Prolog as description and implementation language in computer science teaching

Henning Christiansen
Roskilde University, Computer Science Dept.,
P.O.Box 260, DK-4000 Roskilde, Denmark
E-mail: henning@ruc.dk

Abstract

Prolog is a powerful pedagogical instrument for theoretical elements of computer science when used as combined description language and experimentation tool. A teaching methodology based on this principle has been developed and successfully applied in a context with a heterogeneous student population with uneven mathematical backgrounds. Definitional interpreters, compilers, and other models of computation are defined in a systematic way as Prolog programs, and as a result, formal descriptions become running prototypes that can be tested and modified by the students. These programs can be extended in straightforward ways into tools such as analyzers, tracers and debuggers. Experience shows a high learning curve, especially when the principles are complemented with a learning-by-doing approach having the students to develop such descriptions themselves from an informal introduction.

1 Introduction

Teaching of theoretical aspects of computer science to university students that do not necessarily possess a solid mathematical background may sound like a contradiction. The Advanced Studies in Computer Science at Roskilde University, Denmark, is a part of long tradition of interdisciplinary studies in which the same courses often are offered for classes of students with different backgrounds such as Natural Science, Humanities, or Social Sciences. Certain issues that are important for all sorts of teaching become extra critical in this context, and furthermore stressed by the fact that a tradition of 50% student project work throughout the studies leaves only very little time for regular courses. First of all, the presentation needs to be appealing and fruitful for every single student in this heterogeneous audience. Secondly, extreme care must be made in the selection of topics in order to provide a coherent course with a reasonable covering, considering that each course has few nominal hours. Finally, each course must be designed as a component of a full education comparable with any other five-year university education with computer science as a major subject.

This paper gives an overview of a teaching methodology developed under these conditions in which Prolog plays the combined role of as a study object and, more importantly, as a meta-language for describing and experimenting with different models of computation, including programming language semantics and Turing machines, and tools such as tracers and debuggers. The approach has been developed and successfully applied during the 1990s and used in courses until recently; a full account of the approach can be found in a journal paper [2] that also gives a more comprehensive set of references to related approaches; a locally printed textbook in Danish is available [1].

In the following, we analyze the qualities of Prolog that we have relied on in this approach, and we show how definitional interpreters, compilers and other models of computation can be defined in a systematic way as Prolog programs based on a general model of abstract machines. In this way, formal descriptions become running prototypes that are fairly easy to understand and appealing for the students to test and modify. The approach has turned out to be highly effective when combined with learning-by-doing which has been applied for type-checking and implementation of recursive procedures. A brief listing is given of other items treated in a course based on the these principles, and a sample course schedule is shown.

2 Qualities of Prolog in relation to teaching

Prolog is a wonderful programming language for any teacher of computer science: Students with or without previous programming experience can learn to write interesting programs with only a few hours of introduction and guided experiments in front of a computer. A substantial subset of Prolog exposes a mathematically and intuitively simple semantics and makes a good point to emphasize the distinction between declarative and procedural semantics, and thus also to isolate various pragmatic extensions from the core language.

Computer science as university subject contains many aspects where Prolog can be interesting, independently of whether the students intend to use Prolog in their future careers. First of all, Prolog is an obvious second programming language that shows the diversity of the field for student brought up with a language such as Java. Prolog is a type-less language in which any data structure has a denotation and with no need for constructors and selection methods as these are embedded in Prolog's unification. Java, on the other hand, requires the programmer to produce large collections of classes, interfaces, methods, and a test main method before anything can be executed. The conflict between flexibility, conciseness, and semantic clarity on the one hand, and security and robustness on the other is so obviously exposed in this comparison. Prolog's application as a database language is well-known and we shall not go into details here; in section 5 we mention briefly how an introduction to databases has been incorporated in our approach.

A study of Prolog motivates also considerations about the notion of a meta-language: `assert` and `retract` take arguments that represent program text, the

same goes for Prolog's approximation to negation-as-failure which essentially is a meta-linguistic device within the language. The problematic semantics of these features gives rise to a discussion of what requirements should be made to a meta-linguistic representation. Operator definitions in Prolog comprise syntactic meta-language within the language, and are also a perfect point of departure for a detailed treatment of priority and associativity in programming language syntax. In general, we have relied on the following detailed properties of Prolog.

