Using Prolog as metalanguage for teaching programming language concepts

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

1 Introduction

We describe our experiences with a methodology for teaching programming language concepts in which Prolog is applied as metalanguage. Prolog can be seen as an integration of a logical specification language and a collection of practical accessories expected for an interactive, general purpose programming environment. This, together with the easy-to-extend-and-modify characteristics of Prolog programs, makes Prolog ideal in a teaching context: Assuming the students have grasped the overall flavour of Prolog, specifications written in Prolog are immediately understandable to them and this understanding can be further supported by running small examples, either by themselves or by the lecturer using a projector and running computer session in the lecture room.

We do not pretend to have made new contributions to the theory of programming language descriptions. Our semantic descriptions are closely related to Plotkin's operational semantics [18] (see also [23]). But in our setting, these specifications are actually running interpreters that serve as prototype implementations. In addition, they are easily extended into prototypes of programming tools such as tracers, debuggers, time-measurers, etc. so that these can be characterized in a systematic framework. The familiar relation between typing/type checking and unification is also used for describing concise type-checkers with similar qualities.

In a course on programming languages, Prolog can also serve as an interesting example of a "different" language compared with standard imperative or object-oriented languages. In the teaching of Prolog in this context, it is obvious to emphasize Prolog's inherent meta-circular facilities such as assert-retract and its approximation to negation-as-failure and, at the syntactic level, operator declarations. The obvious imperfections of some of these facilities are not a problem: They can be featured in the course as points of departures for critical discussions of language design and pragmatics.[1] Putting emphasis on a mathematically-logical (or simpler, a set-theoretic) semantics for the core of Prolog helps students accept our in-Prolog specifications as

[1] A mechanism may be simplistic and error-prone but used in a proper way, preserve an acceptable semantics.

"formal" descriptions. Finally, this sort of language definitions serves also the extra purpose as strong evidence for the qualities of declarative programming.

In the following, we describe our experiences with the methodology giving samples of such language descriptions and explain how they are applied in our teaching. We may at this point summarize a few observations.

- The approach appeals to students with or without a mathematical background. It has been developed in a context with quit differentiated student backgrounds with humanity and natural science students in the same class.
- It is easy for all students to grasp these language descriptions, that can be understood as "formal specifications" as well as "running prototypes". The model appeals to the students, much more than explicit mathematical formalisms such as denotational semantics [13] that appear quite heavy to some students.
- Exercises related to these inherently theoretical topics can be given in a practical setting: Extend textbook's example with this and this new construction, modify the semantics of another, etc. — and use the Prolog system for developing and testing the solution.
- More complicated topics such as recursive procedures with side-effects or typing for object-oriented languages can be taught by the learning-by-doing principle: Brief informal introduction is given together with a small toolbox of auxiliary predicate and the students must develop and test an executable specification in Prolog. An introduction to relational algebra has also been giving in such a way.
- The learning methodology has proved very effective and efficient measured in the materiel covered in lecture and the students exam marks. Student's comments indicate that they appreciate this form of learning.

1.1 Related work

Our methodology is currently documented in a locally printed textbook [6] in Danish, which still needs to be matured in to an internally publishable edition.

The textbook by Slonneger and Kurtz [19] applies Prolog in a way quite similar to ours, although they use Definite Clause Grammars extended with semantic and other attributes in contrast to our interpreters and compilers that work directly on abstract syntax trees. Although being a minor detail, the use of abstract syntax trees (as opposed to lists of tokens) seems to be more appropriate to emphasize the structural aspects of languages and to result in a simpler presentation. The biggest difference in the two approaches is at the pedagogical level: Their book addresses a mathematically competent audience where our approach aims at a broader and inhomogeneous audience as mentioned above. They also introduce to lambda calculus as well as to denotational, algebraic and axiomatic semantics. We were unaware of [19] while our material was developed in successive revisions of course notes.

Compiler writing in Prolog has been considered by several authors, e.g., [21, 15]. Semantics of programming languages specified in Prolog or Prolog-like languages (often implemented on top of Prolog) is not uncommon, e.g., [3, 4, 9, 17]. The close

relation between attribute grammars [11] and Prolog has also been inspiring for the referenced works, evident in the notation applied by [22] and formally spelled out by [8].

1.2 Overview

In the following we first state the reasons for our choice of Prolog as metalanguage and indicate the general pattern for the way it is used. In section 3 we give an example of a language definition for a simple machine language; we detail here the way we present it to the students and how we use such language descriptions as a means for emphasizing important points. Sections 4 and 5 show defining interpreter and a small compiler for while-programs. A variation of the methodology, by having the students to develop the language descriptions themselves, is described in section 6 concerned with type-checking and implementation of recursive procedures for a Pascal-like language. Section 7 reviews briefly a series of other examples used in our course including LISP with side-effects, relational algebra, Turing-machines, and programming tools such as tracers and debuggers. Section 8 provides a conclusion including a discussion of related work.

