1 Simple Definite Clause Grammars

The observation that sentences are simply concatenated strings of constituents that can be put together through unification is the basis for a grammar formalism called *Definite Clause Grammars* (DCGs). In Prolog for example, we can use the following grammar to parse the sentence `[he, itches]`:

```
concat([ ], X, X).
concat([A|B], C, [A|D]) :- concat(B, C, D).
/* grammar: */
s(S) :-
  np(NP),
  vp(VP),
  concat(NP, VP, S).

/* lexicon: */
np([he]).
vp([itches]).
```

This grammar works in Prolog by generating NP's and VP's, then trying to match them to the sentence. With only a few NP's and VP's, this will work OK; however, the technique slows down rapidly as more NP's and VP's are added to the lexicon.

A better method is to divide the sentence into pieces first, then try to parse each piece as the appropriate constituent:

```
concat([ ], X, X).
concat([A|B], C, [A|D]) :- concat(B, C, D).
/* grammar: */
s(S) :-
  concat(NP, VP, S),
  np(NP),
  vp(VP).

/* lexicon: */
np([he]).
vp([itches]).
```

While this grammar is more efficient than the first one for parsing, it still tries every possible way of splitting the sentence in two. Better still would be to allow the NP parser to figure out exactly how much of the sentence was needed for an NP, then try to parse the remainder of the sentence as a VP. We can achieve this by giving each predicate two arguments: the first is the list of words from which that type of constituent is to be extracted, and the second is the list of words that's left over after the constituent has been extracted:

```
/* grammar: */
s(S0, S2) :-
  np(S0, S1),
  vp(S1, S2).

/* lexicon: */
np([he | Rest], Rest).
vp([itches | Rest], Rest).
```

We can now parse the sentence [he, itches] with the query:

```
?- s([he, itches], [ ]).
yes.
```

2 Agreement

We can do subject-verb agreement in DCG by adding an additional parameter to the NP and VP predicates:

```
/* grammar: */
s(S0, S2) :-
  np(Agr, S0, S1),
  vp(Agr, S1, S2).

/* lexicon: */
np(Agr, [Word | Rest], Rest) :-
  lex_np(Agr, Word).
vp(Agr, [Word | Rest], Rest) :-
  lex_vp(Agr, Word).

lex_np(agr_3s, he).
lex_np(agr_not_3s, they).
lex_vp(agr_3s, itches).
lex_vp(agr_not_3s, itch).
```

Note that I have added `lex_np` and `lex_vp` predicates to simplify the specification of the lexicon. By using the same variable in the grammar for both the NP agreement and the VP agreement, the two are forced to match.

3 Parse Trees

When used for parsing, these grammars act as acceptors; they tell us whether a sentence parses. They do not produce any kind of parse tree to tell us what the syntactic structure of the sentence is. We can add yet another parameter to our predicates for use in building a parse tree. A parse tree will simply be a list whose first item is the name of the grammatical category at the root of the tree, and whose remaining items are either words or other parse trees:

```
/* grammar: */
s([s, NP, VP], S0, S2) :-
  np(Agr, NP, S0, S1),
  vp(Agr, VP, S1, S2).
np(Agr, [np, Det, N], S0, S2) :-
  det(Agr, Det, S0, S1),
  n(Agr, N, S1, S2).

/* lexicon: */
np(Agr, Tree, [Word | Rest], Rest) :- lex_np(Agr, Tree, Word).
det(Agr, Tree, [Word | Rest], Rest) :- lex_det(Agr, Tree, Word).
n(Agr,  Tree, [Word | Rest], Rest) :- lex_n(Agr, Tree, Word).
vp(Agr, Tree, [Word | Rest], Rest) :- lex_vp(Agr, Tree, Word).

lex_np(agr_3s, [np, he], he).
lex_np(agr_not_3s, [np, they], they).
lex_det(_, [det, the], the).
lex_n(agr_3s, [n, frog], frog).
lex_n(agr_not_3s, [n, frogs], frogs).
lex_vp(agr_3s, [vp, itches], itches).
lex_vp(agr_not_3s, [vp, itch], itch).
```

