AN AI APPROACH TO TEACHING LINGUISTICS*

Nicholas Sobin

University of Arkansas, Little Rock

0. Introduction

Levin (1983) deals with the problem of teaching linguistics to non-majors. Her major suggestion is that this can be done more successfully by de-emphasizing formalism and by focusing more on areas of greater popular interest, such as sociolinguistics and language variation. Levin's suggestions comprise one very reasonable approach to the problem. What I wish to explore here is an alternative approach, one which uses computation and which actually places greater emphasis on formalism, but in a way which is of interest and benefit to beginning students whether or not they are linguistics majors. Using the artificial intelligence programming language Prolog, one can model competing hypotheses about particular linguistic phenomena and demonstrate the consequences (and hence the desirability) of each hypothesis. AI programming in Prolog gives students a grammar machine to play with. Working with these programs aids in teaching some of the particulars of grammatical analysis, helps illustrate the scientific method of inquiry (formulating alternative hypotheses and assessing their predictions/consequences), and initiates students into computing and programming skills whose benefits far out-extend linguistic applications.

Here and in what follows, I assume that the reader has at least a basic familiarity with generative grammar and has to teach it. I will deal with aspects of expressing generative grammar and some elementary problems in linguistics in Prolog, and with using these Prolog grammar descriptions in teaching linguistics to students with no background in linguistics, formal logic, or computers.

Sections 1 through 3 discuss aspects of basic linguistic description in Prolog. Sections 4 and 5 each present a problem in linguistic description as it might be treated in Prolog. While each problem is discussed as though it were being presented to a beginning student, the discussion includes some details of the Prolog treatment which are relevant to the instructor's fuller understanding and which need not (but may) be used in the presentation of the problem to beginning students (where one wants to maintain a focus on the linguistic aspects of the problem). Section 6 discusses this less formal use of these Prolog-rendered problems. Section 7 concludes.

As a final note here, there are a number of 'dialects' of Prolog. The one I am using here is the version of the 'Edinburg' dialect produced by Quintus and available for the VAX/VMS System (Quintus Prolog 2.0). Translation to other versions of Prolog (e.g. those used in personal computers) is straightforward.

1. A C-F grammar in Prolog

Prolog is one of the major AI programming languages used in investigating the computer

---

* A number of people have made various significant contributions to this work. I owe them a great deal of thanks. They include AnneMarie Black, Steve Anderson, Andrea Herrmann, Marilyn Parins, John Talburt, and the students of English 3312. Any errors are my own.
implementation/description of natural language. Of particular interest is its generative facility. Prolog allows one to program generative rules in very nearly the same format as is common in generative linguistics. Thus a PS rule such as (1) has the Prolog form (2):

(1) \[ S \rightarrow NP \ VP \]
(2) \[ s \rightarrow np, \ vp. \]

In (2), the terms are lower case since they represent constant terms (rather than variables). The comma may be read 'and then' or 'and next'. Every Prolog statement ends with a period.

Following this format, (3) is a Prolog grammar for some simple sentences:

(3) \[ s \rightarrow np, \ vp. \]
    \[ np \rightarrow det, \ noun. \]
    \[ verb \rightarrow [loves]; [sees]. \]
    \[ det \rightarrow [a] \]
    \[ noun \rightarrow [girl]; [boy]. \]

One additional convention shown in (3) is the semicolon, which functions as a disjunction. What follows it is an alternative content for the category involved. Thus, either 'loves' or 'sees' is a verb. Also, lexical (terminal) material appears in square brackets.

Grammar (3) is a simple context-free grammar of transitive sentences. Context-sensitive grammars are also possible and will be illustrated below.