2 Why Prolog?

First of all, Prolog is an obvious second programming language for student brought up with a language such as Java in order to indicate the diversity of the field: Concise expression, an interactive language, free of writing thousands of classes, interfaces, and methods before anything can be executed, reversibility, self-modifying programs, etc.

Secondly, a study of Prolog motivates considerations about the notion of a metalanguage: assert-retract take arguments that represent program text, the same goes for Prolog's approximation to negation-as-failure which essentially is a metalinguistic device within the language, and the problematic semantics of these features gives rise to a discussion what requirements should be made to a metalinguistic representation.[2] Operator definitions in Prolog comprise syntactic metalanguage within the language, and are also a perfect point of departure for a detailed treatment of priority and associativity in programming language syntax.

To have Prolog serve as a general purpose metalanguage for characterizing programming language notions, we use the following properties.

- Prolog terms with operator definitions provide an immediate representation of abstract syntax trees in a textually pleasing form, cf. the following expression which with one additional operator definition is a legal Prolog term:

```
while( x<y, (x:= x+y ; y:= y+1))
```

[2] The different answers produced for the two queries `?- \+p(X),X=b` and `?- X=b,\+p(X)` to the program consisting of `p(a)` shows the problem of using Prolog variables as a metalevel representation of Prolog variables; see [7, 10] for more detailed discussions.

- Structurally inductive definitions are expressed straightforwardly in Prolog by means of rules and unification, e.g.,

```
statement(while(C,S),···):- condition(C,···),
      statement(S,···), ···.
```

- Data types for, say, symbol tables, variable bindings, are easily implemented by Prolog structures and perhaps a few auxiliary predicates.

- Last but no least: Prolog is appears as a light-weight framework conceptually speaking compared with, say, set and domain theory. 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.

In the course, we may start characterizing the set of abstract syntax trees for a (context-free) language by a recursive Prolog program consisting of rules of the form

cat_0`(`*op*`(`T_1`,...,` T_n`)):-` cat_1`(`T_1`),...,`cat_n`(`T_n`).`

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 inherited attributes in an attribute grammar [11].

An important kind of definition is what we call a *defining interpreter* which to each syntax associates its *semantic relation* of tuples $\langle s_1, \ldots, s_k \rangle$ by predicates of the form

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

As an example, a defining interpreter for an imperative language may associate with the statement "`x:= x+1`" a relation containing among others the following tuples.

```
                ⟨[x=7]   ,  [x=8]⟩
              ⟨[x=666]   ,  [x=667]⟩
          ⟨[x=1,y=32]    ,  [x=2,y=32]⟩
    ⟨[a=17,x=1,y=32]     ,  [a=17,x=2,y=32]⟩
```

In the course, we introduce also a general model of abstract machines by means of which correct interpreters and translators are characterized; however, these definitions are quite standard and not interesting to present here.

3 Example: Defining interpreter for a machine language

We illustrate the way we intend these language descriptions applied in teaching by a definition for a small machine language that we present here in very much the same way as we do it to the students. The reader should imagine a lecture room with a projector connected to a computer with a running Prolog session plus a blackboard available for writing keywords and improvising different drawings and explanations. Now the lecture is supposed to starts.

Machine languages are characterized as sequences of simple state transformation executed in their textual order, however broken by jump instructions whose meaning depends on the labels in the current program. In order to provide a formal presentation to the students, we introduce a simplified machine language by means of the following sample program represented as a Prolog list. The (yet) uncommented example, by the names chosen for the instructions, is intended to trigger the intuition of the existence of some abstract machine.

```
[     push(2),
      store(t),
   7, fetch(x),
      push(2),
      add,
      store(x),
      fetch(t),
      push(1),
      subtract,
      store(t),
      fetch(t),
      push(0),
      equal,
      n_jump(7)]
```

Without any further introduction, the teacher can execute this program by hand on the blackboard. 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]`. Two auxiliary predicates are introduced in order to work with stores.

`store(`*VariableID*`,` *Value*`,` *Store*`,` *UpdatedStore*`)`

`fetch(`*VariableID*`,` *Value*`,` *Store*`)`

The necessary Prolog code to implement these are shown to the students with the behaviour tested in the lecture room or by having the students to do small exercises themselves.

The central predicate in a defining interpreter is the following. The first argument represents a sequence of instruction (a continuation) to be executed and the second one passes the entire program around to all instructions to give the contextual meaning of labels.

