Learning Programming in Prolog Using Schemata [*]

Mária Bieliková, Pavol Návrat
Slovak University of Technology, Dept. of Computer Science and Engineering,
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 logic programming by program schemata construction and explanation. Program schemata construction is helped by presentation of a few examples. After a particular program schema has been introduced, students are required to instantiate the schema both as a writing exercise and afterwards as a programming exercise. We report on our experiments aimed at verifying our hypotheses about suitability of this approach to learning programming by evaluating the effectiveness of the learning process.

Key words: logic programming, Prolog, program schemata

Introduction

Nowadays, learning programming is one of the basic educational activities of any student of computer or software related curriculum. Besides the "classical" procedural programming paradigm, and the "inevitable" object-oriented paradigm, learning programming functionally and programming logically is a standard component of the curriculum offered by Slovak University of Technology in Bratislava. In the paper, we describe our approach and experience with teaching fundamentals of logic programming by program schemata construction and explanation.

Our approach can perhaps best be expressed as designing, explaining, and applying program schemata. We rely on explaining a relatively large number of examples of predicates. As it is frequently the case, Prolog is our language of choice despite its known shortcomings when considered as a pure logic programming language.

Program schemata (generalizations), programming techniques or plans (abstractions) all represent various kinds of programming (meta-) knowledge. We find it important to make the knowledge explicit for the learner [9, 3, 10]. A schema-based approach facilitates learning of general programming techniques by requiring the student to instantiate templates which are generalizations of a program to solve the assignment. This instructional technique has proven useful when recursion (a very difficult concept for most novice programmers [7]) is taught. It can be argued that one of the causes of difficulties is the lack of a structured programming construct for recursion in programming languages like Prolog and Lisp [1]. A schema-based approach enables to alleviate some of these difficulties by providing novice programmer with a set of standard structures (or program schemata) with which one is able build complex and interesting recursive Prolog programs [6].

In the rest of the paper, we describe the method of teaching logic programming together with some of the schemata that we employ in teaching. We report on our experiments aimed at verifying our hypotheses about suitability of this approach with respect to learning programming by evaluating the effectiveness of the learning process.

Method of teaching

To be able to form relatively large and practically interesting programs in Prolog requires to master only "a few" syntactical constructs. However, it would be most naive to expect that based on that fact, programming in Prolog can be learnt very swiftly and simply. More likely, the opposite is true [2]. To be able to form both elegant and efficient logic (Prolog) 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. Some of them seem to be constrained to the extent that their understanding of the alternative paradigm effectively is blocked (cf. similar experience reported in [8]).

We teach logic programming as a part of the subject *Functional and Logic Programming*. The

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

subject is covered in one semester (13 weeks). The relative modesty of its extent forces to concentrate on the foundations of logic programming (which constitutes only a part of the subject's contents; besides it, functional 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 predicates are written by students when completing their assignments for subjects *Artificial Intelligence* or *Knowledge Based Systems*.

Respecting the above considerations, our approach is an attempt to convey the programming experience to the students in a most useful i.e., comprehensive, effective and complete way. We realise that without compressing in time the experience gathering phase, the student would not be able even to approach the desired level of mastery of Prolog programming in such a short period. What is urgently needed is a means to express the programming knowledge in a possibly standardized way, so that the learners can acquire the accumulated experience via a shortcut route. We propose program schemata as a way of expressing standard programming solutions. As a consequence, we hope to achieve more balance between the inventive dimension of programming (i.e., programs are original programmer's inventions), and the engineering dimension of programming (i.e., programs are original technical solutions developed from standardized skeletal solutions by applying standardized procedures and respecting technical standards).

Our basic assumption in teaching logic programming (but similarly also functional programming) is that many programs (e.g., those which are concerned with list processing) have a similar structure so that they can be considered instances of some program schema. Our approach fosters acquiring basic programming techniques. The learner forms instances of schema (replaces or refines generic parts of a schema) and in such a fashion solves the assigned problems.

Schemata for processing of lists

Our students can be considered novices in programming, especially with respect to the logic paradigm. Therefore we restrict ourselves to problems in which no indirect recursion is involved or 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., meta-predicates, control predicates such as cut, input/output, etc.).

Schemata are based on a typical structure of predicates for certain classes of problems. One of the important issues in teaching them is the order in which the learners become acquainted with the different schemata. In our approach, we distinguish two basic classes and proceed accordingly in two steps:

1. list processing at its top level only,
2. list processing at all levels.

