Language recognition and generation in Prolog[*]

Graeme Taylor

January 29, 2004

Prolog, as the writeups in that node indicate, employs a very different approach to that of imperative programming languages. Whilst this can lead to a lot of frustration and confusion when learning to work with it, there are obviously fields in which it excels compared to, say, C. One such area is the issue of language recognition and generation. Prolog allows for very elegant solutions to some of these problems, and furthermore they serve as an ideal starting point for understanding the quirks of the language. As Irexe[1] observes, the goal of Prolog "programming" is more a case of describing the problem- to *"specify exactly what conditions our solution has to meet in order to be satisfactory. The listener does the rest."* However, whilst being almost structureless, wildly different results in language recognition (from correct solutions to a complete failure) can be achieved simply by altering the order in which our conditions are listed. Thus the weaknesses of Prolog's goal-matching approach to the semantic tableau system can be readily illustrated.

In general, given a formal grammar $G = (\Sigma^*, P, I)$ and some W in Σ^*, deciding whether W is in $L(G)$ is recursively undecidable. An important case in which4 it is decidable is context-free languages. Thus we will look at some examples of these, see how to attack them with Prolog, and see where it falls down.
For those who haven't had to endure logic/semantics courses, a quick translation of terminology: G is a grammar as it consists of an initial string or word I, and a list of rules P which allow us to change parts of the string into others. Σ^* is the set of all finite length combinations of the symbols of finite alphabet Σ - obviously any "word" generated from those symbols will be in Σ^* but it is rarely the case that all such arrangements of symbols are possible to generate. So we consider the set $L(G)$, the language generated by grammar G, which is any terminal string- namely, one which can be altered no more by any of the rewriting rules P. So our problem is, given a legitimate string of symbols (which could spell an english word, denote the positions of chess pieces, indicate the state of a rubiks cube or so on) and an initial string of symbols, can we manipulate the initial string into the given one by applying the rules (be they rules of spelling, chess moves, rotations of

[*]First appeared on Everything2, at *http://www.everything2.com/index.pl?node_id=1516430*
[1]An Everything2 user, *http://www.everything2.com/index.pl?node=Irexe*

a cube). As this problem is hard, we restrict ourselves to rewriting rules where only a single symbol is manipulated at a time (not combinations of symbols).

Example 1: a term language

Suppose we have a term language, that is, a string rewriting system whose symbols can be interpreted as functions in some domain. Then the term language is the set of all the possible applications of the function symbols to some constants, with an arbitrary level of nesting. For instance, given a function f of arity 2, and g of arity 1, and constant terms a and b, some valid terms would be a,b $f(a,b)$, $f(g(f(a,a)),b)$ and so on. As an example, a and b could be the natural numbers 1 and 2, f could denote multiplication and g could denote addition of 1, with the domain being the natural numbers. Then our term language consists of a collection of natural numbers, e.g. $f(1,g(1))$ is $f(1,2)=2$; $g(f(1,2))$ is $g(2)$ is 3 and so on- it should be clear that however deeply nest the functions f and g, when evaluated in this interpretation we get sensible output- another natural number. What wouldn't be sensible is an expression such as $g(f)$, $f(1,,2)$, $2(f)$ or so on- these expressions are in some way malformed. So we want to be able to recognise when our input is a correct combination of the symbols.

More formally, we are testing candidates for membership of $L(G)$ where G is (Σ^*, P, S) and P allows S to be re-written as any of $f(S,S)$; $g(S)$; a ; b . Note that any member of $L(G)$ won't actually have an S in it, as that wouldn't be terminal. So we want to define a Prolog function **inlg(X)** which is true if X is in $L(G)$, and false otherwise. We use the following-

```
inlg(a).
inlg(b).
inlg(g(S)):- inlg(S).
inlg(f(S,T)):- inlg(S), inlg(T).
```

These all appear straightforward and valid- we are working recursively and have two base cases (the facts on line 1 and 2) at our disposal. For anything more complicated than a constant a or b, we look to see if it's f of something or g of something- if not, we'll fail, and if so we check whether the somethings are valid.
But the order is crucial. Lets try some sample tests with this configuration.

- First, a test of a known true member- **inlg(f(g(a),b)).**. We get the output yes.
- Next, something that has the right shape but an illegal constant- **inlg(g(f(c,d))).**. We get the output No.
- How about a completely wrong function symbol? **Try inlg(h(x,y,z)).**. This yields No.

- Now we experiment with the ability of Prolog to generate members of $L(G)$ given it's appreciation of the rules. A simple query for variables X in lg is given by **inlg(X).**. Our first output is $X = a$. Repeating the query (use ;) we are told that $X = b$; $X = g(a)$; $X = g(b)$; $X = g(g(a))$; $X = g(g(b))$; ... hopefully you can see the issue here, that Prolog can recognise correct terms which use the function f but cannot generate them itself with this formulation of the rules.