2. Prolog translations

When one creates a fairly traditional PS grammar such as (3) and loads it into Prolog, it is automatically translated into a different form. This translation is significant in that it dictates the form of query used in 'running' a program. A rule such as (2) (whose intuitive content is that a sentence consists of a noun phrase followed by a verb phrase) is translated into the form (4):

(4) \[ S (A,B) \ :- \]
    \[ np (A,C), \]
    \[ vp (C,B). \]

The symbols ':' and ',' are 'if and 'and then' or 'and next', respectively. The capital letters (A, B, and C) are variables which refer to strings or sequences or lists of words which make up a phrase or sentence. The intuitive content of (4) is the same as that of (2), but (4) overtly refers to lists of words: the list of words A is a sentence if the first part of that list (A minus C) is analyzable as a noun phrase and the rest of the list (C) is analyzable as a verb phrase. In what follows next, I will spell out some of the details of how rules like (4) treat these lists of words.
By convention, the last variable (B) in an expression like (4) is an empty list, as in (5b). A list of words which grammar (3) might generate as a sentence is (5a):

(5)  

(a) [a,girl,loves,a,boy]  
(b) [ ]

One way of isolating a part of a list such as the sublist [a,girl] is by subtracting the word sequence (5c) from (5a):

(5)  
c. [loves,a,boy]

This is what Prolog does in rules like (4). In (4), A, B, etc. are heavily interpreted. In particular, in each expression of the form (6),

(6)  
x(Y, Z)

the variable Y represents a list of words. The variable Z represents a sublist of words which makes up the end of list Y. x is some syntactic category. Expression (6) says that removing the sublist Z of list Y from the right end of list Y should leave a list of words which are the contents of the syntactic category x. (The device of subtracting a sublist to obtain another list is called a difference list.)

Now, to read rule (4) more explicitly, recall that the last variable of the first expression of form (6) is by convention the empty list ([ ]). Then rule (4) may be read as follows: the list A with sublist B removed from its right end (but B=[ ]) is a sentence ('s') if (':-') the list A with a sublist C removed from its right end is an np, and list C with the sublist B removed from its right end (B=[ ]) is a vp. For example, if the list [a,girl,loves,a,boy] is list A and [ ] is list B, then [a,girl,loves,a,boy] (A-B) is a sentence if [a,girl,a,boy] (A-C) is a det and [loves,a,boy] (C-B, which is simply C) is a vp. (Although the list A might be divided differently, this is the only division that will succeed on further analysis by grammar (3).)

The next problem is how to decide whether [a,girl] is an np, and [loves,a,boy] is a vp. This can be done by inspecting the rest of the translation of grammar (3). Its complete translation is given as grammar (7):

(7)  
s(A,B) : -  
np(A,C),  
vp(C,B).

np(A,B) : -  
det(A,C),  
noun(C,B).

vp(A,B) : -  
verb(A,C),  
np(C,B).

verb(A,B) : -  
( 'C' (0,loves,B)  
; 'C' (A,sees,B) ) .

det(A,B) : -  
'C' (A,a,B) .
noun(A,B) : -  
( 'C' (A,girl,B)  
; 'C' (A,boy,B) ) .

The np rule of (7) says that a list A (e.g. [a,girl]) with the sublist B removed from its right end (again, by convention, B=[ ]) is an np if the list A with a sublist C removed from its right end (C= [girl]); A-C= [a]) is a det and list C with list B removed from its right end (B= [ ]) is a noun. We need not go into further detail here, but informal inspection of the vocabulary will show that [a] is a det and [girl] is a noun. Similarly, the vp role will break down [loves,a,boy] into a verb
[loves] and an np [a,boy], and the np rule will analyze [a,boy] into a det [a] and a noun [boy]. The analysis of the original word list [a,girl,loves,a,boy] is complete when [loves], [a], and [boy] are analyzed by the last rules as verb, det, and noun, respectively.

