Logic Programming Didactics

Demosthenes Stamatis[1], Petros Kefalas[2]

[1] Dept of Informatics, Alexander Technological Educational Institute, P.O Box 141, 57400 Thessaloniki, Greece, demos@it.teithe,gr

[2] Dept of Computer Science, City College, Affiliated Institution of the University of Sheffield, 13 Tsimiski Str, 54624 Thessaloniki, Greece, kefalas@city.academic.gr

Logic Programming (LP) courses are part of many Computer Science or Artificial Intelligence related programmes. In this paper, we present a systematic approach on teaching an LP course, using Prolog as the main computational paradigm. We argue that LP is an excellent didactic tool for teaching Intelligent Programming Systems as well as a vehicle for an in depth understanding of the programming methodology activity as a whole, both declarative and imperative. A student model is defined which in turn is used to facilitate the learning outcomes and process. The model is based on student misconceptions, which were identified using action research derived from our long experience on teaching LP. We demonstrate that, by lifting these misconceptions through specifically designed teaching sessions students are led towards a better understanding of Logic Programming both as a tool for developing intelligent systems and program construction in general.

Keywords

Logic Programming, AI logics, Mathematical Foundations

1. Introduction: Logic Programming courses in CS Curricula

Computer Science is an engineering discipline, and as such it should integrate a fair amount of mathematical concepts. CS curricula should be designed such that they include introductory courses that relate to mathematics specific to the domain, e.g. Discrete Mathematics including logic, set theory, graph theory etc. [1,2]. Among those, Logic Programming, although not a core in IEEE/ACM Computing Curricula [1], can be found in many CS Departments programmes, especially those with an Artificial Intelligence flavour. It is thought that logic programming, particularly through the use of Prolog as the main programming paradigm, lead to the development of an improved student model that is more capable to cope with programming methodology in general, both imperative and declarative, as well as with Artificial Intelligence techniques, which are normally introduced later in their studies.

It has been long argued that mere possession of knowledge is not sufficient for students in higher education, if the student does not learn how to use this knowledge effectively. To do this, a learner must possess certain intellectual or cognitive processing skills, e.g. the ability to analyse, synthesize or evaluate. These are clearly identified in Bloom's taxonomy of educational objectives, in which the major categories in the cognitive domain are listed, i.e. knowledge, comprehension, application, analysis, synthesis, and evaluation [3,4]. Logic programming courses offer the opportunity to develop such skills in the context of program construction.

From then on, it is really a matter of choice, which of the following options educators would follow:

(a) introduce logic programming early or later in the curriculum,

(b) put more or less emphasis to knowledge of Prolog than pure logic,

(c) put more or less emphasis to generic or specific programming skills.

Examples of various approaches exist and all are designed to meet their learning objectives through appropriate teaching, learning and assessment methods [5,6]. In a straightforward approach the educators might choose to follow widely accepted and well-written textbooks (such as [7]), but the outcome would be no more than an average to good knowledge of Prolog language, as opposed to an in-depth understanding and use of Prolog for intelligent programming.

In this paper we present a teaching approach to LP, which is based on altering misconceptions of students, which are connected with their background on imperative programming languages. Section 2 discusses the student model we deal with, which is built around a set of misconceptions. In section 3, we present the methodology of identifying and lifting the misconceptions. In section 4, the misconceptions are classified accompanied with appropriate representative examples. Finally, we conclude the paper by discussing our approach compared to the standard approach of teaching Prolog as a logic programming paradigm.

2. Teaching Logic Programming based on a student model

The Informatics Department at the Technological Educational Institute (TEI) of Thessaloniki, Greece, has integrated an LP course in its curriculum for the last 15 years. The course is taught in the 5th semester of the studies. In the 7th semester an Artificial Intelligence course is taught which has as a prerequisite the LP course.

2.1 Student Model

A number of researchers in computer science education, adopting the constructivist approach to learning programming language concepts, have identified that the prior knowledge of the students is the key for further knowledge construction [8,9,10,11,12]. Based on this approach, students often develop some kind of prior knowledge which is intuitive, based on past experiences and may be either imprecise or even totally mistaken. This prior knowledge is not considered wrong but is referred to as knowledge based on misconceptions. These misconceptions and their interrelations form the so called student model which should be taken as a basis for organising a teaching strategy. The most important step in such a strategy is that the misconceptions should be lifted in order for the students to proceed with a more in deep knowledge construction.