`sequence(`*Sequence*, *WholeProgram*`,` *CurrentStack*`,` *CurrentStore*`,` *FinalStack*`,` *FinalStore*`)`

Before giving the details of this, we set up a definition for a whole program as follows.

```
machine_program(Prog, FinalStack, FinalStore):-
      sequence(Prog,Prog,[],[],FinalStack,FinalStore).
```

The meaning of simple statements that transform the state are 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.

```
sequence([push(N)|Rest], Prog, S0, L0, S1, L1):-
      sequence(Rest, Prog, [N|S0], L0, S1, L1).

sequence([fetch(Var)|Rest], Prog, S0, L0, S1, L1):-
      fetch(Var,X,L0),
      sequence(Rest, Prog, [X|S0], L0, S1, L1).

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

These rules are tested on the computer and compared with drawings on the blackboard if needed. The similar rule for subtraction is a good point for discussing the order of the arguments:

```
sequence([minus|Rest], Prog, [X,Y|S0], L0, S1, L1):-
      YminusX is Y - X,
      sequence(Rest, Prog, [YminusX|S0], L0, S1, L1).
```

At this stage, the audience is warmed up for the more interesting cases. The unconditional jump instruction is defined as follows; it is assumed that the diverse use of the `append` predicate has been exercised thoroughly with the students.

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

Executing a few examples, perhaps complemented by a drawing on the blackboard — and within a few minutes the students has grasped the principle of a continuation and continuation semantics.

The following two rules defining conditional jumps serve as an immediate repetition of the principle.

```
sequence([n_jump(E)|_], P, [0|S0], L0, S1, L1):-
     append(_, [E|Continuation], P),
     sequence(Continuation, P, S0, L0, S1, L1).

sequence([n_jump(_)|Continuation], P, [1|S0], L0, S1, L1):-
      sequence(Continuation, P, S0, L0, S1, L1).
```

Now we need only provide the rules for skipping over labels in a sequence and for stopping a run.

```
sequence([Label|Rest], P, S0, L0, S1, L1):-
     integer(Label),
```

```
    sequence(Rest, P, S0, L0, S1, L1).

sequence([],_,S,L,S,L).
```

Having some experience with Prolog programming, it is obvious to the students that these rules are necessary in order to provide a complete definition. The entire defining interpreter is now finished and can be tested on the sample program shown as an introduction (and the students are receptive to the teachers proclamations concerning precise language definitions at the one hand and the implementation of various languages by an interpreter on the other).

As a final piece of candy, the following rule is added as the first one to the interpreter:

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

This turns it into a functioning tracer, thus emphasizing the advantages of having formal specifications integrated in general purpose, interactive programming environment.

The following exercises are given to the students in order to provide a hands-on feeling and to give them an impression of the power in being able to design themselves new language constructions and facilities.

- Extend language and interpreter with instructions for subroutines: `jump_sub(`*to-label*`,` *return-to-label*`)` and `return`. Provide an interesting sample program in the extended language for testing the solution.
- Write a Prolog program checking that labels are used in a consistent way (part of exercise is to define what that means). Prize is given for the most elegant solution.
- Design and implement an extension of the tracer so it becomes a debugger with possibility to change variables, affect outcomes of conditional jumps and allow arbitrary number of undo's of execution steps.[3]
- Examples are shown of how subsequences of instructions can be replaced by other and more efficient ones. Write a Prolog predicate that performs such optimizations.
- Mathematically oriented students are given a few hints so they can work out a mathematical version of what they have seen.

Moving up to the meta-pedagogical level, we conclude that this (part of a) lecture with exercises, built around a seemingly innocent example, in a compact but digestive way has established important pieces of knowledge and methodology that otherwise may be quite an obstacle for some students.

[3] . . . last topic is perfect training for those students who wants to master the powerful control device provided by Prolog's backtracking.

4 Example: Defining interpreter for while-programs

The detailed comments to the previous example have indicated the spirit in which we communicate knowledge of computer languages to the students; the following examples are given in a more compact way. Now we consider while-programs of which the following sample, representing Euclid's algorithm for greatest common divisor, is a prototypical example.

```
a:= 221 ; b:= 493 ;
while( a =\= b,
         if( a>b, a:= a-b, b:= b-a))
```

Abstract syntax trees are represented as Prolog terms but at the same time with an acceptable concrete appearance. 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`}`)`

Some of the rules of this interpreter are shown in the following; the most important ones are for the `if` and `while` statements. Notice that we reuse the storage structure and auxiliaries from the previous example.