3. Queries

One 'runs' a Prolog program by asking it questions about the facts and rules which it 'knows'. A good part of the motivation for the preceding discussion of the Prolog translation of generative rules is to explain the form of such queries. Queries are made on the translated rule form, which Prolog utilizes. As discussed earlier, a generative rule like (8) has the translated form (9):

\[
\text{(8) np } \rightarrow \text{ det, noun.}
\]

\[
\text{(9) np(A,B) :-
\quad \text{det(A,C);}
\quad \text{noun(C,B).}}
\]

If the rule (8) (or grammar (3)) is loaded into Prolog, it is automatically translated into form rule (9) (or grammar (7)). At the Prolog prompt 'I-'), one may ask what np's the grammar will produce by typing in query (10):

\[
\text{(10) I?- np(NP,[ ] ).}
\]

Capital letters in Prolog represent variables. In response to a query containing a variable such as NP in (10), Prolog will give a possible value for/instantiate the variable. Given what has been said in the preceding section, relative to grammar (7), NP in (10) corresponds to a list of words which grammar (7) can analyze as an np since NP of (10) corresponds to A of (9). Thus, Prolog will respond to (10) as in (11):

\[
\text{(11) NP = [a,girl]}
\]

One may simply type Return to get back to the prompt, which will allow a different query, or one may ask for an alternative solution/answer to the present query. Typing a ';' and Return elicits the latter. Such a move here will elicit (12):

\[
\text{(12) NP = [a,boy]}
\]

Doing this again will elicit 'no'. 'No' here simply means that from the facts and rules that Prolog has access to (grammar(7)) there are no more possible np's.

As a final note here, NP in (10) is simply a mnemonic notation for the variable. The variable may be any capital letter (e.g. X) or capital-initial letter-number sequence. What is important is the position of the variable in the query because it is in this second-to-last (penultimate) position that lists of words analyzable as np (or any of the other syntactic categories) are stored/analyzed. If one asked (13), which contains two variables, Prolog would respond as in (14):

\[
\text{(13) I?- np (NP,X).}
\]

\[
\text{(14) NP = [a,girl]}
\]

\[
\quad X = [ ]
\]
Response (14) is giving values for each variable, including the information that the last variable in rule (9) represents the empty list.

Response (14) is also indicative of other possible queries. Notice that (10), in contrast to (14), gives the value of, rather than asking a value for, the last variable (B) of rule (9). In Prolog, it is possible to ask about selected pieces of information, while specifying or ignoring other pieces. Thus, (15a) is a query about the possible words which may follow [a] in an np, and (16a) is a query about the first possible word(s) of a np, ignoring the words that may follow. (Underscores are 'anonymous' variables, for which Prolog will not return a value.) (15-16b) are Prolog's initial responses.

(15) a. \?- np([a,N],[ ]).
    b. N = [girl]

(16) a. \?- VP([V,_,_,],[ ]).  
    b. V = [loves]

If one asks (17a), then one is simply asking whether [a,girl] is a possible np, to which Prolog will simply respond as in (17b):

(17) a. \?- np([a,girl],[ ]).  
    b. yes

Query (18a) will elicit the opposite response, (18b), since the list [girl,a] is not computable from grammar (7).

(18) a. \?- np([girl,a],[ ]).  
    b. no

Thus, Prolog knows not only a vocabulary, but a syntax over that vocabulary. Any of the other syntactic categories may be queried in the same fashion.

The preceding represents an introduction to the grammar facility of Prolog. Much terminology and many alternative analytic possibilities have been ignored. These can be gotten from other sources. What follows are some sample programs which may be useful in teaching or illustrating some important facets of basic linguistic analysis.

4. Problem 1: A morphology with rules of word formation vs. a word list

'Rule' is a central notion in modern linguistic analysis. Rules appear to be involved in many or all of the components of the grammar of a language, including its lexicon. The following problem illustrates the benefits of treating the lexicon as a list of morphemes which undergo rules of word formation, as opposed to treating the lexicon as a simple list of unanalyzed words.

English speakers know that words as in (19) are all words of English.

