A Brief Study of Prolog for Writing Context Free Grammars and a Comparison of the Language with Lisp

Jake Sauter
May 12, 2017

1 Abstract

The use of Prolog will be examined for the purpose of writing context free grammars. The context free grammars mentioned will be a collection of grammar rules that can properly identify a grammatical sentence in the defined language, or with the help of Prolog easily produce all grammatical sentences and sub-sentence phrases in the language. Through this case study, Prolog will be examined as in depth as possible for its wide array of uses, and its strengths and weaknesses will be compared and contrasted with the strengths and weaknesses of lisp.

2 Background and Introduction

Natural Language Processing (NLP) is an area of research and application that explores how computers can be used to understand and manipulate natural language text or speech to do useful things. The foundations of NLP lie in a number of disciplines, namely computer and information sciences, linguistics, artificial intelligence and cognitive science [1].

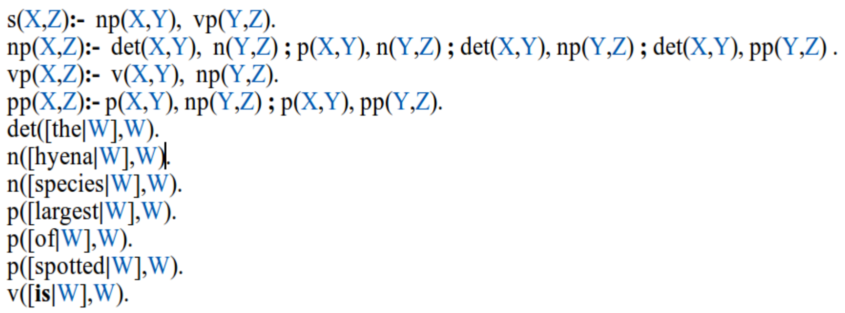
In this study, the logical programming language of Prolog will be reviewed for its use in a small but useful type of NLP program used to implement Context Free Grammars (CFGs). This Prolog program will be used to identify the grammatical sentence that the CFG was designed to describe. In doing so, the CFG is also describing a language, as a language is set of all possible sentences (and only the possible sentences) described by the grammar [2]. In this study five sentences (languages) will be modeled using CFGs, however this can be done using two methods in Prolog. The CFG can be modeled explicitly as a logical system using fundamental Prolog terms, or Definite Clause Grammar (DCG) syntax that can be used to more quickly and easily implement these structures. This being said, the five sentences prior mentioned will be modeled using these two methods.

Much of the fundamental knowledge of Prolog will be needed in order to implement these CFGs in the language, so this adds the possibility of the evaluation of Prolog versus and other arbitrary programming language for their strengths and weaknesses. For this comparison, LISP will be used to compare as it has also been a fundamentally important language in the realm of initial advancement in Artificial Intelligence. It is important to mention that the comparison will be a general comparison, and not in the comparison in the languageâĂŹs abilities to implement a CFG

3 Program Description

The tasks of the study included gathering five sentences to model using CFGs, accurately describe their grammatical decomposition in a manner consistent with CFGs, and implement each of the accurately described CFGs using two different methods. The method of implementation include a “manual“ type method, by only using fundamental Prolog terms and in an “automatic“ type method, by using the internal DCG syntax/structure.

The “manual“ type method was going to be more difficult so this is where the project began, for the reasoning of if the CFG can work using the manual type method, then little effort should be required to model the CFG using the built-in structure. The fundamental idea while implementing efficient CFGs in Prolog (and possibly other programming languages) is the idea of difference lists. The key idea underlying difference lists is to represent the information about grammatical categories not as a single list, but as the difference between two lists [2]. A small example describing a sentence broken up into top level noun phrase, verb phrase composition could be [a,woman,shoots,a,man] [a,woman]. The sentence represented is “A woman shoots a man“. This difference list representation could be describing the top level noun phrase of the sentence, being “a woman“, which is the disjunction of the first list with the second list. This a powerful idea and makes CFGs much more efficient in Prolog than other methods, such as building a sentence bottom-up using an append like action. An example of how grammatical rules can be composed to form a CFG using this method can be seen below.