```
program(P, Storage) :- statement(P, [], Storage).

statement((Var := Expression), L1, L2):-
      expression(Expression, L1, Value),
      store(Var,Value,L1,L2).

statement( (S1 ; S2), L1, L3):-
      statement(S1, L1, L2),
      statement(S2, L2, L3).

statement( if(Cond, Smt1, Stm2), L1, L2):-
      condition(Cond, L1, Value),
      (Value = true -> statement(Stm1, L1, L2)
                           ; statement(Stm2, L1, L2)).

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

expression(Variable, L, V):- atom(Variable),
      fetch(Variable,V,L).
```

```
expression( Tal, _, Tal):- integer(Tal).

expression( (Exp1 + Exp2), L, Res):-
      expression( Exp1, L, V1),
      expression( Exp2, L, V2),
      Res is V1 + V2.

condition( true, _, true).

condition( false, _, false).

condition( (Exp1 = Exp2), L, Res):-
      expression( Exp1, L, V1),
      expression( Exp2, L, V2),
      (V1 = V2 -> Res = true ; Res = false).
```

Presentation of this defining interpreter and the exercises given can follow the pattern we showed above. In case the students have been presented earlier for Hoare logic, e.g., as part of a programming course, there is another good exercise in formalizing this as an interpreter in Prolog.

5 Example: Compiler for while-programs

The structure of our defining interpreters can also be adapted to describe compilers. We considered above 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.[4]

We introduce two auxiliaries, one to generate new unused labels and another one providing syntactic sugar for putting together sequences of instruction sequences and single instructions. They are illustrated in the following example query.[5]

```
?- new_label(L1), new_label(L2), C1 = [push(1),add],
   C2 <- L1 + push(7) + L2 + C1.

L1 = 117
L2 = 118
C1 = [push(1),add]
C2 = [117,push(7),118,push(1),add]
```

Now a simple, non-optimizing compiler for while-programs can be presented as follows (selected rules only).

[4] As mentioned, the correctness of a translator has been defined for the students within a model of abstract machines.

[5] Depending on the level of experience in Prolog programming, the students may be give as an exercise to program these auxiliaries or are given the definitions in the lecture.

```
program(P, K):- statement(P, K).

statement((S1 ; S2), C):-
      statement(S1, C1),
      statement(S2, C2),
      C <- C1 + C2.

statement((Var := Exp), C):-
      expression(Exp, C1),
      C <- C1 + store(Var).

statement( if(Cond, Stm1, Stm2), C):-
      condition(Cond, CondC),
      statement(Statement1, C1),
      statement(Statement2, C2),
      new_label(L2), new_label(L_end),
      C <-    CondC +
              n_jump(L2) +
              C1 +
              jump(L_end) +
         L2 + C2 +
         L_end.

statement( while(Cond, Stm), C):-
      condition(Cond, CondC),
      statement( Stm, C1),
      new_label(Lstart), new_label(Lend),
      C <-  Lstart + CondC +
                     n_hop(Lend) +
                     C1 +
                     hop(Lstart) +
            Lend.

expression(Number, C):-
      integer(Number),
      C <- push(Number).

expression( Variable, C):-
     atom(Variable),
     C <- fetch(Variable).

expression((Exp1 + Exp2), C):-
      expression(Exp1, C1),
      expression(Exp2, C2),
      C <- C1 + C2 + add.
```

These rules can be tested one after one during the lecture as they are introduced. Finally, we can combine the compiler with the defining interpreter for machine programs as follows.

```
?- program( ( a:= 221 ; b:= 493 ;
               while( a =\= b,
                      if( a > b,
                          a:= a-b,
                          b:= b-a))), C),
     sequence( C, _, L).

C = [  stack(221),store(a),stack(493),store(b),
     2,fetch(a),fetch(b),not_equal,n_jump(3),
       fetch(a),fetch(b),greater,n_jump(0),
       fetch(a),fetch(b),minus,store(a),jump(1),
     0,fetch(b),fetch(a),minus,store(b),
     1,jump(2),
     3],
L = [a=17,b=17]
```

Student exercises may consist of adapting this compiler to extensions to the while-language also considered in earlier exercises related to the defining interpreter. The optimizer for machine programs considered in an earlier exercise can be applied at different level of granularity.

The purpose of presenting this little compiler is manyfold: It illustrates the notions of a compiler and of syntax-directed translation and makes the distinction between interpretation and compilation clear, it shows how standard imperative constructs are mapped into machine language, and it can serve as an appetizer for more serious studies of compilers, e.g., [2]. Finally, it serves as an introduction to the larger learning-by-doing exercise described in the following section.

6 Example: Do-it-yourself recursive procedures

Instead of always presenting ready solutions to the students, it is also motivating, once they have become familiar with the principles, to work out nontrivial examples by themselves.

In the following, we sketch a larger exercise in which a class of students had to produce a type checker and an interpreter for a Pascal-like language with arrays and side-effects. The following, recursive quicksort program is prototypical; notice that an "^" operator is used for array-indexing.

