Lecture 8: More DCGs

- Exercises
  - Solution to exercises LPN 7
  - Recall of previous lecture

- Theory
  - Examine two important capabilities offered by DCG notation:
    - Extra arguments
    - Extra tests
  - Discuss how they can be used to:
    - Build parse trees
    - Implement non context-free languages
    - Modularize grammars

Precision on Difference Lists

- In the previous lecture we introduced difference lists
- Note: difference lists are the representation of a concept, an alternative notation for lists.
  - NOT AN OPERATION!

```
?-  X = [a,b,c]-[c].
    X = [a, b, c]-[c].
?-  [a,b,c,d] = [a,b,c]-[X].
    false.
?-  [a,b,c,d]-[X] = [a,b,c|X]-[X].
    X = [d].
?-  s([a,b,c,d],[X]) = s([a,b,c|X],[X]).
    X = [d].
```

Solution to Exercise 7.1

```
s --> foo,bar,wiggle.
foo --> [choo].
foo --> foo,foo.
bar --> mar,zar.
mar --> me,my.
me --> [i].
my --> [am].
zar --> blar,car.
blar --> [a].
car --> [train].
wiggle --> [toot].
wiggle --> wiggle,wiggle.
```

```
s(X-T):- foo(X-B), bar(B-C),
    wiggle(C-T).
foo([choo|T]-T).
foo(X-T):- foo(X-B), foo(B-T).
bar(X-T):- mar(X-B), zar(B-T).
mar(X-T):- me(X-B), my(B-T).
me([i|T]-T).
my([am|T]-T).
zar(X-T):- blar(X-B), car(B-T).
blar([a|T]-T).
car([train|T]-T).
wiggle([toot|T]-T).
wiggle(X-T):-
        wiggle(X-B), wiggle(B-T).
```

```
?- s(X-[]).
X = [choo, i, am, a, train, toot] ;
X = [choo, i, am, a, train, toot, toot] ;
X = [choo, i, am, a, train, toot, toot, toot].
```

Solution to Exercise 7.2

The formal language $a^n b^n - \{\varepsilon\}$ consists of all the strings in $a^n b^n$ except the empty string. Write a DCG that generates this language.

```
s --> [a,b].
s --> l,s,r.
l --> [a].
r --> [b].
```

```
?- s(X,[]).
X = [a, b] ;
X = [a, a, b, b] ;
X = [a, a, a, b, b, b] .
```

Solution to Exercise 7.3

Let $a^n b^{2n}$ be the formal language which contains all strings of the following form: an unbroken block of as of length n followed by an unbroken block of b's of length 2n, and nothing else. For example, *abb*, *aabbbb*, and *aaabbbbbb* belong to $a^n b^{2n}$, and so does the empty string. Write a DCG that generates this language.

```
s --> [].
s --> l,s,r,r.
l --> [a].
r --> [b].
```

```
?- s(X,[]).
X = [] ;
X = [a, b, b] ;
X = [a, a, b, b, b, b].
```

Extra arguments in DCGs

- In the previous lecture we introduced basic DCG notation
- But DCGs offer more than we have seen so far
  - DCGs allow us to specify **extra arguments**
  - These extra arguments can be used for many purposes

Extending the grammar

- This is the simple grammar from the previous lecture
- Suppose we also want to deal with sentences containing pronouns such as

  she shoots him

  and

  he shoots her
- What do we need to do?

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

Extending the grammar

- Add rules for pronouns
- Add a rule saying that noun phrases can be pronouns

- Is this new DCG any good?
- What is the problem?

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

Some examples of grammatical strings accepted by this DCG

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

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

Some examples of ungrammatical strings accepted by this DCG

```
?- s([a,woman,shoots,he],[ ]).
yes
?- s([her,shoots,a,man],[ ]).
yes
s([her,shoots,she],[ ]).
yes
```

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

What is going wrong?

- The DCG ignores some basic facts about English
  - *she* and *he* are <u>subject pronouns</u> and cannot be used in object position
  - *her* and *him* are <u>object pronouns</u> and cannot be used in subject position
- It is obvious what we need to do: extend the DCG with information about subject and object
- How do we do this?