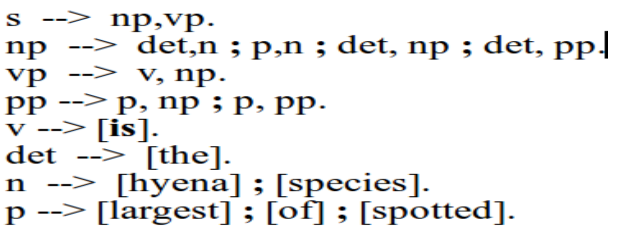


```
s(X,Z):-  np(X,Y),  vp(Y,Z).
np(X,Z):-  det(X,Y),  n(Y,Z) ; p(X,Y), n(Y,Z) ; det(X,Y), np(Y,Z) ; det(X,Y), pp(Y,Z) .
vp(X,Z):-  v(X,Y),  np(Y,Z).
pp(X,Z):- p(X,Y), np(Y,Z) ; p(X,Y), pp(Y,Z).
det([the|W],W).
n([hyena|W],W).
n([species|W],W).
p([largest|W],W).
p([of|W],W).
p([spotted|W],W).
v([is|W],W).
```

Figure 3.1: A representation of a ”manual” CFG in Prolog

This method allows a recursive action to take place in Prolog, breaking a sentence up into its components to hopefully form the sentence given (in list form to the first argument of s, with the second argument being the empty list).

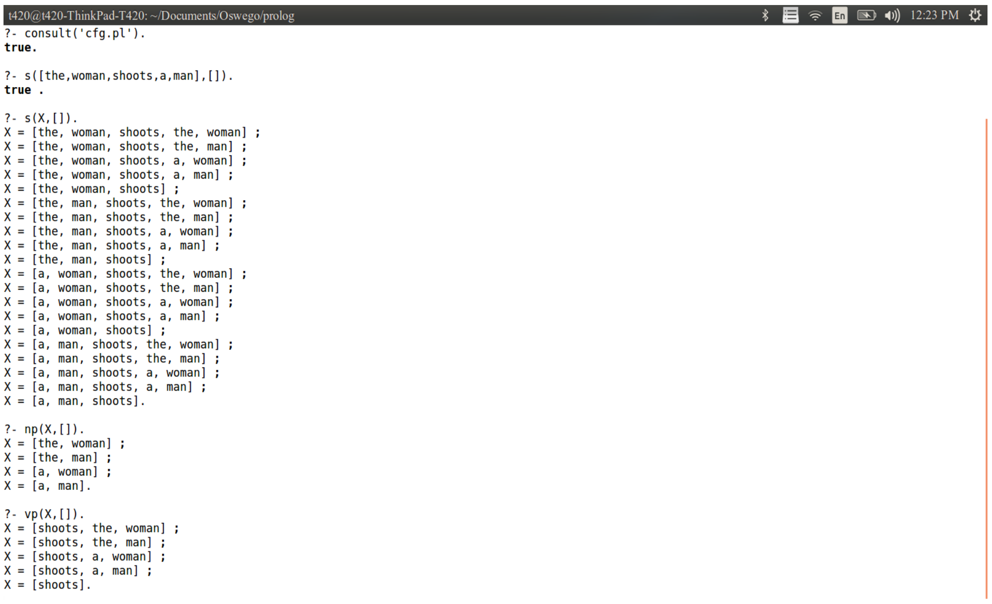
The “automatic“’ type method uses the same concept of difference lists and the idea of constitutionality as well, however it is all hidden. A CFG can be modeled using the DCG syntax in Prolog that can be more comfortably read by anyone who is familiar with the typical structure given to CFGs in the modeling process. An example of how grammatical rules can be composed to form a CFG using this method can be seen below.



```
s  -->  np,vp.
np  -->  det,n ; p,n ; det, np ; det, pp.
vp  -->  v, np.
pp --> p, np ; p, pp.
v --> [is].
det  -->  [the].
n  -->  [hyena] ; [species].
p --> [largest] ; [of] ; [spotted].
```

