Lecture 7: Definite Clause Grammars

- Theory
  - Introduce **context free grammars** and some related concepts
  - Introduce **definite clause grammars**, the Prolog way of working with context free grammars (and other grammars too)
- Exercises
  - Exercises of LPN: 7.1, 7.2, 7.3
  - Practical work

Context free grammars

- Prolog offers a special notation for defining **grammars**, namely DCGs or definite clause grammars
- So what is a grammar?
- We will answer this question by discussing **context free grammars**
- CFGs are a very powerful mechanism, and can handle most syntactic aspects of **natural languages** (such as English or Italian)

Example of a CFG

s → np vp
np → det n
vp → v np
vp → v
det → *the*
det → *a*
n → *man*
n → *woman*
v → *shoots*

Ingredients of a grammar

- The → symbol is used to define the <u>rules</u>
- The symbols **s**, **np**, **vp**, **det**, **n**, **v** are called the <u>non-terminal</u> symbols
- The symbols in italics are the <u>terminal</u> symbols:
  the, a, man, woman, shoots

s → np vp
np → det n
vp → v np
vp → v
det → *the*
det → *a*
n → *man*
n → *woman*
v → *shoots*

A little bit of linguistics

- The non-terminal symbols in this grammar have a traditional meaning in linguistics:
  - **np**: noun phrase
  - **vp**: verb phrase
  - **det**: determiner
  - **n**: noun
  - **v**: verb
  - **s**: sentence

More linguistics

- In a linguistic grammar, the non-terminal symbols usually correspond to **grammatical categories**
- In a linguistic grammar, the terminal symbols are called the **lexical items**, or simply words (a computer scientist might call them the **alphabet**)

Context free rules

- The grammar contains nine context free rules
- A context free rule consists of:
  - A single non-terminal symbol
  - followed by $\rightarrow$
  - followed by a finite sequence of terminal or non-terminal symbols

s $\rightarrow$ np vp
np $\rightarrow$ det n
vp $\rightarrow$ v np
vp $\rightarrow$ v
det $\rightarrow$ *the*
det $\rightarrow$ *a*
n $\rightarrow$ *man*
n $\rightarrow$ *woman*
v $\rightarrow$ *shoots*