- Prolog terms with operator definitions provide an immediate representation of abstract syntax trees in a textually pleasing form; see the following expression which with an operator definition for ":=" is a Prolog term:
  `a:= 221; b:= 493; while(a =\= b, if(a>b, a:= a-b, b:= b-a))`
- Structurally inductive definitions are expressed straightforwardly in Prolog by means of rules and unification, e.g.,
  `stmnt(while(C,S),···):- condition(C,···), stmnt(S,···), ···.`
- Data types for, say, symbol tables and variable bindings, are easily implemented by Prolog structures and a few auxiliary predicates.
- Specifications are directly executable and can be monitored in detail using a tracer; they can be developed and tested incrementally and interactively. Students can easily modify or extend examples and test their solutions. Prolog invites to an interactive and incremental style of program development, not only for students but also for the teacher to do this during the lecture using a computer attached to a projector.
- The characterization of various pragmatic issues can be developed in direct relation to "ideal" formal descriptions. An interpreter, for example, is easily extended into a tracer or debugger, and code optimization can be incorporated in a small compiler written in Prolog.
- Last but no least: Prolog appears as an easily accessible framework compared with, say, set and domain theory. Although basically representing the same universal concepts, the combined logical and operational nature of Prolog-based specifications gives an incomparable intuitive support.

3 A basic model of abstract machines

An unsophisticated model of abstract machines is a central element in our methodology, used for the general characterization of computer languages and computational models.

A particular *abstract machine* is characterized by its *input language* which is a collection of phrases or sentences, a *memory* which at any given time contains a value from some domain of values, and finally a *semantic function* mapping a phrase of the input language and memory state into a new memory state. For

simplicity, output is not explicit part of the definition but considered as part of the "transparent" memory whenever needed.

The framework includes a general notion of *implementation* of one machine in terms of another, and three different modes are defined, *interpretation*, *translation* and use of *abstraction mechanisms* in standard programming languages. Interpreters and translators themselves, as well as program modules, can be explained as particular abstract machines.

Abstract and concrete syntax are introduced and distinguished in an informal way, and the representation of abstract syntax trees by Prolog terms (as above) is emphasized. The abstract syntax of a context-free language is characterized by a recursive Prolog program consisting of rules of the form

cat_0`(`*op*`(T1,..., Tn)):-` cat_1`(T1),...,`cat_n`(Tn).`

where *op* names an operator combining phrases of syntactic categories cat_1, ..., cat_n into a phrase of category cat_0.

Syntax-directed definitions can be specified by adding more arguments corresponding to the synthesized as well as inherited attributes of an attribute grammar [5]. Consistent with our abstract machine model, we introduce what we call a *defining interpreter* which to each syntax tree associates its *semantic relation* of tuples $\langle s_1, \ldots, s_k \rangle$ by predicates of the form

cat_i`(`*syntax-tree*`,`s_1`,...,`s_k`)`

As an example, a defining interpreter for an imperative language may associate with each statement a relation between variable state before and after execution, which for a statement such as "`x:= x+1`" contains among others the following tuples: $\langle$`[x=7],[x=8]`$\rangle$, $\langle$`[x=1,y=32]],[x=2,y=32]`$\rangle$,

4 Imperative and procedural languages

In the following we show how standard programming languages are characterized in our Prolog-based style, indicating the spirit in which it is communicated in the teaching. We proceed by introducing a defining interpreter for a simple machine-like language giving a continuation-style semantics for jumps and control points. This serves the dual purposes of making the semantics of such languages explicit and of introducing continuations as programming technique and semantic principle. Next is shown a defining interpreter for while-programs and a compiler of while-programs into machine language. Finally we describe an assignment where the students developed type checker and interpreter for a simple Pascal-like language from a brief, informal introduction.