In both classes, we define pairs of similar problems e.g., substituting a symbol in a list by another one at the top level of the list, and similarly, substituting 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 program schemata for processing of lists is outlined. The schemata identified in Fig. 1 can be found in Appendix. In the case of processing of lists at all levels, the classes of problems are the same. We use the following general schema of processing of lists at all levels:

$\textit{DList}([H|T]\ \&V_1)$:-
    $\textit{DList}(H\ \&V_2)$,
    $\textit{DList}(H\ \&V_3)$.
$\textit{DList}(E\ \&V_1)$:-
    $\textit{Process}(E\ \&V_2)$.

How the program schema is represented? 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. We use two kinds of variables: programming language variables (Prolog variables) and schema variables (variable predicate symbols). Variables begin with a capital letter. Schemata that express classes of predicates with various numbers of clauses or various numbers of conditions in clauses are represented using an additional optional symbol. This symbol allows to mark place for a

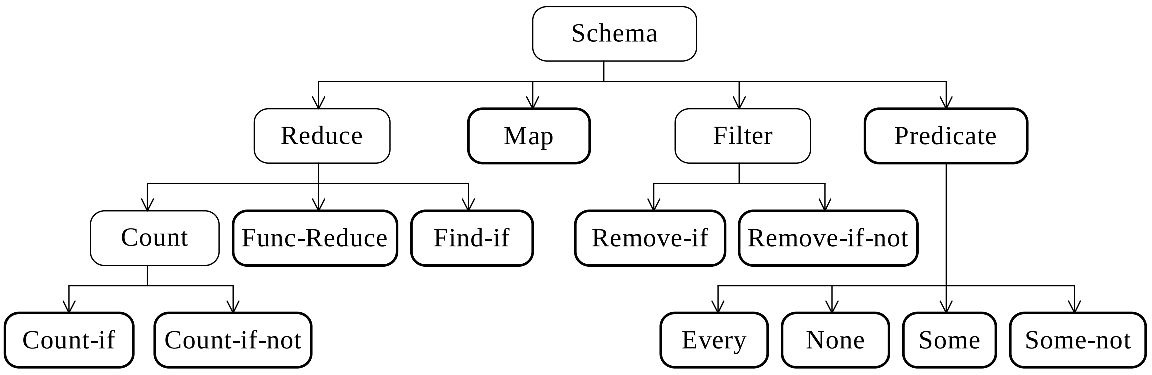

Fig. 1. Hierarchy of program schemata for processing of lists.

clause or a condition that may but need not occur in a definition of a particular predicate of the class expressed by the schema. The optionality is represented by brackets. Optional arguments in terms are represented by the symbol &. For example in the above schema the occurences of the expression &V_1 can be replaced by a sequence of any number of arguments (including none). The condition in the third clause is optional.

In all the schemata we use a generalised optional argument (in form of &V_i). It expresses an option that the schema instance (i.e., the particular predicate) can involve additional arguments. Our position is that such additional arguments are not crucial from the point of view of the programming knowledge that is expressed by the schema. To add them is more or less a matter of routine. Therefore, we shall not mention them from now on.

Experiments

To find out more about the effectivenes of our approach, we conducted some experiments with the spring of 1997 class. It is useful to repeat here the fact that before coming to learn programming in Prolog, the students spend the first part of the subject learning functional programming (Lisp). The reason why we use this particular order has been more or less pragmatic. However, we note that to discuss and ask about the order in which a learner studies different programming paradigms is a legitimate question in its own right.

Experimentation method

There were two groups of students. Only to one of them a catalogue of program schemata was made available during the test. The catalague included several schemata (e.g., five for one particular problem). Some problems could be solved using some of the schemata directly. There were other problems that could be solved by combining some schemata from the catalogue, or by specialising them. In order to eliminate possible influence of other factors, schemata are named so that the names do not reveal the essence of the processing. For example, the schema for counting those elements of a list at the top level which share the given property was presented to students in the following form:

```
Schema-1([], 0).
Schema-1([H | T], C) :-
    Predicate(H), Schema-1(T, CL),
    C is CL + 1.
Schema-1([H | T], C) :-
    [Not-Predicate(H),]
    Schema-1(T, C).
```

Across both groups, there were students with two different preconditions. Attending lectures is not obligatory for the students and consequently, there is usually a part of the class not present. Some students were present at the previous week's lecture when the 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 available at that time did not include program schemata).

