# Some Thoughts on How to Teach Prolog[*]

Manuel V. Hermenegildo[1,2], Jose F. Morales[1,2], and Pedro Lopez-Garcia[2,3]

[1] Universidad Politécnica de Madrid (UPM)
[2] IMDEA Software Institute
[3] Spanish Council for Scientific Research (CSIC)
{manuel.hermenegildo,josef.morales,pedro.lopez}@imdea.org

**Abstract.** Prolog, and (Constraint) Logic Programming in general, represent a unique programming paradigm. Prolog has many characteristics that are not present in other styles of programming, and this is one of the reasons why it is taught. At the same time, and precisely because of this uniqueness, teaching Prolog presents some special challenges. In this paper we present some lessons learned over many years of teaching Prolog, and (C)LP in general, mostly to CS college students, at several universities. We address how to show the beauty and usefulness of the language, and also how to avoid some common pitfalls, misconceptions, and myths about it. The emphasis of our discussion is on how, rather than what. Despite some focus on CS college students, we believe that many of the ideas that we propose also apply to teaching Prolog at any other education level.

**Keywords:** Teaching Prolog, Prolog, Prolog Myths, Prolog Beauty, Prolog Playgrounds, Active Logic Documents, Logic Programming, Constraint Logic Programming.

## 1 Introduction

(Constraint) Logic Programming, (C)LP, and Prolog in particular, represent a unique programming paradigm with many characteristics that are not present in other styles of programming, such as imperative, object-oriented, or functional programming. Most notably the paradigm is based on logic and includes search as an intrinsic component, as well as the use of unification, generalizing pattern matching. This brings about other interesting and also different aspects, such as for example reversibility of programs or being able to represent knowledge and reason about it, including formulating specifications and algorithms within the same formalism. It is thus not only a unique *programming* paradigm, but also a modeling and reasoning tool.

These unique characteristics, coupled with its usefulness in many application areas, are fundamental reasons why Prolog is taught, certainly in many top institutions. Quite simply, a CS graduate is not complete without knowledge of one of the handful of *major* programming paradigms that we have come up with in CS to date. However, precisely because it is a quite different paradigm, teaching Prolog presents some particular challenges. Our first and perhaps most important consideration about teaching Prolog is that if it is done *it definitely should be done right.* Learning a programming paradigm that is quite different from what students have typically seen before and have already adapted to cannot be done lightly, in a few days, and certainly not in the same way that one moves from one imperative programming language to another. Fortunately, very good material exists (books, slides, web sites, exercises, videos) for the task. Our objective in this paper is to present a few complementary lessons learned over many years of teaching Prolog, and (C)LP in general, mostly to CS college students, at several universities, including T.U. Madrid, U. of New

[*] Partially funded by MICINN projects PID2019-108528RB-C21 *ProCode*, TED2021-132464B-I00 *PRODIGY*, and FJC2021-047102-I, by the Comunidad de Madrid program P2018/TCS-4339 *BLOQUES-CM*, and by the Tezos foundation. The authors would also like to thank the anonymous reviewers for very useful feedback on previous drafts of this paper.

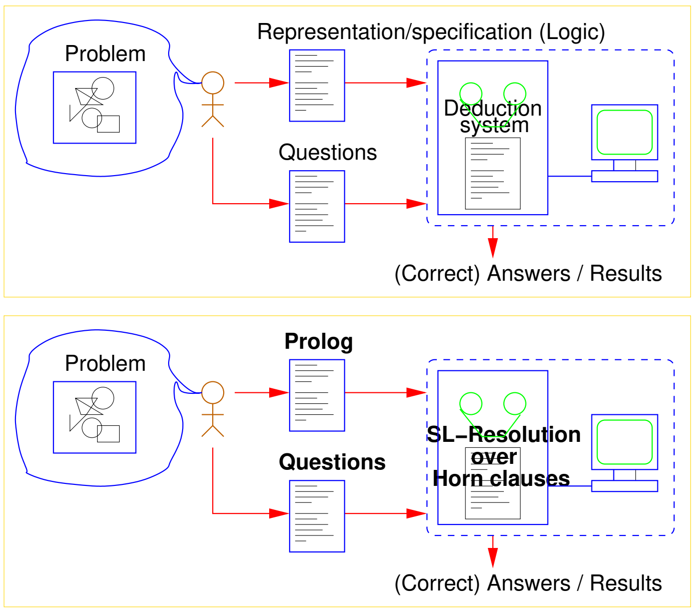
Problem
Representation/specification (Logic)
Deduction system
Questions
(Correct) Answers / Results
Problem
Prolog
SL–Resolution over Horn clauses
Questions
(Correct) Answers / Results

**Fig. 1.** A motivational view of (C)LP and Prolog.

Mexico, and U.T. Austin.[1] In this context, the students have typically already been exposed to other programming languages, as well as hopefully some concepts of logic and proofs, at the point in time in which they are exposed to Prolog and (C)LP. An important objective in this scenario is then to make the material attractive, intriguing, and challenging to such an audience. We offer some ideas on how to show the beauty and usefulness of the language, and also to how to avoid some common pitfalls, misconceptions, and myths about it. Our emphasis is more on methodological issues, rather than, e.g., on syllabus design, for which there is comparatively more material available. We also make no attempt to cover all aspects of how to teach Prolog, which would of course require a book onto itself, but rather provide a few ideas that we have found useful. Despite some focus on CS college students, we believe that most of the ideas that we propose also apply to teaching Prolog at any other education level.[2] Finally, while our discussion centers primarily around Prolog, given the theme of this volume, we believe many of the considerations and ideas are applicable to the (constraint) logic programming paradigm in general.