4.1 A defining interpreter for a machine language

The following Prolog list is an abstract syntax tree for a program in a simplified machine language. Presenting this sample to the students is sufficient to indicate the existence of an abstract machine, and it gives good sense to execute

this program by hand on the blackboard from the intuition provided by the instruction names.

```
[     push(2),
      store(t),
   7, fetch(x),
      ...
      equal,
      n_jump(7)]
```

The semantics of such programs assumes a stack (that we can represent as a Prolog list) and a storage of variable bindings (represented conveniently as lists of "equations", e.g., `[a=17,x=1,y=32]`). The central predicate in a defining interpreter is the following. The first argument represents a sequence of instructions (a continuation) to be executed and the second one passes the entire program around to all instructions to give the contextual meaning of labels.

$$\texttt{sequence}(\mathit{Seq}, \mathit{Prog}, \mathit{Stack}_{\mathit{current}}, \mathit{Store}_{\mathit{current}}, \mathit{Stack}_{\mathit{final}}, \mathit{Store}_{\mathit{final}})$$

The meaning of simple statements that transform the state is given by tail-recursive rules such as the following: Do whatever state transition is indicated by the first instruction and give the resulting state to the continuation. Example:

```
sequence([add|Cont], Prog, [X,Y|S0], L0, S1, L1):-
     YplusX is Y + X,
     sequence(Cont, Prog, [YplusX|S0], L0, S1, L1).
```

The unconditional jump instruction is defined as follows; it is assumed that the diverse usages of the `append` predicate have been exercised thoroughly with the students at an earlier stage.

```
sequence([jump(E)|_], P, S0, L0, S1, L1):-
     append(_, [E|Cont], P),
     sequence(Cont, P, S0, L0, S1, L1).
```

Executing a few examples, perhaps complemented by a drawing on the blackboard — and within a few minutes the students have grasped the principle of a continuation and continuation semantics. The remaining rules that complete the interpreter are straightforward.

A little aside can be made, turning the interpreter into a functioning tracer by adding the following rule as the first one to the interpreter:

```
sequence([Inst|_],_,_,_,_,_):- write(Inst), write(' '), fail.
```

Students are given the following exercises that serve the twofold purpose of familiarizing them with the material and introducing other important aspects: extend language and interpreter with instructions for subroutines; write a Prolog program checking that labels are used in a consistent way; write a Prolog predicate that optimizes selected subsequences of instructions; design and implement an extension of the tracer with debugging commands.

4.2 A defining interpreter for while-programs

As a next step up the ladder of languages moving away from the machine and closer to "problem-oriented" languages, we consider while-programs whose semantics also can be specified in terms of a defining interpreter. A defining interpreter consists of the following predicates.

`program(`*program*`,` *final-storage*`)`
`statement(`*statement*`,` *storage-before*`,` *storage-after*`)`
`expression(`*expression*`,` *storage*`,` *integer*`)`
`condition(`*condition*`,` *storage*`,` {`true`, `false`})`

Most rules are straightforward, the most complicated one being the following defining the meaning of a while statement.

```
statement( while(Cond, Stm), L1, L2):-
  condition(Cond, L1, Value),
  (Value = true -> statement((Stm ; while(Cond, Stm)),L1,L2)
                ; L1=L2).
```

The following exercises are given to the students: run a sample program including a while loop with Prolog's debugger switched on and record all primitive actions; extend the language with expressions of the form `result_is(` *statement*`,` *variable*`)`; extend the language with a `for` loop; extend the interpreter with a simple tracing facility.

4.3 A compiler for while-programs

The structure of our defining interpreters can also be adapted to describe compilers. Above, we considered a semantics for while-programs defined in terms of state transformations and now we consider an alternate semantics capturing meanings by means of sequences of machine instructions. Two auxiliary predicates are introduces, one for creating unused machine language labels and another one to facilitate the composition of sequences of instructions; illustrated below. The following rule specifies the compilation of a while statement.

```
statement( while(Cond, Stm), C):-
      condition(Cond, CondC),
      statement( Stm, C1),
      new_label(Lstart), new_label(Lend),
      C <-  Lstart + CondC +
                    n_jump(Lend) +
                    C1 +
                    jump(Lstart) +
            Lend.
```

The compiled code for the while statement is composed by the code for its constituents, two new labels created by `new_label` and specific instructions; the predicate denoted by "`<-`" puts together the sequence indicated by "`+`" in its

second argument and unifies it with the first argument. Notice that `n_jump` is a conditional jump to the specified label whenever the previous computation has placed a value representing false on top of the stack. The code produced can be executed by the interpreter shown in section 4.1. As before, exercises are given that involve testing and extending this compiler in various ways.

