Demonstrating Multiple Prolog Programming Techniques Through a Single Operation

Nick Bassiliades[1](✉), Ilias Sakellariou[2], and Petros Kefalas[3]

[1] School of Informatics, Aristotle University of Thessaloniki, Thessaloniki, Greece
nbassili@csd.auth.gr

[2] Department of Applied Informatics, University of Macedonia, Thessaloniki, Greece
iliass@uom.edu.gr

[3] Department of Computer Science, City College, University of York Europe Campus, Thessaloniki, Greece
kefalas@york.citycollege.eu

Abstract. Without doubt Prolog, as the most prominent member of the logic programming (LP) approach, presents significant differences from the mainstream programming paradigms. However, demonstrating its flexibility to larger audiences can indeed be a challenging task, since the declarative style of LP lies outside the mainstream programming languages most are familiar with. In this paper, we demonstrate how alternative implementations of a single list operation can prove to be a rather helpful tool for demonstrating a plethora of Prolog programming aspects and techniques, and some issues associated with these, such as efficiency, readability and writability of code.

Keywords: Logic Programming techniques · Prolog Flexibility · Prolog Efficiency · Prolog Education

1 Introduction and Motivation

Communicating the flexibility of the logic-based approach to problem solving supported by Prolog, to wider audiences, such as students, programmers and in general people less familiar with LP, can prove to be a somewhat complicated task. Thus, the motivation behind this paper is to demonstrate the flexibility of Prolog in coding a simple operation, commonly found in all programming environments supporting lists, by employing a variety of programming techniques. As educators, we have been teaching Prolog for many years, following a fairly standard approach of introducing the language constructs and techniques that differentiate it from other mainstream programming languages. By starting from pure Prolog syntax, declarative and operational semantics, then moving towards recursion and lists and concluding with extra-logical features and practical meta-programming techniques, gave us the opportunity to teach Prolog through a good number of examples of various complexity.

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2023
D. S. Warren et al. (Eds.): Prolog: The Next 50 Years, LNAI 13900, pp. 71–81, 2023.
https://doi.org/10.1007/978-3-031-35254-6_6

What was missing was a standard reference to a single operation that could be implemented in different ways, thus demonstrating the flexibility of the language which, however, may raise issues for discussion around declarative versus procedural approach, readability versus writability of code and simplicity versus efficiency. We found that `list_min` predicate could serve this purpose. We followed different approaches in our own institutions; either to present this example throughout the semester as we introduce concepts together with other examples, or to devote a revision session towards the end of the semester to summarise the language potential before the final exam. Results were encouraging in both approaches.

Therefore, we demonstrate how a single list operation, i.e. the minimum of a list, can prove to be a rather helpful tool to refer to a plethora of Prolog programming aspects and techniques. The benefits are numerous: (a) it is a simple operation, with which students are familiar with and thus presents no barriers in understanding its specification, (b) it has a surprising number of alternative implementations, (c) benchmarking of the alternatives is a straightforward task, allowing discussion of complexity issues that arise. The current paper contributes towards demonstrating Prolog flexibility, especially in audiences just getting familiar with Prolog, by:

- discussing how a single simple list operation can demonstrate a number of programming techniques, through alternative implementations,
- presenting how such simple code can expose a number of lower level issues of some techniques, such as garbage collection and tail recursion optimization.

The paper is not aimed to be a tutorial in Prolog, as this is presented earlier in this book [14]. It is assumed that the reader has some knowledge of the language. Additionally, our audience includes educators who would like to adopt this example throughout their courses on top of all existing material they teach as a single point of reference suitable for discussing the flexibility of the language.

2 The Challenges of Learning Prolog

Learning Prolog can indeed be a challenging task, usually attributed to the declarative style of LP, which lies outside the mainstream programming languages. Indeed, in CS curricula, students are exposed early to concepts such as while-for loops, destructive assignment variables, types, common functions with a single return value, etc. in the usual first "Introduction to Programming" course. This programming paradigm mindset is further deepened, for example by object oriented programming courses that although they introduce new concepts such as inheritance, still follow the well established procedural vein and the usually OO focused software engineering courses. The distance between the paradigms is further enlarged because students are normally instructed to avoid as much as possible recursion due to inefficiency issues, present in most mainstream languages that lack optimization support. Consequently, as educators we must overcome a number of obstacles and in fact "reset" the way students think

about programming. Although a number of classic introductory and advanced books and tutorials exist, such as [1,3,9,10,12], the pedagogical issues faced in class remain a great challenge [6,11,13], and require novel approaches and environments [4,8].

