insight (SIGPLAN Notices), V. 39, N. 4 (Dec 2007)

Prolog as the First Programming Language

Arthur Fleck, Professor Emeritus
Computer Science Department
University of Iowa
Iowa City, Iowa 52242
fleck@cs.uiowa.edu

Abstract
The adoption of logic programming in an introductory course on computer programming offers several rewards. The higher conceptual level of this programming paradigm guides students to an early appreciation for abstraction. The descriptive character of programs makes them more understandable than conventional programs, and the relational basis makes them more versatile. This incremental and highly interactive style of programming leads to early student success and motivates active participation by students.

Keywords
Prolog, logic programming, first programming language

1. INTRODUCTION

This paper advocates the use of logic programming, specifically Prolog, for the language in a first computer programming course. The elements of precise logical thinking provide a basis, and logical assertions themselves effectively serve as a computer program. Logic programming is a unique programming paradigm that accomplishes a shift in emphasis from *how to* compute an answer to *what constitutes* an answer. This raises student thinking to a substantially higher level of abstraction than traditional programming languages involve.

Traditional programming languages require our attention to be on controlling numerous individual steps of a complicated computing mechanism. They require a student to first assimilate a plethora of "contraptions", many of which have no direct connection to a problem of interest. Then each problem requires the creation of a representation of the problem domain in terms of these devices. The dominance of these factors in traditional programming languages diverts intellectual attention from the basic problem-solving task.

In traditional programming, a specification and a corresponding program are two completely separate things. Creation of a program requires an intellectual leap from a specification understood in the problem domain to computation steps that manipulate the elements of a computing mechanism. Logic programming enjoys the economy of the unification of the specification and the program as a collection of logical assertions.

The importance of abstraction to the programming process has repeatedly been recognized (e.g., Kramer[6]). Logic programming tightly integrates programming with problem solving, rather than treating them as two distinct activities. Focus on the conceptual aspects of program construction facilitates student conceptualization of the multilayered character of abstraction essential to true understanding of computer systems. This provides strong support for subsequent development of programming skills, regardless of what language may later be employed.

It is widely acknowledged that Prolog is not synonymous with logic programming. However, the predominance of the language in technical literature and the widespread availability of high quality implementations make Prolog a compelling choice. By making careful selections among the features of Prolog that are introduced, it is possible to utilize facilities for which logic provides a meaningful (though sometimes imperfect) explanation. A few essential constructs not modeled in logic are utilized to provide a bridge between logic and procedural programming ideas.

2. BASIC COURSE CONTENT

Of course, some time must be devoted to basic computing facilities (e.g., editor, browser, operating system) at hand for students. This is no different for this course than any other, and is not pursued in this discussion.

In a traditional language, a *program* constitutes a set of instructions for manipulating a computer representation of the elements of a problem, and a computer executes these instructions to obtain a solution. By contrast, in Prolog a program is a set of logic formulas asserting known properties of a problem domain, and a solution is deduced from these assertions. So in logic programming, computation is synonymous with deduction.

To pursue logic programming, one must have a basic appreciation of logic. But to fully develop logic takes more than a single course itself. Therefore, in this course we start with logic, but devote a relatively small amount of time to briefly introduce its basic ideas. We strive for expanded understanding of logic to be ultimately developed hand-in-hand with the use of Prolog to “animate” logical assertions. Each of these viewpoints enhances a student's understanding of the other.

Our development of the course begins with propositional logic and the basic logical connectives. Students quickly assimilate the 'and'. 'or', and 'not' operations, and the syntax rules for well-formed formulas. Because of the central conceptual role it plays in logic programming, the 'implies' connective is strongly emphasized. Basic ideas such as satisfiability and tautology are presented. The extraordinary flexibility to phrase other (NP-complete) questions as satisfiability of suitable formulas is used to illustrate the computational ramifications of these seemingly abstract logic ideas.

