Prolog via Grammars

Tom Cornell

April 28, 1998

Contents

Chapter 1

Proplog—Propositional Grammars

We will illustrate the resolution rule with the simple theory in propositional logic given in table 1.

Table 1 A Proplog Theory.

```
*/
s :- np, vp.
s :- np, aux, vp.

np :- det, n.   np :- name.

det :- the.     det :- every.

n :- prisoner.

% No clauses for name.

aux :- has.

vp :- v.

v :- escaped.

% Lexicon:
the.   every.
prisoner.
has.
escaped.
/*
```

This theory looks a lot like a phrase structure grammar, but note that it means something a little funny, since it just treats what we think of as grammatical categories as if they were simply facts. So all we can really do is establish whether or not `s` is a true fact. Roughly, the lexicon is a collection of facts and the program can tell us whether or not these facts can be combined in such a way as to make an `s` or a `vp`, for example. So ultimately the only interesting questions we can ask are questions like "Are there any S's," or "Are there any NP's." We cannot, for example, find out what those S's and NP's actually are. Still, it makes a good starting place: the subject matter is familiar, and the form is simple.

Suppose then that our initial goal is the query "`?- s.`" We apply our selection function to select an atom from the goal. The leftmost atom in the goal is, of course, `s`. Then we search the program for clauses whose heads are `s`. There are two of them:

```
s :- np, vp.
s :- np, aux, vp.
```

According to our Prolog-style implementation of the Resolution Rule, we must try the first clause first. So we save a choice point recording the fact that we have not yet tried clause 2, and we remove the atom `s` from the goal and replace it with the body of the first clause, giving the new goal "`?- np, vp.`"

Now the cycle repeats itself. We apply the selection function to the goal, yielding `np`. We search the program for clauses headed by `np`, finding now

```
np :- det, n.
np :- pron.
np :- name.
```

We start with the first clause, as always. So we push a new choice point onto the stack, recording that clauses 2 and 3 remain untried, and we replace the selected atom in the goal with the body of the selected clause, yielding the new goal "`?- det, n, vp.`" We select `det` next, pick the first clause in its definition, and replace it in the goal with the body of that clause, yielding the new goal "`?- the, n, vp.`" We select `the`, pick the first and only clause in its definition and replace it in the goal with the body of its clause. The body of its clause is empty, so the result of this "replacement" is "`?- n, vp.`" It should be clear from this point how the rest of the proof proceeds. We summarise it as follows.

```
?- s.
?- np, vp.
?- det, n, vp.
?- the, n, vp.
?- n, vp.
?- entity, vp.
?- vp.
?- v.
?- exists.
?- .
```

In the last step we derive the empty clause, so the original goal succeeds.

Just as the program looks a lot like a context-free phrase-structure grammar, so this proof looks a lot like a derivation from that grammar. The only real difference is that we throw away the terminal categories `the`, `entity`, and `exists`, rather than glueing them together into a string. We will see later how to capitalize on this correspondence between definite clause proofs and phrase structure derivations to develop an extremely powerful grammar formalism, known as Definite Clause Grammars (DCGs). For the moment, however, we are just trying to understand how the Resolution Rule, as implemented in Prolog, works.

1.1 Backtracking

Our previous proof went very smoothly. What happens when things go wrong? Suppose, for example, that we changed the definition of `np` to the following.

```
np :- name.
np :- det, n.
np :- pron.
```

All we have done here is to reorder the clauses so that the one defining noun phrases as names (i.e., proper nouns) comes first. Observe that there are no clauses in the definition of `name`. So the derivation does not proceed very far. We give the first few steps of the derivation below, together with the choice points we create; choice points are represented by pairs of the form (<selected-atom>,<next-untried-clause>), and written so that the leftmost choice-point is always the most recent.

```
Goal:               Choice Point Stack:
?- s.
?- np, vp.          (s,2)
?- name, n, vp.     (np,2),(s,2)
```

Now, the search for clauses defining `name` fails, so we have to backtrack. Backtracking just means that we pop the most recent choice-point off of the top of the stack, and go back to the point at which we selected that atom. In this example, the last choice we made was which clause to resolve `np` against, and the next untried clause is clause number two. So we choose that clause, and replace `np` with `det, n`. Now the proof looks as follows:

```
Goal:               Choice Point Stack:
?- s.
?- np, vp.          (s,2)
?- name, n, vp.     (np,2),(s,2)
<fail; backtrack!>
?- np, vp.          (s,2)
?- det, n, vp.      (np,3),(s,2)
```

Now the proof continues as in the first example.