Students often ask the question "*Why Prolog?*" Some are convinced by typical justifications that refer to foundations of the discipline as well as teaching programming principles with a minimum set of constructs. It is expected, however, that a tighter integration, both in terms of facilities and available examples, with other programming languages and tools will be sought towards the perception of students for its applicability. Additionally, the current growth of Knowledge Graphs [7] and their associated Semantic Web technologies that rely mostly on logic, raises some extra arguments in favor of learning LP.

For as long as Logic forms the foundations of Computer Science, LP, through its main representative (Prolog) will remain current. It is no surprise that Prolog is perhaps one of the few programming language that persists for many years in Computer Science curricula; while other programming languages come and go, Prolog remains as a paradigm for declarative thinking (programming) that can be used to teach a plethora of programming principles and techniques. We anticipate that this fact will also remain true in the future.

The power of simplicity (syntax and constructs) gives us, educators, the opportunity to focus on programming principles that span from purely declarative to procedural. In this paper, we attempt to demonstrate a showcase on how this is feasible through alternative implementations of a single operation that can be referenced throughout the learning process or used to summarise and revise the language abilities and flexibility.

3 Logic Programming Techniques

The term "programming technique" has been used in different contexts: we adopt the definition that a *technique* is a pattern, applied to a wide range of problems [2], as for example *failure driven loops*, *generate-and-test* and *accumulator pairs*, among many others.

A *schema* is a representation of a technique that allows developers to represent a general structure solution code to a problem, so that, given the specific context, the schema may be instantiated to executable code that solves the particular problem. The schema approach to teaching Prolog was introduced rather early, as for instance in [5], where also a single example was used. Although we follow the same approach, our focus is different, since we contend that the single task used can serve to discuss a wider number of issues, as for example efficiency and non-recursive techniques.

Probably the most widely used approach for logic programs is *recursion*. It is derived through the mathematical concept of induction [15] and assumes that a problem of a certain size, say N, can be solved if the same problem of a smaller size, say N-1, is solvable. Although a powerful approach, that provides for simple solutions of rather complex problems even when considered in a procedural

mindset, it is still a skill that most novice programmers lack, or have not been exposed and exercised to a great extent.

4 One Problem - Many Solutions

The problem in question is to find the minimum element of a list of numbers, what we refer to as *list_ min/2* predicate. We will show the recursive approach first, and then we will explain some more complex non-recursive schemata. In the following, in order to distinguish easily between the different implementations of *list_ min/2*, we adopt a numbering in the name of the predicate, i.e. `list_minX/2` with X ranging from 1 to 9.

4.1 Recursive Super-Naive Declarative Implementation

The first attempt to a solution, focuses on presenting a rather simple view of recursion, that reads as follows: "*To find the minimum of a list of N elements,* **assume** *that you know the minimum of its N−1 elements and compare with the Nth*". This leads to a (super-naive) recursive implementation of the *min_ list* predicate, shown below as predicate (`list_min1/2`).

```
list_min1([Min],Min).
list_min1([H|T],H):-
    list_min1(T,MinTail),
    H =< MinTail.
list_min1([H|T],MinTail):-
    list_min1(T,MinTail),
    H > MinTail.
```

This implementation offers the grounds for raising a number of issues: (a) **list decomposition** in the head of the clause, (b) an introduction to the structure of a predicate definition with **recursive and terminal cases** (rules) and (c) placing **alternative choices** in separate rules, as a straightforward rule-of-thumb instead of OR (`;`) within the body of the clause that leads to several concerns. List decomposition on the head of the clause, provides a syntactically simple list operation, whereas alternative cases in a definition enhance readability and make the resulting code more extensible.

