chose the Prolog predicates with the tail recursion to be the first ones to learn using schemata in learning logic programming [8].

Evaluation of our experiment shows that from 105 students, only 5 students tried to apply the schema with the tail recursion, and only 2 of them successfully. The hypothesis that students prefer the tail recursion has been refuted.

Another question which is frequently discussed with respect to learning programming concerns the *role of computer* and importance of its direct usage when solving a programming problem. A quite frequent pattern of a student's problem solving procedure includes an initial sketch of a design of the solution on a paper and then an interaction with a computer during which implementation of the solution is attempted to be completed. We shall refrain in this paper from commenting on such a pattern with respect to its appropriateness etc. We wanted to find out whether the peculiar variety of the trial and error method is applied because of greater comfort or because of inability to devise a correct solution by using paper and pen only.

We have found out that 54% of all students had the correct solution already sketched on a paper. The remaining students had not and most of them could not improve the original sketch during computer implementation. Their modifications concerned usually some minor changes like amending of completing tests in *cond* forms. The likely conclusion is that in most cases, direct usage of a computer when solving a programming problem does not have a clearly positive effect.

Another hypothesis which we attempted to get confirmed or refuted was the one related to the question if problems of processing of lists at all levels are harder to solve for learners than processing of lists at the top level only. Our hypothesis is: *processing of a list at all levels is more difficult to program for learners*. We based our hypothesis on our several years' experience.

All the problems of processing of a list such that they preserve the structure of the list can be generalized to processing of an arbitrary s-expression (i.e., an atom or a dotted-pair). In such a case, the general schema *DS-exp* should be used. Other (more special) schemata are listed in Appendix.

Having been explained these schemata, the students could apply them equally (easily) as the schemata for processing of lists at the top level. The choice of problems did not influence the results.

Now, in the experiments reported here we specifically evaluated the success rate of students for the two classes of problems. In Tab. 3, the results show that the overall average of marks for solving problems of processing of a list at all levels is even slightly higher than of processing of a list at the top level only. However, when we adjusted the results by deleting the total failures, the relation became reversed in a very slight way. We can conclude that students perform equally well for both kinds of problems.

	overall results	Adjusted results (total failures not considered)
	ϕ marks	ϕ marks
top level only	**2.79**	**3.27**
all levels	**2.93**	**3.14**

Tab. 3. Success rates of solving processing of lists at the top level and at all levels.

One interesting result with respect to processing of a list at all levels emerged after tracing how successful were those students who made use of the idea of considering a binary tree instead of a list. From those students who applied the corresponding schema only one student (0.9 per cent) made an error. All the rest of them devised correct solutions. On the other hand, when students applied the idea known from processing of a list at the top level (process the empty list, depending on the test applied to the first element process the first element and process the rest of the list), their error rate increased significantly. The students forgot very often to process the first element of the list, or they prescribed to process without testing if it was a list (but their function could not process an atom).

correct solution

		yes	no
the schema list = binary tree applied	yes	**33.6%**	**0.9%**
	no	**23.6%**	**41.8%**

Tab. 4. Solving processing of lists at all levels.

Finally we wish to summarize briefly the errors that occured most often during the experiment. We wish to emphasize that many kinds of errors which were commited mainly due to acquired habits from other programming paradigms (especially the procedural one) could be prevented by restricting the set of functions of the language Lisp that were authorized to use. For example, the assignment (*set*, *setq*, *setf*) was not allowed.

The errors that we list below are the commonest ones according to our experience. We are convinced that with even more emphasis on explaining and practising with schemata, the errors will be still reduced in the future. The errors are:

- not a completed stopping condition;
- not recursing into the rest of the list (when processing of the list at all levels, students frequently considered recursion for the first element only and forgot completely about the rest);
- not recursing into the first element of the list (when not regarding the list to be a binary tree);
- inappropriate constructor;
- complete conditions in each branch;
- conditions that can never be true;
- use of the form *cond* in connection with the logical functions *and, or*, i.e. (cond ((and ……..) t) (t nil))
- nesting *cond* forms into several levels.