What if the definition of `np` had consisted only of the single clause "`np:- name.`"? There are no more clauses to try, so there is no reason to push a choice-point.

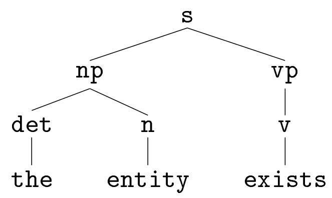
```
Goal:               Choice Point Stack:
?- s.
?- np, vp.          (s,2)
?- name, n, vp.     (s,2)
```

So, on backtracking, we would go all the way back to the `s`-rule, to try its second clause. Of course, in this grammar every sentence requires an NP, so there is ultimately no way past this faulty `np`-rule! In this case we will ultimately run out of choices and, at some point, the proof will fail and the choice point stack will be empty. At that point, Prolog finally admits failure, and prints out the rather uninformative answer "`no`".

1.2 Proof Trees and Search Trees

A tool which is often useful in the design and analysis of logic programs is the `proof tree`. Intuitively, a proof tree just records the successful proof of a goal in tree-like form. A little more formally, if we think of arbitrary Proplog clauses as grammar rules, then proof trees are just the trees generated by that "grammar". And when a Proplog program (a "propgram"?) really does represent a grammar then, well, the resemblence between proofs and derivations becomes that much more obvious. The proof tree in figure 1 can be read as follows. In order to

Figure 1 Proof tree for the goal `?-s.`



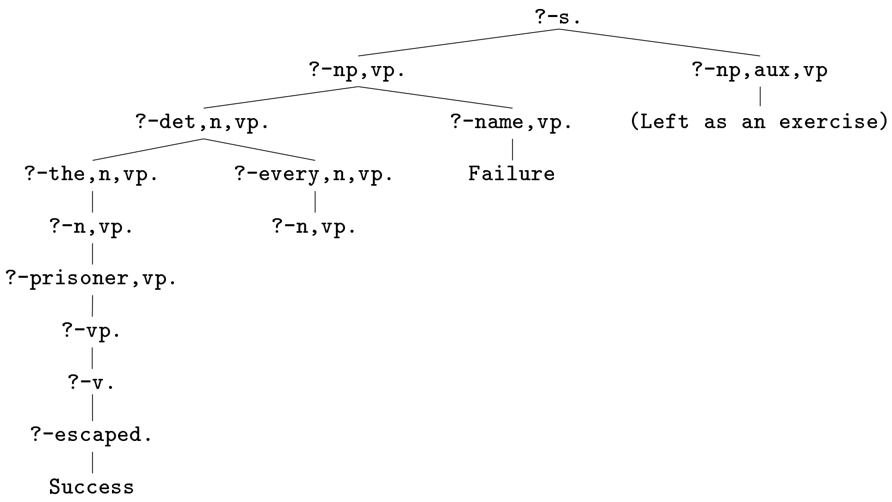
prove `s` we had to prove `np` and we had to prove `vp`. To prove `np` we had to prove `det` and `n`. To prove `det`, we had to prove `the`, and the proof of `the` was immediate from the program.

Proof trees can be very helpful when you are designing a new program. They help one to reason about a problem in terms of its subproblems: in order to prove my overall goal, what things do I need to accomplish first? The tree shape reminds one that, once one knows that `np` must be established, the problem of how to do so can be solved in a similar fashion, and in isolation from other problems, like how to prove `vp`.

A proof tree is read conjunctively: a parent node is established by establishing its first daughter, *and* its second daughter, and so forth. This focuses attention on what logical resources are *required* in order to prove a goal. However, it does not tell us what gets done with other resources in our program. That is, the proof tree focuses on a successful proof, and describes how that proof went, but it abstracts away from all the searching the Prolog engine might have had to do to find that proof. To give us a better idea of how much work the Prolog engine has to do we use *search trees*.

Search trees are read disjunctively. Nodes are labeled with goals, and each daughter node represents one way of attempting to prove the goal labeling its parent node. The shape of a search tree depends very heavily not only on the program but also on the interpreter that runs it. In particular, we can only draw a search tree once we know the selection function our interpreter uses. Since Prolog uses a leftmost-first selection function, we develop a search tree as follows. Given a node labeled with the goal `?-G1, ... ,Gn.` we search the program for clauses whose heads match `G1` and, for each such clause, we give our node a daughter labeled with the resolvent of the parent goal with that clause. That is, we replace `G1` with the body of the clause we have found.