## 2 Showing the Beauty of the Language and the Paradigm

Perhaps the most important objective when teaching Prolog is to succeed in showing the great beauty of the (C)LP paradigm in general and of Prolog in particular. To this end, we believe that it is important to transmit (already in the first class) the original *motivations* behind the language. The following approach has worked well for us in our courses:

1. Prolog, an acronym of *Pro*gramming and *Log*ic, represents an answer to the fundamental question of what is the best way to *program* computers, in order to get them to solve problems and do what we need, and in particular of how *logic* can help us in this task.
2. There are many standard ways in which logic is used in the context of programming, e.g., as a means for defining the semantics of programs, writing specifications, and proving

[1] See `https://cliplab.org/logalg` for a collection of our teaching materials. We would like to thank the many contributors to these materials which have influenced this paper, including Francisco Bueno, Manuel Carro, Isabel García Contreras, Daniel Cabeza, María José García de la Banda, David H. D. Warren, Ulrich Neumerkel, Michael Codish, and Michael Covington.

[2] See also other papers in this volume, which address the subject of teaching Prolog to school children [2,19,3].

properties of programs with respect to those specifications. But here we are concerned with using logic *directly as the programming language.*

3. Now, time for the real *overall vision*: if we assume we have an *effective deduction procedure*, i.e., a mechanical proof method that, given a description of a problem written in logic, can provide answers to questions about this problem (prove theorems about it), then *a different view of problem solving and computing is possible* (Figure 1, top):
   (a) First, we program once and for all this *automated deduction procedure* in the computer;
   (b) then, for each problem we want to solve, we find a suitable *representation* for the problem in logic (which would be just the specification of the problem);
   (c) and, to obtain solutions, we simply ask questions and let the deduction procedure do the rest.

   Prolog (Figure 1, bottom) is the realization of this "dream."[3]
4. Time now to illustrate all this practically with one or more examples. The level of complexity of these initial examples depends on the background of the students. In general, simple examples (such as the classic family relations or, more broadly, examples with bounded search tree), are good starters. However, for students with some programming and mathematical background we have found it motivating to consider the task of specifying precisely what a simple imperative program should compute, in order to eventually prove its correctness.
5. E.g., consider a simple imperative program that calculates the squares of the first five naturals. After looking at the imperative code and how remote it is from the specification, we will develop gradually the intended semantics (post-condition) from first principles using Peano arithmetic, encoded as Horn clauses, starting with defining the naturals, then addition, then multiplication, etc. (Figure 2).[4] We develop each of these predicate definitions by reasoning about the (infinite) set of (ground) facts that we want to capture (introducing thus informally the notion of declarative semantics), and work with the students on generating the definitions by generalization from the tables of facts, thinking inductively, etc.[5] Finally, we show that, by loading these definitions into an LP system, one can use this specification by itself, not only to do the task specified (generate the squares of the naturals $< 5$), but also to subtract using the definition of addition, or even compute square roots. I.e., the specification can be explored and debugged! And, since the logic is executable, one does not need to prove that the imperative program adheres to the specification, or in fact to write the imperative program at all.
6. This presentation should be motivating, but at the same time it is also a good moment for expectation management. We discuss (informally) for what logics we have effective deduction procedures, and the trade-offs between expressive power and decidability, justifying the choice of first-order logic and SLD-resolution in classical LP, and giving at least the intuition of what semi-decidability entails.[6]
7. Having shown this *declarative* view of logic programs, it is of course important to also show the operational semantics, i.e., unification, resolution, etc. Some members of the LP community have argued that only the declarative semantics are important,[7] but this is

[3] A historical note can be useful at this or a later point, saying that this materialization was done by Colmerauer (with colleagues in Marseilles and in collaboration with Kowalski and colleagues in Edinburgh) [4,14], and was made possible by the appearance of Robinson's resolution principle [18], Cordell Green's approach to resolution-based question answering [7], the efficiency of Kowalski and Kuhnen's SLD resolution [12], Kowalski's combination of the procedural and declarative interpretations of Horn clauses [11], and the practicality brought about by Warren et al.'s Dec-10 Prolog implementation [23,17].

[4] The `:- use_package(sr/bfall).` directive (an expansion) activates breadth-first execution, which we find instrumental in this part of the course; see also the discussion in Section 3.

[5] See also [21], in this same volume, for an ample discussion of how to build programs inductively.

[6] See also the discussion in Section 3 on termination, the shape of the tree, search strategies, etc.

