Teaching Prolog through Grammars

David S. Warren*,†

Stony Brook University, New York, USA

Abstract

This short paper describes how one might teach Prolog to undergraduate students using grammars as the motivating examples. Grammars for natural and formal languages provide great examples with which students are already familiar. They require only positive rules and display a fully declarative framework. This paper shows that perhaps the original motivation of natural language processing for the invention of Prolog was not happenstance.

1. Introduction

This informal short paper describes the idea of teaching Prolog using context-free grammars. Grammars are a perfect framework to display and explore the declarativity and generality of the Prolog language. Grammars underlay the original motivation for the invention of Prolog by Alain Colmerauer. Context-free grammars (CFGs) are an independently formulated, studied, and understood formal mathematical structure, with much known about them and their processing [1]. So, there is an independent body of knowledge to which we can compare what Prolog does with grammars. We can compare what Prolog does to what is done by the best independently, specially developed, algorithms. Also, from the point of view of Prolog, grammars are positive programs so they don't raise the complicated issues of negation. Students are familiar with and understand the idea of a (formal) language so they have introspective access to data that they can use to determine if a grammar is correctly representing a known language. Also grammars fundamentally involve inductive definitions, where phrases are embedded in other phrases. Students intuitively understand this property of language and can use that understanding as an entry to understanding induction and recursion in programming.

Also grammars are, in some intuitive sense, "Datalog complete" in that many kinds of complexities of Datalog programs can be exemplified, and made visual, in grammars. Thus, grammars are a good focus of study to learn about what Datalog (and Prolog) programs can do. And when we use lists to represent input strings, students are introduced to Prolog structures. Also building parse trees is a natural grammatical problem, and that introduces the more general Prolog data structure.

Finally, the study of grammars in Prolog highlights the importance of tabled evaluation. Many grammars for familiar languages are naturally left recursive. Native Prolog does not process them correctly. But with tabled evaluation, Prolog does. And even when native Prolog evaluation does terminate, it is often exponential where tabled evaluation is guaranteed to be polynomial. And well-known general context-free recognition algorithms, such as Earley and CKY, are the algorithms carried out by tabled evaluation. The student can see that in fact, tabled evaluation is the more natural and complete Prolog evaluation strategy, with native Prolog evaluation being best thought of as an optimization that in some cases improves time and/or space performance.

There is a long history of good ideas about how to teach Prolog, recent ones including those presented in this workship, and some presented in an edited collection [2]. We hope to contribute the approach described in this paper to the mix.

PEG 2024: 2nd Workshop on Prolog Education, October, 2024, Dallas, USA.

warren@cs.stonybrook.edu (D. S. Warren)

0000-0001-7567-8156 (D. S. Warren)

2. Initial Considerations

Prolog is noted for its natural handling of grammars, so natural in fact that it includes a special syntax for writing grammars, the "Definite Clause Grammar" (DCG) syntax. We do *not* use the DCG syntax in our initial presentation and development of grammars in Prolog, but instead use native Prolog syntax. The DCG notation, while convenient for knowledgeable users, can obscure the basic workings of Prolog for beginners. And since we are trying to teach Prolog through grammars, the DCG notation is just a distraction or source of confusion. At the very end of teaching grammars, one could introduce the students to the DCG notation when their relationship to Prolog is clearly understood.

Also, normally grammars in Prolog use lists to represent the strings in a language. In fact difference lists are used. But for beginning students lists are relatively complex data structures, and their representation of difference lists, with their rather sophisticated use of unification to efficiently implement concatenation, is not so easy to understand. So we avoid the list representation of strings and rather use a Datalog representation [3]. Given a string, we number the spaces between the symbols in the string and store 3-ary facts to represent the words in the various positions in the string. For example, the string consisting of the words "mary kicked the ball" is represented by four Prolog database facts:

```
word(0,mary,1).
word(1,kicked,2).
word(2,the,3).
word(3,ball,4).
```

In this way, a subsequence of an input sequence of symbols can be represented by the pair of position integers that begin and end the subsequence. With this representation the representation of grammars in Prolog is in fact in the Datalog sublanguage. This treatment allows students to build an understanding of the declarative meaning and computational characteristics of Prolog before undertaking the complexity of complex data structures and structure unification. Then, after grammars in Datalog are well understood, students can be introduced to lists and the power of structure unification and see how the same Prolog grammar can be used for generation as well as recognition.