So we already have some idea of how the order does matter. Had we interchanged the third and fourth rows of our program, then we could cheerfully generate terms in f all day long without getting a term in g- whilst retaining the ability to identify terms featuring g if they are supplied. What if we had put the more complex rules before the very simple facts about a and b being members of our language? For this we shall move to a second example, as this was my first Prolog programming exercise and it failed for that very reason!

Example 2: statement form

If we have some list of predicates, we could consider the logical combinations of them. These would be: $(P \vee P)$, $(P \wedge P)$, $(P \Rightarrow P)$, $(P \Leftrightarrow P)$, $(\neg P)$; where P is a placeholder for something in statement form (namely a predicate or logical combination of predicates- once again our definition of correct form is recursive in nature. It is important to realise that we aren't after the same entry for each P, just something from the collection of valid replacements of P). Now, we could proceed as before, defining functions of appropriate arity- say use **and(P,Q)** to denote $(P \wedge Q)$, or **neg(P)** to indicate $(\neg P)$. But this seems ugly, as the natural reading of them is destroyed: $not(and(P, OR(R, S)))$ is already rather difficult to decipher. Instead, we can use Prolog's list structure to set up a cleaner isomorphism between strings possibly in statement form and lists-

- If we have $p \Rightarrow q$, use the list [p, imp, q] in Prolog.
- Similarly define $\vee$, $\wedge$ and $\Leftrightarrow$ by [p, or, q] , [p, and, q], [p, iff, q] respectively.
- For negation use [neg, p] to indicate $\neg p$.
- Predicates can be written as they are e.g. p.

Now the question of whether $p \Leftrightarrow (q \vee r)$ is in statement form becomes a test of whether [p, iff, [q, or, r]] is in statement form (note the list within a list.). So we'll try the following set of rules to test $s(X)$ - that is, if X (a list) is in statement form:

```
s([A, or, B]):- s(A), s(B).
s([A, and, B]):- s(A), s(B).
s([A, imp, B]):- s(A), s(B).
s([A, iff, B]):- s(A), s(B).
```

```
s([neg, A]):- s(A).
s(X):- member(X,[p,q,r]).
```

The first five lines represent the translation described above between prolog lists and statements; the last line gives us an intial stock of predicates (the constant terms p,q,r). The first four lines are very similar and we could clean things up more by introducing a concept of an operator and then checking whether the middle list entry is an operator, but this will suffice.

Or will it? If we try a query, such as s([p, imp, q]). we get the correct answer, yes. But if we try for a general query of things in statement form, $s(X)$:

```
ERROR: Out of local stack
```

In other words the search is non-terminal. Why is this? The first example's refusal to generate terms with the function f should have given a clue to the approach Prolog employs when looking to solve these general queries- it utilises the first match it can to solve them, even if other options are possible. So when we ask for an X in statement form, it matches at the first line and concludes that X would be in statement form if X was a list consisting of [A, or, B] where A and B where in statement form. This seems like progress- all it need do now is find some A and B in statement form. So it starts looking for an A in statement form. The first line tells it that A would be in statement form if A was a list [C, or, D]- so it sets off to find a C in statement form: and so on forever- or more accurately, until it exhausts the stack.

Yet all the lines we used were correct, and in fact the only work we need to do is shift the last line to the top of the list. Then our search will terminate firstly with the predicates p,q,r ; *then* we get (p or p), (p or q), (p or r), (p or (p or p)) and so on... still no progress beyond lots of or conditions, but we prevent non-termination.

Thus we have seen that correct language recognisers can be constructed which purely by altering their order allow for wide variation in the ability to generate such terms. Furthermore once more than a single line is required to define recursively our desired terms, we lose the ability to generate all but the permutations of the constants and nestings of that first rule's symbol.

A final example doesn't add much to this discussion but illustrates how to work with strings of arbitrary length rather than arbitrary nesting of recognised functions/connectives.

Example 3: arbitrary strings of a's and b's

Given an alphabet $\Sigma = S, a, b$ and a desire to generate some strings of a's and b's, we could use the formal grammar $G = (\Sigma^*, P, S)$ where P allows us to rewrite S as $aSbS$, bSS, a or b.

Now we need a way to notate a string in Prolog. Again, we will use lists. So we want a function f which takes strings and churns out Prolog lists. Again this will be recursive-

- Let $h(emptystring) = []$ (the empty list)
- Then, $h(aX) = [a|h(X)]$ - so a string consisting of a followed by some other symbols is the list with head a and tail comprised of the Prolog representation of X.
- Similarly $h(bX) = [b|h(X)]$.

So the question of whether a string $pqrs$ is in $L(G)$ becomes the question of whether the list [p,q,r,s] is. We can use the following Prolog rules:

```
inlg([a]).
inlg([b]).
inlg([b|X]) :- append(S,T,X), inlg(S), inlg(T).
inlg([a|X]) :- append(S,[b|T],X), inlg(S), inlg(T).
```

This formulation corresponds to the string-rewriting rules adopted earlier. Bear in mind that $append(S,T,X)$ isn't an operation, it's a predicate which is true if X consists of T concatenated to S.