[7] And even some LP languages have been proposed that explicitly did not have an operational semantics, such as, e.g., the Goedel language.

```
:- use_package(sr/bfall).                                              run ▶

natural(0).
natural(s(X)) :- natural(X).

less(0,s(X)) :- natural(X).
less(s(X),s(Y)) :- less(X,Y).

add(0,Y,Y) :- natural(Y).
add(s(X),Y,s(Z)) :- add(X,Y,Z).

mult(0,Y,0) :- natural(Y).
mult(s(X),Y,Z) :- add(W,Y,Z), mult(X,Y,W).

nat_square(X,Y) :- natural(X), natural(Y), mult(X,X,Y).

output(X) :- natural(Y), less(Y,s(s(s(s(s(0)))))), nat_square(Y,X).
```

**Fig. 2.** Horn-clause specification (and program) for squares of naturals $< 5$ (click on run to load).

clearly not so in our view. Instead, we believe that it is important to present Kowalski's declarative/procedural duality, i.e., that (constraint) logic programs are, at the same time, logical theories (that thus have a declarative meaning) and procedural programs that can be debugged, followed step by step, etc. like any other language. How could we otherwise reason about complexity (or, going further, even memory consumption, etc.)? To say that these things don't matter does not make sense in the world of programming languages. In other words, without an operational semantics, we do not (also) have a programming language, but rather just a specification and/or knowledge representation formalism – the beauty of Prolog is that it is *both*. And the argument for the procedural interpretation goes further: natural language also involves procedural forms of expression. Thus, elimination of the ability to represent procedures also eliminates the ability to represent some types of knowledge.

8. Finally, it is of course important to relate the declarative and procedural views, explaining that the declarative meaning of a set of rules is a (possibly infinite) set of ground facts and that these constitute precisely the set of ground literals for which, when posed as queries, Prolog (possibly needing a fair search rule) answers yes.

Once motivation is established, and we can understand programs both declaratively and operationally, we can show other unique aspects that contribute to the elegance and beauty of the language:

1. We can show with examples (and benchmarking them) how in Prolog it is possible to go from executable specifications to efficient algorithms gradually, as needed. For example:[8]
   (a) The mathematical definition (i.e., the specification) of the modulo operation, `mod(X,Y,Z)` where `Z` is the remainder from dividing `X` by `Y`, i.e., $\exists Q s.t.\ X = Y * Q + Z \land Z < Y$, can be expressed directly in Prolog:

   ```
   mod(X,Y,Z) :- less(Z, Y), mult(Y,Q,W), add(W,Z,X).              run ▶
   ```

   This version is clearly correct (since it is directly the specification) and (using breadth-first search) works in multiple directions, always finding all solutions:

   ```
   ?- op(500,fy,s).
   yes
   ?- mod(X,Y, s 0).
   X = s 0,
   Y = s s 0 ? ;
   ```

[8] Clicking on the run ▶ links is perfectly safe!

```
?-  X=f(K,g(K)),
    Y=a,
    Z=g(L),
    W=h(b,L),
    % Heap memory at this point ⟶
    p(X,Y,Z,W).
```

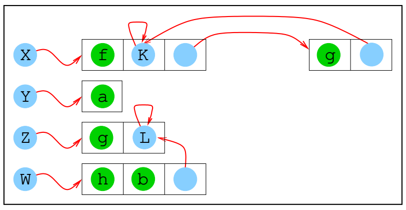

**Fig. 3.** Using unification to build data structures with declarative pointers.

```
X = s 0,
Y = s s s 0 ? ;
X = s s s 0,
Y = s s 0 ? ;
...
```

but it can be quite inefficient.

(b) At the same time we can write a version such as this one:

```
mod(X,Y,X) :- less(X, Y).                          run ▶
mod(X,Y,Z) :- add(X1,Y,X), mod(X1,Y,Z).
```

which is much more efficient, and works well with the default depth-first search rule. One can time some queries or reason about the size of the proof trees to show this.[9]

2. It is also important to show the power and beauty of unification, not only as a generalization of pattern matching, but also as a device of *constructing and accessing (parts of) complex data structures* and passing them and returning them from predicates. This can be illustrated to students by building data structures piecemeal in the top level, as illustrated in Fig. 3, which shows graphically the process of building some data structures in memory via unifications, just before performing a call to the p/4 procedure. The idea is to illustrate that logical variables can be seen as "declarative pointers" [9]. I.e., in the same way a set of Prolog clauses constitute at the same time statements in logic and a program, the data structures of Prolog can be seen at the same time as (Herbrand) terms of the logic and as traditional data structures with (declarative, i.e., single assignment) pointers. We have found that explaining this duality is also very enlightening.

There are of course many other beautiful and elegant aspects to show (e.g., higher-order, meta-interpretation, or types and program properties in general, all of which in (C)LP once more can be covered within the same language), but our space here is limited. As a compelling example, in (C)LP, types (and properties in general) can be defined as (runnable) predicates. E.g., the *type* natlist can be defined as:

```
    natlist([]).                                     run ▶
    natlist([H|T]) :- natural(H), natlist(T).
```

and this predicate can be used to check that an argument is a list of naturals (dynamic checking) or "run backwards" to generate such lists (property-based testing for free!). Some of these aspects are covered in other papers in this volume [10].