Next the additional ideas of predicate logic are covered. Of course, the idea of a predicate or relation is fundamental in logic as well as in Prolog. This is one of the major departure points of Prolog from traditional languages, and it's natural to develop the relational view in this logic setting. The concepts of universal and existential quantification of predicate logic are central to Prolog and are first explored in the logic context. The most challenging aspect of logic to get across to students is the independence of logic formulas from any specific context, and to understand that truth or falsity will generally vary with an interpretation.

Finally, axiom and proof rule ideas must also be covered. Students need to understand that axioms are chosen to reflect a context of interest, and deduction can show what else must necessarily be true in *that* context. Introduction of the familiar "modus ponens" proof rule has proven sufficient for the purposes of this course. Students are shown the impossibility of exhaustive truth analysis for assertions of predicate logic, and the resulting necessity of using a proving process instead. The connection between truth analysis and proof of formulas, and the efficacy of replacing the former by the latter is emphasized. This paves the way for describing Prolog's use of mechanical proof construction to accomplish computation.

Formal logic provides a means for describing conditions of interest with complete precision. Identifying the key properties that determine a correct answer to a problem is the first step. Viewing things from this perspective is a substantial aid in gaining deeper insights into a problem that has been posed. Understanding how to use logic to express the critical properties as a logical formula is the second step. The basic idea of logic programming is that a resulting logic formula can be "executed" by a computer to produce an answer.

Prolog has an especially simple syntax that is readily explained, especially after the idea of logical formulas has already been presented. Since the initial semantic ideas of Prolog are directly drawn from logic, no further discussion is needed to explain the basis of the language.

Prolog provides logic with a "procedural" interpretation. The goal-oriented, backtracking model of execution offers a dual view of the proving process, and reinforces and expands the formal logic perspective. It is vital to compare and contrast Prolog's goal satisfaction approach with logical proof construction. This also provides a basis for explaining other Prolog facilities for which logic is inadequate.

A particularly important attribute of virtually every Prolog implementation is the ability to "trace" the steps of a query as Prolog carries out its goal reduction process. This is an especially important aid to student understanding as it reveals steps in a search for a proof. It also shows how deduction steps accumulate an incremental effect on logic variables to provide answers as side-effects.

Repetitious computations are described in three distinct ways in Prolog. The 'findall' predicate is easily introduced early since it has an adequate (higher order) logical explanation, and provides a looping construct that is natural to the logic viewpoint. So called "success-failure" loops usually appear as simple assertions (i.e., straight-line code), but especially with the aid of tracing, students soon see how this technique effects a repetitious computation (and possibly an infinite loop) through Prolog's backtracking. Finally, recursion is very natural in Prolog and also is conceptually simple from the logical perspective (see next section).

3. TECHNICAL COURSE DETAILS

A critical concept in logic programming (and Prolog) is the idea of a logic variable. It's essential to continually reinforce the role of logic programming variables as "unknowns" in the sense of mathematics, especially for those (numerous) college freshmen who have already had some experience with a traditional programming language and retain the misleading idea of a variable from that context. A related matter is ensuring that students realize that the values taken on by logic variables are normally treated as unevaluated *terms* (or trees). An appreciation of these two points, and their interrelationship, is vital to the understanding of logic programming.

Prolog has two very important attributes for beginning students. One is that it is highly interactive. That means that for the most part, students obtain immediate useful feedback. Also, students receive prompts and responses from the system that, especially in early stages, relieve the burden of programming their own input-output. The second important attribute of Prolog is the completely incremental way in which programs can be constructed and tested. One does not need to attempt a complete problem solution, or even insert stubs, before trying out code. One need not supply a large boilerplate – a few isolated lines treating a single special case is already a usable Prolog program! If you have written out just one case of one predicate, you can immediately try it and simply add more code later. These features lead to early student success, and thus greatly enhance student confidence and motivation.

Prolog programs define the behavior of predicates and ideally enjoy an absence of "orientation requirements". That is, programs make no distinction between arguments

and results. For instance, the single Prolog 'append' predicate can be used to concatenate two lists producing the result, to divide a given list into two parts which are produced, to test membership of an item in a list, etc. – each of these uses would require a separate program in a conventional language. For this reason Prolog instills thinking about the generality and reusability of programs in a very concrete way.