```
program(
    (var(n,int); var(a, int_array(4))),
    declare_proc( qsort, left, right,
       (var(i,int); var(j,int); var(x,int); var(w,int)),
       (i:= left; j:= right; x:= a^( (left+right)//2) ;
```

```
    repeat( (while(a^i<x, i:= i+1) ;
             while(x<a^j, j:=j-1) ;
             if(i=<j, (w:=a^i; a^i:= a^j; a^j:= w;
                       i:= i+1; j:= j-1))),
    % until
             i > j); % end repeat
    if( left<j, proc_call(sort,left, j)) ;
    if( i < right, proc(sort,i,right)) )
), % end proc qsort

% main program:
(n:= 4; a:= [30,10,40,20];
 proc(qsort,1,n); write(a)))
```

The syntax, including scope and type principles, and semantics of the language was described informally to the students and their task was to produce type-checker and interpreter to be tested on a number of sample programs, including the one shown above.

The students had programming experience in advance with this kind of languages but the first systematic introduction to types and type-checking were given to them by the text (plus brief introduction lecture) of the present exercise. In order to simplify their work, they were given auxiliary predicates for working with symbol tables and runtime stacks, but with only a sketchy explanation of how to use these tools for the tasks. So the students' task was to put the whole machinery together and test it.

The prescribed time for the work was one week on half time, including writing a small report documenting their solutions; they could work in groups of up to three students. The most experienced students had the type-checker and interpreter running after four or five hours, and all students within a class of some 30 students solved the task within the prescribed time. All solutions were of good quality and there was no obvious difference between those produced by students with a mathematical background and by those without. In the general, the students characterized this exercise as difficult and challenging, but one of the most interesting ones from which they had learned a quite lot.

For our reader we show some fragments of a possible solution. Let us make precise some assumptions about the language. Procedures take always two integer parameters and local (as well as global) variable declarations may introduce integer and array variables. There are no local procedures, so the runtime stack can be organized as a list of stack frames, each being a list of bindings; looking up a variable can be done by looking first in the topmost frame and if not found, in the bottom frame.

The anticipated solution makes recursive calls to same and previous procedures possible. The type checker can be defined by a predicate `tc_`*cat*`(`tree`,` *current-table*`,` *updated-table*`)` for those syntactic categories *cat* whose phrases are intended to introduce new nomenclature, and with fewer arguments for other syntactic categories. A sufficient type checker rule for a single procedure declaration is the following.

```
tc_proc_decl(
  declare_proc(ProcId,ParId1,ParId2,LocalVarDecls, Stm),
```

```
  Table1, Table2):-
    tc_identifier(ProcId),
    Table2 = [(ProcId, procedure2) | Table1],
    tc_identifier(ParId1), tc_identifier(ParId2),
    Table3 = [(ParId2, int),(ParId1, int)|Table2],
    tc_var_decl(LocalVarDecls,Table3,Table4),
    tc_statement(Stm,Table4).
```

Correct typing of a procedure call is expressed in the following way.

```
tc_statement( proc_call(ProcId, Exp1, Exp2),Table):-
    symbol_tabel_find(ProcId, Table, procedure2),
    tc_expression(Exp1,Table,int),
    tc_expression(Exp2,Table,int).
```

For the interpreter, we give the flavour of a solution by showing the most complicated rule which is the one for procedure calls. Each statement is executed relative to a table of procedure closures and a runtime stack and produces an updated runtime stack. The procedural meaning of local variable declarations is to extend a current stack frame with "locations" for the variables as to produce a new frame.

```
statement(proc_call(ProcId, Exp1, Exp2),
                  ProcTable, Stack1, Stack2):-
    member( proc(Id,ParId1, ParId2, LocalVarDecls, Stm),
                  ProcTable),!,
    expression(Exp1, Stack1, ParValue1),
    expression(Exp2, Stack1, ParValue2),
    var_decl(LocalVarDecls,
      [(ParId2,ParValue2),(ParId1,ParValue1)], StackFrame),
    statement(Stm, [StackFrame|Stack1],
      ProcTable, [_|Stack2]).
```

7 Other examples

We sketch briefly a number of other examples which have been used in our course. Logic circuits modelled in Prolog is a standard example used in many Prolog text books and is obvious to apply in our context due to metalinguistic aspects (modelling the language of logic circuits); we refrain from giving details. The following examples show different aspects of Prolog as well as other languages and programming tools.

7.1 LISP modelled with assert-retract

This example goes to the limit of our paradigm of using Prolog as a logical specification language. In this way, the presentation becomes a bit provocative and can initiate discussions about what requirements should be made in general to a specification and to metalanguages.

By means of Prolog's assert and retract facilities, we define an interpreter for a small LISP-like language with function definitions and variable assignments modelled as side-effects. This way we provide a model of an interactive LISP environment [12]. Here follow a few rules that show the principle; notice also the pragmatic aspect present in error messages in some rules.

