Symbolic Programming - Chapter 8 - More DCGs

Varjak Wolfe

November 26, 2021

These notes follow the online coursebook Learn Prolog Now.

Two important capabilities offered by DCG notation:

- Extra arguments
- Extra tests

Extra Arguments

CDGs allow us to specify extra arguments.

Let us extend our previous grammar from the last chapter to include sentences containing pronounts like she,he.

s −− > np, vp.
np −− > det, n.
vp −− > v, np.
vp −− > v.
det −− > [the].
det −− > [a].
n −− > [man].
n −− > [woman].
v −− > [shoots].

We need to add rules for ponouns and add a rule for saying that noun phrases can be pronouns.

s −− > np, vp.
np −− > det, n.
np −− > pro.
vp −− > v, np.
vp −− > v.
det −− > [the].
det −− > [a].
n −− > [man].
n −− > [woman].
v −− > [shoots].
pro −− > [he].
pro −− > [she].
pro −− > [him].
pro −− > [her].

Some sample queries:

?- s([she,shoots,him],[]).

yes

?- s([a, woman, shoots, him],[]).

yes

These are grammtoca strings accepted by the DCG. However, there are also examples of ungrammatical strings that it accepts:

?- s([a, woman, shoots, he],[]).

yes

?- s([her, shoots, she],[]).

yes

The DCG ignroes some basic facts about english:

- she and he are subject pronouns while her and him are obejct pronouns. We can do this using extra arguments.

s $--$ > np(subject), vp.

np(_) $--$ > det, n.

np(X) $--$ > pro(X).

vp $--$ > v, np(object).

vp $--$ > v.

det $--$ > [the].

det $--$ > [a].

n $--$ > [man].

n $--$ > [woman].

v $--$ > [shoots].

pro(subject) $--$ > [he].

pro(subject) $--$ > [she].

pro(object) $--$ > [him].

pro(object) $--$ > [her].

?- s([she, shoots, he],[])

no

So how does it work?

Recall that the rule

s $--$ > np,vp

is syntactic sugar for:

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

So the rule:

s $--$ > np(subject), vp.

translates into:

s(A,B):- NP(subject,A,C), vp(C,B).

Listing noun phrases:

?- np(Type, NP, []).

Type =_

NP = [the, woman];

Type =_

NP = [the, man];