Discussion of both Prolog arithmetic and negation as failure are deferred until the goal-oriented model has been discussed. This model is required to provide a cogent explanation of these facilities since they reflect logic imperfectly. Leaving these topics until later in the course requires some care in the earliest stages, but it has proven to be quite workable.

The early avoidance of arithmetic is facilitated by Prolog's natural suitability for non-numeric applications. Such problems are common in everyday situations, and allow the immediate pursuit of the use of logic description as a program. However, even non-numerical problems sometimes rely on basic counting, so there is motivation to introduce the goal-solving model without undue delay.

Avoiding general negation in the early stages of the course is compensated for through reliance on 'not identical' (\==) and 'not unifiable' (\=) predicates for terms. While much less expressive than general goal negation, these primitives do allow numerous aspects of negation to be expressed, and are readily explained in the logic context. The introduction of negation as failure is therefore less pressing than the arithmetic facilities. Explaining when it can reliably be regarded as logical negation requires a careful development. This not only requires consideration of Prolog's goal reduction process, but explanation of the "Closed World Assumption". However, once its basis has been presented, negation as failure expands expressive power and allows inherently simple programs for some complex problems.

Recursion is often puzzling to beginning students, but in Prolog it is completely natural. For instance, from the logic point of view, the implication "if X is an ancestor of Y, and Y is a parent of Z, then X is an ancestor of Z" makes obvious sense logically. However, as a program, this introduces recursion. Of course, recursion still warrants discussion as it is another means to express repetitive computation (and possibly an infinite loop). But the discussion can begin from a clear and unequivocal logic basis.

In the latter stage of the course, it is useful to present the unification algorithm. The algorithm is sufficiently simple that students readily grasp it, and they thereby gain a better understanding of the procedural process that occurs in Prolog. It's also then possible to explain the "occur-check" concession made by Prolog.

When Dijkstra[4] invented the language he used in developing his predicate transformer methodology for program construction, he indicated that he "shuttered at the thought of introducing nondeterminicy" in his programming language. However, his formalism convinced him otherwise. The central role of predicates in Prolog also makes non-determinism completely natural (one might say logical), and it is one of the few languages to incorporate it. The uniform adoption of the relational view in Prolog makes non-determinism quite unremarkable, and multiple answers and backtracking are quite natural.

Prolog provides just two data structuring facilities, but they are two that are of central importance in computing. One is the list, and the student is bound to gain substantial experience with lists. The second data structure is the tree, rarely seen as a first-class data structure in programming languages. Every value in Prolog is a tree. It's instructive for students to encounter (and understand) the natural correspondence between Prolog's term presentation and the more common diagram presentation of trees. In the latter part of the course students are quite prepared to understand tree construction and tree-walking processes.

The author has found that a class segment on logic puzzles (e.g., the familiar Zebra, or Einstein's, Puzzle) is a useful conclusion that brings all of the discussed elements into play. These problems consist of the presentation of a collection of facts about some situation, followed by a question whose answer is to be deduced from these facts. Sounds rather simple in the abstract. However, in practice these puzzles are carefully designed to make it *barely* possible to reach the sought after conclusion, and if it is a good puzzle, it appears at first glance (and second glance too) that there is insufficient information to answer the question posed. A well-constructed logic puzzle presents a paramount deductive challenge in that we are asked to make a maximal conclusion from the bare minimum of premises. Students enjoy such problems and genuinely appreciate having a computer program generate the solution.

In the spirit of logic programming, several features of Prolog are *not* covered. Most notable would be the Prolog 'cut' operation. While this can be explained using the goal reduction model of Prolog, it does not have a meaningful description at the logic level, and for our purposes is better omitted. Also omitted are term inspection predicates (e.g., var(X)), DCG grammars, and character-level input-output. Finally, the emphasis on a higher level of abstraction offers limited circumstances that raise the issue of efficiency considerations.