Results and evaluation

There were 101 students who took part in the experiments, all of them level 3 students of our baccalaureate Informatics course in the software engineering track. There were 53 students among them who have already heard about schemata, but the remaining haven't. During the test, 47 students had available the catalogue of schemata, but the remaining had not. Distribution of the whole set of students into the four categories is shown in Tab. 1.

		Schemata available?	
		yes	no
Explanation received?	yes	**22**	**31**
	no	**25**	**23**

Tab. 1. Distribution of students into groups.

Perhaps the most important question to answer when trying to evaluate the approach to learning logic programming based on program schemata is concerned with a positive identification of their influence on students' learning process. In our formulation, does the use of schemata in learning logic programming influence results of novice learners? Can it speed up the process of learning the paradigm?

We can formulate our first hypothesis: *The results after using schemata are better*. To judge about the results, we tested the whole class. All the students (in one place at one time) were supposed to solve very simple problems by devising appropriate predicates.

In Tab. 2 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?	
		yes	no
Explanation received?	yes	**3.66**	**3.45**
	no	**3.15**	**2.62**

a

Explanation received?	**77%**	**74%**
	56%	**28%**

b

Tab. 2. Overall results of tests.

The results in Tab. 2 allow some conclusions. Best results were achieved by the students who had prior information on schemata and at the same time they had the catalogue of schemata at hand during the test. 77% out of them produced correct solutions. Second best results (74%) were achieved by those students who had the catalogue of schemata at hand during the test but did not hear their presentation and explanation. Comparing to them, students who heard the presentation and explanation but did not have the catalogue of schemata at hand during the test performed worse (56%). By far the worst results were achieved by the subgroup without prior information and without schemata.

The results of the first two subgroups show clearly that schemata influence positively the students' programming performance. It is perhaps surprising that their results are so close. However, here it is worth noting that the whole class was at that point already having some experience with program schemata, because in the first part of the semester, schemata for functional programming were taught in a similar way. This is indirectly endorsed by similar experiments of ours related to using schemata in teaching functional programming. The second and third subgroups scored almost precisely the same there. Here (after some experience with program schemata), for the students it was more valuable to receive the catalogue of Prolog related schemata and to have it during the test.

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 39% of all students had the correct solution already sketched on a paper. Next 30% of all students not only did not have a correct solution designed on a paper, but could not arrive to one even during the interaction with a computer. The remaining 31% was able to make some minor changes like amending or completing tests in clauses' bodies. 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. It is partially in conflict with our experience from the previous years when logic programming was taught without program schemata.

Another hypothesis which we attempted to get confirmed or refuted was the one related to the question if problems of processing lists at all levels are harder to solve for learners than processing lists at the top level only. Our hypothesis is: *Processing a list at all levels is more difficult to program for learners*. We based our hypothesis on our several years' experience. 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 indeed the students perform better when programming processing of lists at the top level only. However, the difference is quite small, perhaps thanks also to the previous experience with functional programming. On the other hand, a similar evaluation with respect to functional programming did not confirm the hypothesis. The students treated the lists to be processed at all levels as binary trees which they were able to program equally "easily" as processing lists at the top level only. We claim that this result also supports positive influence of the use of program schemata.

	ϕ marks out of 4
processing at the top level only	**3.24**
processing lists at all levels	**3.18**

Tab. 3: Average success rate of solving processing of lists.

Students' feedback

At the end of semester, we initiated an enquette in order to solicit their feedback regarding the subject Functional and logic programming and specifically our approach based on program schemata. The table below summarizes the distribution of responses to one of the questions in the questionnaire. It shows that a majority of students uses the program schemata (first three answers, i.e. 78% of all students).

"Which is the way you use program schemata in designing simple functions/predicates for processing lists?"	
I write it on a paper	17%
I recollect it in my mind	27%
I think my using of schemata is automated and I do not consciously realise it	34%
I know what schemata are but I do not use them	21%
I do not know what schemata are	1%

Conclusions

In the paper we present the program schemata which we use in teaching logic (Prolog) programming and the way how to represent them. Similar approach to logic programming has been described by Gegg-Harrison [5, 6]. He described 14 basic schemata for recursive predicates together with a way of organizing the dialog in a tutoring system. In [13] is presented an extension of Gegg-Harrison's proposal for schema language. Our approach differs in choosing a different set of schemata and in focusing to various identified kinds of problems related to list processing. After all, list processing can be viewed as a very general class of computations. Moreover, our approach reflects consequently our ultimate goal – to support the learning process.