4.4 A learning-by-doing approach to recursive procedures and type-checking

The detailed semantics and implementation of recursive procedures and type-checking are usually consider very difficult by students. We have had good success with these topics by means of a larger learning-by-doing assignment continuing the material presented so far.

The students were presented for a simple Pascal-like language by means of example programs with a recursive quicksort program as a prototypical representative. Type requirements and a standard stack-based implementation principle for recursive procedures were described informally, and the assignment was to implement both type-checker and compiler in Prolog.

The prescribed time for the work was one week on half time, including writing a small report documenting the solutions. The most experienced students had type checker and interpreter running after four or five hours, and all students in a class of some 30 students solved the task within the prescribed time. All solutions were acceptable and there was no obvious difference between those produced by students with a mathematical background and by those without.

5 Other course elements

Here we list other topics integrated with the previous material in different versions of our course; more details including program samples can be found in [2].

Logic circuits modeled in Prolog is a standard example used in many Prolog text books. This is obvious to apply in our context due to the meta-linguistic character (modeling the language of logic circuits).

LISP modeled with assert-retract. Function definitions and variable bindings are implemented using Prolog's assert-retract. Illustrates dynamic binding and different levels of binding times plus introduces functional programming. The use of assert-retract as opposed to explicit state arguments makes it possible to model an interactive Lisp environment with few lines of codes.

Turing machines. An introduction to computability theory is given, based on Turing machines and Turing completeness. An interpreter made up by a few lines of Prolog is an excellent way to illustrate a Turing-machine and to provide a truly dynamic model, especially when a tracing facility is added. The existence of the interpreter shows that Prolog is Turing-complete, and having played with it makes it easier for the students to understand the proof of undecidability of the halting problem.

Vanilla and Prolog source-to-source compilation. The familiar Vanilla self-interpreter for Prolog [7] is a perfect example to illustrate the notion of a self-interpreter. Appearing a bit absurd and useless to the students in the first place, they begin to see the point of a self-interpreter when a few lines of additional code makes it into a tracer and debugger. Source-to-source compilation is illustrated in terms of a profiling tool that inserts additional code to record the number of entrances, successes and failures of each clause in a Prolog program.

Relational algebra in Prolog. The course described here has in some years been integrated with a standard database course. As an introduction to relational database technology, students were given the assignment of implementing an interpreter for relational algebra. The conditions were the same as for the task on type-checking and recursive procedures described above, one week on half time, including writing a small report documenting the solutions. This task has been given to several classes of students and all students usually succeed in producing an acceptable solution, although `join` often causes problems.

Syntax analysis. Traditional methods for lexical analysis and parsing are integral components of our course. Prolog is used as a ready-at-hand tool for the students to implement finite state machines, deterministic as well as nondeterministic. Top-down parsing is illustrated perfectly by Prolog's built-in Definite Clause Grammars [6], and bottom up-parsers by an analogous grammar formalism CHRG [3] developed on top of Constraint Handling Rules [4] which is a recent extension to some Prolog versions that provides a natural paradigm for bottom-up evaluation. Now quick and effective introductions can be given to standard implementation principles for finite state machines and parsing.

Dissecting a Prolog implementation in Java. As a conclusion of the course, the students are shown a full implementation in Java of a subset of Prolog, including lexical analysis, parsing, representation of abstract syntax trees in an object-oriented language, and an interpreter which exposes a detailed implementation of Prolog's unification procedure.

