CS 603: Programming Languages

Lecture 28

Spring 2004

Department of Computer Science
University of Alabama
Joel Jones

Overview

- More about Definite Clause Grammars
  - Adding arguments
  - Building parse trees

More on Definite Clause Grammars

- Taken from:
  - `http://www.coli.uni-sb.de/~kris/prolog-course/html/node66.html`
- The grammar from last time didn't handle pronouns, so how do we fix it?
- Extend the grammar *-or-*
- Use additional arguments

New Sentence DCG

- Add pronouns to support sentences like
  - "She shoots him" and "He shoots her"

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

Becomes:

```
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].
```

But what's wrong with the sentences generated by the new grammar?

Fixing Sentence DCG

- Accepts incorrect sentences, such as:
  - "A woman shoots she", "Her shoots she"
- Problem is there is no encoding of *subject* pronouns, such "she" and "he" or of *object pronouns*, such as "her" and "his"
- We can fix this with new rules

Pronouns using Rules

```
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_object --> [him].
pro_subject ---> [she].   pro_object --> [her].
```

But this seems pretty clumsy. We added a small feature and most of the rules had to change.

Pronouns using 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 grammar adds no new rules as compared to the first one.

Intuitively, the argument is used to carry which kind of noun phrase (np) is expected and which pronouns (pro) are appropriate for which noun phrases.

DCG Argument Implementation

```
s --> np,vp.
```

Is syntactic sugar for:

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

And

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

Is syntactic sugar for:

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

```
np(_) --> det,n.
np(X) --> pro(X).
```

Is syntactic sugar for:

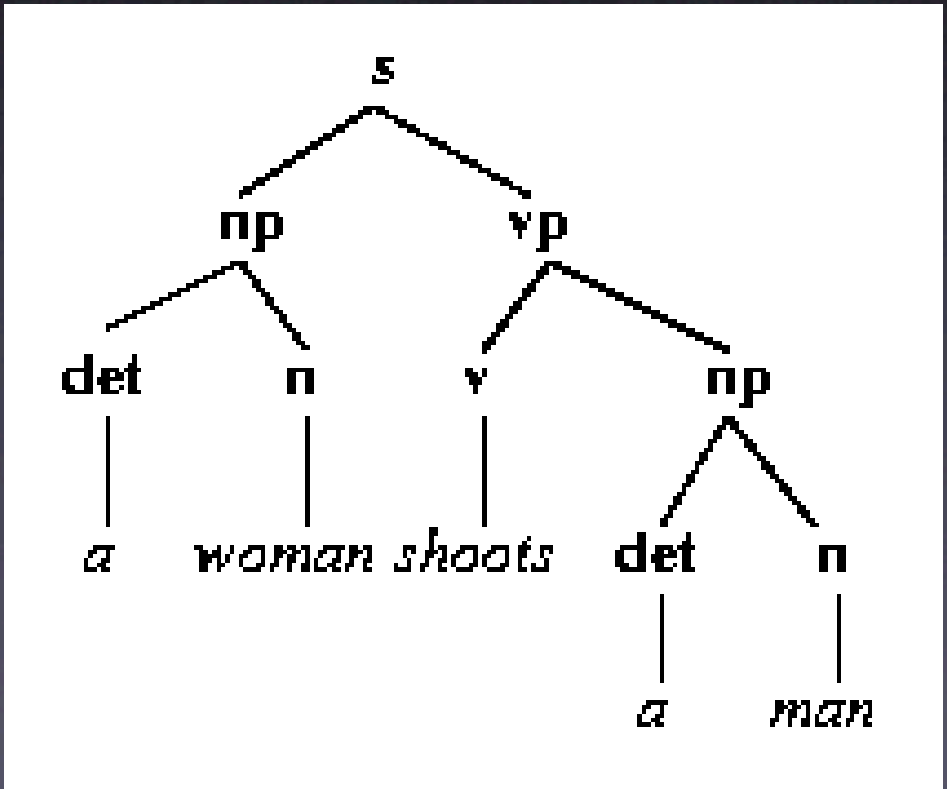
```
np(A,B,C) :-
    det(B,D),
    n(D,C).
np(A,B,C) :-
    pro(A,B,C.
```

New Sentence DCG at Work

```
? np(X,NP,[]).
X = _2625
NP = [the,woman];

X = _2625
NP = [the,man];

X = _2625
NP = [a,woman];

X = _2625
NP = [a,man];

X = subject
NP = [he];

X = subject
NP = [she];

X = object
NP = [him];

X = object
NP = [her];

no
```

Building Parse Trees

- How do we represent and build a parse tree?
  - In other words, for the tree below on the left, produce the term below on the right.




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

How do we solve this problem?

What did we do the last time we had a problem?

Building Parse Trees (cont.)

```
s(s(NP,VP))  -->  np(NP),
                  vp(VP).
np(np(DET,N))  -->  det(DET),
                    n(N).
vp(vp(V,NP))  -->  v(V),
                   np(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].
```

Build parse trees for the syntactic categories on the left-hand side out of the right-hand side. For example:

vp(vp(V,NP)) --> v(V),np(NP).

Then v might be instantiated to v(shoots) and NP will be instantiated to np(det(a),n(woman)).

So, s(T,[a,woman,shoots],[]). yields

T = (s(np(det(a),n(woman)),
vp(v(shoots)))

yes

DCGs for non-CFGs

- Consider the formal language:
  - $a^n b^n c^n$ - which consists of all non-null strings which consist of *n* '*a*'s followed by *n* '*b*'s followed by *n* '*c*'s

Is this language context-free?

DCGs for non-CFGs (cont.)

```
s(Count) --> ablock(Count),
             bblock(Count),
             cblock(Count).
ablock(0) --> [].
ablock(succ(Count)) --> [a],ablock(Count).
bblock(0) --> [].
bblock(succ(Count)) --> [b],bblock(Count).
cblock(0) --> [].
cblock(succ(Count)) --> [c],cblock(Count).
```

What is succ? A way of representing numbers without using numbers, also known as Peano numbers—succ(0) = 1, succ(succ(0)) = 2, etc.

DCGs for non-CFGs Example

```
?- s(Count,L,[]).

Count = 0
L = [] ;

Count = succ(0)
L = [a, b, c] ;

Count = succ(succ(0))
L = [a, a, b, b, c, c] ;

Count = succ(succ(succ(0)))
L = [a, a, a, b, b, b, c, c, c] ;
```

```
?- listing(s),
   listing(ablock).

s(A, B, C) :-
        ablock(A, B, D),
        bblock(A, D, E),
        cblock(A, E, C).

ablock(0, A, A).
ablock(succ(A), B, C) :-
        'C'(B, a, D),
        ablock(A, D, C).
```