And finally, we use *tabled evaluation* as the default evaluation strategy for computing with grammars. We want our development to emphasize the declarative aspects of grammars, that they are a general way to inductively define languages, i.e., sets of sequences of symbols. One needn't think about an algorithm that recognizes whether a sequence is in the language defined by a grammar in order to understand what sequences are indeed in the language. But if we use Prolog's native evaluation strategy we have to explain why Prolog does not process left-recursive grammars correctly, but will loop infinitely on some strings. So, either we make the seemingly arbitrary (and in fact, unnecessary and awkward) restriction to grammars that do not contain left-recursion, or we use the stronger evaluation strategy of tabling. We could, of course, begin with an explanation of Prolog's native strategy and show why the non-left-recursion restriction is necessary. However, that would intertwine the declarative understanding of grammars with the computational limitations of Prolog's native strategy, in the end potentially confusing the students. So we strongly suggest using the tabled evaluation strategy when introducing Prolog to students using grammars. Native Prolog evaluation can come later.

3. The Development

In this section we provide an outline of a sequence of topics and exercises one might use to introduce Prolog using grammars.

It is important to use a very few motivating example languages throughout the development. The example languages should be already relatively familiar to students, so they can draw on their intuitions to understand the various concepts. I think perhaps the best general language is the students' natural language. I would use English. Of course English, and any natural language, is much too big and complex to be mostly covered in a single unit. So the grammar would be highly simplified covering

only a small, formalized fragment of the language. For knowledgeable computer science students a simple programming language, or fragment of a more complex programming language, might be appropriate. Even a mathematical language, like the language of algebraic expressions or equations, might be appropriate. We will first use English in this presentation and later introduce simple arithmetic expressions.

We start with the introduction of the idea of a CFG using the example grammar. We use the usual notation of CFGs. The beginning of an example grammar for English might be:

```
sentence --> noun_phrase verb_phrase
noun_phrase --> proper_noun
noun_phrase --> determiner noun
verb_phrase --> intransitive_verb
verb_phrase --> transitive_verb noun_phrase
verb_phrase --> verb_phrase prepositional_phrase
prepositional_phrase --> preposition noun_phrase
proper_noun --> john | maria | manuel
determiner --> the | a | every
noun --> man | woman | student | dog | park | telescope
noun --> noun prepositional_phrase
intransitive_verb --> walks | talks
transitive_verb --> loves | hits | walks | sees
preposition --> in | with | to
```

Depending on student backgrounds, one would describe how the grammar generates sentences: defining terminals and nonterminals, how rules replace in symbol sequences nonterminals by other symbol sequences, and sentences are such generated sequences of terminals starting with the start symbol.

An example sentence in this grammar is "A man sees a student in the park with a telescope." It would be represented in the Prolog database as:

```
word(0,a,1).
word(1,man,2).
word(2,sees,3).
word(3,a,4).
word(4,student,5).
word(5,in,6).
word(6,the,7).
word(7,park,8).
word(8,with,9).
word(9,a,10).
word(10,telescope,11).
```

Possible questions at this point:

1. How many sentences are in this language?
2. What are the noun phrases in this sentence? Give the pairs of integers that are the endpoint of the noun phrase subsequences. Note that some noun phrases are contained in other noun phrases.

Note that each nonterminal determines a set of finite sequences of terminal symbols: those generated from the rules by starting with that single nonterminal as the start symbol.

Next we introduce Prolog rules. Prolog defines relations, and here we want to define a relation consisting of pairs of integers. For each nonterminal we want to define a relation that contains the pair $\langle i, j \rangle$ if and only if the input string from i to j can be generated by that nonterminal.