However, a number of other issues are eminent. Probably the one with "*what is hidden*" in the previous implementation is the fact that there is a choice point in the recursive rule embedded in the predicate; the choice between the minimum among the head of the list and the minimum of the tail of the list, that occurs after the recursive call, leads to **inefficiency issues** and provides a great chance to discuss the **execution tree** and placing checks as early as possible. The response to the obvious reaction for **subgoal reorderding** in the body of the rules, leads to a discussion on the **non-logical handling of numbers** in classical (non CLP) Prolog implementations, that expects ground variables in any arithmetic expression. This discussion provides an excellent prompt to introduce Constraint LP.

4.2 Naive Declarative Implementation

The inefficiency issue manifested in the previous implementation, demands a better recursive definition, that of "*the minimum element of a list is the minimum among its head and the minimum of its tail*", implemented as predicate `list_min2/2`, shown below.

```
list_min2([Min],Min).
list_min2([H|Tail],Min) :-
    list_min2(Tail,MinTail),
    min(H,MinTail,Min).

min(X,Y,X) :- X=<Y.
min(X,Y,Y) :- X>Y.
```

Although, arithmetic checks are delayed and placed "after" the recursive call, committing to the min value is delayed, i.e. variable `Min` is instantiated at the last call, after the arithmetic check. This leads to a *linear complexity*, albeit some memory inefficiencies. A puzzling point to novice learners is that comparisons take actually place backwards, from the last element of the list to the first, usually referred to as building the solution "on the way up".

This version offers itself to talk about the **if-then-else** construct (without explicit reference to the hidden cut) by rewriting the code for `min/2` as follows:

```
list_min21([Min],Min).
list_min21([H|Tail],Min) :-
    list_min21(Tail,MinTail),
    (H>MinTail -> Min=MinTail; Min=H).
```

Although, some may find that the above presents a more readable implementation, it does contain explicit unification, i.e. the use of the "=" operator, which can be easily (mis)taken for assignment.

4.3 The "Standard" Algorithmic Implementation

Following the implementation of the "naive" program where numbers are compared in the reverse order, this predicate allows for an explanation of its memory inefficiency, due to its inability to take advantage of **recursion optimization** techniques. Having been exposed in Compiler and Computer Architecture courses to the function call mechanisms, one can easily understand that having to execute code after the recursive call, a lot of information must be kept in the *memory stack* (i.e. values of variables at each recursive step), so this implementation is memory demanding. The latter offers an excellent chance to discuss **tail recursion optimization**, and the need to place the recursive call last in the body of the predicate.

The implementation shown below (predicate `list_min3/2`), is based on the "standard" algorithm that is taught in programming courses. Thus, the **accumulator pair** technique [2,5], is introduced that offers a great opportunity

to discuss *single assignment variables* in the Prolog programming context. The technique of introducing an **auxiliary predicate** is common in Prolog. It is interesting to notice that the auxiliary predicate may have the same functor name as well since it has different arity.

```
list_min3([H|T],Min) :-
    list_min3_aux(T,H,Min).

list_min3_aux([],Min,Min).
list_min3_aux([H|T],TempMin,Min) :-
    min(H,TempMin,NextMin),
    list_min3_aux(T,NextMin,Min).
```

Alternatively, since now the comparison is done before the recursive call, we could avoid the use of the `min/3` predicate and have instead two recursive calls, without causing so much inefficiency this time, as shown in predicate `list_min4/2`). However, to maintain efficiency, it is needed to explicitly insert the **cut operator** leading to a **check-and-commit** technique.

```
list_min4([H|T],Min) :-
    list_min4_aux(T,H,Min).

list_min4_aux([],MSF,MSF).
list_min4_aux([H|T],MSF,Min):-
    H < MSF, !,
    list_min4_aux(T,H,Min).
list_min4_aux([H|T],MSF,Min):-
    H >= MSF, !,
    list_min4_aux(T,MSF,Min).
```

Although the cut in the code above is inserted to take advantage of the **tail recursion optimization**, since checks are mutually exclusive, the second check could be eliminated, leading to a reduced number of checks in the code. Alternatively, as mentioned above, cut can be implicitly replaced by the "more declarative" **if-then-else** construct.