(19) jumps, loving, walked, walks

There are various ways to model this knowledge of words. One way is simply to list these words, as in Prolog grammar (20a):
An alternative model of the English speaker's knowledge of the words in (19) is grammar (20b). Grammar (20b) does not list any of these words directly. Rather, it analyzes them as consisting of v(erb)stem and a v(erb)suf(fix), and then it lists these two sets of items:

\[
(20) \begin{align*}
\text{a. verb} & \rightarrow [\text{jumps}]; \\
& [\text{loving}]; \\
& [\text{walked}]; \\
& [\text{walks}].
\end{align*}
\]

Now, if we ask either grammar whether each of the words in (19) are possible, by posing queries such as (21a) for grammar (20a) or (21b) for grammar (20b), each grammar will respond 'yes' for each word in (19).

\[
(21) \begin{align*}
\text{a. I?- verb(}[\text{jumps}], [\text{ }] ). \\
\text{b. I?- verb(}[\text{jump}, -s], [\text{ }] ).
\end{align*}
\]

Thus, both grammars are equally effective in characterizing the English speaker's knowledge of the fact that the items in (19) are English words. If the speaker possessed as part of her/his knowledge of English either (20a) or (20b), then she/he would 'know' the items in (19).

However, additional considerations support grammar (20b) as the Superior model of the English speaker's knowledge of the words in (19). First, consider the fact that any mature speaker who knows the words in (19) will also know the words in (22).

\[
(22) \text{jumped, jumping, loves, walking}
\]

Grammar (20a) fails to predict that the words of (22) are possible. If the English speaker's knowledge of (19) were due to knowing grammar (20a), then that speaker would not know the words in (22). Grammar (20a) fails to relate those two sets of words. This can be verified by querying grammar (20a) with query (23a), and then by typing Return after each Prolog response until Prolog responds 'no'. The latter is illustrated in (23b).

\[
(23) \begin{align*}
\text{a. I?- verb(v, [ ] ).} \\
\text{b. V = [jumps];} \\
& V = [loving]; \\
& V = [walked]; \\
& V = [walks];
\end{align*}
\]
In contrast, grammar (20b) correctly predicts that the words in (22) are possible. This can be verified by querying grammar (20b) with query (23a), and continuing the query with ';' as above. The result of doing this is (23c):

\[
(23) \begin{align*}
V &= [\text{jump}, -\text{s}] ; \\
V &= [\text{jump}, -\text{ed}] \\
V &= [\text{jump}, -\text{ing}] \\
V &= [\text{love}, -\text{s}] \\
V &= [\text{love}, -\text{ed}] \; ; \\
V &= [\text{love}, -\text{ing}] ; \\
V &= [\text{walk}, -\text{s}] \\
V &= [\text{walk}, -\text{ed}] \\
V &= [\text{walk}, -\text{ing}] \\
V &= \text{no}
\end{align*}
\]

Thus, if the English speaker 'knew' grammar (20b), not only would she/he know the words of (19), but the speaker would know automatically and without further specification that the words in (22) are also possible. Therefore, since grammar (20b) makes better predictions about the additional words known by English speakers who know those in (19), grammar (20b) is the better hypothesis about/model of this part of the speaker's knowledge of English grammar.

At this point, one might observe that it is easy to modify grammar (20a) so that it will also account for the words in (22) simply by adding these words directly to (20a). However, this would not really solve the problem. These two grammars (20a and 20b) make very different claims about how it is that speakers of English know the words that they know. Since (20a) is simply a word list, it asserts that speakers know/learn each word or word form one at a time. With (20a), knowledge of one particular word or word form has no implication for whether the speaker knows any other word or word form. In contrast to this, grammar (20b) characterizes knowledge of words as the result of building words with what we will call word formation rules (WFRs; e.g. 'verb -- vstem, vsuf.'), so that a word like jumped is not a single learned item, but the result of combining smaller items.

Now, that (20b) is preferable can be seen by considering how English speakers are able to treat new words. If an English speaker is simply given a new stem form, such as blonk, and is told that blonk is a verb, without being shown any other forms of the word, that speaker will know 'automatically' that this new word has all the possible forms in (24):

\[
(24) \text{blonks, blonked, blonking}
\]

Grammar (20b) can account for this phenomenon, and grammar (20a) cannot. If we augment (20b) by simply adding [blonk] to the list of vstem's, query (23a) will elicit, in addition to (23c), the forms of blonk in (24). The speaker who 'knows' grammar (20b) 'automatically' knows the various forms of blonk, though she/he has only been 'told' the stem form. By contrast, if we make the same addition of [blonk] to grammar (20a), query (23a) will yield only [blonk], and will yield none of its related forms. Grammar (20a) fails to mirror the speaker's 'automatic' knowledge of the other forms of a new verb. The reader is invited to verify this result in Prolog.