The prior knowledge of students in our case is that developed due to their contact with imperative programming languages. In the first couple of years of their studies, after the students have been exposed to many imperative language paradigms as well as algorithmic design and quite heavily engaged in programming with C++ and Java, they have inductively developed a specific mind-set about programming, which is characterised by the following:

- any variable used in a programme should be declared by type,
- any variable can change its value,
- any function/method has a type, can be part of any expression and returns one (and only one) value at a time,
- all parameters of functions/methods should have a value when called,
- arithmetic expressions as parameters can be evaluated,

- recursive data structures are rarely used,
- recursions is not-preferred over iteration, etc.

All the above, in the context of LP form misconceptions that should be lifted in a LP course, with Prolog as the main programming paradigm.

A similar approach to ours [13] shows how Prolog can be taught based on common misconceptions of imperative programmers, a study which is focused specifically on teaching recursion through appropriately designed templates

2.2 Learning Outcomes

In our LP course, we decided not to put emphasis into Prolog knowledge per-se (although basic elements of the language should be taught), but to focus on all the above aspects. We believe the students appreciate more the skills acquired through this course, which can be used to change the mind-set of the programming task as a whole. By the end of the course should be able to:

- understand the basic principles of logic programming theory and symbolic reasoning,
- demonstrate good knowledge of the basic Prolog language by constructing small programs,
- make sense of more complicated Prolog programs, predict and describe what they do,
- modify existing code to perform a similar task,
- identify the advantages of declarative programming and evaluate its shortcomings in comparison with imperative languages,
- comprehend the basic principles of programming languages, like procedural abstraction, program design and development, parameter passing, recursion, variable binding etc.,
- adapt declarative programming techniques to other programming paradigms.

These learning outcomes are assessed through coursework and final examinations.

3. Methodology

Students enrolling in the LP course and after having covered the basic elements of LP (normally 4 lectures) are requested to fill-in a questionnaire. The questionnaire contains a number of simple problem definitions together with example codes in Prolog that represent alternative solutions to a problem. These fall in three (3) categories.

The first one contains incorrect examples (incorporating characteristics of an imperative language) which are opposed to correct ones. The following is a typical example of this category:

Problem: Find the length of a list

`lengthA([],0).`
`lengthA([H|T],L):-`
`  L is lengthA(T)+1.`
	`lengthB([],0).`
	`lengthB([H|T],L):-`
	`  length(T,R),`
	`  L is R+1.`

Students are asked to compare the two codes and answer if one or both of them are correct.

In this case most of the students chose the correct code (lengthB) but a number of them (25%) find both codes correct, which means that they expect length to be a predicate and an integer function at the same time.

In the second category both examples given are syntactically correct but one of them deviates from problem definition. The following is a typical example of this category:

Problem: Find the sum of the elements of a list containing integers.

```
sumA([],0).
sumA([H|T],H+R):-
      sumA(T,R).
```

```
sumB([],0).
sumB([H|T],S):-
      sumB(T,R),
      S is H+R.
```

Students are asked to compare the two codes and answer if one or both of them are (a) syntactically correct and (b) logically correct.

In this case almost all of the students (95%) consider sumB as correct code both from the syntactic and the logical point of view. This means that although they accept the fact that Prolog does not evaluate its arguments it is difficult for them to escape from the arithmetic nature of imperative languages. They are really surprised when they are presented with the answer:

`S = 4+(5+(3+(6+0)))` to the query `?- sumA([4,5,3,6], S)`

Their surprise is increased when one or more numbers from the above list of integers are replaced by variables.

In the third category both examples given are syntactically and logically correct but one of them is based on imperative programming style. The following is a typical example of this category:

Problem: Concatenation of lists

```
appendA(X,Y,Z):-
      X=[], Y=L, Z=L.
appendA([X|Y],L,Z):-
      appendA(Y,L,NL),
      Z=[X|NL].
```

```
appendB([],L,L).

appendB([X|Y],L,[X|NL]):-
      appendB(Y,L,NL).
```

Students are asked to compare the two codes and answer if one or both of them are correct.

In the case they find both of them correct they are asked to choose the one which is best suited to their programming style. In this occasion, most of the students (65%) either chose appendA as the correct code or the one with the best programming style. The conclusion is that most of the students are stacked with the way assignment and parameter passing techniques are used in imperative languages

The outcome of the questionnaire is to identify specific cases of misconceptions. With the goal of clarifying further their misconceptions students are also interviewed. The result of this process is also to gather a set of characteristic examples that are going to be used later in class in order to lift these misconceptions.

During the rest of the semester we split the cohort of students into two groups: students participate in different laboratory and tutorial classes. In the first group we apply our teaching methodology as described below, and in the second we follow a standard LP textbook approach.