Figure 2 Partial search tree for ?-s.



The tree in figure 2 shows most of the search tree for the goal `?-s.`, given the program in table 1. (For the most part the missing pieces are just copies of parts of the search tree given in the figure.) We have ordered daughter nodes from left to right so as to reflect the order of clauses in the program. Accordingly, Prolog can be seen as executing a depth-first, left-to-right search of the search tree, looking for leaf nodes labeled “Success” (i.e., looking for instances of the empty clause). If we reorder the clauses in the program so as to force backtracking, and reorder the subtrees of the search tree accordingly, we move the branch that ends in “Failure” to the left of the success branch. Prolog's backtracking can now be seen as just climbing back up the search tree to the most recent branching node and trying another daughter of that node until it runs out of daughters to try; then it must climb back up a little higher. In this way, Prolog will eventually search the entire tree if it has to, and if it finds no success-nodes, then it will report failure and halt. That is, it will halt as long as the search tree is finite. If the search tree is infinite, well then, things are a little more complicated, as we see in the next section.

1.3 Left Recursion and Infinite Proof Searches

There are two famous problems that can arise due to compromises that were made in the design of Prolog. One of them is as follows. I asserted of the program with the faulty `np`-rule that at some point Prolog will run out of choices. However, there are programs for which Prolog will never, ever run out of choices. Often this is not a problem: it just means there are an infinite number of distinct solutions, and if you want them all you must have infinite patience!

But sometimes this means that an otherwise logically correct program, given a true goal with a finite number of distinct answers will nonetheless fail to halt. Worse yet, it may fail to reach some of the correct answers before it goes off on its infinite search. Suppose we add the following clause to the `s`-rule.

```
s :- s, and, s.
```

We add "`and.`" to the lexicon as well. Now, given the faulty `np`-rule, we will ultimately have to use this new clause and expand the goal "`?- s.`" to "`?- s, and, s.`" Ultimately, the other two clauses for `s` will fail again, for lack of any proofs of `np`, and we will work our way down to clause 3 of `s` again, expanding the goal to "`?- s, and, s, and, s.` Every time we expand "`s`" to "`s, and, s`" we get a whole new fresh set of choices for how to expand the leftmost literal. Clearly this program will never halt: it will even fail to fail. In terms of search trees, the search tree for this goal now contains an infinite branch.

The problem is that we have a rule in our program which is *left recursive*: it calls itself first, before it does anything else. Left recursive programs will (almost) always eventually fall into an infinite proof-search. This is a severe problem, but it can be remedied. It is always possible to convert a left recursive program into an equivalent, non-left-recursive program, but the result is not always so readable. In our case we could redefine the grammar so that it contains a new non-terminal symbol, called `foo`, and redefine `s` as follows.

```
s :- foo.
s :- foo, and, s.

foo :- np, vp.
foo :- np, aux, vp.
```

What we have done is to package all of the unproblematic `s`-rules from the old grammar under the new symbol `foo`. Now the `s`-rule will only call itself after it has found a `foo`. In our example with the faulty `np`-rule, we are guaranteed to run out of ways to prove `foo`, and, failing to find a proof for `foo`, we will not look for further ways to prove `s`.

The solution works, but at some cost: in our case, we have introduced a symbol which has no part to play in our theory of grammar. It is meaningless; or rather, it only has a procedural meaning. As a result, our proofs are no longer identical to derivations in the corresponding grammar. Even worse is the following. Normally a sentence of the form "S and S and S and S," can have a number of interpretations, corresponding to the bracketings [S and [S and [S and S]]], [[S and S] and [S and S]], and so on. Our modified program will find only one way to prove that such a sentence is an `s`, corresponding to the first of these bracketings. These problems can be gotten around, but the example still serves to show that, even when efficiency is not a concern, we still have to know something about Prolog's procedural interpretation just to get our programs to halt. Efficiency may not be a concern at the moment, but *effectiveness* always is.

*/

Chapter 2

Lists—Recognizing and Generating Strings

With a propositional grammar like the previous one we can already make some interesting observations. For example, every possible proof of "`?- s.`" ultimately rests on a certain collection of facts. Put another way, we can think of Prolog goals as propositions which must ultimately be reduced to, or verified by, a particular collection of facts. If we collect all of the facts that are used to verify the goal "`?- s.`" in the order in which we uncovered them, then we see that they correspond in the obvious way to sentences of the language weakly generated by the grammar of table 1.