## 3 Dispelling Myths and Avoiding Misconceptions

In addition to showing the beauty of the language, another aspect that we believe is important to cover during the course is to dispel the many *unfounded myths and misconceptions* that still circulate about Prolog and the (C)LP paradigm in general, and to which students may be exposed. While some of these views may have been at least partially true of early implementations of Prolog, both the language and its implementations have come a long way over many years (actually decades) of evolution, and it is easy to show how the shortcomings of early Prologs have been addressed in most current systems. The following is an

[9] See also [1], in this same volume, for another interesting example which can be used similarly.

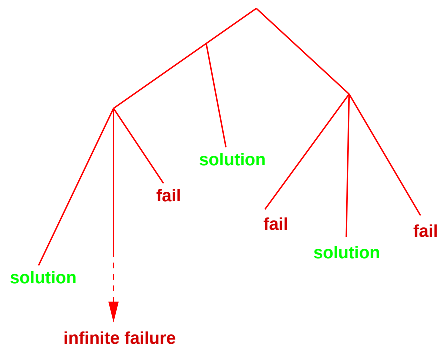

**Fig. 4.** Possible cases in the search.

incomplete list of some of these perceived shortcomings and some suggested dispelling facts or actions:

***Explaining termination.*** As mentioned in the previous section, it is certainly a good idea to start teaching the declarative view, i.e., using the logical reading when writing and reading clauses. In this view of the language, if the logic that we write is correct the system will answer any question. As mentioned before, here examples with bounded search tree are great starters (e.g., using only constants or avoiding recursion). However, trouble begins at the very beginning once structures and recursion are introduced: students soon run into termination problems. This is true of course of any programming language or proof system. However, non-terminating executions are likely to discourage beginners if their origins are not explained well and no remedies provided.

For example, let us define a pair of natural numbers in Peano representation:

```
natural(0).
natural(s(X)) :- natural(X).

pair(X,Y) :- natural(X), natural(Y).
```
run ▶

A query such as `?- pair(X,Y),X=s(0).` will hang with the standard Prolog depth-first rule, because the search never progresses beyond `X=0`, since there are infinite possible solutions for `Y` that will be explored first. In contrast, if the program is executed with a breadth-first strategy (which explores all branches fairly), it produces the expected solutions quickly.

A solution that has worked well for us is the following:

1. Provide students with a means for *selectively* switching between search rules, including in particular at least one that is fair. E.g., being able to run programs in breadth-first, depth-first, iterative deepening, tabling, etc. This comes built-in in certain Prologs[10] but it is in any case easy to implement in any Prolog for example via a meta-interpreter.
2. Without giving too many details, start by running all predicates in breadth-first mode - all examples work great! This will allow students to gather confidence with recursion, search, and the power of a logic-based programming language (specially if they have already taken an introductory logic course), by simply thinking logically and/or inductively.
3. After students have been exposed to and written a few examples, we have found figures such as Figs. 4–5 useful to introduce them in a graphical way to the basic theoretical results at play, i.e., the soundness and (refutation-)completeness of the SLD(NF)-resolution proof procedure used by Prolog. The practical implication to convey is that the search

[10] E.g., in Ciao Prolog, in which we have added over time a number of features to facilitate teaching Prolog and (C)LP, one can for example use `:- use_package(sr/bfall).` to run all predicates breadth-first. Also, many Prologs have tabling nowadays.

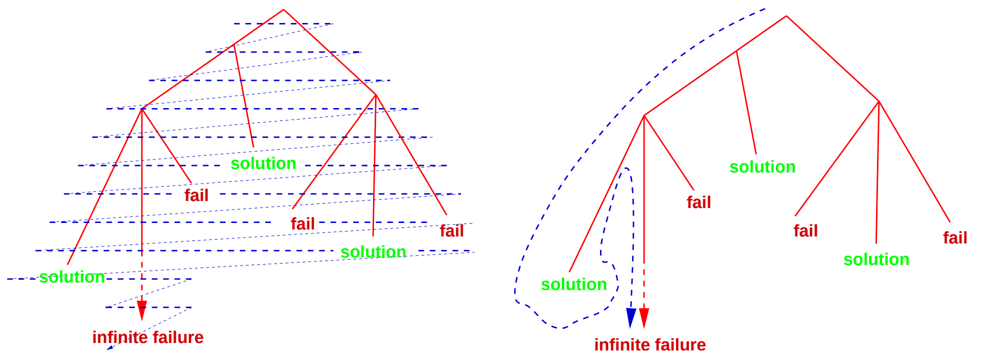

**Fig. 5.** Breadth-first (left) and depth-first (right) exploration.

tree has the shape of Fig. 4, i.e., that all solutions and some failures are at finite depth, but that there are branches leading to failure that may be infinite and that it is not always possible to detect this. This summary depiction makes it easy to explain why breadth-first (or iterative deepening, or any other fair search rule) is guaranteed to produce all solutions if they exist in finite time (Fig. 5, left), and why depth-first sometimes may not (Fig. 5, right). Of course, neither one of them is guaranteed to always finish after producing the positive answers.