4 Subcategorization Frames

We can now add subcategorization frames to our verbs (`verb_i` represents intransitive verbs, `verb_m` represents monotransitive verbs, and `verb_d` represents ditransitive verbs):

```
/* grammar: */
s([s, NP, VP], S0, S2) :-
  np(Agr, NP, S0, S1),
  vp(Agr, VP, S1, S2).
np(Agr, [np, Det, N], S0, S2) :-
  det(Agr, Det, S0, S1),
  n(Agr, N, S1, S2).
vp(Agr, [vp, Verb], S0, S1) :-
  verb_i(Agr, Verb, S0, S1).
vp(Agr, [vp, Verb, Dobj], S0, S2) :-
  verb_m(Agr, Verb, S0, S1),
  np(_, Dobj, S1, S2).
vp(Agr, [vp, Verb, Iobj, Dobj], S0, S3) :-
  verb_d(Agr, Verb, S0, S1),
  np(_, Iobj, S1, S2),
  np(_, Dobj, S2, S3).


/* lexicon: */
np(Agr, Tree, [Word | Rest], Rest) :- lex_np(Agr, Tree, Word).
det(Agr, Tree, [Word | Rest], Rest) :- lex_det(Agr, Tree, Word).
n(Agr,  Tree, [Word | Rest], Rest) :- lex_n(Agr, Tree, Word).
vp(Agr, Tree, [Word | Rest], Rest) :- lex_vp(Agr, Tree, Word).
verb_i(Agr, Tree, [Word | Rest], Rest) :- lex_verb_i(Agr, Tree, Word).
verb_m(Agr, Tree, [Word | Rest], Rest) :- lex_verb_m(Agr, Tree, Word).
verb_d(Agr, Tree, [Word | Rest], Rest) :- lex_verb_d(Agr, Tree, Word).


lex_det(_, [det, the], the).
lex_det(agr_3s, [det, a], a).
lex_det(agr_not_3s, [det, some], some).
lex_n(agr_3s, [n, frog], frog).
lex_n(agr_not_3s, [n, frogs], frogs).
lex_n(agr_3s, [n, basketball], basketball).
lex_n(agr_not_3s, [n, basketballs], basketballs).
lex_verb_i(agr_3s, [vp, itches], itches).
lex_verb_i(agr_not_3s, [vp, itch], itch).
lex_verb_m(agr_3s, [vp, touches], touches).
lex_verb_m(agr_not_3s, [vp, touch], touch).
lex_verb_d(agr_3s, [vp, tosses], tosses).
lex_verb_d(agr_3s, [vp, toss], toss).
```

5 Semantic Interpretation

Semantic interpretation can be added to a definite clause grammar simply by augmenting each structure in the lexicon with an additional field. In the following grammar, each constituent has the form `[name, agreement, interpretation]`. First, here's the parser:

```
parse(Type, Sem, String) :-
  constituent([Type, _, Sem | _], String, []).

constituent(Con, S0, S1) :-
  rule(Con, RHS),
  constituents(RHS, S0, S1).

constituent(Lex, [Word | S0], S0) :-
  word(Word, Lex).

constituent(Lex, [Word | S0], S1) :-
  word(Word, Lex, Complements),
  constituents(Complements, S0, S1).

constituents([], S0, S0).

constituents([Con | Rest], S0, S2) :-
  constituent(Con, S0, S1),
  constituents(Rest, S1, S2).
```

In these rules, the substring information (*i.e.* `S0`, `S1`, and `S2`) has been separated from the grammar.

Here are the grammar rules. Notice that in building an interpretation for the sentence, the rule `s` $\Rightarrow$ `np vp` forces the agent of the action to be the interpretation of the noun phrase:

```
rule([s, _, [Action, [agent, Subj] | Rest]],
     [[np, Agr, Subj], [vp, Agr, [Action, [agent, Subj] | Rest] | _]]).

rule([np, Agr, Sem], [[det, Agr], [n, Agr, Sem]]).
```

Finally, most of the information about semantic interpretation is embedded in the lexical entries:

```
word(itches, [vp, agr_3s, [itch, [agent, _]]]).
word(touches, [vp, agr_3s,
                   [touch, [agent, _], [theme, Obj]]], [[np, _, Obj | _]]).
word(persuades, [vp, agr_3s,
                     [persuade, [agent, _],
                                [theme, Person],
                                [co_theme, [Pred, [agent, Person] | Rest]]]],
                [[np, _, Person | _],
                 [vp, infinitival, [Pred, [agent, Person] | Rest] | _]]).
word(to, [vp, infinitival, Sem], [[vp, nonfinite, Sem | _]]).
word(smell, [vp, nonfinite, [smell, [agent, _], [theme, Smellee]]],
                 [[np, _, Smellee | _]]).
word(sleep, [vp, nonfinite, [sleep, [agent, _]]]).
word(sleeps, [vp, agr_3s, [sleep, [agent, _]]]).
word(likes, [vp, agr_3s, [like, [agent, _], [theme, Likee]]],
            [[np, _, Likee | _]]).
word(likes, [vp, agr_3s, [like, [agent, Person],
                                [theme, [Pred, [agent, Person] | Rest]]]],
            [[vp, infinitival, [Pred, [agent, Person] | Rest] | _]]).
word(carry, [vp, nonfinite, [carry, [agent, _], [theme, Item]]],
            [[np, _, Item | _]]).
word(the, [det, _]).
word(cow, [n, agr_3s, cow]).
word(bobo, [np, agr_3s, bobo]).
```

Each of the above grammars is available on the departmental machines in the directory ~`mayfield/dcg`.