A naïve way…

```
s --> np_subject, vp.
np_subject --> det, n.              np_object --> det, n.
np_subject --> pro_subject.         np_object --> pro_object.
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].
```

Nice way 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 --> [woman].
n --> [man].
v --> [shoots].
pro(subject) --> [he].
pro(subject) --> [she].
pro(object) --> [him].
pro(object) --> [her].
```

This works…

```
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([she,shoots,him],[ ]).
yes
?- s([she,shoots,he],[ ]).
no
?-
```

What is really going on?

- Recall that the rule:

  s --> np,vp.

  is really syntactic sugar for:

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

- The rule

  s --> np(subject),vp.

  translates into:

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

Listing noun phrases

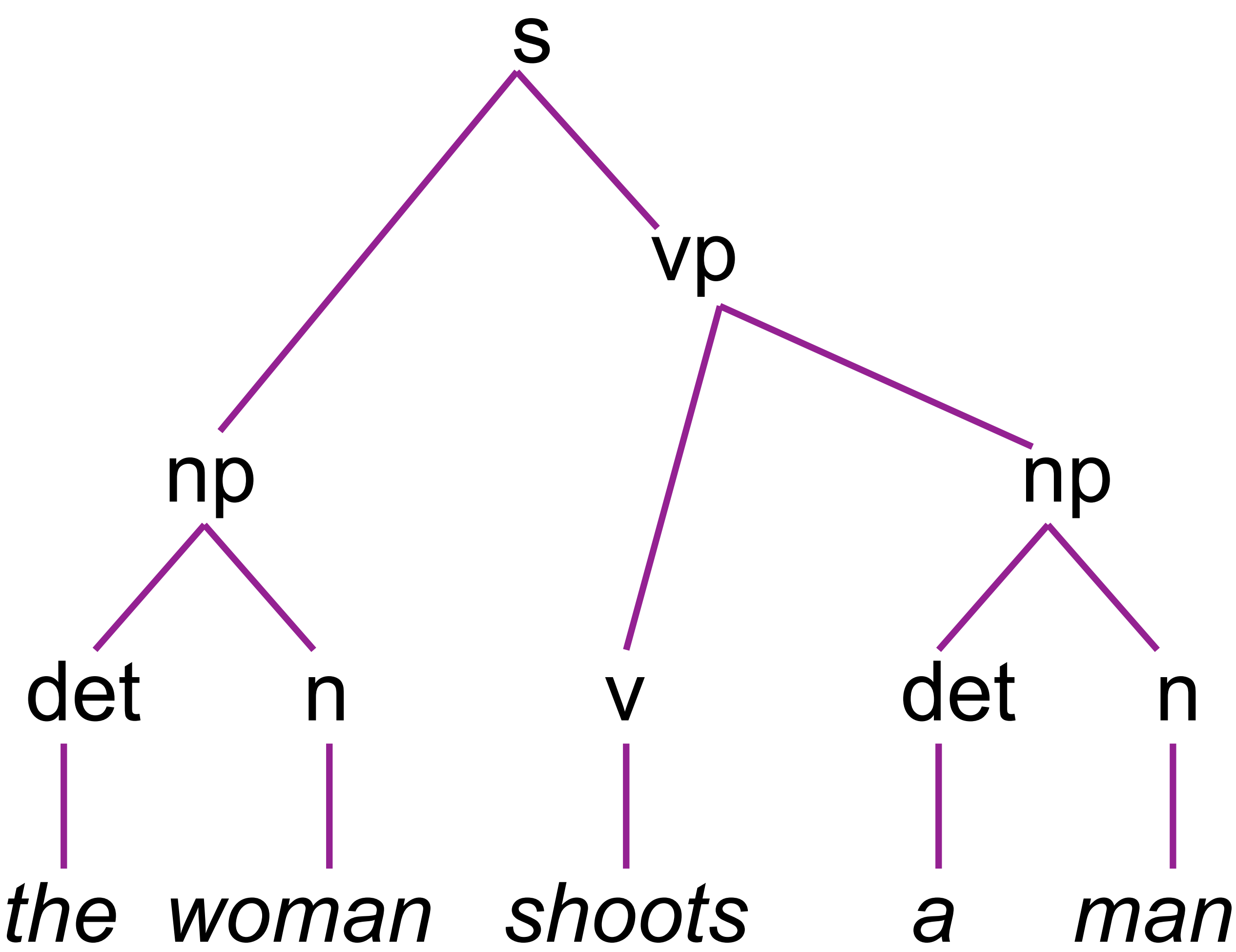
```
s --> np(subject), vp.
np(_) --> det, n.
np(X) --> pro(X).
vp --> v, np(object).
vp --> v.
det --> [the].
det --> [a].
n --> [woman].
n --> [man].
v --> [shoots].
pro(subject) --> [he].
pro(subject) --> [she].
pro(object) --> [him].
pro(object) --> [her].
```