5 Conclusions

In the paper we present the program schemata which we use in teaching functional (Lisp) programming and the way how to represent them. The method is based on building a catalogue of program schemata (ultimately to be stored in learners' mind). This is the reason why the simplicity of program schemata representation is crucial. We realise that formal methods for representing prog-

ram schemata, such as those presented in [7, 23] for logic programming, are important and inevitable when exploring properties of the created solution, its correctness, etc. On the other hand, when teaching novices (especially when very short period of time is available), very simple set of schemata should be devised. To speed up learning, ready to use knowledge should be presented to learners. Our program schemata were devised with this point in mind. They are purposely "incomplete" in the sense that their specialization cannot be for some problems completed at the syntactic level.

The very basics of Lisp programming is covered in 2 three hour lectures (the language) and 3 three hour lectures (programming with schemata). The lectures are by no means sufficient to learn programming so a big stress is put on exercises. An important benefit is that the students gain better fundamentals and less students remain "totally lost". In discussions, their questions are more relevant and show more insight.

We have been aware of possible shortcomings of the use of schemata in teaching programming, as voiced for example by Bowles and Brna [3]:

- *when there are too many schemata or plans, it becomes difficult to learn them.* We use only a limited number of them and focus to master the fundamentals faster and better;
- *use of schemata supports the tendency to concentrate to structural properties of the solution.* Therefore, we complement the teaching by discussing programming techniques, too (similarly to Bowles and Brna);
- *there exists a danger that the student has not understood the essence of the schema, but is simply able to apply syntactically one.* We guide the students to solve also more complex problems where a pure syntax based application cannot lead to a correct solution. Schemata (such as those that we mentioned in this paper) help form a bottom layer of the program. Frequently, schemata must be combined using various programming techniques. Moreover, the schemata are not available directly on a paper, but the students are expected to achieve a level of mastery when they are able to devise new schemata and maintain known schemata in mind.

Our approach is based on the idea that by arriving swiftly to a correct un-

derstanding of how to solve a rather modest number of basic classes of problems, the students will be able to solve also other more or less similar problems. We have experimentally verified the fact that the novices are able to devise correct functional programs in a shorter time using schemata when comparing to the "traditional" approach. The use of schemata makes teaching functional programming more effective. One interesting result is also that when learning with schemata, students have less difficulty with processing of a list at all levels.