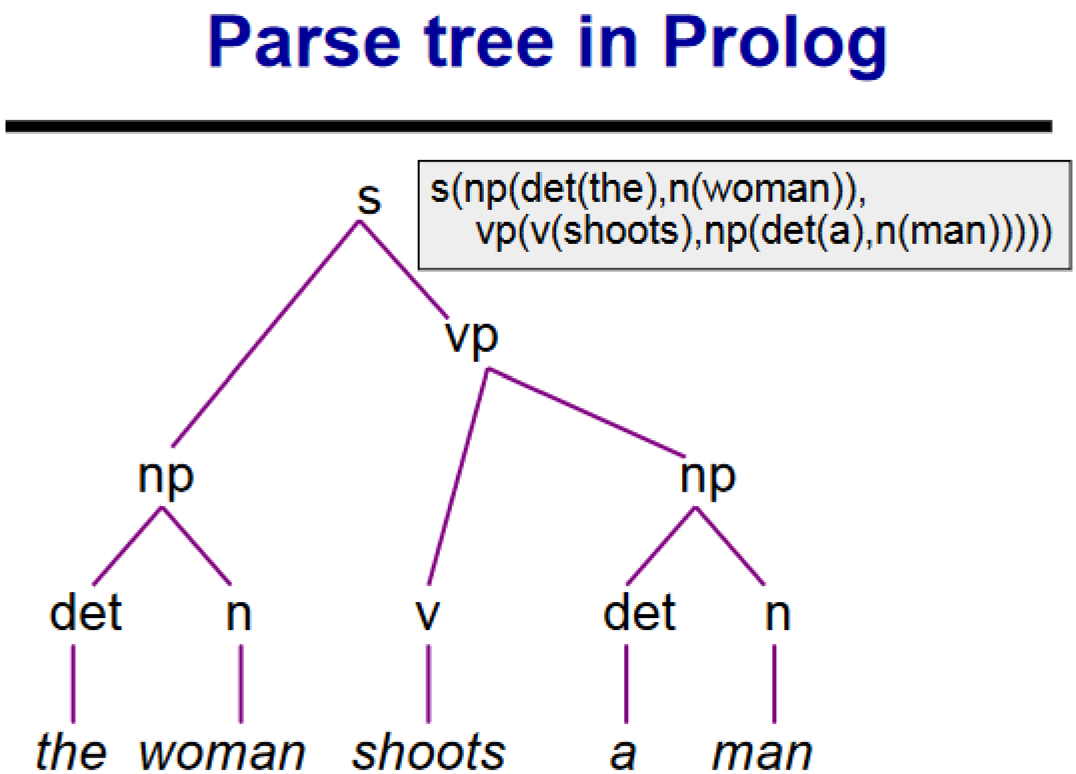
Type =_

NP = [a, woman];

Type =_

NP = [a, man];
Type =_
NP = [he];

Building Parse Trees



Lets make our DCG build a parse tree.

```
s --> np(subject), vp.
np(_) --> det, n.
np(X) --> pro(X).
vp --> v, np(object).
vp --> v.
det --> [the].
det --> [a].
n --> [woman].
n --> [man].
v --> [shoots].
pro(subject) --> [he].
pro(subject) --> [she].
pro(object) --> [him].
pro(object) --> [her].
```

```
s(s(NP,VP)) --> np(subject,NP), vp(VP).
np(_,np(Det,N)) --> det(Det), n(N).
np(X,np(Pro)) --> pro(X,Pro).
vp(vp(V,NP)) --> v(V), np(object,NP).
vp(vp(V)) --> v(V)).
det(det(the)) --> [the].
det(det(a)) --> [a].
n(n(woman)) --> [woman].
n(n(man)) --> [man].
v(v(shoots)) --> [shoots].
pro(subject,pro(he)) --> [he].
pro(subject,pro(she)) --> [she].
pro(object,pro(him)) --> [him].
pro(object,pro(her)) --> [her].
```

?- s(T,[he,shoots],[]).
T = s(np(pro(he)),vp(v(shoots)))
yes

Beyond Context-Free Languages

In the previous lecture we showed DCGs as useful for working with context free grammars. However, they can deal with a lot more than that. The extra arguments allow us to cope with any computable language.

An example using $a^n b^n c^n/[\epsilon]$

This language consists of strings such as abc, aabbcc, aaabbbccc, aaaabbbbcccc and so on.

This is not context-free- but can we still write a DCG to produce these strings.

s(Count) $-->$ as(Count), bs(Count), cs(Count).
as(0) $-->$ [].
as(succ(Count)) $-->$ [a], as(Count).
bs(0) $-->$ [].
bs(succ(Count)) $-->$ [b], bs(Count).
cs(0) $-->$ [].
cs(succ(Count)) $-->$ [c], cs(Count).

We can also call any prolog predicate from the right side of a DCG rule using curly brackets

s(Count) $-->$ as(Count), bs(Count), cs(Count).
as(0) $-->$ [].
as(NewCnt) $-->$ [a], as(Cnt), NewCnt is Cnt + 1.

bs(0) −− > [].
bs(NewCnt) −− > [b], bs(Cnt), NewCnt is Cnt + 1.
cs(0) −− > [].
cs(NewCnt) −− > [c], cs(Cnt), NewCnt is Cnt + 1.

Seperating Rules from the Lexicon

This means eliminating all mention of individual words in the DCG and to record all info about individual words in a seperate lexicon.