```
?- np(Type, NP, [ ]).
Type =_
NP = [the,woman];

Type =_
NP = [the,man];

Type =_
NP = [a,woman];

Type =_
NP = [a,man];

Type =subject
NP = [he]
```

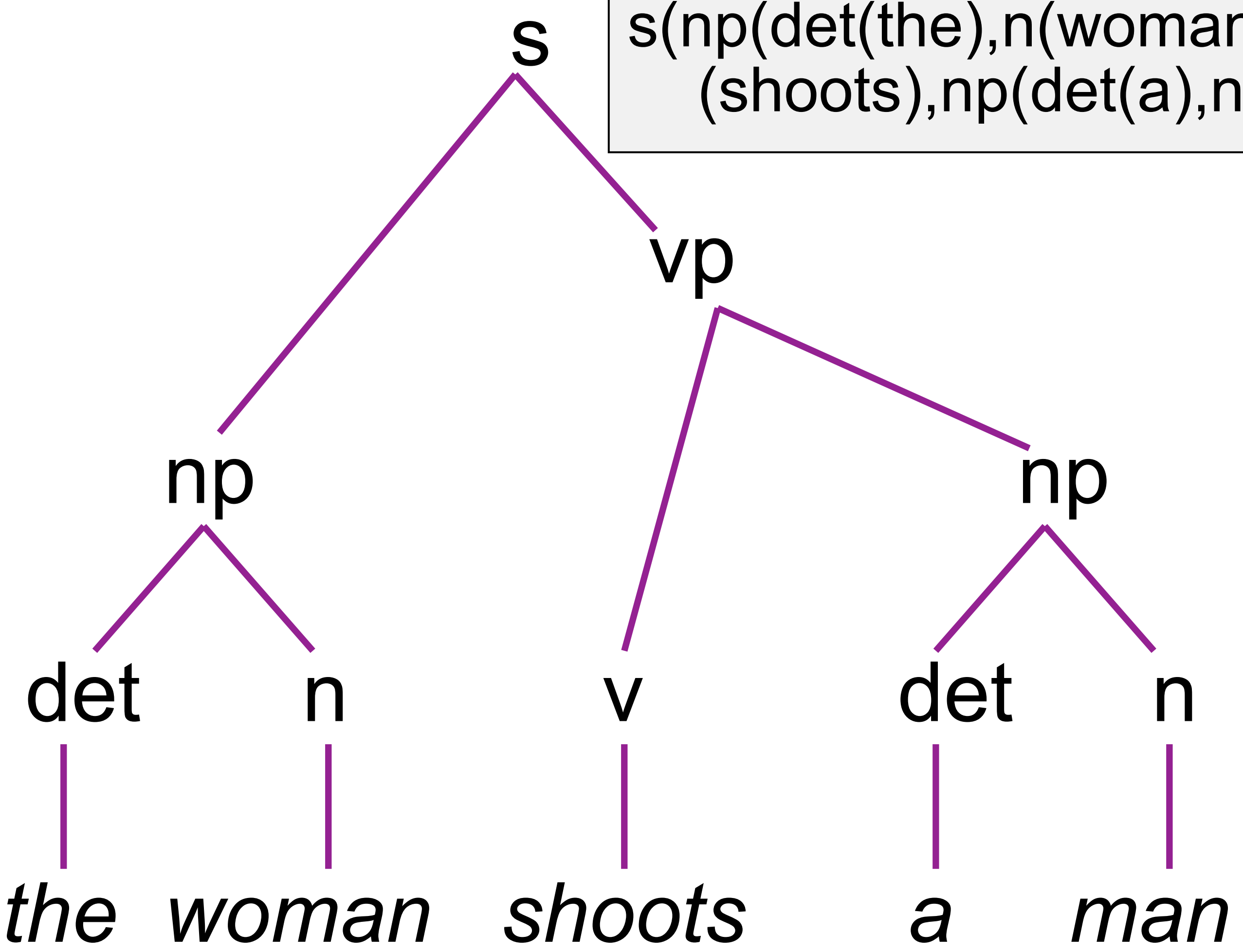
Building parse trees

- The programs we have discussed so far have been able to recognise grammatical structure of sentences
- But we would also like to have a program that gives us an analysis of their structure
- In particular we would like to see the trees the grammar assigns to sentences

Parse tree example



Parse tree in Prolog



s(np(det(the),n(woman)), vp(v
(shoots),np(det(a),n(man)))))

DCG that builds 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].
```