Appendix – program schemata used in experiments

Processing of lists at the top level only	**Processing of lists at all levels**
(defun *Map* (List) (cond((null List) nil) (t (cons (*Transf* (car List)) (*Map* (cdr List))))))	(defun *DMap* (S-exp) (cond((*Test* S-exp) (*Transf* S-exp)) [(atom S-expr) S-exp)] (t (cons (*DMap* (car S-exp)) (*DMap* (cdr S-exp))))))
(defun *Reduce* (List) (cond((null List) *Neutral-Value*) (t (*Reduction* (car List) (*Reduce* (cdr List))))))	(defun *DReduce* (S-exp) (cond((*Test* S-exp) S-exp) ((atom S-exp) *Neutral-Value*) (t (*Reduction* (*DReduce* (car S-exp)) (*DReduce* (cdr S-exp))))))
(defun *Count-if* (List) (cond((null List) 0) ((*Test* (car List)) (+ 1 (*Count-if* (cdr List)))) (t (*Count-if* (cdr List)))))	(defun *DCount-if* (S-expr) (cond((*Test* S-expr) 1) ((atom S-expr) 0) (t (+ (*DCount-if* (car S-expr)) (*DCount-if* (cdr S-expr))))))

```
(defun Count-if-not (List)
  (cond((null List)  0)
       ((Test  (car List))
            (Count-if-not (cdr List)))
       ( t  (+ 1
            (Count-if-not (cdr List)))
       ) ))
```

```
(defun DCount-if-not (S-exp)
  (cond((Test S-exp)  0)
       ((atom S-exp)  1 )
       ( t ( +
           (DCount-if-not (car S-exp))
           (DCount-if-not (cdr S-exp))
       )) ))
```

```
(defun Find-if (List)
  (cond((null List)  Fail-Value)
       ((Test  (car List))  (car List))
       ( t  (Find-if (cdr List)))  ))
```

```
(defun DFind-if (S-exp)
  (cond((Test  S-exp)  S-exp)
       ((atom S-exp)  nil)
       ( t  (or  (DFind-if (car S-exp))
                 (DFind-if (cdr S-exp)))
  ) ))
```

```
(defun Every (List)
  (cond((null List)  t)
       ((Test (car List))
                  (Every (cdr List)))
       ( t  nil)  ))
```

```
(defun DEvery (S-exp)
  (cond((Test  S-exp)  t)
       ((atom S-exp)  nil)
       ( t  (and (DEvery (car S-exp))
                 (DEvery (cdr S-exp)))
       ) ))
```

```
(defun Some (List)
  (cond((null List)  nil)
       ((Test (car List))  t )
       ( t  (Some (cdr List)))  ))
```

```
(defun Dsome (S-exp)
  (cond((Test  S-exp)  t)
       ((atom S-exp)  nil)
       ( t  (or (DSome (car S-exp))
           (DSome (cdr S-exp))))  ))
```

```
(defun None (List )
  (cond((null List)  t)
       ((Test (car List))  nil )
       ( t  (None (cdr List)))  ))
```

```
(defun Dnone (S-exp)
  (cond((Test S-exp)  nil)
       ((atom S-exp)  t)
       ( t  (and  (DNone (car S-exp))
            (DNone (cdr S-exp))))  ))
```

```
(defun Some-not (List)
  (cond((null List)  nil)
       ((Test (car List))
                 (Some-not (cdr List)))
       ( t  t)  ))
```

```
(defun DSome-not (S-exp)
  (cond((null S-exp)  nil)
       ((Test S-exp)  nil)
       ((atom S-exp)  t)
       ( t (or
            (DSome-not (car S-exp))
            (DSome-not (cdr S-exp)))
       ) ))
```

```
(defun Remove-if (List)
  (cond ((null List) nil)
        ((Test (car List))
                (Remove-if (cdr List)))
        ( t  (cons (car List)
                (Remove-if (cdr List)))
        ) ))
```

```
(defun DRemove-if (List)
  (cond ((null List)  nil)
        ((Test (car List))
            (DRemove-if (cdr List)))
        ((atom (car List))
            (cons (car List)
              (DRemove-if (cdr List))))
        ( t  (cons
              (DRemove-if (car List))
              (DRemove-if (cdr List)))
        ) ))
```

References

1. Bieliková, M. and P. Návrat (1997). A schema-based approach to teaching programming in Lisp and Prolog. In P. Brna and D. Dicheva (Eds.), Proc. PEG'97 Int. Conf., pp. 22-29.
2. Bieliková, M. and P. Návrat (1997). Functional and logic programming. STU Bratislava. 243pp. (in Slovak)
3. Bowles, A. and P. Brna (1993). Programming plans and programming techniques. In P. Brna, S. Ohlsson and H. Pain (Eds.), Proc. World Conf. on Artificial Intelligence in Education. AACE Press. pp. 378-385.
4. Buschmann, F., R. Meunier, P. Sommerhand and M. Stal (1996). Pattern oriented software architecture: A System of patterns. John Wiley & Sons. 457pp.
5. Davies, S. and A.M. Castell (1993). Embodying theory in intelligent tutoring systems: an evaluation of plan-based accounts of programming skill. Computers and Education, 20(1), 89-96.
6. Davies, S. (1996). The role of external information sources in computer programming - a framework for understanding programming strategies. In P. Vanneste, K. Bertels, B. De Decker and J.M. Jaques (Eds.), Proc. 8th Annual Workshop Psychology of Programming, pp. 167-173.
7. Flenner, P., K.K. Lau and M. Ornaghi (1997). Correct-schema-guided synthesis of steadfast programs. In M. Lowry and Y. Ledru (Eds.), Proc. IEEE Conf. on Automated Software Engineering. IEEE Press.
8. Gegg-Harrison, T.S. (1991). Learning Prolog in a schema-based environment. Instructional Science, 20, 173-192.
9. Gegg-Harrison, T.S. (1996). Extensible logic program schemata. In I. Gallagher (Ed.), Proc. of the 6th Int. Conf. on Logic Program Synthesis and Transformation. Springer-Verlag.