4. At the same time, one should discuss the advantages and disadvantages of these search rules in terms of time, memory, etc. I.e., that the price to pay for breadth-first execution's advantages is very large (potentially exponential) memory and time consumption, while depth-first can be implemented very efficiently with a stack, with iterative deepening representing an interesting middle ground. This can be shown very practically by benchmarking actual examples, motivating the practical choices made for Prolog, which bring in great efficiency at the (reasonable) price of having to think about goal and clause order.
5. For example, simply changing the goal order (in this case in the query) to `?- X=s(0),pair(X,Y).` modifies the search space and the program can produce all the answers with the standard Prolog search. This leads naturally to discussing how one needs to reason about the behavior (cost and termination) of predicates depending on the different *modes* (see [20]) in which they will be called.
6. And it can also be pointed out that there exist other techniques like delays and, specially, tabling, which helps detect some of the infinite failures in finite time, and avoid repeated answers (even infinite ones), in addition to changing dynamically the goal order.
7. More generally, while using small examples, sophisticated Prolog implementations, and modern fast computers can create the misconception that Prolog provides solutions effortlessly, it should be stressed that, as a Turing complete language, also Prolog programmers eventually need to care about algorithmic time and memory complexity of both their programs and the libraries/features they utilize, and, at the limit, termination.
8. Thus, this is also a good moment to introduce informally the notion of *undecidability* and the *halting problem* and relate them to the graphical depictions of the search tree. Some students may not be aware that there exist *decision problems* that admit no algorithmic solution. One should underline that this is of course not a particular problem of Prolog, but rather the essence of computability, and no language or system (Prolog, logic, nor any other Turing-complete programming language) can provide a magic formula that guarantees termination.
9. Needless to say it is also important to explain how to control search, via clause order and literal order, as well as pruning (see cut later). And, if time permits, it is important to discuss the use of the constraint systems built into most current Prolog systems (Q, R, fd, ...), and how they can bring about many other improvements to search (e.g., generate and test vs. constrain and generate), arithmetic (see below), etc.

***Showing that Prolog arithmetic can also be reversible.*** The opposite of course is true if the discussion is limited to the ISO-Prolog arithmetic built-ins. However, this is far from the only answer in modern Prolog systems. The approach that we take is:

1. Present all the initial examples using Peano arithmetic (as shown in the `nat/1` example and in Section 2): it is beautiful and reversible (if run with a fair search rule of course). Although obviously inefficient and slow for number crunching, Peano arithmetic is an excellent tool for the first steps in writing numeric and recursive programs.
2. Then, the ISO-Prolog arithmetic built-ins can be introduced, explaining that they were added for practical efficiency, but at the cost of losing some reversibility. For example, using `Y is X+1` instead of `Y=s(X)` is no longer always reversible.
3. Then, (arithmetic) constraint domains can be introduced showing that they can represent the best of all worlds: at the same time beautiful, powerful, and efficient! For example, in the case of CLP{Q,R} (or CLP(fd)) our simple increment becomes `Y .=. X+1` which is both reversible and efficient. An alternative that we have also used is to start from the beginning using constraint-based arithmetic instead of Peano, but it can be more cumbersome because for the student to fully understand what is really going on one should really discuss the workings of the constraint solver. In that sense Peano is simpler because with only unification and Horn clauses all operations are fully defined. Still, both approaches are fine.

***The occur check is available (if needed).*** A misconception is that Prolog must be unsound because there is no occur check.

1. First, it is important to explain why the decision to leave out the occurs check was made in the first place: in addition to reducing the cost of general unification, removing the occurs check allows the complexity of the *variable-value* unification case to be constant time (instead of linear, since there is no need to check that the left hand side does not occur in the value), which is arguably a basic requirement for a practical programming language.
2. It is also important to point out that the lack of occurs check is rarely an issue in actual applications, and that, furthermore, there is in any case a built-in for unification with occurs check. In most cases, including e.g., implementation of theorem provers, one can selectively use the unification with occurs check built-in only when needed. It can also be useful to provide a package (easy to implement in most Prologs with a term expansion) that calls unification with occurs check by default for all unifications. This can also be provided in some systems via an engine/compiler flag.
3. Furthermore, it should be mentioned that many Prolog systems in fact now support infinite tree unification (stemming all the way back to Prolog II) as the default. In this extension to unification (actually, a constraint domain), infinite terms are actually supported and the unifications that would be the subject of the occurs check produce such terms. For example, the answer to `?- X = f(X).` is simply `X = f(X)`, and the system does not go into a loop when performing such unifications or printing such infinite terms.

***Prolog can be pure (despite cut, assert, etc.)***

1. In the same way that we start with a powerful yet not necessary efficient search rule such as breath-first, it is convenient to restrict the discussion initially to *pure* fragments of Prolog without side-effects, cuts, or the dynamic manipulation of database.
2. In this phase it can be convenient to have a mechanism to enable a *pure mode* in the Prolog implementation where impure built-ins simply are not accessible. However, this is not strictly necessary and the objective can also be achieved by simply not allowing the use of impure built-ins, or, in fact, any built-ins, initially. Peano arithmetic is again handy here.
3. Later, the ISO-Prolog built-ins can be introduced. Here, it is convenient to treat each kind of *impurity* separately:

(a) Cuts: a first consideration is that using if-then-else is often preferable. Then, regarding cuts, it is important to explain at least the difference between (green) cuts added as optimizations to discard unnecessary choice points (but which do not alter the declarative semantics) from (red) cuts that are only meaningful for some calling modes and whose removal would make the program incorrect. Explain that argument indexing makes many green cuts unnecessary.