In [11], too, there is defined a number of constructs suitable for building Prolog programs (*program skeletons* and *techniques* which extend skeletons and express steps in program construction). Separation of control flow and technique is the key idea of this approach. Student selects a skeleton and then enhances it, using standard techniques, to solve his or her particular problem. We feel that to understand skeletons can be rather difficult for novices without having some preceeding experience in building and fully understanding some programs.

Our approach is focused on learning programming with schemata. The method is around building 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 program schemata, such as those presented in [4, 12, 13], are important and inevitable when exploring properties of a created solution, its correctness, etc. Formalising is necessary also for programming tools such as Prolog Techniques Editor [12]. 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 purposly "incomplete" in the sense that their specialization cannot be for some problems completed at the syntactic level.

As the main result, we report on experiments that allow to judge quite favorably the approach to teaching Prolog programming when the student learns a set of program schemata and how to apply them.

References

1. M. Bieliková, P. Návrat. A schema-based approach to teaching programming in Lisp and Prolog. In: *Proc. PEG'97 Int. Conf.,* P. Brna and D. Dicheva (Eds.), 22-29. 1997.
2. A. Bowles, P. Brna. Programming plans and programming techniques. In *Proc. World Conf. on AI in Education*, P. Brna, S. Ohlsson and H. Pain (Eds.), 378-385. AACE Press, 1993.
3. P. Brna. Teaching Prolog Techniques. *Computers and Education,* 20(1):111-17. 1993.
4. P. Flenner, K.K. Lau, M. Ornaghi. Correct-schema-guided Synthesis of Steadfast Programs. In *Proc. 1997 IEEE Conf. on Automated Software Engineering.* M. Lowry and Y. Ledru (Eds.), IEEE Press. 1997.
5. T.S. Gegg-Harrison. Representing logic program schemata in λProlog. In *Proc. of the 12th Int. Conf. on Logic Programming*, L. Sterling (Ed.), 467-481. MIT Press, 1995.
6. T.S. Gegg-Harrison. Extensible logic program schemata. In *Proc. of the 6th Int. Conf. on Logic Program Synthesis and Transformation*, I. Gaallagher (Ed.). Springer-Verlag, 1996.
7. S.M. Haynes. Explaining recursion to the unsophisticated. *SIGSCE Bulletin*, 27(3):3-6. 1995.
8. S. Joosten (Ed.), K. van den Berg, G. van der Hoeven. Teaching functional programming to first-year students. *J. Functional Programming*, 3(1):49-65. January, 1993.
9. P. Návrat, V. Rozinajová. Making programming knowledge explicit. *Computers and Education*, 21(4):281-299. 1993.
10. C. Sollohub. Programming Templates: Professional programmer knowledge needed by the novice. *Computer Science Education,* 3:255-266. 1991.
11. L.S. Sterling, M. Kirschenbaum. Applying techniques to skeletons. In *Constructing Logic Programs*, J.M.Jacquet (Ed.), 127-140. John Wiley, 1993.
12. W.W. Vasconcelos, M. Vargas-Vera, D.S. Robertson. Building Large-Scale Prolog Programs using a Techniques Editing System. In *Int. Logic Programming Symposium.* The MIT Press. 1993
13. W.W. Vasconcelos, N.E. Fuchs. Prolog Program Development via Enhanced Schema-Based Transformations. In *Proc. of 7th Workshop on Logic Programming Environments*, Portland, 1995.

Appendix – program schemata used in experiments

Processing lists at the top level only	**Processing lists at all levels**

Map([], []).
Map([H_1 | T_1], [H_2 | T_2]) :-
    [*Element-Test*(H_1),]
    Transform(H_1, H_2), *Map*(T_1, T_2).
[*Map*([H | T_1], [H | T_2]) :-
    [*Not-Element-Test*(H),] *Map*(T_1, T_2).]
	DMap([H_1 | T_1], [H_2 | T_2]) :-
	    DMap(H_1, H_2),
	    DMap(T_1, T_2).
	DMap(E_1, E_2) :-
	    Test(E_1 [, E_2]), *Transform*(E_1, E_2).
	DMap(E, E &1) [:- *Not-Test*(E &2)].