```
lisp([quote,X], X).

lisp( [plus,X,Y], Value):-
    lisp(X, Xvalue),
    lisp(Y, Yvalue),
    Value is Xvalue + Yvalue.

lisp([car,X], Value):-
    lisp(X, Xvalue),
    (Value = [Value | _] -> true
     ;
     nl, write('CAR of non-list: '),
     write(Xvalue), abort).

lisp([setq,Var,X], Xvalue):-
    lisp(X, Xvalue),
    asserta((lisp(Var, Xvalue):- !)).
```

The last rule gives rise to a discussion of binding times and a critique of Prolog for the lack of indication of different binding times for `Var`, `X` and for `Xvalue`; this is another way of showing the problems inherent in the nonground representation of Prolog in itself.

The following rule for function definitions with a single parameter emphasizes the problem but shows also many interesting programming languages aspects such as extensibility, parameter transmission and, again, different binding times.[6]

```
lisp([defun, F, Param, Body], F):-
    asserta((lisp( [F, Arg], Value):- !,
                lisp(Arg, ArgValue),
                asserta((lisp(Param, ArgValue):- !)).
                lisp( Body, Value),
                retract((lisp(Param, ArgValue):- !)) )).
```

Here the teacher has a good occasion for critisizing the nonground representation for very practical reasons: It makes the specification almost unreadable. This suggest the design of a new syntax (from [5]) for a ground representation with one or more prefix asterisks to indicate binding time for represented variables.[7]

[6] Running a definition for a recursive LISP function in a Prolog environment that reflects assertions immediately in the program window illustrates in an effective way the principle of a recursion stack.

[7] The present rule does not show multiple asterisks, but we can illustrate their use by the an alternative way of indicating the rule to be asserted when the actual parameter has been evaluated: `lisp(Param, **result):- **result = *argValue.`

```
lisp([defun, F, Param, Body], F):-
    new_asserta(
        (lisp([F, *arg], *value):- !,
           lisp(*arg, *argValue),
           new_asserta((lisp(Param, *argValue):- !)).
           lisp(Body, *value),
           new_retract((lisp(Param,*argValue):- !)))).
```

The possible exercises to be given to the students following the lecture include:

- Extend the interpreter to handle the `eval` function and test it on given examples.
- Implement a version with call-by-name parameters.
- Examine the given interpreter and add complete error messages; what does "complete" mean?
- Analyze the interpreter to figure out what happens when formal parameters are `setq`ed inside the body of a functions. Discuss different possible semantics and test them.
- Write a definition of a nullary predicate `run_lisp` that adds a read-eval-print loop upon the interpreter.
- Add a debugging facility to the interpreter.
- Implement the suggested `new_asserta` and `new_retract` predicates.
- Write program transformers that can apply to the body of function definitions, e.g., getting rid of explicit parameter references and using substitution by means of Prolog variables instead.

7.2 Turing machines

Turing machines are an interesting topic in itself in a programming language course, and showing a Turing-machine interpreter in Prolog is an obvious way to provide a truly dynamic model of a Turing machine, especially when a tracing facility is added. The definition of such an interpreter is straightforward and not shown here. The existence of the interpreter shows that Prolog is Turing-complete and playing with it warms up the student for the proof of undecidability of the halting problem. Exercises consist of writing small Turing-machines (including a "copy machine" often used in the mentioned proof) and extending the interpreter to handle multi-tape machines.

7.3 Playing with Vanilla and Prolog source-to-source compilation

The familiar Vanilla self-interpreter for Prolog [20] is a perfect example to illustrate the notion of a self-interpreter.

```
solve(true).
solve((A,B)):- solve(A), solve(B).
solve(A):- clause(A,B), solve(B).
```

It may appear a bit absurd and useless to the students until we begin modifying it into a tracer by adding the following stuff to its last rule.