Critical to the success of grammar (20b) is its analysis of words into smaller parts, and its use of a WFR. The greater success of grammar (20b) in dealing with the above facts/observations suggests that it is a better model of word storage than (20a) with its simple listing of words. Thus we conclude that speakers of natural languages (English) learn/store words not as simple lists of
whole words but break them down into their meaningful smaller parts (morphemes), and use WFRs to 'create' words from these smaller parts.

5. Subcategorization frames

Consider a simple grammar of transitive and intransitive sentences such as (25):

\[
(25) \quad s \rightarrow np, vp.
\]

\[
vp \rightarrow \text{verb}, np;
\text{verb}.
\]

\[
np \rightarrow \text{noun}
\]

\[
noun \rightarrow [\text{mary}]; [\text{max}].
\]

\[
\text{verb} \rightarrow [\text{adores}]; [\text{arrived}].
\]

In (25), the vp rule says that a vp consists of either a verb and then an np or just a verb. Querying this grammar with (26) and ';' will reveal a difficulty.

\[
(26) \quad !?- s(Sent, []).
\]

While grammar (25) will produce the grammatical sentences (27), it will also produce the ungrammatical sentences in (28). Thus, it is over-generating; it is mispredicting that some actually ungrammatical sentences are grammatical.

\[
(27) \quad \text{Sent} = [\text{mary, adored, max}]
\]

\[
\text{Sent} = [\text{mary, arrived}]
\]

\[
(28) \quad \text{Sent} = [\text{mary, adores}]
\]

\[
\text{Sent} = [\text{mary, arrived, max}]
\]

One solution to the problem is to break verb information into separate syntactic categories, transitive and intransitive, as in grammar (29).

\[
(29) \quad s \rightarrow np, vp.
\]

\[
vp \rightarrow \text{vtr}, np;
\text{vintr}.
\]

\[
np \rightarrow \text{noun}
\]

\[
noun \rightarrow [\text{mary}]; [\text{max}].
\]

\[
\text{vtr} \rightarrow [\text{adores}].
\]

\[
\text{vintr} \rightarrow [\text{arrived}].
\]

Grammar (29) solves the initial problem. Using query (26) and ';' with grammar (29) will show that it successfully generates only grammatical sentences as in (27) and none of the ungrammatical sentences of (28).
However, this sort of solution engenders another significant problem. Both grammars (25) and (29) have treated inflected verbs as though they were atomic units—units not analyzable into smaller parts. This, of course, is false. All verbs, including transitives and intransitives, show the same fundamental breakdown into a verb stem and a verb suffix. (Recall grammar (20b) above and the accompanying discussion). If we were to try to further develop the description in (29) along these lines (that is, if we were to try to integrate the analyses in (29) and (20b)), the resultant grammar would be problematic in that it would contain what is usually labeled as a missed generalization, a situation where what is obviously one and the same rule or process is specified in a grammar more than once, as though the two (plus) specifications had nothing to do with one another, or were not related, or were not a single rule or process but separate, ones. A description without missed generalizations, that is, one which describes each general rule, and so forth, in a single statement over the whole grammar, is preferable to a description of the same data containing missed generalizations. We always try to capture the notion that some rule or process is a single one by stating it once and only once in a grammar. To put this all differently, when we encounter a set of data/phenomena which we are convinced is due to a single rule or process, we must be able to give a single, unified descriptive statement of that set, or we will have failed to describe or explain its unity.

In what follows next, we will consider in more detail the potential missed generalization problem with grammar (29). Then we will consider the solution from a linguist-theoretic standpoint, and deal with its implementation in Prolog.

As noted above, grammar (29) replaces the single syntactic category verb of grammar (25) with two distinct (formally unrelated) categories, namely, vtr and vintr. In grammar (25), any new rule affecting the category verb would affect all verbs, since they are all members of this single category. However, in grammar (29), a rule affecting, say, vtr, would not affect intransitive verbs, since they are in a distinct, formally unrelated syntactic category, vintr. Thus, if we constructed a partial grammar like (30a), which is based on (29), analyzing transitive verbs into verb stems and verb suffixes, we will have said nothing about the analysis of intransitive verbs.