(b) Assert/retract: some programming patterns using dynamic program manipulation, like explicit memoization or algorithms that explicitly interleave phases of database modifications with pure deductions, do not sacrifice declarativeness. The classic Fibonacci program with a double recursion, implemented with and without memoing, is a good example here. Assert and retract in modern Prolog systems are module-aware, which makes it easier to encapsulate their effects. Also, it should be noted that there are other approaches, such as mutables or condition-action rules, that offer more structured ways to express state change; see, e.g., [13,6].

4. It is also useful to develop pure libraries (e.g., implicit states *ala* DCGs, monad-like styles), built-ins, or semantics where effects and actions are properly captured.
5. Finally, it is also important to point out that sometimes impurity is just necessary, but one should strive to keep it encapsulated as much as possible as libraries or program expansions.

***Negation.*** Explain negation as failure, possibly from the theoretical point of view (least-Herbrand-model semantics, see also [20,22]) and through its classical implementation using Prolog built-ins. And devote time to discussing the limitations, either at a simple level (avoid calling negation on non-ground goals) or delving more deeply into stratification, etc. A good idea is to suggest to the students that they guard themselves from mistakes by defining their own negation that performs some checks, for example that the negated goal is ground:

```
not(G) :- ground(G), !, \+ G.                                        run ▶
not(_) :- throw(error).
```

In an advanced course one can also go into more complex topics, commenting on the alternatives that Prolog systems offer nowadays (such as the very interesting s(CASP) approach [8]) and on ASP.

***Prolog is in many ways as other languages, but adds unique, useful features.***

1. It is interesting to show that Prolog is not a "strange" language, and is in fact completely "normal" when there is only one definition per procedure which is called in one mode. But that at the same time it can also have several definitions for the same procedure, and can thus support natively search, several modes, running "backwards," etc.
2. Moreover, Prolog (and specially modern Prolog systems) actually *subsume* both functional and imperative programming. At least theoretically, this is well known since functions can be easily encoded as relations and since mutable variable changes can be declaratively encoded by state-threading or other means. (And that this idea is very useful in practice for analysis of other languages using Horn clauses!)
   In practice, translating functional or imperative constructs to Prolog is relatively easy, specially when using special syntactic extensions (such as *logical loops* or *functional notation*). Performance-wise, most Prolog implementations are optimized enough to execute recursions as efficiently as loops (e.g., *last-call optimization*), use *logical mutable variables* (equivalent to implicit arguments), or as a last resort store mutable states via the dynamic database.
3. For students that have some notions of programming language implementation, it helps to explain that, when running "forward", Prolog uses a stack of activation records for local variables, and return addresses (forward continuations), as every language, that allow knowing where to return when a procedure returns (succeeds). Then, Prolog also has a stack of *backwards continuations*, to know where to go if there is a failure (previous *choice point*), and this (coupled with other memory and variable binding management techniques) implements Prolog's backtracking semantics very efficiently.

4. College students that are already familiar with younger languages (Erlang, Haskell, or even Rust) often recognize striking similarities in syntax and semantics with Prolog (e.g., "pattern matching"). Most of them tend to be amazed by Prolog simplicity and expressive power and recognize that Prolog is still unique.

***Prolog has many applications / uses / ...***

1. Show some of the many examples of great applications that Prolog has. There are great collections on line, and some new ones have been gathered for the Prolog 50th anniversary, but in particular there are some really excellent examples in the volume in which this paper appears. An excellent example is ProB, winner of the first Colmerauer Prize [15]. See also the excellent related discussion [11] on the advantages of using Prolog.
2. Give the students as homework real, challenging, and interesting projects where they can experience the power and elegance of the language.
3. Another thing to perhaps point out in this context is that modern applications are almost never written in a single language and some Prolog implementations can be easily embedded as part of more complex systems, and this is done routinely.

***Prolog can also have "types" (if needed).*** Prolog is in principle a dynamically typed language, and this is not without its advantages –the success of Python attests to this. Second, as mentioned before, types and many other properties can indeed be defined very elegantly within the same language of predicates, and used as both run-time tests and generators. Furthermore, there are Prolog systems that also check these types and properties statically (again, see [10] in this volume).

And, to end this section on myths and misconceptions, a final mention is due to the **Fifth Generation (FG) project**. It is probably unlikely for current students to be aware of this project, but since the subject of its success or failure does comes up with some periodicity, e.g., in online forums, it suffices to say that one hand there were many interesting outcomes of this project (and there is ample literature on the subject) and on the other the fate of the FG Project is in any case quite unrelated to Prolog and (C)LP, simply because, contrary to widespread belief, the FG did not use Prolog or "real LP" anyway! It used flat committed choice languages,[12] which arguably did not contribute to make the FG as successful as it could have been.