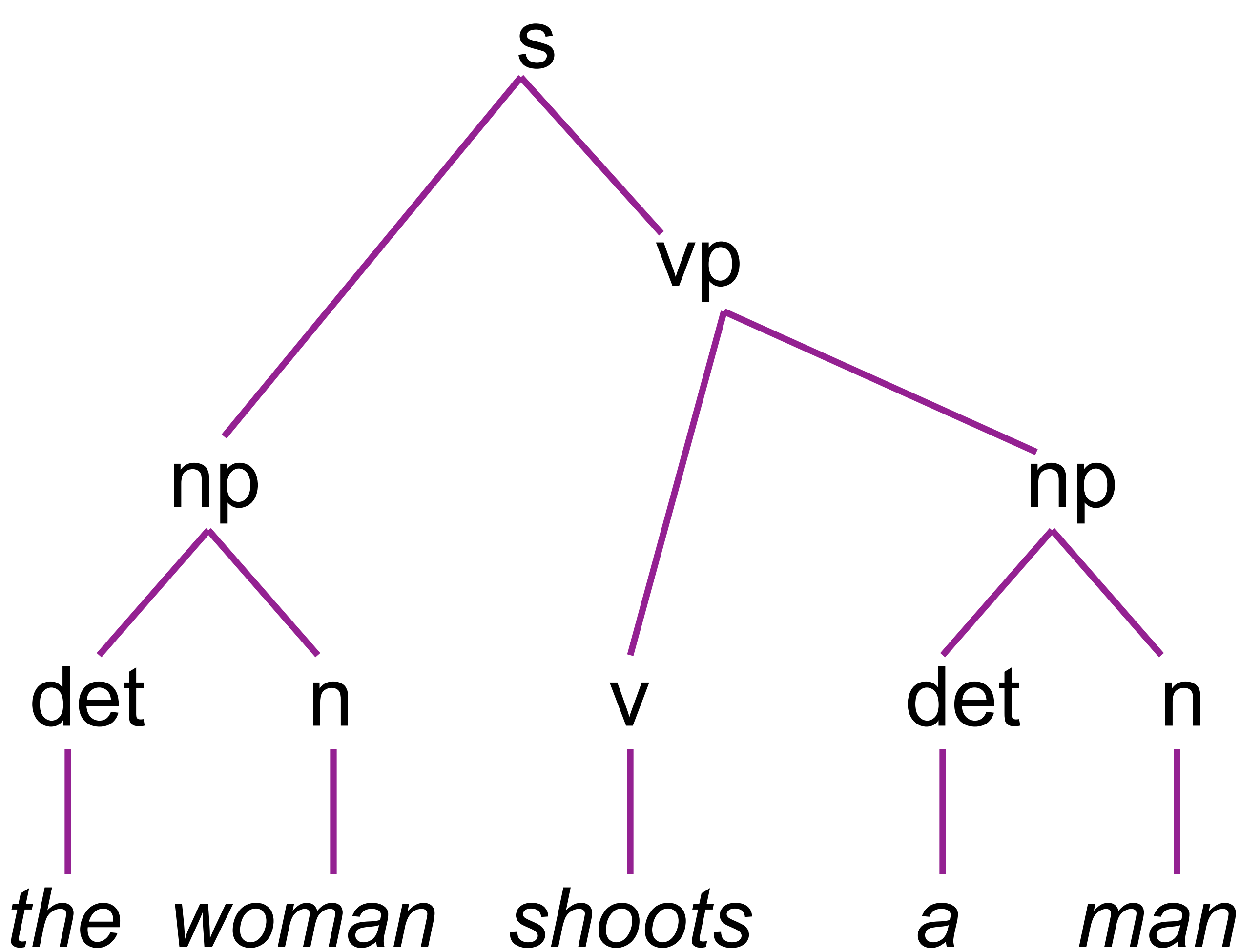
Grammar coverage

- Consider the following string:

  the woman shoots a man

- Is this string grammatical according to our grammar?
- And if it is, what syntactic structure does it have?

Syntactic structure



s → np vp
np → det n
vp → v np
vp → v
det → *the*
det → *a*
n → *man*
n → *woman*
v → *shoots*

Parse trees

- Trees representing the syntactic structure of a string are often called **parse trees**

- Parse trees are important:
  - They give us information about the string
  - They give us information about structure

Grammatical strings

- If we are given a string of words, and a grammar, and it turns out we can build a parse tree, then we say that the string is **grammatical** (with respect to the given grammar)
  – E.g., *the man shoots* is grammatical

Grammatical strings

- If we are given a string of words, and a grammar, and it turns out we can build a parse tree, then we say that the string is **grammatical** (with respect to the given grammar)
  - E.g., *the man shoots* is grammatical

- If we cannot build a parse tree, the given string is **ungrammatical** (with respect to the given grammar)
  - E.g., *a shoots woman* is ungrammatical

Generated language

- The **language generated by a grammar** consists of all the strings that the grammar classifies as grammatical

  For instance

  a woman shoots a man

  a man shoots

  belong to the language generated by our little grammar

Recogniser

- A context free **recogniser** is a program which correctly tells us whether or not a string belongs to the language generated by a context free grammar
- To put it another way, a **recogniser** is a program that correctly classifies strings as grammatical or ungrammatical

Information about structure

- But both in linguistics and computer science, we are not merely interested in whether a string is grammatical or not
- We also want to know *why* it is grammatical: we want to know what its structure is
- The parse tree gives us this structure

Parser

- A context free **parser** correctly decides whether a string belongs to the language generated by a context free grammar
- And it also tells us what its structure is
- To sum up:
  - A recogniser just says *yes* or *no*
  - A parser also gives us a parse tree

Context free language

- We know what a context free grammar is, but what is a context free language?
- Simply: a context free language is a language that can be generated by a context free grammar
- Some human languages are context free, some others are not
  - English and Italian are probably context free
  - Dutch and Swiss-German are not context free

Theory vs. Practice

- So far the theory, but how do we work with context free grammars in Prolog?
- Suppose we are given a context free grammar
  - How can we write a recogniser for it?
  - How can we write a parser for it?
- In this lecture we will look at how to define a recogniser

CFG recognition in Prolog

- We shall use lists to represent a sequence of tokens
  [a,woman,shoots,a,man]
- The rule **s → np vp** can be thought as concatenating an **np**-list with a **vp**-list resulting in an **s**-list
- We know how to concatenate lists in Prolog: using append/3
- So let's turn this idea into Prolog

CFG recognition using append/3

```
s(C):- np(A), vp(B), append(A,B,C).
np(C):- det(A), n(B), append(A,B,C).
vp(C):- v(A), np(B), append(A,B,C).
vp(C):- v(C).
det([the]).        det([a]).
n([man]).          n([woman]).        v([shoots]).
```

CFG recognition using append/3

```
s(C):- np(A), vp(B), append(A,B,C).
np(C):- det(A), n(B), append(A,B,C).
vp(C):- v(A), np(B), append(A,B,C).
vp(C):- v(C).
det([the]).        det([a]).
n([man]).          n([woman]).        v([shoots]).
```

```
?- s([the,woman,shoots,a,man]).
yes
?-
```

CFG recognition using append/3

```
s(C):- np(A), vp(B), append(A,B,C).
np(C):- det(A), n(B), append(A,B,C).
vp(C):- v(A), np(B), append(A,B,C).
vp(C):- v(C).
det([the]).        det([a]).
n([man]).          n([woman]).        v([shoots]).
```

```
?- s(S).
S = [the,man,shoots,the,man];
S = [the,man,shoots,the,woman];
S = [the,woman,shoots,a,man]
…
```