At the end of the course, we request students of both groups to answer a final questionnaire. This questionnaire is similar to the one given at the beginning of the course containing problem definitions used in intelligent programming systems and relevant Prolog codes. To

justify their answers students are also asked to give the answers resulting from the execution of specific queries.

4. Classification of Misconceptions and Teaching Approach

Based on the analysis of the results of the questionnaires and the interviews of the students we have grouped the misconceptions into five categories, namely:

- Logical variable
- Unification
- Predicates vs functions
- Execution
- Recursion

Table 1: Categorising Misconceptions

Misconception Category	Description	Student Achievement if misconception is lifted
The logical Variable		
Scope of Variables	Students believe that the scope of a variable is the whole program, or the set of predicates having the same name.	Incremental Programming Top-Down Design
Variable binding	Students do not understand that variables cannot be assigned, but can only be instantiated and they don't loose their value afterwards.	Programming based on a sound logical approach - Referential Transparency
Uninitiated Variables	Students believe that a variable could not remain uninitiated during execution.	The ability to deal with incomplete data - Intelligence
Unification		
Parameter Passing 1	Students do not understand the dual role of variables as parameters.	Intelligent programming Software reusability
Parameter Passing 2	Students do not understand the possibility of using parameters with their analytic – term form.	Intelligent programming
Parameter Evaluation	Students do not understand the real meaning of the fact that parameters are not evaluated.	Symbolic Processing
Predicates vs functions		
	Students do not understand that predicates are not able to be used as functions.	Procedure oriented programming - top down design
Execution		
Automatic backtrack mechanism	Students do not understand the automatic generate and test execution mechanism of LP.	Intelligent programming
Pattern directed execution	Students do not understand the dynamic order of predicate definitions in a program.	Pattern matching - Procedure oriented programming
Recursion		
Combining Recursion with backtracking	Students do not list alternative cases using separate clauses in the body of recursion.	Abstraction
Recursion as a methodology	construction of recursive predicates vs interpretation of recursive predicates.	Abstraction – Declarative style of programming

The classification of these categories of misconceptions together with their subcategories is depicted in Table 1. We give a description of each misconception and for each subcategory we list student achievements expected if the specific misconception is lifted.

We designed our teaching around the student mode, which is based on the misconceptions categorisation, described above. We use a number of examples that drive our teaching classes and lab sessions. We ask the students to devise their solution and compare it with the one we suggest. Table 2 shows some characteristic examples of Prolog code parts predicate are not functions.

Table 2 A number of examples for lifting student misconceptions.

Misconceptions	**Student Solution**	**Prolog solution**

Unification:
- concatenation of lists
	`conc(X,Y,Z):-`
	`  X=[], Y=L, Z=L.`
	`conc([X|Y],L,Z):-`
	`  conc(Y,L,NL),`
	`  Z=[X|NL].`
		`conc([],L,L).`
		`conc([X|Y],L,[X|NL]):-`
		`  conc(Y,L,NL).`

Multiple use of predicates:
- splitting a list,
- adding elements to a list
	Do not know how
	`add(X,[],[X]).`
	`add(X,[H|T],[X,H|T]).`
	`add(X,[H|T],[H|R]):-`
	`  add(X,T,R).`
		`split(L,L1,L2):-`
		`  conc(L1,L2,L).`
		
		`add(X,L,NL):-`
		`  delete(X,NL,L).`

Unification, return value by predicates:
- length of a list
	`length([H|T],L):-`
	`  L = length(T)+1.`
		`length([H|T],L):-`
		`  length(T,R),`
		`  L is R+1.`

Arithmetic, symbolic process:
- sum of numbers
	`sum([H|T],S):-`
	`  sum(T,R),`
	`  S is H+R.`
		`sum([H|T],S):-`
		`  sum(T,R),`
		`  S = H+R.`

Terms (not evaluated):
- Evan's analogy
	`figure(1):-`
	`middle(`
	`triangle,square)).`
	`or`
	`figure(1,middle,`
	`triangle,square).`
		`figure(1,`
		`middle(triangle,square)).`

Generate and test:
- zebra problem
- map colouring
	Do not know how
	(in most cases)
		`generate_test(L,X,Sol):-`
		`  member(X,L),`
		`  test(X,Sol).`

Students also have the opportunity to develop their sense of equivalence in various ways. For instance, the last part of course is about graphs, but it takes a while for them to realise that finding a path through a graph is the same program as the one finding the ancestor in a family tree, something that they have seen in the very first lesson. Numerous practices are encouraged throughout the 12 lab sessions that exist in the course. These practices include:

- **Top Down Design**: The Backward Chaining technique used by Prolog in running mode, can lead students to start designing the structure of the computer program in a top-down approach. The students will eventually start solving problems by breaking them into smaller sub problems that are easier to implement and handle. This also leads towards implementation of more well-structured programs.
- **Incremental Programming**: Since Prolog does not have (in its pure version) keywords-commands, the situation that the students get in, is much different than that of other popular programming languages. At least at the beginning, they believe that they have to build everything they need from scratch. Although this is true, they

shortly understand that the extensive reuse of code that they wrote, will make their task much easier.