Unfortunately, concepts like "proof" and "facts" and so forth are *metalogical.* Our program has certain *logical* consequences, like that the proposition `s` is true, and the Prolog interpreter can help us find out what those logical consequences are. But the fact that `s` can be proved from the collection of facts `the`, `prisoner`, `escaped`, or that those facts will be discovered by the Prolog interpreter in the order just given, *are not* logical consequences of the program. Essentially, they are consequences of the fact that we are using the SLD resolution inference rule, with leftmost-first selection function, and not consequences of the fact that `np` and `vp` taken together imply `s`, for instance.

The *recognition problem*, however, is precisely this problem, of how to tell whether some particular string is in the language described by a grammar or, in our terms, which lexical facts were used in what order to prove an `s`-goal. Of course, Prolog would not be very useful to us if we could not even solve the recognition problem! We consider in this section how to do it.

The key is to move from a propositional logic to a predicate logic. Now we can use argument terms to record important features of a proof. This is a very powerful technique, which we will continue to use to solve harder problems, like the parsing problem. We introduce it here, however, while the problem is still fairly simple.

We will represent linguistic strings with the Prolog *list* data structure. The data structure definition is as follows.

```
*/
list([]). % The empty list is a list.
list([_Hd|Tl]):- list(Tl).
```

```
/*
```

The symbol [] is a constant, sometimes called "nil". It does not obey the rules for forming symbolic constant names, but it has the advantage of looking like an empty list ought to look. Likewise, the expression [Hd|Tl] does not look much like a legal Prolog structure, but it corresponds to the perfectly ordinary structure

```
     '.'
    /   \
  Hd     Tl
```

The '.' functor is sometimes referred to as "cons". ("Nil" and "cons" are terms borrowed from LISP.) Note that the expression [a|b] is perfectly legal Prolog, but it is not a list. That is, the goal "?- list([a|b])." will fail.

One thing we will want to do very often with lists is to concatenate two lists together to form a third.

```
*/
append([],Bs,Bs).
append([X|As],Bs,[X|Cs]):- append(As,Bs,Cs).
/*
```

Now we take our grammar from table 1 and add a single argument to every nonterminal symbol, together with appropriate calls to append/3. Note that this allows us the slight optimization of getting rid of our "lexical facts"; these are now just rolled up into the definitions of the preterminal categories.

The grammar in table 2 can be used for recognizing strings given by the user so that, for example, the goal

```
?- s([every,prisoner,has,escaped]).
```

succeeds. There is a slight problem with using it to generate strings, however.

```
| ?- s(L).

L = [the,prisoner,escaped] ? ;

L = [every,prisoner,escaped] ? ;

L = [the,old,prisoner,escaped] ? ;

L = [every,old,prisoner,escaped] ? ;

L = [the,old,old,prisoner,escaped] ? ;

L = [every,old,old,prisoner,escaped] ? ;

L = [the,old,old,old,prisoner,escaped] ? ;

L = [every,old,old,old,prisoner,escaped] ?

yes
```

Table 2 A Grammar for Recognition and Generation of Lists.

```
*/
s(String):- append(Left,Right,String), np(Left), vp(Right).
s(String):-
        append(Left,Rest,String), append(Mid,Right,Rest),
        np(Left), aux(Mid), vp(Right).

np(String):- append(Left,Right,String), det(Left), n1(Right).
%np(String):- name(String).

n1(String):- n(String).
n1(String):- append(Left,Right,String), adj(Left), n1(Right).

det([the]).     det([every]).

n([prisoner]).

adj([old]).

% No clauses for name.

aux([has]).

vp(String) :- v(String).

v([escaped]).
/*
```

Observe that there is no finitely reachable point in time at which this sequence will include sentences like `[the,prisoner,has,escaped]`. We can fix this easily though. We observe the following behavior of the data type definition `list/1`.

```
| ?- list(L).

L = [] ? ;

L = [_A] ? ;

L = [_A,_B] ? ;

L = [_A,_B,_C] ? ;

L = [_A,_B,_C,_D] ?

yes
```

That is, this predicate will enumerate all of the lists in order of increasing length. Therefore the goal

```
?- list(L), s(L).
```

will enumerate all of the sentences of a given length before backtracking all the way to `list/1` and considering lists of the next higher length. So, to sum up, we have the following.

```
*/
recognize(String):- s(String).

generate(String):- list(String), s(String).
/*
```