Func-Reduce([], *Neutral-Value*).
Func-Reduce([H | T], R) :-
    [*Element-Test*(H),]
    Func-Reduce(T, RL), *Element-Reduce*(H, RL, R).
[*Func-Reduce*([H | T], R) :-
    [*Not-Element-Test*(H),] *Func-Reduce*(T, R).]
	DFunc-Reduce([H | T], R) :-
	    DFunc-Reduce(H, RH),
	    DFunc-Reduce(T, RT),
	    Reduction(RH, RT, R).
	DFunc-Reduce(E, E) :- *Test*(E).
	DFunc-Reduce(E, *Neutral-Value*) [:- *Not-Test*(E)].

Count-if([], 0).
Count-if([H | T], C) :-
    Element-Test(H),
    Count-if(T, CL), C is CL + 1.
Count-if([H | T], C) :-
    [*Not-Element-Test*(H),] *Count-if*(T, C).
	DCount-if([H | T], C) :-
	    DCount-if(H, CH),
	    DCount-if(T, CT),
	    C is CH + CT.
	DCount-if(E, 1) :- *Test*(E).
	DCount-if(E, 0) [:- *Not-Test*(E)].

Processing lists at the top level only	**Processing lists at all levels**

Count-if-not([], 0).
Count-if-not([H | T], C) :-
    Element-Test(H), *Count-if-not*(T, C).
Count-if-not([H | T], C) :-
    [*Not-Element-Test*(H),]
    Count-if-not(T, CL), C is CL + 1.
	DCount-if-not([H | T], C) :-
	    DCount-if-not(H, CH),
	    DCount-if-not(T, CT),
	    C is CH + CT.
	DCount-if-not(E, 0) :- *Test*(E).
	DCount-if-not(E, 1) [:- *Not-Test*(E)].

Find-if(E, [H | _]) :-*Element-Test*(E, H). *Find-if*(E, [_ | T]) :- *Find-if*(E, T). *Every*([]). *Every*([H | T]) :- *Element-Test*(H), *Every*(T). *Some*([H | _]) :- *Element-Test*(H). *Some*([H | T]) :- [*Not-Element-Test*(H),] *Some*(T). *None*([]). *None*([H | T]) :- *Element-Test*(H), !, fail. *None*([H | T]) :- [*Not-Element-Test*(H),] *None*(T). *Some-not*([H | T]) :- *Element-Test*(H), *Some-not*(T). *Some-not*([H | _]) [:- *Not-Element-Test*(H)].	*DFind-if*(E, [H | T]) :-*DFind-if*(E, H) ; *DFind-if*(E, T). *DFind-if*(E, H) :- *Test*(E, H). *DEvery*([]). *DEvery*([H | T]) :- *DEvery*(H), *DEvery*(T). *DEvery*(E) :- *Test*(E). *DSome*([H | T]) :- *DSome*(H) ; *DSome*(T). *DSome*(E) :- *Test*(E). *DNone*([H | T]) :- *DNone*(H), *DNone*(T). *DNone*(E) :- *Test*(E), !, fail. *DNone*(E). *DSome-not*([H | T]) :- *DSome-not*(H) ; *DSome-not*(T). [*DSome-not*([]) :- !, fail.] *DSome-not*(E) :- *Test*(E &2), !, fail. *Dsome-not*(E).

Remove-if([], []).
Remove-if([H | T1], T2) :-
    Element-Test(H), *Remove-if*(T1, T2).
Remove-if([H | T1], [H | T2]) :-
    [*Not-Element-Test*(H),] *Remove-if*(T1, T2).
 	
Remove-if-not([], []).
Remove-if-not([H | T1], [H | T2]) :-
    Element-Test(H),
    Remove-if(T1, T2).
Remove-if([H | T1], T2) :-
    [*Not-Element-Test*(H),]
    Remove-if(T1, T2).
	DRemove-if([H_1 | T_1], R) :-
	    DRemove-if(H_1, H_2), *DRemove-if*(T_1, T_2),
	    combine(H_2, T_2, R).
	DRemove-if(E, *remove-flag*) :- *Test*(E).
	DRemove-if(E, E) [:- *Not-Test*(E)].
	
	[*DRemove-if-not*([], []).]
	DRemove-if-not([H_1 | T_1], R) :-
	    DRemove-if-not(H_1, H_2), *DRemove-if-not*(T_1, T_2),
	    combine(H_2, T_2, R).
	DRemove-if-not(E, E) :- *Test*(E).
	DRemove-if-not(E, *remove-flag*) [:- *Not-Test*(E)].
	
	combine(*remove-flag*, T, T).
	combine(H, T, [H | T]).