DCG that builds parse tree

```
?- s(T,[he,shoots],[]).
   T = s(np(pro(he)),
         vp(v(shoots)))
   yes.
?- s(Tree,S,[]).
   …
?- np(Type, Tree, S, []).
   Tree = np(det(the), n
         (woman)),
   S = [the, woman] ;
   …
   Type = subject,
   Tree = np(pro(he)),
   S = [he] .
```

```
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].
```

Beyond context free languages

- In the previous lecture we presented DCGs as a useful tool for working with context free grammars
- However, DCGs can deal with a lot more than just context free grammars
- The extra arguments gives us the tools for coping with any computable language
- We will illustrate this by looking at the formal language $a^nb^nc^n$

An example

- The language $a^n b^n c^n$ consists of strings such as ε, abc, aabbcc, aaabbbccc, aaaabbbbcccc, and so on
- This language is not context free – it is impossible to write a context free grammar that produces exactly these strings
- But it is very easy to write a DCG that does this

DCG for $a^n b^n c^n$

```
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).
```

```
?-  s(Count, X, []).
    Count = 0,
    X = [] ;
    Count = succ(0),
    X = [a, b, c] ;
    Count = succ(succ(0)),
    X = [a, a, b, b, c, c] .
```

Extra goals

- Any DCG rule is really syntactic structure for ordinary Prolog rule
- So it is not really surprising we can also call any Prolog predicate from the right-hand side of a DCG rule
- This is done by using curly brackets { }

Example: DCG for $a^n b^n c^n$

```
s(Count) --> as(Count), bc(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}.
```

Separating rules and lexicon

- One classic application of the extra goals of DCGs in computational linguistics is separating the grammar rules from the lexicon
- What does this mean?
  - Eliminate all mention of individual words in the DCG
  - Record all information about individual words in a separate lexicon

The basic grammar

```
s --> np, vp.

np --> det, n.

vp --> v, np.
vp --> v.

det --> [the].
det --> [a].

n --> [woman].
n --> [man].

v --> [shoots].
```

The modular grammar

```
s --> np, vp.

np --> det, n.

vp --> v, np.
vp --> v.

det --> [the].
det --> [a].

n --> [woman].
n --> [man].

v --> [shoots].
```

➔

```
s --> np, vp.

np --> det, n.

vp --> v, np.
vp --> v.

det --> [Word], {lex(Word,det)}.

n --> [Word], {lex(Word,n)}.

v --> [Word], {lex(Word,v)}.
```

+

```
lex(the,     det).
lex(a,       det).
lex(woman,   n).
lex(man,     n).
lex(shoots,  v).
```

Concluding Remarks

- DCGs are a simple tool for encoding context free grammars
- But in fact DCGs are a full-fledged programming language and can be used for many different purposes
- For linguistic purposes, DCG have drawbacks
  - Left-recursive rules
  - DCGs are interpreted top-down
- DCGs are no longer state-of-the-art, but they remain a useful tool

Exercise LPN Chapter 7

8.1, 8.2

Next lecture

- A closer look at terms
  - Introduce the identity predicate
  - Take a closer look at term structure
  - Introduce pre-defined Prolog predicates that test whether a given term is of a certain type
  - Show how to define new operators in Prolog