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(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 --> blar, car.

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 $a^n b^n \setminus \{\varepsilon\}$ 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 $a$s of length $n$ followed by an unbroken block of $b$’s of length $2n$, and nothing else. For example, $abb$, $aabbba$, 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?
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 subject pronouns and cannot be used in object position
  – *her* and *him* are object pronouns 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:
  \[ s \rightarrow np, vp. \]
  is really syntactic sugar for:
  \[ s(A,B) :- np(A,C), vp(C,B). \]

• The rule
  \[ s \rightarrow np(subject), vp. \]
  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

```
the  woman  shoots  a  man
```

```
\[
\begin{array}{c}
\text{s} \\
\text{vp} \\
\text{np} \\
\text{det}  \quad \text{n}  \quad \text{v}  \\
\text{the}  \quad \text{woman}  \quad \text{shoots} \\
\text{np} \\
\text{det}  \quad \text{n}
\end{array}
\]
```
Parse tree in Prolog

s(np(det(the), n(woman)), vp(v (shoots), np(det(a), n(man))))
**DCG that builds parse tree**

<table>
<thead>
<tr>
<th>Rule</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>s → np(subject), vp.</td>
<td>s(s(NP,VP)) → np(subject, NP), vp(VP).</td>
</tr>
<tr>
<td>np(_) → det, n.</td>
<td>np(_,np(Det,N)) → det(Det), n(N).</td>
</tr>
<tr>
<td>np(X) → pro(X).</td>
<td>np(X,np(Pro)) → pro(X,Pro).</td>
</tr>
<tr>
<td>vp → v, np(object).</td>
<td>vp(vp(V, NP)) → v(V), np(object, NP).</td>
</tr>
<tr>
<td>vp → v.</td>
<td>vp(vp(V)) → v(V).</td>
</tr>
<tr>
<td>det → [the].</td>
<td>det(det(the)) → [the].</td>
</tr>
<tr>
<td>det → [a].</td>
<td>det(det(a)) → [a].</td>
</tr>
<tr>
<td>n → [woman].</td>
<td>n(n(woman)) → [woman].</td>
</tr>
<tr>
<td>n → [man].</td>
<td>n(n(man)) → [man].</td>
</tr>
<tr>
<td>v → [shoots].</td>
<td>v(v(shoots)) → [shoots].</td>
</tr>
<tr>
<td>pro(subject) → [he].</td>
<td>pro(subject, pro(he)) → [he].</td>
</tr>
<tr>
<td>pro(subject) → [she].</td>
<td>pro(subject, pro(she)) → [she].</td>
</tr>
<tr>
<td>pro(object) → [him].</td>
<td>pro(object, pro(him)) → [him].</td>
</tr>
<tr>
<td>pro(object) → [her].</td>
<td>pro(object, pro(her)) → [her].</td>
</tr>
</tbody>
</table>
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 $a^n b^n c^n$
An example

- The language $a^n b^n c^n$ consists of strings such as $\varepsilon$, abc, aabbcc, aaabbbccc, aaaabbbbc, 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 $a^n b^n c^n$

\[
\begin{align*}
\text{s(Count)} &\rightarrow \text{as(Count)}, \text{bs(Count)}, \text{cs(Count)}. \\
\text{as}(0) &\rightarrow [].
\end{align*}
\]

\[
\begin{align*}
\text{as}(\text{succ(Count)}) &\rightarrow [a], \text{as(Count)}.
\end{align*}
\]

\[
\begin{align*}
\text{bs}(0) &\rightarrow [].
\end{align*}
\]

\[
\begin{align*}
\text{bs}(\text{succ(Count)}) &\rightarrow [b], \text{bs(Count)}.
\end{align*}
\]

\[
\begin{align*}
\text{cs}(0) &\rightarrow [].
\end{align*}
\]

\[
\begin{align*}
\text{cs}(\text{succ(Count)}) &\rightarrow [c], \text{cs(Count)}.
\end{align*}
\]

?- \text{s(Count, X, [])}.
Count = 0, 
X = [] ;
Count = \text{succ(0)}, 
X = [a, b, c] ;
Count = \text{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 $a^n b^n c^n$

\[
\begin{align*}
\text{s(Count)} & \rightarrow \text{as(Count)}, \text{bc(Count)}, \text{cs(Count)}. \\
\text{as(0)} & \rightarrow [\]. \\
\text{as(\text{NewCnt})} & \rightarrow [a], \text{as(\text{Cnt})}, \{\text{NewCnt is Cnt + 1}\}. \\
\text{bs(0)} & \rightarrow [\]. \\
\text{bs(\text{NewCnt})} & \rightarrow [b], \text{bs(\text{Cnt})}, \{\text{NewCnt is Cnt + 1}\}. \\
\text{cs(0)} & \rightarrow [\]. \\
\text{cs(\text{NewCnt})} & \rightarrow [c], \text{cs(\text{Cnt})}, \{\text{NewCnt is Cnt + 1}\}.
\end{align*}
\]
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

\[
s \rightarrow \text{np, vp.}
\]

\[
\text{np} \rightarrow \text{det, n.}
\]

\[
\text{vp} \rightarrow \text{v, np.}
\]

\[
\text{vp} \rightarrow \text{v.}
\]

\[
\text{det} \rightarrow \text{[the].}
\]

\[
\text{det} \rightarrow \text{[a].}
\]

\[
\text{n} \rightarrow \text{[woman].}
\]

\[
\text{n} \rightarrow \text{[man].}
\]

\[
\text{v} \rightarrow \text{[shoots].}
\]
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