## 4 Some Thoughts on Systems

Some thoughts regarding the different types of Prolog systems that we fortunately have currently freely available for teaching. This includes:

1. The classical systems with traditional installation, which in general are most appropriate for more advanced users and intensive use. In the context of teaching, they have the advantage that the best implementations portray a serious language which is competitive with the most popular languages in performance, features, environment, libraries, embeddability, small executables, etc. The drawback is of course that this type of system requires installation, but that should not be any hurdle for college students (or at least not for those in CS).
2. At the same time, there are now fortunately also very good Prolog playgrounds and notebooks that require no installation. Examples are, e.g., the Ciao Playgrounds and Active Logic Documents, SWISH, $\tau$-Prolog, s(CASP) playground, etc. These no-installation alternatives can be very attractive for beginners, young students, or simply casual users, and they are in all cases very useful for supporting executable examples in manuals and

[11] `https://prob.hhu.de/w/index.php?title=Why_Prolog%3F`

[12] An interesting topic that is however out of our scope here –let's just say for this discussion that they used "something similar to Erlang."

tutorials, as done here. These systems can be server-based or browser-based, each having advantages and disadvantages. A good discussion on this topic can be found in [16] and [5], within this same volume.

3. As far as functionality, ideally the system (or systems) to be used should allow covering ISO-Prolog and some constraint domains –most current Prolog systems are capable of this. Other desirable features in our opinion are the possibility of restricting programs to pure LP (and supporting several search rules and tabling), modern forms of negation (such as, e.g., ASP/s(CASP)), functional syntax, extended support for higher-order and libraries, etc. Again, many current Prolog systems provide at least some of these characteristics.

## 5 The Programming Paradigms Course

So far, we have generally assumed that a reasonable number of lectures are available for teaching the language. However, a particularly challenging case is the standard "programming paradigms" course, now quite widespread in CS programs, where each programming paradigm, including often (C)LP and Prolog, is devoted perhaps two or three weeks. This scenario has some risks, and in extreme cases could even be counter-productive. A first risk, mentioned before, is that it is simply not easy, even for experts, to, in just very few classes, teach Prolog and the (C)LP paradigm well enough that the students really "get it." Other potential pitfalls in practice include, for example, that in some cases a "logic programming" library from a non-LP language (e.g., emulating Prolog in Scheme) may be used instead of a "real" Prolog system, which will have much more competitive speed and memory efficiency (plus an advanced programming environment, being capable of generating efficient executables, etc.). Shortcuts such as these, coupled with a superficial presentation in a few classes, can run the risk of leading to misconceptions about the language, its capabilities, applications, performance, and ultimately, its beauty. All this brings about the obvious and important question of what to do if the programming paradigms course is really the only slot available in the curriculum to teach Prolog. This is in our opinion a topic that deserves more attention from the (C)LP community. While, as mentioned before, quite good material exists for full-size Prolog courses, this is arguably less so for the (C)LP part of a programming paradigms course. So, in parallel with arguing for the presence of specific full courses devoted to (C)LP, it would be very useful to develop material (slides, notes, a short book, etc.) aimed specifically at teaching this "programming paradigms slot" well. In addition, we hope that the reflections in this paper can also help in this challenging context.

## 6 Conclusions

We have presented some lessons learned from our experience teaching Prolog, and (C)LP in general, over the years. We have covered some methodological issues that we feel are important, such as how to show the beauty and usefulness of the language, and also to how to avoid some common pitfalls, misconceptions, and myths about it. However, teaching Prolog and (C)LP is an extremely rich subject and there are of course many other aspects that we have not been able to address for lack of space. We still hope that at least some of our suggestions are of use to other instructors that are teaching or plan to teach Prolog. For a more complete picture, as mentioned before, much of our experience over the years is materialized in a) the courses that we have developed, for which, as pointed out before, the material is publicly available[13], and b) the many special features that we have incorporated over time in our own Ciao Prolog system in order to aid in this important task of teaching logic programming. Again, we have touched upon some of them, such as being able to choose different search rules, the playground, or active logic documents, but there are many others. We hope that all the ideas present in these materials and systems are helpful and inspiring to both Prolog instructors and students.