(30) a. s --> np,vp.
    vp --> vtr,np;
        vintr.
    vtr --> vstem, vsuf.
    etc.

We show the (incorrectly claimed) unrelatedness of vtr and vintr in this grammar by considering one possible complete version of it, (30b):

(30) b. s --> np,vp.
    vp --> vtr,np;
        vintr.
    vtr --> vstem, vsuf.
    np --> noun
    noun --> [mary];[max].
    vstem --> [adore].
    vsuf --> [-s];[-ed].
    vintr --> [arrived].
The query (30c) along with ';' will elicit all of the output of (30b):

(30) c. \text{?} \text{s(Sent, [])}.

The reader should operate (30b) and the other possibilities to be discussed below.

Some comments on (30b) are in order. First, the last rule is necessary for the intransitive verb to be in the grammar. If we substitute [arrive] for [arrived], all we will get is [arrive] in the output, since the vstem-vsuf analysis here applies only to transitive verbs. To analyze intransitive verbs into stem and suffix will require a separate rule, as in grammar (30d):

(30) d. s --> np, vp.
     vp --> vtr, np;
     vintr.
     vtr --> vstem, vsuf.
     vintr --> vstem, vsuf.

Grammar (30d) succeeds in analyzing both transitive and intransitive verbs into stem and suffix, but the cost is quite high. First, the vtr and vintr rules clearly represent a missed generalization: we have here two statements rather than a single statement of what is clearly a single word formation process, the analysis of verbs into stem and suffix. Second, grammar (30d) has actually lost the transitive/intransitive distinction and will over-generate just as grammar (25) did. In (30d), this over-generation is due to the fact that both vtr and vintr are constructed on the same category vstem, which is equally either [adore] or [arrive]. This problem may be solved as in grammar (30e) by creating different vstem categories, but we will still not have succeeded in giving a single, unified statement of the analysis of all verbs into stem and suffix.

(30) e. s --> np, vp.
     vp --> vtr, np;
     vintr.
     vtr --> vtrstem, vsuf.
     vintr --> vintrstem, vsuf.
     np --> noun

In current linguistic analysis, distinctions such as transitive/intransitive are treated not in terms of distinct syntactic categories, but as secondary information about particular members of a single general class, here, verb. Thus, while arrive and adore are both verbs (e.g., they both take verb suffixes which all verbs and only verbs do), adore must appear as the verb of a transitive vp, that is, one with an object np, and arrive must not, since it is intransitive. This information about the necessary compliment material of particular lexical items is encoded in what are called subcategorization frames. A subcategorization frame is information about the syntactic context in which a particular lexical item can appear. We might construct lexical entries for each of these verbs as in (31):

(31) adore (verb, '_np')
arrive (verb,'_')

For adore in (31), the expression 'np' is to be interpreted as follows. Take the blank to be the position where the verb is to go. Further, assume that the single quotes enclose all of the material which can appear with the verb in vp. Then the expression 'np' says that this verb (adore) can appear in a vp with an np to its right (the direct object). Since this is the only subcategorization frame for adore, this verb must appear in this context. Similarly, the expression '_' for arrive indicates that it can (must) appear as the sole member of its immediate vp. Thus, both verbs are analyzed as verbs, but with certain unique distributional features involving complementary syntactic material, such as the required presence or absence of an object np.

A subcategorization frame description of transitivity can be carried out in Prolog. It is given here as grammar (32):

(32) s --> np,vp.
    np --> noun.
    vp --> verb('np'),np;
    verb('_').
    vstem('np') --> [adore].
    vstem('_') --> [arrive].
    np --> noun.
    vsuf --> [-s]; [-ed].
    verb(x) --> vstem(x), vsuf.