10. Gamma, E., R. Helm, R. Johnson and R. Vlissides (1995). Design patterns: elements of reusable object-oriented software. Addison-Wesley. 395pp.
11. Haynes, S.M. (1995). Explaining recursion to the unsophisticated. SIGSCE Bulletin, 27(3), 3-6.
12. Joosten, S. (Ed.), K. van den Berg and G. van der Hoeven (1993). Teaching functional programming to first-year students. J. Functional Programming, 3(1), 49-65.
13. Khazaei, B., J. Siddiqi, A. Harnack, R. Osborn and C. Roast (1996). Further investigations into transfer effect of moving from procedural to logic programming. In P. Vanneste, K. Bertels, B. De Decker and J.M. Jaques (Eds.), Proc. 8th Annual Workshop Psychology of Programming, pp. 25-42.
14. Koziak, I. (1997). Analyzer: automated deriving of descriptions of student programs. In P. Brna and D. Dicheva (Eds.), Proc. PEG'97 Int. Conf., pp. 47-54.
15. Návrat, P. and V. Rozinajová (1993). Making programming knowledge explicit. Computers and Education, 21(4), 281-299.
16. Návrat, P. (1996). A closer look at programming expertise: Critical survey of some methodological issues. Information and Software Technology, 38(1), 37-46.
17. Pree, W. (1994). Design patterns for object-oriented software development. Addison-Wesley. 268pp.
18. Shaw, M. and D. Garlan. (1996). Software architecture – perspectives of an emerging discipline. Prentice Hall. 243pp.
19. Sterling, L.S. and M. Kirschenbaum (1993). Applying techniques to skeletons. In J.M. Jacquet (Ed.), Constructing Logic Programs, John Wiley, pp.127-140.
20. Sollohub, C. (1991). Programming templates: professional programmer knowledge needed by the novice. Computer Science Education, 3, 255-266.
21. Soloway, E. and K. Ehrlich (1988). Empirical studies of programming knowledge. IEEE Trans. on Software Engineering, 10(5), 595-609.
22. Vasconcelos, W.W. (1994). Designing Prolog programming techniques. In Proc. of the 3rd Int. Workshop on Logic Program Synthesis and Transformation. Springer-Verlag.
23. Vasconcelos, W.W. and N.E. Fuchs (1995). Prolog program development via enhanced schema-based transformations. In Proc. of 7th Workshop on Logic Programming Environments.
24. Weber, G. and A. Möllenberg. ELM programming environment: A tutoring system for Lisp beginners. In K.F. Wender, F. Schmalhofer, H.D. Böcker (Eds.), Cognition and Computer Programming. Albex. pp. 373-407.

Date of submission: 26 November 1997

MÁRIA BIELIKOVÁ was born in Slovakia in 1966. She received her Ing. (MSc.) in 1989 from Slovak University of Technology in Bratislava and her CSc. (PhD.) degree in 1995 from the same university. She is an assistant professor at the Department of Computer Science and Engineering at Slovak University of Technology in Bratislava, where she is a member of the Intelligent support of software development research group. Her research interests include functional and logic programming, knowledge engineering, software development and management of versions and software configurations. She is a member of IEEE Computer Society and the Slovak Society for Informatics.

PAVOL NÁVRAT was born in 1952 in Bratislava, Slovakia. He received his Ing. (MSc.) summa cum laude in 1975, and his CSc. (PhD.) degree in Computing Machinery in 1983 both from Slovak University of Technology in Bratislava. He has been with Slovak University of Technology since 1975. In 1989-1990, he spent 5 months as a visiting associate professor with the Department of Informatics, University of Athens, Greece. In 1992-1994, he joined as an associate professor the Department of Mathematics, University of Kuwait for two academic years. Since 1996, he is a full professor of Computer Science and Engineering. He (co-)authored two books and numerous scientific papers. His scientific interests include automated programming and software engineering, knowledge-based methods for assistance in programming and software development, as well as other topics in Software Engineering. He is a member of the IEEE and its Computer Society, the American Association for Artificial Intelligence, and the Slovak Society for Informatics. He is also a member of the Association for the Advancement of Computing in Education, for which he volunteers as a regional liaison officer.