6 A sample course schedule

The following table shows the schedule for a version of a course designed according to our methodology as it was given in spring 2001. The actual course has changed slightly from semester to semester so not all items mentioned above are included. The course corresponds to 25% of a student's work in one semester (7.5 ECTS) and is concentrated on 10 full course days. Each course day consists of lectures and practical problem solving related to the day's lecture. A considerable amount of homework is expected from the students.

1	Introduction: Abstract and concrete syntax, semantics, pragmatics, language and meta-language. Prolog workshop I: The core language, incl. structures.
2	Prolog workshop II: Lists, operators, assert/retract, cut, negation-as-failure.
3	Abstract machines: Definitions of a.m., interpreter, translator, etc. Prolog workshop II contd.
4	Language and meta-language, Prolog as meta-language. Semantics of sequential and imperative languages; defining interpreters and a small compiler.
5	Declarations, types, type checking, context-dependencies, recursive procedures.
6	Introduction to and practical work with large exercise: do-it-your-self recursive procedures, interpreter and type checker.
7	Conclusion and comments to large exercise. Turing-machines, decidability and computability, Turing universality, the halting problem, Turing machines in Prolog.
8	Constraint logic programming: Introduction to CLP(R) and CHR; CHR Grammars for bottom-up parsing.
9	Syntax analysis: Lexical analysis and parsing; recursive-descent parsing
10	Overview of phases in a traditional compiler. Dissection of an implementation of Prolog in Java. Evaluation of the course.

7 Conclusion

We have explained a methodology based on a combination of a simple, underlying model of abstract machines and the use of Prolog as general definition and implementation language. Prolog is well suited for this purpose: Conceptual simplicity and high expressibility with a core language consistent with a subset of first-order logic; syntactic extensibility that allows a direct notation for abstract syntax trees in a textually acceptable form; a rule-based structure that fits perfectly with an inductive style of definition. Last but not least: Prolog is an interactive language that appeals to incremental development, testing, and experimentation with an extremely short turn-around time from idea → implementation → observation → revision or extension of idea.

Our experience have shown that theoretical issues of computer science can be taught in this way in an entertaining and concrete way which, unlike traditional approaches, appeals to a wide range of students for which a uniform mathematical background cannot be taken for granted.

A critical remark may be that this form of learning is very compact, with many important aspects covered by one minimalist and seemingly innocent example as was the case with the interpreter for machine language. One might fear that students tend to remember only the example and not the points that the teacher had in mind. We have not applied any scientifically based evaluation principle, but it is our clear impression that the practical work in exercises and larger assignments serves fully to avoid this potential danger. Informal evaluations with the students have indicated a high degree of satisfaction with the teaching principle. Especially the larger learning-by-doing assignments (type-

checking plus recursion; relational algebra in Prolog) were characterized as difficult and challenging, but also some of the most interesting ones from which the students had learned quite a lot.

Acknowledgment: This research is supported in part by the IT-University of Copenhagen.

References

[1] Henning Christiansen. Sprog og abstrakte maskiner, 3. rev. udgave [in Danish; eqv. "Languages and abstract machines"]. Datalogiske noter 18, Roskilde University, Roskilde, Denmark, 2000.

[2] Henning Christiansen. Teaching computer languages and elementary theory for mixed audiences at university level. *Computer Science Education Journal*, 14, 2004. To appear.

[3] Henning Christiansen. CHR Grammars. *Int'l Journal on Theory and Practice of Logic Programming*, 2005. To appear.

[4] Thom Frühwirth. Theory and Practice of Constraint Handling Rules, Special Issue on Constraint Logic Programming. *Journal of Logic Programming*, 37(1–3):95–138, October 1998.

[5] Donald Knuth. Semantics for Context-Free Languages. *Mathematical Systems Theory*, 2:127–145, 1968.

[6] F. C. N. Pereira and D. H. D. Warren. Definite clause grammars for language analysis — a survey of the formalism and a comparison with augmented transition networks. *Artificial Intelligence*, 13:231–278, 1980.

[7] D. H. D. Warren. Implementing Prolog - Compiling Predicate Logic Programs – Volumes 1 & 2. D.A.I. Research Report 39, 40, University of Edinburgh, May 1977.