```
solve(A):-
    trace_code((write('Enter '), write(A), nl),
               (write('Fail '), write(A), nl)),
    clause(A,B),
    trace_code((write('Try '), write((A:- B)), nl ),
               (write('Drop '), write((A:- B)), nl)),
    solve(B),
    trace_code((write('Succeed '), write(A), nl)).

trace_code( Forwards, Backwards):-
    Forwards ; Backwards, fail.
```

Further extensions make it into a debugger which allows the user to affect program execution similarly to standard Prolog debuggers.

Efficiency measuring of programs can also be incorporated, but we can also use source-to-source compilation instead (and thus provide an opportunity to show this phenomenon). Half a page of Prolog code can implement a translator that `retract`s each clause of the form

Head `:-` *Body*

and `assert`s another one of the form

Head `:-` *CountClauseEntranceAndBacktrack*`,` *Body*`,`
CountClauseExitAndRe-entrances

where the added pieces of code maintains global counters for each clause.

This is an entertaining and systematic way to study and characterize programming tools and interesting exercises can be given of implementing similar and other tools.

7.4 Do-it-yourself relational algebra

In section 6 we showed how type-checking and implementation of recursive procedures can be taught by having the students to develop an implementation in Prolog. We have also applied a similar approach for an introduction to relational algebra. A small example of a database is informally introduced with the notions of a relational schema (with named, untyped attributes) and database tuples, and operations `union`, `intersect`, `where` ⟨*simple-condition*⟩, and `join` where the latter is defined in terms of coinciding attribute names. A representation of base relations is shown with schema and tuples given as Prolog facts, e.g.:

```
schema(costumer,
           [costumer_no, costumer_name, costumer_city]).
tuple(costumer, [k17, jensen, roskilde]).
tuple(costumer, [k29, hansen, copenhagen]).
```

The students' tasks is now to complete the definitions for the `schema` and `tuple` predicates so that an arbitrary relational expression can be interpreted as first argument.

The conditions were the same as for the task described in section 6, 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 acceptable solution, although `join` often causes problems to some of them. The students' comments on working with this task are usually very positive.

7.5 Other applications of Prolog

Our course on programming languages includes also elementary material on lexical analysis and parsing. Here Prolog can be used as the ready-at-hand tool for the students to implement finite state machines (comparing with a generic interpreters for regular expressions[8], and different parsing methods.

8 Conclusion

We have shown how Prolog can be used as metalanguage in the teaching of programming language concepts in a way that employs the core of Prolog as an executable, formal specification language combining it with the advantages of using an interactive Prolog programming environment. Specification of language semantics are written as interpreters that can be easily extended into tracers and debuggers and interactive and extensible aspects of programming environment can also be charactericed in this way. Simple compilers and type-checkers can also be described in a similar way. This application of Prolog makes it also obvious to use in parallel Prolog as an interesting object of study in itself discussing its advantages and slight imperfections with the students.

These specifications written in Prolog are concise, easy to communicate to the students and — what is the most important for an effective learning — the students can execute and modify these specifications, and develop nontrivial specifications on their own, once they have grasped the principles. The method has been applied over a number of years at Roskilde University where the education structure is such that teaching in computer science addresses in the same class both "traditional" computer science students with a mathematical background and students from humanities (who have other strong qualifications, not to forget). This is a big challenge from a pedagogical point of view and it motivated our development of this methodology that we believe to be useful also in other contexts where an effective introduction to program language studies is needed. This can be students (of computer science or other studies) who need a compact treatment because other material takes a higher priority or as an introduction to more formal studies of programming languages, their semantics and implementation.

[8] Also a good point to discuss program specialization and partial evaluation.

We should also emphasize that working in this way gives the student a first-hand impression of declarative programming as a powerful tool for fast prototype implementation of new language constructs. The students should be able to transfer these experiences to other domains as well.

We gave an overview of related work in section 1.1 and we could identify a similar approach in the textbook [19]. As already mentioned, that book addresses exclusively a mathematically competent audience where we approach a broader audience, including students from humanities. In this light, our main contribution is to prove that Prolog as metalanguage for programming languages (and their semantics) is a very effective means for teaching these inherently complicated and technical items to this wider audience.

Prolog textbooks tend to concentrate on artificial intelligence applications, including (in most cases very simplified) natural language processing. It is also interesting to notice the difference between our context and the Prolog textbook case: Our purpose of is clearly to be descriptive using executable speciation. We are in this way not constrained by requirements of robustness, scalability and ultimate efficiency, thus having a context in which the qualities of logic programming comes best to their right.

On the theoretical side on programming language semantics, we do not add anything new: The *core* of most of our examples can be identified using other symbols within existing textbooks, e.g., [23]. However, the possibility to extend the specifications to become running programming tools such as tracers and debuggers seems to be new.

It seems obvious that the functional programming language ML to a large extent could take over the role we have given Prolog in our methodology; we have not considered this in detail and at the time of writing we are not aware of such an approach. A related approach using Scheme, although with different goals, is [1]. Finally we can mention tools for executing formal language definitions based on a particular formalism; we can mention Mosses' SIS system [14] as an early such system which is based on the lambda calculus and Centaur [3].

Acknowledgment: This research is supported in part by the IT-University of Copenhagen. The do-it-yourself relational algebra task of section 7.4 has been developed in collaboration with my colleague at Roskilde University, Troels Andreasen.

References

[1] Abelson, H., Sussman, G.J., *Structure and Interpretation of Computer Programs.* MIT Press & McGraw-Hill, 1985.

[2] Aho, A.V., Sethi, R., and Ullman, J.D., *Compilers, Principles, Techniques and Tools.* Prentice-Hall, 1986.

[3] Borras, B., Clément, D., Despeyroux, Th., Incerpi, J., Kahn, G., Lang, B., Pascual, V.. CENTAUR: The System. Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments 1988, *SIGPLAN Notices* 24(2), pp. 14–24, 1989.

[4] Bryant, B.R., Pan, A., Rapid prototyping of programming languages semantics using Prolog. *Proc. of COMPSAC 89, Int'l Computer Software and Applications Conference.* IEEE, pp. 439–446, 1989.

[5] Christiansen, H., Declarative semantics of a meta-programming language. *Proceedings of the Second Workshop on Meta-programming in Logic*. April 4–6, 1990, Leuven, Belgium. Bruynooghe, M., ed., pp. 159–168, 1990.

[6] Christiansen, H., Sprog of abstrakte maskiner, 3. rev. udgave. [In Danish; eqv. "Languages and abstract machines"] 267 pp. *Datalogiske noter* 18 (3rd ed.), Roskilde University, Denmark, 2000.

[7] Christiansen, H. and Martinenghi., D. Symbolic constraints for meta-logic programming. *Journal of Applied Artificial Intelligence*, pp. 345–368, 2000.

[8] Deransart, P., Maluszynski, J., Relating Logic Programs and Attribute Grammars. *Journal of Logic Programming* 2, pp. 119–155, 1985.

[9] Despeyroux, T. Typol: A Formalism to Implement Natural Semantics. *INRIA Rapport de Recherche* 94, 1988.

[10] Hill, P.M., and J. Gallagher, J. Meta-programming in logic programming. In: *Handbook of Logic in Artificial Intelligence and logic Programming.* Gabbay, D.M., Hogger, C.J., and Robinson, J.A. (eds.), volume 5, pp. 421–498. Oxford University Press, 1998.

[11] Knuth, D.E., Semantics of context-free languages. *Mathematical Systems Theory* 2, pp. 127-145, 1968.

[12] McCarthy, J., Abrahams, P.W., Edwards, D.J., Hart, T.P., and Levin, M.I., *Lisp 1.5 Programmer's Manual*, MIT Press, 1962.

[13] Milne, R. and C. Strachey. A Theory of Programming Language Semantics. (Two vol's). Chapman and Hall, 1976.

[14] Mosses, P.D. SIS — semantics implementation system, reference manual and user guide. *Technical Report MD-30, DAIMI,* Computer Science Department, Aarhus University, Denmark, 1979.

[15] Paakki, J., Prolog in Practical Compiler Writing. *The Computer Journal* 34, pp. 64–72, 1991.

[16] Pereira, F.C.N., Warren, D.H.D.. Definite Clause Grammars for Language Analysis — A Survey of the Formalism and a Comparison with Augmented Transition Networks. *Artificial Intelligence* 13, pp. 231–278, 1980.

[17] Pettersson, M., RML - A New Language and Implementation for Natural Semantics. Proc. Programming Language Implementation and Logic Programming, 6th International Symposium, PLILP'94. *Lecture Notes in Computer Science* 844, Springer-Verlag, pp. 117–131, 1994.

[18] Plotkin, G.D., Structural operational semantics. Lecture Notes, *DAIMI FN*-19, Aarhus University, Denmark, 1981.

[19] Slonneger, K., Kurtz, B.L., *Formal Syntax and Semantics of Programming Languages. A Laboratory Based Approach*, Addison-Wesley Publ. Co., 1995.

[20] Warren, D.H.D., Implementing Prolog — Compiling Predicate Logic Programs. *D.A.I. Research Report*, no. 39, 40. Edinbourgh University, 1977.

[21] Warren, D.H.D., Logic programming and compiler writing. Software—Practice and Experience 10, pp. 97–125, 1980.

[22] Watt, D.A., Madsen, O.L., Extended Attribute Grammars. *The Computer Journal* 26, pp. 142–153, 1983.

[23] Winskel, G., *The Formal Semantics of Programming Languages: An introduction* MIT Press, 1993.