4.4 A Reduction Approach

The next implementation is in fact an ad-hoc application of the **reduce** operator commonly found in functional languages and recently in many Prolog implementations (predicate `list_min5/2`).

```
list_min5([M],M).
list_min5([H1,H2|T],Min):-
    H1 > H2, !,
    list_min5([H2|T],Min).
list_min5([H1,H2|T],Min):-
```

```
    H1 =< H2, !,
    list_min5([H1|T],Min).
```

Selecting two elements instead of a single from the list supports an early arithmetic comparison, leading to an immediate **pruning of the non-successful branch**. What is left to be decided is the repeated execution of the operation for the rest of the elements, achieved by "pushing" the result (i.e. the minimum between the two) elements to the top of the list for the next recursive call.

Alternatively, an even more compact version of the predicate relies on the `min/3` predicate mentioned previously (predicate `list_min6/2`).

```
list_min6([Min],Min).
list_min6([H1,H2|T],Min):-
    min(H1,H2,M),
    list_min6([M|T],Min).
```

The introduction of the latter provides the necessary ground to demonstrate the implementation of the *reduce* operator using **variable call**, that can work on any binary operation (e.g. min, max, etc.). The latter is achieved by simply adding one more argument to hold the predicate name of the operation and the term construction subgoal using the **univ/2** operator, as shown below:

```
list_reduce([Value],_,Value).
list_reduce([H1,H2|T],Operation,Value):-
    C =.. [Operation,H1,H2,Next],
    call(C),
    list_reduce([Next|T],Operation,Value).
```

We usually call this implementation an elegant, declarative "hack", taking advantage of the list itself to deliver the temporary result to the end. Although novice learners find this implementation rather ingenious, they rarely reproduce it in future programming tasks. This is probably due to the fact that they are not used to a *functional style* of programming; rather they prefer the more traditional "array" style of iterating the list and keeping the temporary result in a separate variable as an extra argument.

4.5 A Non-recursive Declarative Definition

A verbal description of a complete definition of `list_min/2` could be *the minimum of a list, is a member of the list such that no other member of the same list exists smaller than it*. Interestingly enough this can be directly implemented in Prolog (predicate `list_min7/2`).

```
list_min7(List,Min):-
    member(Min,List),
    not((member(X,List), X < Min)).
```

This is probably the most declarative version of `list_min/2`, reported here, and is in fact an application of the **generate-and-test** technique. Understanding, however, the operation of the predicate presents significant challenges. First of all, it demands a good understanding of **backtracking** and **negation**, i.e. the fact that once an element smaller than the current `Min` is found then the second subgoal fails, leading to a *re-instantiation* of the `Min` to the next element of the list. The process is repeated until the argument inside the negation in the second subgoal fails, for all instantiations of `X`, leading to the solution.

However, this elegant indeed definition suffers from high *computational complexity*. It does not take long to realize that it has $O(N^2)$ complexity whereas all other previous solutions (except the "super-naive" one) have linear complexity.

4.6 Using Solution Gathering Predicates

Another version (`list_min8/2`) mainly used for illustrating the operation of the `setof` **solution gathering** predicate, stressing that it is a clever trick, but with higher-than-needed computational cost.

```
list_min8(List,Min) :-
    setof(X,member(X,List),[Min|_]).
```

This version exploits the builtin predicate `setof` and follows the naive algorithmic thinking of sorting a list in ascending order to return its first element. However, one needs to realize that sorting has a larger average complexity $O(n \log n)$ than finding the minimum $O(n)$, so in general it should be avoided.

4.7 Using Assert/Retract and Failure-Driven Loops: The One to Avoid

No matter how simple a programming language can be, some of its features may be used to create the "unthinkable". The same happens with Prolog's ability to alter its program while the program is executed, i.e. asserting and retracting clauses on the fly. We refer to this version (predicate `list_min9`) as the "one to avoid", since it relies on a **"global" variable** implemented as a separate dynamic predicate, to simulate *destructive assignment*. It offers the opportunity to present a number of issues regarding assert/retract, as well as the necessity of **side-effects** inside a **failure-driven loop**.

```
list_min9([H|_T],_Min) :-
    assert(temp_min(H)),
    fail.
list_min9(List,_Min) :-
    member(X,List),
    retract(temp_min(TmpMin)),
    min(X,TmpMin,NextMin),
    assert(temp_min(NextMin)),
```

```
    fail.
list_min9(_List,Min) :-
    retract(temp_min(Min)).
```

We do not present this as a technique that someone should adopt. We just mention it as an extreme example of how flexible and "dirty" Prolog programming can get!