Example rules would be:

```
noun(I,J) :- word(I,man,J).
..
noun_phrase(I,J) :- determiner(I,K), noun(K,J).
..
```

And so on. We get one rule for each grammar rule, and explain the grammar vertical bar (or) symbol and what to do about it in Prolog. I would probably say it's shorthand for multiple simpler rules, rather than introduce Prolog's disjunction at this point.

Next, we talk about Prolog's evaluation strategy of top-down backward chaining with tabling (or memo-ization), which is done in XSB Prolog by adding `:- auto\_table` command to indicate tabling. Then we can pose the query `:- sentence(0,10)`, which asks whether the 10-word sentence represented by the `word/3` facts above is indeed a sentence generated by our grammar. After experimentation, changing the `word/2` facts, we can modify the Prolog program by adding `write` statements at the end of the Prolog rules that define the nonterminal `noun_phrase` to print out the nonterminal and endpoints of recognized noun phrases. We can run this revised program see if Prolog finds the same noun phrases we do, and we can discuss the differences, if any.

If we have available a bottom-up evaluator (or XSB's subsumptive table indexing capability by for each nonterminal `NT`, `:- table NT/2 as subsumptive,index(0).`), we can run the program bottom up. We can start with a diagram that consists of the input sentence as a labeled chain graph, the nodes are integers, and the words are labels on the appropriate edges. We draw this labeled graph. Then for each new fact added by the bottom-up evaluation, we add a new edge, labeled by the nonterminal symbol and connecting the integer nodes of the added fact. This algorithm can be seen to be the CYK algorithm (originally by Itiroo Sakai [4] and rediscovered by each of Cocke, Younger, Kasami, and Schwartz).

We can do the same for the original tabled evaluation of the Prolog program (seeing the added edges by adding `write` statements at the ends of clauses). This expanded graph is called a "chart", and this is known as recognition using the chart parsing strategy. If the grammar is in Chomsky form (at most two symbols on the right-hand-side of any rule), then one gets the Earley recognition algorithm [5] for that grammar.

4. Possible Lectures and/or Exercises

In the following we describe other aspects and uses of grammars that can be used as ways to explore Prolog.

4.1. Parse Trees

We can add an argument to the nonterminal predicates (say as a new first argument) that constructs the parse tree of the string recognized. For example, we would define the rule:

```
sentence(sentence(S,V),I,J) :- noun_phrase(S,I,K), verb_phrase(V,K,J).
```

We can then experiment with generating parse trees for various sentences (and phrases). This introduces Prolog terms in a natural way, since terms are naturally the right kind of tree. It also introduces multiple answers when parse trees for ambiguous sentences are generated. This is a good point to discuss nondeterminism, and how it arises and is treated by Prolog.

4.2. Number Agreement Constraints in English

We could explore refinements of our, admittedly very simple, grammar for English. We might elaborate it by adding plural nouns and third person plural verb forms, and then adding additional arguments to the necessary nonterminals to ensure number agreement. By adding nouns with the same singular and plural forms (e.g., "sheep"), we can show the use of unification to infer the number of some particular uses of those nouns, e.g., "the sheep walks" marks this subject as singular from the third-person verb.

4.3. Pronouns, Case, and Antecedents

We could continue with the English grammar to add more complex elaborations, such as a treatment of pronouns of the appropriate case, and even finding antecedents to pronouns by accumulating a list of possible antecedent noun phrases and then choosing one of the appropriate gender from the accumulated list to assign as the coreferent of a later pronoun. This, however, would take us further into the innards of Prolog and its list processing and is probably best left for later, if at all.

4.4. Exploring an Expression Grammar

An important grammar elaboration is the association of some semantics to the strings of its language. While much interesting work has been done with English grammars to this end, these concepts are probably easier for students to understand in a more formal, mathematical setting.

So, to discuss issues of language semantics, my choice would be to introduce a new grammar to explore these aspects of grammar processing. I would introduce a simple arithmetic expression grammar, such as:

```
expr --> expr + term
expr --> term
term --> term * primary
term --> primary
primary --> <integer>
primary --> <atom>
primary --> ( expression )
```

This is again left-recursive, so we need tabling to get a complete recognition strategy. With this grammar we can discuss the grammatical concepts of operator priorities and association, with overriding with the use of parentheses.