Grammar (32) is what is called a definite clause grammar. Unlike the earlier context-free grammars, it contains additional or extra information (e.g. subcategorization frames) which may be used to further limit the possible combinations of words which the grammar can generate. As above, Prolog automatically translates such a grammar into a different form. To understand further how this grammar works and how to query it, consider its translation given as (33):

(33) s(A,B) :-
    np(A,C),
    vp(C,B).

np(A,B) :-
    noun(A,B).

vp(A,B) :-
    ( verb('np',A,C),
      np(C,B);
      verb('_',A,B)
    ).

verb(A,B,C) :-
    vstem(A,B,D),
    vsuf(D,C).

noun(A,B) :-
    ( 'C'(A,mary,B)
      ; 'C'(A,max,B)
    ).

vstem('np',A,B) :-
    'C'(A,adore,B).

vstem('_',A,B) :-
    'C'(A,arrive,B).

vsuf(A,B) :-
    ( 'C'(A,-s,B)
      ; 'C'(A,-ed,B)
    ).

yes

The s rule and the np rule of (33) are to be interpreted as before. (Note that the contents of np here is only proper names, so the list, A, is an np if that same list A is a noun.) An inspection of the vp rule and the verb rule will reveal more about the general format of Prolog expressions. Recall the general form (6) of Prolog expressions given above:

(6) x(Y,Z)
In (6), Y is a list of words with Z as the last part of that list. Expression (6) states that list Y with Z removed is the contents of syntactic category x. Actually, expression (6) is only a subtype of the expressions available in a definite clause grammar such as (33). To the left of Y, we can carry other information. The general DCG expression form is (34):

\[(34) \ x(A, B, \ldots, Y, Z)\]

In (34), Y and Z represent a difference list representation of the list of words making up syntactic category x, just as in (6). In (34), 'A,B,..' represent arbitrarily many other information positions to the left of the difference list. Thus, we interpret the vp rule as follows: List A minus list B (B=[ ]) is a vp if a) list A minus list C is any verb with the designation '_np' and the rest of the list (C-B) is an np, or b) list A minus list B (B=[ ]) is a verb with the designation '_'. Thus, the only lists analyzable as np's will be those containing a '_np'-designated verb and an np, or only a '_'-designated verb. A glance at the vocabulary rules will indicate that only adore will appear in a vp with an np object; arrive will appear only in vp's with no object.

This glance at the lexicon should also reveal something else. The vp rule restricts '_np'-designated verbs to transitive vp's and '_'-designated verbs to be intransitive vp's, but it is only verb stems (vstem) which directly carry this feature. Verbs are composed from vstem and vsuf by the verb rule. Since verbs must carry the '_np' or '_-' designation to comply with the vp rule, how do they get it? The answer lies in the verb rule. Consider the verb rule from (32), with the difference list material removed, given here as (35):

\[(35) \ \text{verb}(X) \rightarrow \text{vstem}(X), \text{vsuf}.\]

In (35), the repeated use of X means that the two instances of X are bound; when one has a particular value, the other must also have it. X with vstem is either '_np' or '_-', as inspection of the vstem lexical rules of (32) will show. Thus, a verb containing a '_np' vstem will also be '_np', and so forth. That is, the verb inherits its transitivity designation from its vstem.

As was the case earlier, any query of (32) must have the structure of the expressions found in the Prolog-translated form (33). Thus, (36a-d) are queries about the possible lists of words which (33) would generate as s's, np's, vp's, and verbs:

\[(36) \ a. \ |?- \ s(\text{Sent}, [ ]).\]
\[\quad b. \ |?- \ np(\text{NP}, [ ]).\]
\[\quad c. \ |?- \ vp(\text{VP}, [ ]).\]
\[\quad d. \ |?- \ \text{verb}(\text{TR}, V, [ ]).\]

Notice that (36d) contains two variables, the first corresponding to the transitivity designation of the verb. Query (36e) will elicit only intransitive verbs:

\[(36) \ e. \ |?- \ \text{verb}('_', V, [ ]).\]

Query (36f) will elicit a response to whether adore is a transitive or intransitive vstem:

\[(36) \ f. \ |?- \ \text{vstem}(\text{TR}, \text{[adore]}, [ ]).\]