As an entirely unconventional beginning programming course, there are few texts currently available. The author found that the standard Prolog book by Clocksin and Mellish[3] required only a modest amount of supplementary material. There are numerous implementations of Prolog available covering all platforms, including good quality implementations that students can obtain for their personal computers without charge. The author found SWI-Prolog[4] quite suitable. It's important

that whatever implementation is selected should provide a good execution tracing facility. This is an invaluable aid to students, and promotes better understanding of the goal-oriented, backtracking reduction process of Prolog.

4. CONCLUSIONS

Early developments in knowledge representation (see e.g., Brachman & Levesque[2]) and deductive database (see e.g., Ramamohanarao & Harlan[7]) were based on formal logic. Formal specification and proof of programs utilized formal logic at its outset (Hoare[5]). Dijkstra[4] initiated "predicate transformer" development methodology of programs, and provided an early caution on the detrimental effect of conventional programming languages on clear thinking. Using logic as a language of programming is not as unconventional as it may seem at first glance.

The intention of this approach in a first programming course is not to establish exceptional proficiency in Prolog programming (though this is not discouraged). Rather, the goal is to reveal the intrinsic link between analytical thinking and computer-generated solutions to problems, and to impart critical skills of logical analysis. This will provide the student with a solid conceptual foundation on which to build a deep understanding of computer programming. And the goal-oriented emphasis on achieving the solution to a problem by a combination of the solutions to simpler problems instills the top-down, divide and conquer strategy as a cornerstone in student thinking.

Just as in conventional programming, before a program can be written, students must learn to express characteristics of a problem domain with abstract computer representations. However, in contrast to conventional programming, the representations do not involve bytes or pointers, or other language specific "contraptions". Instead they consist of formulas utilizing predicates (i.e., relations) that interrelate significant aspects of the problem domain and this more directly connects the programming representation to the problem domain.

While it is contrary to traditional programming education to teach a first programming course using logic programming, experience has led me to the conclusion that this is a superior approach. Logic programming uses a level of abstraction that focuses on stating precisely what the problem is, rather than requiring contemplation of a variety of computing contraptions and their manipulation in solving a problem. This establishes a pattern of thinking that promotes sound analytical habits.

I have had the rare opportunity to employ the logic programming approach in a first programming course for college freshmen over a period of six years, and was delighted by how effective this approach proved to be. Early concerns of how well beginning students would fair with this material were quickly and consistently dispelled. The material was well received by students, and overall performance consistently exceeded that of students in a traditional beginning programming class.

The Computer Science Teacher's Association (CSTA) has recently formulated a K-12 model CS curriculum[1], and my experience with entering college freshmen indicates that Prolog would be nicely suited to a role at this earlier level as well. Since the K-12 model guides the preparation of our entering students, this is an important issue for college educators to address. In fact, the first programming course will soon be routinely taken before students enter college.

In a first programming course, logic programming has significant advantages. For students who continue study of programming using conventional languages, it promotes a systematic logical analysis of problems and thereby provides an invaluable foundation for all further programming. For those who continue no further in programming, it provides a widely available tool that is readily applicable to a broad range of computing tasks. The experience with careful logical thinking provides benefits in innumerable other ways.

5. REFERENCES

1. ACM K-12 CS Model Curriculum (2nd ed.). http://csta.acm.org/Curriculum/sub/ACMK12CSModel.html
2. Brachman, R. & Levesque H. *Knowledge Representation and Reasoning*. Morgan Kaufmann, 2004, 381 pp.
3. Clocksin, W. & Mellish C. *Programming in Prolog* (5th ed.). Springer, 2003, 299 pp.
4. Dijkstra E. *A Discipline of Programming*. Prentice-Hall, 1976, 217 pp.
5. Hoare C. An axiomatic basis for computer programming. *Commun. ACM* 12, 10 (1969), 376-380, 383.
6. Kramer. J. Is Abstraction the Key to Computing? *Commun. ACM*, V. 50, 4 (April 2007), 36-42.
7. Ramamohanarao K. & Harlan J. An Introduction to Deductive Database Languages and Systems. *VLDB Jour.* 3 (1994), 107-122.
8. SWI-Prolog Foundation, http://www.swi-prolog.org/