One final note of caution, which is really beside the points we are considering here: this scheme works smoothly as long as the grammar generates an infinite set of sentences. This is because the search tree for the generation problem has an infinite branch, regardless of how many sentences the grammar generates. Given the definition here, this is so because `list(String)` has infinitely many solutions, but even without this "watchdog predicate", the calls to `append/3` contained in the grammar, if called with all arguments unbound, will also act to generate ever larger and larger lists. Once we run out of sentences we will encounter only failure nodes for these longer "sentence templates", and there will be infinitely many of them. So after generating all the sentences in the language, Prolog will go off on an infinite search for more of them, never to return (at least not under its own control). This can be fixed by putting the calls to `append/3` at the end of the clause bodies in which they appear. But that scheme in turn cannot be used for an infinite language, since a call to a goal like "`?- s([]).`" or some other ungrammatical string will lead the grammar to generate all of its strings, of which there are infinitely many, failing every time to match the input. All of these problems go away if we use *difference lists* instead of ordinary Prolog lists, but that is a topic for another day. */

Chapter 3

Trees—The Parsing Problem

3.1 The Context-Free Case

We have already observed that the structure of a proof tree for our propositional grammar was isomorphic to the structure of the corresponding derivation tree. So, just as we could recognize strings by keeping track of which axioms were used in a proof, we can parse strings by keeping track of which rules were used in the proof. We used lists to implement strings. To implement trees we will use the following data type, using the operator symbol '/' to represent immediate dominance and using lists to hold the subtrees of a given node.

```
*/
tree(_/[]). % Leaf nodes dominate no subtrees.
tree(_/Subs):- forest(Subs).

forest([A]):- tree(A).
forest([A,B|Bs]):- tree(A), forest([B|Bs]).
/*
```

Note that forests of subtrees are required to be nonempty, so as to distinguish interior nodes from terminals. Note also that our notation for trees is very reminiscent of that for lists, which reflects the facts that trees are generalizations of lists in a rigorously formal sense. So, for example, the tree term for a tree with only a single branch labeled with the elements `a, b, c, d` would look as follows.

```
a/[b/[c/[d/[]]]]
```

Compare

```
[a|[b|[c|[d|[]]]]].
```

Unfortunately, there is no built in syntactic sugar to compare with the list notation `[a,b,c,d]`.

The grammar in table 3 adds tree terms to the grammar of table 2. Just as with the simpler recognition grammar, these extra terms bear an absolutely

Table 3 A Grammar for Recognition and Generation of Lists.

```
*/
s(s/[T_NP,T_VP],String):- append(Left,Right,String),
        np(T_NP,Left), vp(T_VP,Right).
s(s/[T_NP,T_Aux,T_VP],String):-
        append(Left,Rest,String), append(Mid,Right,Rest),
        np(T_NP,Left), aux(T_Aux,Mid), vp(T_VP,Right).

np(np/[T_Det,T_N1],String):- append(Left,Right,String),
        det(T_Det,Left), n1(T_N1,Right).

n1(n1/[T_N],String):- n(T_N,String).
n1(n1/[T_Adj,T_N1],String):- append(Left,Right,String),
         adj(T_Adj,Left), n1(T_N1,Right).

det(det/[the/[]],[the]).     det(det/[every/[]],[every]).

n(n/[prisoner/[]],[prisoner]).

adj(adj/[old/[]],[old]).

% No clauses for name.

aux(aux/[has/[]],[has]).

vp(vp/[T_V],String) :- v(T_V,String).

v(v/[escaped/[]],[escaped]).
/*
```

mechanical relation to the underlying grammar rules. In particular, the parse tree constructed by any given non-terminal expansion rule is a function of the rule and the parse trees for the grammatical category predicates that it calls. Thus it keeps track of the essential features of the corresponding proof tree. The only difference is that the proof tree must contain the many different proofs of calls to `append/3`, which we systematically ignore in building the parse tree.

As a generator, the program of table 3 gives us exactly the same problem, and has exactly the same solution, as the program of table 2. That is, we simply use calls to `list/1` as a source of external control over the behavior of the grammar predicates like `s/2`.

3.2 And Beyond ...

As long as we restrict our tree terms to recording the shape of the particular rule in which they are constructed (that is, they contain no information which is not local to a single clause), and as long as we make no appeal to extra-grammatical predicates other than `append/3`, then our grammars are pretty much guaranteed to give us just the context free sets of strings and their derivation trees. But Prolog is a Turing-capable programming language; any computable function can be implemented as a Prolog program. So where is this extra power coming from? Mainly from the ability to carry around arbitrary amounts of information from one node in a proof tree to another arbitrarily distant node.

*/