Definite Clause Grammars

Human Language Technology

Mike Rosner, Dept ICS
October 2012
Contents

1 Definite Clause Grammars
   - Grammars and Logic
   - Handling Text Segments
   - The DCG Formalism
   - Building a Parser

2 Top Down Recognition Algorithm in Prolog
References

Fernando C. N. Pereira and Stuart M. Shieber.  
Prolog and natural-language analysis.  
Center for the Study of Language and Information, Stanford, CA, USA, 1987.

Fernando C. N. Pereira and David H. D. Warren. 
Definite clause grammars for language analysis - a survey of the formalism and a comparison with augmented transition networks. 
Definite Clause Grammars (DCGs)

- DCG's are a Prolog-based grammar formalism
- Basic intuition is that a grammar rule like

\[ S \rightarrow NP \, VP \]

can be viewed first as a piece of *propositional calculus* i.e.

\[ NP \land VP \supset S \]

ie if NP is true and VP is true then you can infer that S is true
- But there is something missing here
Definite Clause Grammars (DCGs)

- DCG’s are a Prolog-based grammar formalism
- Basic intuition is that a grammar rule like

\[ S \rightarrow NP \ VP \]

can be viewed first as a piece of *propositional calculus* i.e.

\[ NP \land VP \supset S \]

ie if NP is true and VP is true then you can infer that S is true

- But there is something missing here
- The notion *what it is* these predicates are true of.
Definite Clause Grammars (DCGs)

- DCG’s are a Prolog-based grammar formalism
- Basic intuition is that a grammar rule like
  \[ S \rightarrow NP \ VP \]
  can be viewed first as a piece of *propositional calculus* i.e.
  \[ NP \land VP \supset S \]
  ie if NP is true and VP is true then you can infer that S is true
- But there is something missing here
- The notion *what it is* these predicates are true of.
- They are true of text segments
- So let’s give each predicate an an argument
We provide text-segment arguments n, v and s as follows

\[ NP(n) \land VP(v) \supset S(s) \]

Now instead of the meaning of each predicate being atomic, is more like *NP is true of text segment n* etc.

But there is still something missing . . .
DCG and Prolog

- We provide text-segment arguments n, v and s as follows

\[ NP(n) \land VP(v) \supset S(s) \]

- Now instead of the meaning of each predicate being atomic, is more like *NP* is true of text segment *n* etc.

- But there is still something missing . . .

- Namely the **adjacency relationships** between the segments
We provide text-segment arguments $n$, $v$ and $s$ as follows

\[ NP(n) \land VP(v) \supset S(s) \]

Now instead of the meaning of each predicate being atomic, is more like *NP is true of text segment n* etc.

But there is still something missing . . .

Namely the adjacency relationships between the segments

We also have that $n$ and $v$ are adjacent, and when concatenated together, they form $s$.

How can this be stated?

One way is state the fact as a predicate e.g. `append(n,v,s)`.

But there is a neater way
Text Indices for Text Segments

- John
- booked
- that
- flight

1 2 3 4 5

NP(x, y) \land VP(y, z) \supset S(x, z)

NP(1, 2) \land VP(2, 5) \supset S(1, 5)

- This is equivalent to the following definite clause:
  \( s(X,Z) :- np(X,Y), \; vp(Y,Z). \)

- We can define a whole grammar in this way
A Complete Grammar

\[
\begin{align*}
  s(X,Z) & : - \ np(X,Y), \ vp(Y,Z). \\
  np(X,Z) & : - \ d(X,Y), \ n(Y,Z). \\
  np(X,Z) & : - \ n(X,Z). \\
  vp(X,Z) & : - \ v(X,Y), \ np(Y,Z). \\
  vp(X,Z) & : - \ v(X,Z).
\end{align*}
\]

But what is missing?
A Complete Grammar

\[
\begin{align*}
  \text{s}(X, Z) & : \neg \text{np}(X, Y),\ \text{vp}(Y, Z). \\
  \text{np}(X, Z) & : \text{d}(X, Y),\ \text{n}(Y, Z). \\
  \text{np}(X, Z) & : \text{n}(X, Z). \\
  \text{vp}(X, Z) & : \text{v}(X, Y),\ \text{np}(Y, Z). \\
  \text{vp}(X, Z) & : \text{v}(X, Z).
\end{align*}
\]