- **Recursion**: Since iteration constructs are not provided in Prolog, recursion should be used instead. Although recursion is also used in other programming languages, this technique is heavily used in Prolog and so the student will have to start thinking and solving problems recursively. A student, who is used in recursion, can apply the same technique to other programming languages. This will make students more competent in the way they solve problem and write programs.
- **Intelligent Programming - Non-determinism**: In the execution of a Prolog program the non-determinism feature is more apparent. Although the Prolog rule that will execute is the first one that matches the goal, we can ask for more than on solution. Using the backtracking mechanism other valid solutions can be found. This introduces: a) "don't know non-determinism" implying that all possible ways to find the solutions will be followed since the execution does not know how to find the solution, b) "don't care non-determinism" meaning that we just need one solution and we do not care which solution is that among the many that exists.

3. Conclusions

We have presented our approach in teaching an LP course based on a student model that takes into account the mind-set developed through earlier courses on imperative programming. We classified the misconceptions in several categories and found examples that are used in teaching in order to facilitate the in-depth understanding of Prolog. The final questionnaire handed in to students and the answers returned show that Group A (in which we apply our teaching methodology) outperforms Group B (which is taught through the standard textbook). The programming skills acquired from students in Group A match our initial aims for being able to cope with intelligent systems programming, something that is demonstrated when they enrol in the Artificial Intelligence course later in their studies. There is a number of students' wrong ideas that proved resistant to our didactic activity since their presence was also apparent after the whole course of study and the evaluation of the final questionnaire. These are connected mainly with the referential transparency of the logical variable and the ability of using predicate parameters in multi I/O format.

Based on the first results there is an experimental confirmation that such teaching approach is beneficial in helping students learn logic programming. Future work will require long test beds to formally validate our experimental-empirical results, by gathering data from students and educators in multiple institutions and bringing these together for analysis.

References

1 Joint ACM/IEEE Task Force, Computing Curricula, 2001

2 P.Kefalas, A.Sotiriadou, Logic & Sets in Computer Science Curriculum, Proceedings of the 2nd Panhellenic Logic Symposium, 191-196, Delphi, 1999

3 B.S.Bloom., D.Krathwohl. Taxonomy of Educational Objectives: The Classification of Educational Goals, by a committee of college and university examiners. Handbook I: Cognitive Domain. New York, Longmans, Green, 1956.

4 N.Jackson, Programme Specifications and its role in promoting an outcomes model of learning, Active learning in Higher Education, 132-151, 2000

5 P.Szeredi, Teaching Logic Programming at the Budapest University of Technology, First International Workshop on Teaching Logic Programming – TeachLP, 2004

6 H.Christiansen, Prolog as Description and Implementation Language in Computer Science Teaching, First International Workshop on Teaching Logic Programming – TeachLP, 2004

7 I.Bratko, Prolog Programming for Artificial Intelligence, Addison-Wesley/Pearson Education, 3rd edition, 2001

8 M. Ben-Ari, Constructivism in Computer Science Education, ACM SIGCSE-Bulletin, Vol.30, Issue 1, 257 – 261, 1998
9 J. C. Spohrer, E. Soloway, Novice mistakes: are the folk wisdoms correct?, Communications of the ACM, Vol. 29, Issue 7, 624-632, 1986
10 C. Holmboe, A Cognitive Framework for Knowledge in Informatics:The Case of Object-Orientation, Proceedings of the 4th annual SIGCSE/SIGCUE ITiCSE '99 (Cracow, Poland), 17-20, ACM Press, 1999
11 M.Hristova, A.Misra, M.Rutter, R.Mercuri, "Identifying and correcting Java programming errors for introductory computer science students," Proceedings of the 34th SIGCSE Technical symposium on Computer science education, Reno, Navada, USA, ACM Press, 2003
12 S. Madison, G. Gifford "Modular Programming: Novice Misconceptions", Journal of Research on Technology in Education Vol. 34, No 3, 217-229, 2003.
13 A.N.Kumar, Prolog for imperative programmers, Journal of Computing Sciences in Colleges archive, Vol. 17, Issue 6, 167–181, 2002