Try these and construct others.
In sum, grammar (32-33) represents an analysis of transitivity which is superior to (25), (29), (30b), (30d), and (30e). Grammar (25) simply failed to describe the transitivity distinction and over-generated both transitive and intransitive vp's. Grammars (29-30e) resolved the over-generation problem, but the tactic of dividing verbs into two distinct syntactic classes resulted in a missed generalization, namely, that all verbs are uniformly composed of a stem and a suffix. Grammar (32-33) resolves this difficulty. By calling all verbs 'verb', it is able to state in a single rule that verbs are composed of a stem and a suffix. By use of subcategorization frames, it is able to distinguish the type of vp in which each type of verb can participate. Since this grammar correctly describes what the possible sentences are, and does so in a maximally general way (without missed generalizations), we take it to be the best model of English of the ones considered. Until some arguably better model comes along, we take this model as telling us how English speakers analyze transitivity—in terms of subcategorization frames (rather than in terms of distinct verb categories). Thus we further conclude that subcategorization frames are among the devices which make up the grammars of natural languages.

6. Teaching with Prolog

The preceding sample problems are more elaborately discussed than they need be with beginning students. The elaboration here is due to the attempt to lay out some of the workings of Prolog's grammar facility. In fact, students can get along fine with Prolog hearing much less about its workings, and Prolog is capable of dealing with many more aspects of linguistic analysis than I have dealt with here. Let me pursue each point in turn.

As regards how much students have to be told to work with the grammar facility, they need not be told a lot beyond generative rule form and the form of queries. A grammar course need not become preoccupied with particulars of the AI language. That is a major advantage of the generative rule facility. For example, students can be given a grammar of transitivity like (29), and can be told how to query it without making strong reference to the translated form of (29) (e.g.,
To ask what word sequences the grammar will produce for a particular syntactic category (s, np, etc.), at the Prolog prompt just type a) that category in small letters, b) open parenthesis, c) the same category in caps, d) comma, e) an opening bracket and a closing bracket, f) close parenthesis, and g) period (e.g. s(S,[]). to elicit sentences or np(NP,[]). to elicit np's, etc.). I have found that this 'magic formula' approach to queries works pretty well for situations where I want to use Prolog grammars to illustrate points of grammar rather than discuss Prolog. Those students who find Prolog itself an object of interest can pursue it further individually. In general, I have found that once students get past the initial period of getting acquainted with the computer, Prolog grammar programs prove very useful not only in illustrating particular points of analyses, but also in providing something very concrete For them to deal with in a subject area which is usually considered abstract.

Regarding Prolog's linguistic description capabilities, these outextend what I have discussed here. Without going into programming details here, I have used Prolog programs to deal with such things as derivational versus inflectional morphology, the concept of stem (versus base) as the basis of inflectional morphology, the analysis of possessive phrases as a class of determiner (versus, say, adjective—a traditional characterization), recursion and recursive nodes, and number agreement within NP and within S. In addition, (using DCG) one can write programs which will return not only a phrase or sentence, but its structural analysis or logical form. Some of these analyses require additional programming techniques (e.g. depth bounding in analyses involving left-branching recursion), but these are quite manageable, and are easily 'isolated out' for beginning students to whom these details are not relevant. The generative rule facility allows one to focus pretty clearly on the linguistic issues being dealt with.
7. Final remarks

The type of grammatical study I have sketched here represents another viable approach to teaching linguistic subject matter to non-majors. It deals with linguistic formalism head-on, and the instructor may deal with the programming language if that is of interest, but need not. That is the advantage of the generative rule facility, which I have partially described here. It allows one to construct grammars in very nearly traditional generative format and to query these grammars for recognition or output by reasonably straightforward 'formulas'. It turns an otherwise very abstract subject into a much more concrete one, which makes non-majors much more comfortable dealing with the subject matter. Prolog materials are frequently aimed at audiences with background in logic, linguistics, or computer science. However, I have found that it is quite possible to use the Prolog generative rule facility successfully with an audience untrained in any of these.

References