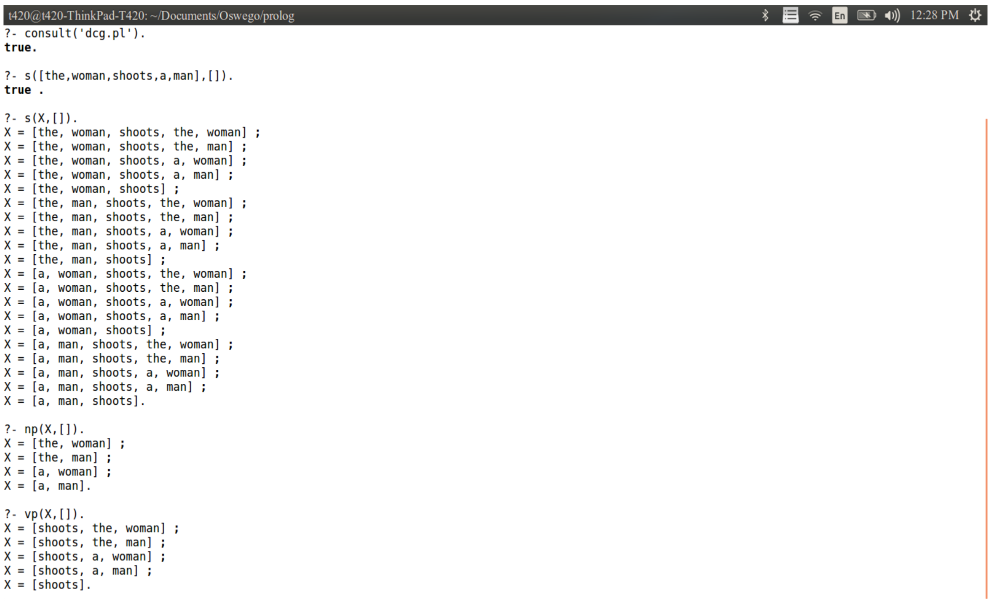
Figure 3.2: A representation of an ”automatic” CFG in Prolog

4 DEMOS



```
?- consult('cfg.pl').
true.

?- s([the,woman,shoots,a,man],[]).
true .

?- s(X,[]).
X = [the, woman, shoots, the, woman] ;
X = [the, woman, shoots, the, man] ;
X = [the, woman, shoots, a, woman] ;
X = [the, woman, shoots, a, man] ;
X = [the, woman, shoots] ;
X = [the, man, shoots, the, woman] ;
X = [the, man, shoots, the, man] ;
X = [the, man, shoots, a, woman] ;
X = [the, man, shoots, a, man] ;
X = [the, man, shoots] ;
X = [a, woman, shoots, the, woman] ;
X = [a, woman, shoots, the, man] ;
X = [a, woman, shoots, a, woman] ;
X = [a, woman, shoots, a, man] ;
X = [a, woman, shoots] ;
X = [a, man, shoots, the, woman] ;
X = [a, man, shoots, the, man] ;
X = [a, man, shoots, a, woman] ;
X = [a, man, shoots, a, man] ;
X = [a, man, shoots].

?- np(X,[]).
X = [the, woman] ;
X = [the, man] ;
X = [a, woman] ;
X = [a, man].

?- vp(X,[]).
X = [shoots, the, woman] ;
X = [shoots, the, man] ;
X = [shoots, a, woman] ;
X = [shoots, a, man] ;
X = [shoots].
```

Figure 4.1: An example run of a loaded ”manual” cfg



```
?- consult('dcg.pl').
true.

?- s([the,woman,shoots,a,man],[]).
true .

?- s(X,[]).
X = [the, woman, shoots, the, woman] ;
X = [the, woman, shoots, the, man] ;
X = [the, woman, shoots, a, woman] ;
X = [the, woman, shoots, a, man] ;
X = [the, woman, shoots] ;
X = [the, man, shoots, the, woman] ;
X = [the, man, shoots, the, man] ;
X = [the, man, shoots, a, woman] ;
X = [the, man, shoots, a, man] ;
X = [the, man, shoots] ;
X = [a, woman, shoots, the, woman] ;
X = [a, woman, shoots, the, man] ;
X = [a, woman, shoots, a, woman] ;
X = [a, woman, shoots, a, man] ;
X = [a, woman, shoots] ;
X = [a, man, shoots, the, woman] ;
X = [a, man, shoots, the, man] ;
X = [a, man, shoots, a, woman] ;
X = [a, man, shoots, a, man] ;
X = [a, man, shoots].

?- np(X,[]).
X = [the, woman] ;
X = [the, man] ;
X = [a, woman] ;
X = [a, man].

?- vp(X,[]).
X = [shoots, the, woman] ;
X = [shoots, the, man] ;
X = [shoots, a, woman] ;
X = [shoots, a, man] ;
X = [shoots].
```

Figure 4.2: An example run of a loaded ”automatic” cfg

5 Comparison of Languages

Throughout the course of this project the basic fundamental concepts were reviewed. It was discovered that Prolog‘s claim of being a logically motivated programming language was certain true, and discovered that it operated very differently from most programming languages. However it was also clear that LISP and Prolog could both be characterized as recursive list processing languages.