CFG recognition using append/3

```
s(C):- np(A), vp(B), append(A,B,C).
np(C):- det(A), n(B), append(A,B,C).
vp(C):- v(A), np(B), append(A,B,C).
vp(C):- v(C).
det([the]).       det([a]).
n([man]).         n([woman]).       v([shoots]).
```

```
?- np([the,woman]).
yes
?- np(X).
X = [the,man];
X = [the,woman]
```

Problems with this recogniser

- It doesn't use the input string to guide the search
- Goals such as np(A) and vp(B) are called with uninstantiated variables
- Moving the append/3 goals to the front is still not very appealing --- this will only shift the problem --- there will be a lot of calls to append/3 with uninstantiated variables

Difference lists

- A more efficient implementation can be obtained by using **difference lists**
- This is a sophisticated Prolog technique for representing and working with lists
- Examples:

[a,b,c]-[ ]	is the list [a,b,c]
[a,b,c,d]-[d]	is the list [a,b,c]
[a,b,c|T]-T	is the list [a,b,c]
X-X	is the empty list [ ]

CFG recognition using difference lists

```
s(A-C):- np(A-B), vp(B-C).
np(A-C):- det(A-B), n(B-C).
vp(A-C):- v(A-B), np(B-C).
vp(A-C):- v(A-C).
det([the|W]-W).          det([a|W]-W).
n([man|W]-W).   n([woman|W]-W).     v([shoots|W]-W).
```

CFG recognition using difference lists

```
s(A-C):- np(A-B), vp(B-C).
np(A-C):- det(A-B), n(B-C).
vp(A-C):- v(A-B), np(B-C).
vp(A-C):- v(A-C).
det([the|W]-W).          det([a|W]-W).
n([man|W]-W).   n([woman|W]-W).     v([shoots|W]-W).
```

```
?- s([the,man,shoots,a,man]-[ ]).
yes
?-
```

How does this work?

- Are there any tricks involved?
  Draw search tree!

CFG recognition using difference lists

```
s(A-C):- np(A-B), vp(B-C).
np(A-C):- det(A-B), n(B-C).
vp(A-C):- v(A-B), np(B-C).
vp(A-C):- v(A-C).
det([the|W]-W).        det([a|W]-W).
n([man|W]-W).   n([woman|W]-W).     v([shoots|W]-W).
```

```
?- s(X-[ ]).
S = [the,man,shoots,the,man];
S = [the,man,shoots,a,man];
….
```

Summary so far

- The recogniser using difference lists is a lot more efficient than the one using append/3
- However, it is not that easy to understand and it is a pain having to keep track of all those difference list variables
- It would be nice to have a recogniser as simple as the first and as efficient as the second
- This is possible: using DCGs

Definite Clause Grammars

- What are DCGs?
- Quite simply, a nice notation for writing grammars that hides the underlying difference list variables
- Let us look at three examples

DCGs: first example

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

DCGs: first example

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

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

DCGs: first example

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

```
?- s(X,[ ]).
S = [the,man,shoots,the,man];
S = [the,man,shoots,a,man];
….
```

What is going on?

- A DCG rule such as:

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

is really a syntactic variant of:

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

- DCGs simplify notation!

DCGs: second example

```
s --> s, conj, s.          s --> np, vp.
np --> det, n.             vp --> v, np.          vp --> v.

det --> [the].             det --> [a].
n --> [man].               n --> [woman].         v --> [shoots].
conj --> [and].            conj --> [or].         conj --> [but].
```

- We added some recursive rules to the grammar…
- What and how many sentences does this grammar generate?
- What does Prolog do with this DCG?

DCG without left-recursive rules

```
s --> simple_s, conj, s.
s --> simple_s.
simple_s --> np, vp.
np --> det, n.
vp --> v, np.
vp --> v.

det --> [the].          det --> [a].
n --> [man].            n --> [woman].          v --> [shoots].
conj --> [and].         conj --> [or].          conj --> [but].
```

DCGs are not magic!

- The moral: DCGs are a nice notation, but you cannot write arbitrary context-free grammars as a DCG and have it run without problems
- DCGs are ordinary Prolog rules in disguise
- So keep an eye out for left-recursion!

DCGs: third example

- We will define a DCG for a formal language
- A formal language is simply a set of strings
  - Formal languages are objects that computer scientist and mathematicians define and study
  - Natural languages are languages that human beings normally use to communicate
- We will define the language $a^n b^n$

DCGs: third example

- We will define the formal language $a^n b^n$

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

```
?- s([a,a,a,b,b,b],[ ]).
yes
?- s([a,a,a,a,b,b,b],[ ]).
no
```

DCGs: third example

- We will define the formal language $a^n b^n$

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

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

Exercises

- LPN 7.1
- LPN 7.2
- LPN 7.3

Summary of this lecture

- We explained the idea of grammars and context free grammars are
- We introduced the Prolog technique of using difference lists
- We showed that difference lists can be used to describe grammars
- Definite Clause Grammars is just a nice Prolog notation for programming with difference lists

Next lecture

- More Definite Clause Grammars
  - Examine two important capabilities offered by DCG notation
    - Extra arguments
    - Extra tests
  - Discuss the status and limitations of definite clause grammars