5 Evaluation of Efficiency vs. Perception

An interesting aspect of demonstrating the `list_min` operation in many different versions is that it leads to commenting on the efficiency of each version using automatically generated lists of random integers, best and worst case scenarios, e.g. ordered or reverse ordered lists, and Prolog statistics. Thus, a novice learner can see how each technique affects the performance.

Having completed all the classes, we requested our students to conduct experiments with all versions of `list_min`, with lists of various sizes in ascending, descending and random order. Having gathered the results in terms of cputime and number of inferences, they were asked to express their opinion which are the best three versions, by reconciling efficiency, readability and writability of code.

The results obtained by different lists sizes, ranging from 1000 to 100000 elements (1K, 30K, 60K, 100K), present some interesting aspects regarding the different predicate versions. All experiments were conducted using SWI Prolog. With respect to efficiency, we have found that the predicates above can be classified in three groups: The first group contains `list_min1`, which in fact fails to report a solution for large lists in descending order; for instance, the execution time for a list of only 30 elements is 218.6 s. Obviously, results for ascending lists are comparable with those of other predicates; however, the decision was to exclude the predicate from further testing since it would not provide any significant results with respect to random and descending lists.

The fully declarative solution (`list_min7`, second group), although it performs better, it still follows the generate and test strategy, yielding high execution times and a large number of inferences to reach a solution for the descending worst case: starting with 5.3 s for a list of 10K elements up to 548 s approximately for the list of 100K elements. This is expected, since in the descending case, the solution generator (`member/2`) produces the correct solution last, yielding the highest number of iterations. For the same reason, best results are obtained for lists in ascending order, followed by those for the random.

The third group contains all other predicates. We avoided reporting execution times, since, even for 100K lists, the former are less than a quarter of a second, across all predicates in the group, yielding no interesting (or safe) results for comparison. Instead we opted to measure *number of inferences per list element*, just to give an indication how close in terms of performance versions are. Results averaged between all tests, for each predicate are presented in Fig. 1. Finally, `list_min8` is not included in the figure, since `setof` is implemented at a lower level, so the exact number of inferences is not correctly reported by SWI-Prolog.

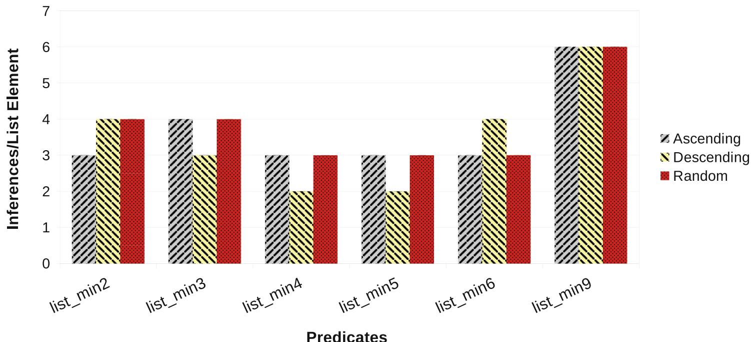

Fig. 1. Results showing Inferences/List Element on the third group of predicates.

All predicates we tested (most of them recursive) scan the whole list once no matter the type of list. So, with minor deviations in any of the ascending, descending, or random order, the number of inferences is more or less the same (the extra lines from one predicate to the other cause the extra inferences but play a minor role). Thus, performance is independent of the type of list.

Minor differences among predicates in the figure are attributed to the order of checks. For instance, in `list_min2` arithmetic comparisons occur "backwards", whereas in `list_min3` occur on the "way down" to the base recursive case, thus showing slightly different behaviour on the extreme cases (ascending/descending order). It should be noted that the `assert/retract` version seems to be unaffected by the order of elements in the list and yields the higher number of inferences, due to constantly accessing Prolog memory.