- But what is missing?
- Information about actual words.
A Complete Grammar

\[
s(X,Z) :- \text{np}(X,Y), \text{vp}(Y,Z).
\]
\[
\text{np}(X,Z) :- \text{d}(X,Y), \text{n}(Y,Z).
\]
\[
\text{np}(X,Z) :- \text{n}(X,Z).
\]
\[
\text{vp}(X,Z) :- \text{v}(X,Y), \text{np}(Y,Z).
\]
\[
\text{vp}(X,Z) :- \text{v}(X,Z).
\]

- But what is missing?
- Information about actual words.
- This is contained in a lexicon
%grammar
s(X,Z) :- np(X,Y), vp(Y,Z).
np(X,Z) :- d(X,Y), n(Y,Z).
np(X,Z) :- n(X,Z).
vp(X,Z) :- v(X,Y), np(Y,Z).
vp(X,Z) :- v(X,Z).

%lexicon
d([the|X],X).
n([man|X],X).
n([kim|X],X).
n([mia|X],X).
n([mia|X],X).
v([shot|X],X).
Difference Lists

- A list can be represented as the difference between two other lists. For example `[aaa]` is the difference between `[aaa, bbb, ...]` and `[bbb, ...]`. In this case, the list `[bbb, ...]` serves as a kind of pointer to the end of the prefix `[aaa]` inside the list `[aaa, bbb, ...]`. 
Difference Lists

- Key idea: we can represent a text segment as the difference between two lists.
- So a single word $W$ is the difference between a list $[W|X]$ and the list $X$.
- Hence the clause $\nu([\text{shot}|X],X)$ states that the text segment consisting of the word “shot” is a $\nu(\text{verb})$.
- So how would we state that there is a $s(\text{entence})$ for the text segment “mia shot the man”.

Difference Lists
Key idea: we can represent a text segment as the difference between two lists.

So a single word $W$ is the difference between a list $[W|X]$ and the list $X$.

Hence the clause

$$v([\text{shot}|X], X).$$

states that the text segment consisting of the word “shot” is a $v(verb)$.

So how would we state that there is a $s(entence)$ for the text segment “mia shot the man”. 

$$s([\text{mia, shot, the man}], []).$$
The DCG Formalism

% grammar
s  -->  np, vp.
np -->  d, n.
np -->  n.
vp -->  v.
vp -->  v, np.

% lexicon
d -->  [the].
n -->  [man].
n -->  [mia].
n -->  [kim].
v -->  [shot].
How to Compile a Grammar

- Use a text editor to create a file with a “.pl” extension and save in a folder - e.g. g0.pl
- Open a shell and cd to the grammar folder
- Run Prolog
How to Compile a Grammar

?- [g0].
g0 compiled 0.00 sec, 4,248 bytes
true.
?- s([mia,shot,the,man],[]).
true .
?- s([mia,shot,the,woman],[]).
false.
?- listing(s).
s(A, C) :-
np(A, B),
vp(B, C).
We give each symbol in the grammar an argument which holds the parse tree.

Each grammar rule indicates how to build the LHS tree from the trees of the RHS constituents,

For example, e.g.

before:  \( s \rightarrow np, \ vp \)
after:  \( s(\ s(NP, VP)) \rightarrow np(NP), \ vp(VP) \).

Each lexical rule indicates the terminal tree

For example, e.g.

before:  \( n \rightarrow [mia] \).
after:  \( n(\ n(mia)) \rightarrow [mia] \).
Grammar that Builds a Parse Tree

% grammar
s(s(NP,VP)) --> np(NP), vp(VP).
np(np(D,N)) --> d(D), n(N).
np(np(N)) --> n(N).
vp(vp(V)) --> v(V).
vp(v(V,NP)) --> v(V), np(NP).

% lexicon
d(d(the)) --> [the].
n(n(man)) --> [man].
n(n(mia)) --> [mia].
n(n(kim)) --> [kim].
v(v(shot)) --> [shot].
How it Works

?- [g1].
% g1 compiled 0.00 sec, 5,344 bytes
true.

?- s(PT,[mia,shot,kim],[]).
PT = s(np(n(mia)), v(v(shot), np(n(kim))))
Top Down Recogniser in Prolog

```
parse(C, [Word|S], S) :-
    word(C, Word).

parse(C, S1, S) :-
    C --> Cs,
    parse_list(Cs, S1, S).

parse_list([], S, S).
parse_list([C|Cs], S1, S) :-
    parse(C, S1, S2),
    parse_list(Cs, S2, S).
```