4.4.1. Semantics

With this grammar we can begin to discuss the concepts of the semantics, or meaning, of phrases, and the translation of phrases into other languages. It is natural to add an argument to each nonterminal of this expression grammar to contain the computed numeric value of the recognized expression, the "meaning" of the expression. For this we must introduce Prolog's treatment of arithmetic and the `is/2` builtin predicate. We would constrain the expressions to be evaluated to have only integer leaves. (We might extend this to handle atom constants as variable names by adding a parameter of a variable-value mapping and use it to retrieve the values of the variables as identified by atom leaves. This would require the introduction of at least simple lists and their use.)

4.4.2. Translation

As an example of translation, we could add an extra argument of a sequence (list) of 3-operand intermediate code instructions that when executed by an appropriate machine would evaluate the input expression. This might be of interest to computer science students interested in compilers.

4.4.3. Interpretation

A bit further afield, but perhaps interesting to more advance computer science students, is an interpreter for a simple procedural programming language. The programming language grammar would generate an abstract syntax tree, which would be used by an interpreter to carry out the program provided an initial state, or perhaps a sequence of inputs.

5. Data Structures

This all raises the question of how and when to introduce data structures to Prolog programmers. I would introduce general terms by adding an argument to the nonterminal predicates of a recognizer to build the parse tree, as suggested above. This is a very natural thing to do for grammars and it introduces terms in their full generality.

After that I would introduce lists as a special form of term expression, a right-linear tree with the ./2 structure symbol and the `[]` constant as the final tail (properly explained). I would give them the library list predicates of `member/2` (and maybe `append/3`) and give them simple exercises to use them in the context of grammars. For example, some of the suggested possible elaborations above could be done in this way.

After students have some simple familiarity with using lists, I would then introduce the append/3 predicate and its definition by using it to go from a parse tree back to the string that it generates (or generates it). This problem can be used to explore complex uses of `append/3` in its multiple modes. So first take the program that recognizes and constructs a parse tree, and eliminate the two input string arguments. The predicates of this program have just one argument and simply tests whether an input parse tree is indeed a parse tree for the nonterminal of the grammar. Then add one new argument that is the list of terminals generated by the associated parse tree. Here one uses `append/3` to put together the strings associated with the component (sub) parse trees. So now one can give a valid parse tree and get back a list of terminals that it would generate. Note one has to be careful that the `append/3` calls all terminate in the modes in which they will be called. This can be pointed out to students and used to introduce the idea of modes and their importance in keeping computations finite. Then one can think about running this parse-tree-to-string backwards, giving it a string (list) and generating a parse tree. This will assuredly break the mode requirements of the existing order of the `append` calls in the previous definition. So we reorder the `append` calls so they all comply with the finitary requirements for the modes. Now this program can be run given a string and generate a parse tree. This is another parser with the input representation of a list rather than as a chain graph in the database.

At this point it may be appropriate to introduce the difference list input string representation of Prolog's DCGs. Rather than initially explaining difference lists in their full generality, I would explain the two arguments as list of tokens passed into a predicate and the remaining list of tokens passed out of the predicate after the matched sublist has been recognized. This, I think, accords better with programming students understanding of parameters. Once it is seen how this works, then we might try to explain this as an example of the general difference list data structure, and explain how it allows concatenation to be done with simple unification. That's why the `append/3` calls that we saw above are no longer necessary. And we can do sentence recognition and generation with the same program. At this point, we might introduce and explain the DCG notation.

5.1. More with DCGs

There are, of course, many more ideas we could explore using grammars and many more exercises to work through. For example, we could point out that the same DCG can use either the `word/3` input string representation or the difference list representation; only the definition of (the invisible) `'C'/3` predicate needs to be changed. In fact, DCG's can be used for defining quite general state-changing algorithms, the two added arguments being the state at entry to the predicate and the state at exit from the predicate. The "terminals" are parameters to the `'C'/3` predicate that can describe state changes.