Regarding student perception, the definitions of `list_min8`, `list_min2` and `list_min7` were among the first three preferences, gathering 72%, 48% and 40% respectively of the students who preferred them in their top-three choices, although the last one requires a considerable number of inferences compared to all the rest. All other versions were roughly equally preferred. It was surprising that 16% of the students declared as a top-three choice `list_min1`; it takes an enormous amount of cputime to complete which makes it practically useless but it was preferred for its readability. Even more surprising is that 20% included `list_min9` in their best three choices; it is extremely complex and far from purity but it may match the programming style that learners have been exposed to in previous courses.

6 Conclusions

We presented the flexibility of Prolog by using a single operation and multiple programming techniques that result in different implementations. Each version of the predicate `list_min` allows space to discuss all interesting features of Prolog. The code variations gave us the opportunity to discuss declarativeness versus efficiency issues as well as readability, purity and dirty characteristics of the

language. As educators, we make use of those examples in our class of novice Prolog learners and we showed their perceptions and evaluations.

References

1. Bratko, I.: PROLOG Programming for Artificial Intelligence, 4th edn. Addison-Wesley Longman Publishing Co., Inc, USA (2012)
2. Brna, P., et al.: Prolog programming techniques. Instr. Sci. **20**(2), 111–133 (1991). https://doi.org/10.1007/BF00120879
3. Clocksin, W.F., Mellish, C.S.: Programming in Prolog, 5 edn.. Springer, Berlin (2003). https://doi.org/10.1007/978-3-642-55481-0
4. 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: 50 Years of Future, LNAI 13900, pp. 184–193. Springer, Cham (2023)
5. Gegg-Harrison, T.S.: Learning prolog in a schema-based environment. Inst. Sci. **20**(2), 173–192 (1991). https://doi.org/10.1007/BF00120881
6. Hermenegildo, M.V., Morales, J.F., Lopez-Garcia, P.: Some thoughts on how to teach prolog. In: Warren, D.S., Dahl, V., Eiter, T., Hermenegildo, M., Kowalski, R., Rossi, F. (eds.) Prolog: 50 Years of Future, LNAI 13900, pp. 107–123. Springer, Cham (2023)
7. Hogan, A., Blomqvist, E., Cochez, M., D'amato, C., Melo, G.D., Gutierrez, C., Kirrane, S., Gayo, J.E.L., Navigli, R., Neumaier, S., Ngomo, A.C.N., Polleres, A., Rashid, S.M., Rula, A., Schmelzeisen, L., Sequeda, J., Staab, S., Zimmermann, A.: Knowledge graphs. ACM Comput. Surv. **54**(4), 1–37 (2022). https://doi.org/10.1145/3447772
8. Morales, J.F., Abreu, S., Hermenegildo, M.V.: Teaching prolog with active logic documents. In: Warren, D.S., Dahl, V., Eiter, T., Hermenegildo, M., Kowalski, R., Rossi, F. (eds.) Prolog: 50 Years of Future, LNAI 13900, pp. 171–183. Springer, Cham (2023)
9. O'Keefe, R.A.: The Craft of Prolog. MIT Press, Cambridge (1990)
10. Ross, P.: Advanced Prolog: Techniques and Examples. Addison-Wesley (1989)
11. Sekovanić, V., Lovrenčić, S.: Challenges in teaching logic programming. In: 2022 45th Jubilee International Convention on Information, Communication and Electronic Technology (MIPRO), pp. 594–598 (2022). https://doi.org/10.23919/MIPRO55190.2022.9803530
12. Sterling, L., Shapiro, E.: The Art of Prolog (2nd Ed.): Advanced Programming Techniques. MIT Press, Cambridge (1994)
13. Van Someren, M.W.: What's wrong? Understanding beginners' problems with Prolog. Instr. Sci. **19**(4), 257–282 (1990). https://doi.org/10.1007/BF00116441
14. Warren, D.S.: Introduction to Prolog. In: Warren, D.S., Dahl, V., Eiter, T., Hermenegildo, M., Kowalski, R., Rossi, F. (eds.) Prolog: 50 Years of Future, LNAI 13900, pp. 3–19. Springer, Cham (2023)
15. Warren, D.S.: Writing correct prolog programs. In: Warren, D.S., Dahl, V., Eiter, T., Hermenegildo, M., Kowalski, R., Rossi, F. (eds.) Prolog: 50 Years of Future, LNAI 13900, pp. 62–70. Springer, Cham (2023)