[13] `https://cliplab.org/logalg`

## References

1. Bassiliades, N., Sakellariou, I., Kefalas, P.: Demonstrating Multiple Prolog Programming Techniques through a Single Operation. In: Warren, D.S., Dahl, V., Eiter, T., Hermenegildo, M., Kowalski, R., Rossi, F. (eds.) Prolog - The Next 50 Years. No. 13900 in LNCS, Springer (July 2023)
2. Cecchi, L.A., Rodríguez, J.P., Dahl, V.: Logic Programming at Elementary School: Why, what and how should we teach Logic Programming to children. In: Warren, D.S., Dahl, V., Eiter, T., Hermenegildo, M., Kowalski, R., Rossi, F. (eds.) Prolog - The Next 50 Years. No. 13900 in LNCS, Springer (July 2023)
3. Cervoni, L., Brasseur, J., Rohmer, J.: Simultaneously teaching mathematics and prolog in school curricula: a mutual benefit. In: Warren, D.S., Dahl, V., Eiter, T., Hermenegildo, M., Kowalski, R., Rossi, F. (eds.) Prolog - The Next 50 Years. No. 13900 in LNCS, Springer (July 2023)
4. Colmerauer, A.: The Birth of Prolog. In: Second History of Programming Languages Conference. pp. 37–52. ACM SIGPLAN Notices (March 1993)
5. Flach, P., Sokol, K., Wielemaker, J.: Simply Logical - The First Three Decades. In: Warren, D.S., Dahl, V., Eiter, T., Hermenegildo, M., Kowalski, R., Rossi, F. (eds.) Prolog - The Next 50 Years. No. 13900 in LNCS, Springer (July 2023)
6. Genesereth, M.: Dynamic Logic Programming. In: Warren, D.S., Dahl, V., Eiter, T., Hermenegildo, M., Kowalski, R., Rossi, F. (eds.) Prolog - The Next 50 Years. No. 13900 in LNCS, Springer (July 2023)
7. Green, C.C.: Application of Theorem Proving to Problem Solving. In: Walker, D.E., Norton, L.M. (eds.) Proceedings IJCAI. pp. 219–240. William Kaufmann (1969)
8. Gupta, G., Salazar, E., Arias, J., Basu, K., Varanasi, S., Carro, M.: Prolog: Past, Present, and Future. In: Warren, D.S., Dahl, V., Eiter, T., Hermenegildo, M., Kowalski, R., Rossi, F. (eds.) Prolog - The Next 50 Years. No. 13900 in LNCS, Springer (July 2023)
9. Hermenegildo, M.: Parallelizing Irregular and Pointer-Based Computations Automatically: Perspectives from Logic and Constraint Programming. Parallel Computing **26**(13–14), 1685–1708 (December 2000)
10. Hermenegildo, M., Morales, J., Lopez-Garcia, P., Carro, M.: Types, modes and so much more – the prolog way. In: Warren, D.S., Dahl, V., Eiter, T., Hermenegildo, M., Kowalski, R., Rossi, F. (eds.) Prolog - The Next 50 Years, chap. 2. No. 13900 in LNCS, Springer (July 2023), `http://cliplab.org/papers/AssertionsAndOther-PrologBook.pdf`
11. Kowalski, R.A.: Predicate Logic as a Programming Language. In: Proceedings IFIPS. pp. 569–574 (1974)
12. Kowalski, R., Kuehner, D.: Linear resolution with selection function. Artificial Intelligence **2**(3), 227–260 (1971)
13. Kowalski, R., Sadri, F., Calejo, M., Dávila-Quintero, J.: Combining Prolog and Imperative Computing in LPS. In: Warren, D.S., Dahl, V., Eiter, T., Hermenegildo, M., Kowalski, R., Rossi, F. (eds.) Prolog - The Next 50 Years. No. 13900 in LNCS, Springer (July 2023)
14. Kowalski, R.A.: The Early Years of Logic Programming. Communications of the ACM **31**(1), 38–43 (1988)
15. Leuschel, M.: ProB: Harnessing the Power of Prolog to Bring Formal Models and Mathematics to Life. In: Warren, D.S., Dahl, V., Eiter, T., Hermenegildo, M., Kowalski, R., Rossi, F. (eds.) Prolog - The Next 50 Years. No. 13900 in LNCS, Springer (July 2023)
16. Morales, J., Abreu, S., Ferreiro, D., Hermenegildo, M.: Teaching Prolog with Active Logic Documents. In: Warren, D.S., Dahl, V., Eiter, T., Hermenegildo, M., Kowalski, R., Rossi, F. (eds.) Prolog - The Next 50 Years, chap. 14. No. 13900 in LNCS, Springer (July 2023), `http://cliplab.org/papers/ActiveLogicDocuments-PrologBook.pdf`
17. Pereira, L., Pereira, F., Warren, D.: User's Guide to DECsystem-10 Prolog. Dept. of Artificial Intelligence, Univ. of Edinburgh (September 1978)
18. Robinson, J.A.: A Machine Oriented Logic Based on the Resolution Principle. Journal of the ACM **12**(23), 23–41 (January 1965)
19. Tabakova-Komsalova, V., Stoyanov, S., Stoyanova-Doycheva, A., Doukovska, L.: Prolog Education in Selected High Schools in Bulgaria. In: Warren, D.S., Dahl, V., Eiter, T., Hermenegildo, M., Kowalski, R., Rossi, F. (eds.) Prolog - The Next 50 Years. No. 13900 in LNCS, Springer (July 2023)
20. Warren, D.S.: Introduction to Prolog. In: Warren, D.S., Dahl, V., Eiter, T., Hermenegildo, M., Kowalski, R., Rossi, F. (eds.) Prolog - The Next 50 Years. No. 13900 in LNCS, Springer (July 2023)

21. Warren, D.S.: Writing Correct Prolog Programs. In: Warren, D.S., Dahl, V., Eiter, T., Hermenegildo, M., Kowalski, R., Rossi, F. (eds.) Prolog - The Next 50 Years. No. 13900 in LNCS, Springer (July 2023)
22. Warren, D.S., Denecker, M.: A better logical semantics for prolog. In: Warren, D.S., Dahl, V., Eiter, T., Hermenegildo, M., Kowalski, R., Rossi, F. (eds.) Prolog - The Next 50 Years. No. 13900 in LNCS, Springer (July 2023)
23. Warren, D.: Applied Logic—Its Use and Implementation as Programming Tool. Ph.D. thesis, University of Edinburgh (1977), also available as SRI Technical Note 290