We might also note that we could extend the `word/3` input representation to define a general labeled graph, not just a chain as we have done here. We could then interpret that graph as a finite state automaton with a single start state and single final accepting state. With this interpretation our grammar recognizer actually tests for intersection of the two languages: the context-free language of the DCG and the finite-state language of the automaton.

Depending on the sophistication of our students, we might want to explore the computational complexity of recognition with various grammars. The following simple grammar has interesting

complexity properties. It can be coded as a DCG and then explored by students generating and running example inputs. They can get a sense of complexity by generating longer and longer input strings to recognize and seeing the time it takes to recognize each.

The grammar:

```
A --> a A A
A -->
```

generates all nonempty sequences of "a". It is not left recursive so recognition does terminate under the native Prolog evaluation strategy. However, it is exponential. With tabled evaluation, it is polynomial, cubic in fact (when the `word/3` input representation is used).

We might also compare the tabled evaluation of recognition for the following two grammars for all strings consisting of "a"s, one left recursive and the second right recursive:

```
A --> A a
A -->
```

and

```
B --> a B
B -->
```

One can build the chart for an input string of "a"s and see how the recognition complexity differs for the two grammars. The first is linear and the second is quadratic.

Another interesting grammar to consider with tabled evaluation is:

```
A --> A a
A --> A b
A -->
```

It generates all sequences of "a"s and "b"s. It is left recursive, so terminates only when tabled evaluation is used. This example shows that tabled evaluation requires "full" coroutining, since the first and second rules must be applied alternately to recognize strings with both "a"s and "b"s. There is no top-down stack-based scheduling that will allow this grammar to be properly processed.

6. Conclusion

We enumerate several advantages of introducing Prolog to students through grammars.

1. Language is a familiar domain: students come with strong intuitions that can be used to advantage in the development.
2. The problems are intrinsically interesting. Precise, formal specification of infinite sets of interesting sequences and problems of recognition and parsing are natural and interesting.
3. Recursion (and induction) are fundamental concepts for languages. They are made visible through embeddings; students know and see that noun phrases may contain other noun phrases. Recursion is demystified. Another advantage in introducing recursion and induction through grammars is that it is not obvious how to use iteration. When introducing recursion in a context in which a student knows iteration works can result in confusion and rejection. With grammars, there is no iteration, only natural recursion.
4. Unification has natural application in English, as in subject-verb number agreement: number can be determined from the subject or the verb, and if both, then they must agree. This is exactly implemented with simple unification and it is easy to see why it is useful.
5. Grammars are inherently declarative and thus center declarativity. They don't require cut or assert or any nonlogical features. Students see how much can be done within the declarative framework, and tend to not to resort to nonlogical constructs quickly when frustrated.

6. Grammars exemplify the power of evaluation in multiple modes. Their declarativity makes changing modes easier even if the subgoals have to be reordered to satisfy mode requirements.
7. Grammars show the criticality of tabled evaluation to obtain the full benefit of declarativity. Without tabling, this entire development would feel (and be) significantly more contrived.

References

[1] M. A. Harrison, Introduction to Formal Language Theory, Addison-Wesley, 1979.

[2] D. S. Warren, V. Dahl, T. Eiter, M. V. Hermenegildo, R. A. Kowalski, F. Rossi (Eds.), Prolog: The Next 50 Years, volume 13900 of *Lecture Notes in Computer Science*, Springer, 2023.

[3] R. A. Kowalski, Logic for problem solving, volume 7 of *The computer science library : Artificial intelligence series*, North-Holland, 1979. URL: https://www.worldcat.org/oclc/05564433.

[4] I. Sakai, Syntax in universal translation, in: Proceedings 1961 International Conference on Machine Translation of Languages and Applied Language Analysis, volume II, Teddington, England, 1962, pp. 593–608.

[5] J. Earley, An efficient context-free parsing algorithm, Commun. ACM 13 (1970) 94–102. URL: https://doi.org/10.1145/362007.362035. doi:10.1145/362007.362035.