The basis of Prolog can tied to first order (predicate) logic. An individual (constant) in a Prolog program can be given attributes by stating a rule in a knowledge base that a predicate function, if given this constant, will return true. Prolog also makes use of implication, having more complex rules that make use of the ”if âĂę then” logical concept. The rules are in head:- body form. It is said that the head of a rule is true if the body of the rule is true, therefore Prolog can encapsulate inference. The body of rules can become more complex as well with the addition of logical conjunction and disjunction with the “,“’ and “;“ operators respectively. A variable in Prolog is identified by either starting with a special character (such as a capital letter of an underscore) and can take any value. Recursive term structure and variable unification is the source of much of Prolog‘s power [2]. Programmatic unification in Prolog is the act of exploring the space (initializations of variables) to find a situation that a given rule with included variables is true. This idea of unification can be used as a theorem prover in a way, so any powerful programs in Prolog are really unification queries about the recursively defined knowledge base. This makes Prolog fundamentally different than and other programming language. The final powerful concept in Prolog that this project makes use of is its lisp processing ability. In Prolog it is very simple to recursively process lists to convert them or do any other operation needed, because of the “|“ operator, which allows for the splitting of a list into its ”head” and ”tail” with the head of a list being the first element and the tail of the list being the rest of the elements in a list, encapsulated in a list. When writing a Prolog Program one must think of first defining the knowledge base in a declarative way, then must think in procedural way to see how queries will interact with the world.

The basis of Lisp lies in list processing and has a core concept similar to Prolog. In Lisp, one can take the CAR of a list (being similarly defined as the head of a list in Prolog) or one can take the CDR of a list (being similarly defined as the tail of a list in Prolog). Lisp also does a really good job with recursion, and Lisp programs are commonly built using lots of predicate functions (similarly to Prolog), which can be passed to other functions quite simply to aid in calculations and processing. A very powerful concept that Lisp makes use of is mapping. In Lisp it is quite simple to map an operation to every element in a list, or multiple lists, with the list or lists being the operation‘s arguments. Lisp more closely relates to more common programming languages as it can be used functional language, however it can also be used in a straightforward procedural way, which is different from Prolog as it is very hard (however possible) to initially think of writing a Prolog program a procedural process. More ties exist from Prolog to Lisp than may be seen at first glance. The act of programming in Prolog makes the programmer think declaratively at first, then procedurally, however this is very similar to what skilled programmers do when writing larger programs all the time. The programmer must think about the world they are modeling (Whether programming in an object oriented manner or not) then think about how this world can be modeled procedurally. In conclusion, one may think that different thought processes may be had when programming in Lisp or Prolog, however when a higher level of understanding is reached it is possible to understand that they are very similar processes if both are done in a ”proper” manner.

6 Reflections and Conclusions

I had a lot of fun learning the big moves of Prolog, and learned that Prolog and Lisp are a lot more similar than I initially thought. When first looking at Prolog it seemed very foreign, however once I started learning about the use of unification to determine a resultant list through rules that were defined using similar list processing methods in Lisp, I felt at home. A prior idea that I had was strengthened through this study, being that the more languages a programmer is exposed to the better, not necessarily because they will know the tool for the job (even though this is very helpful) but because it forces the programmer to think in a different way about similar tasks, which allows for the mind to be strengthened and new views of procedures to be had.

7 Works Cited

[1] Chowdhury, Gobinda G. "Natural Language Processing." Annual Review of Information Science and Technology. Wiley Subscription Services, Inc., A Wiley Company, 31 Jan. 2005. Web. 12 May 2017.

[2] Blackburn, Patrick, Johannes Bos, and Kristina Striegnitz. Learn Prolog Now! London: College Publications, 2006. Print.