# Contents

# Context Free Grammars

- Prolog offers a special notation for defining **grammars**, namely DCGs or definite clause grammars
- What is a grammar?
- CFGs are a very powerful mechanism and can handle most syntactic aspects of natural languages (such as English or Italian)

```
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 *rules*
- The symbols `s`, `np`, `vp`, `det`, `n`, `v` are called the *non-terminal* symbols
- The symbols `the`, `a`, `man`, `woman`, `shoots` are the *terminal* symbols

## 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
- 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
  - Following by `->`
  - Followed by a finite sequence of terminal or non-terminal symbols

## 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?

## 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

## Grammatric 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 thestrings that the grammar classifies as ungrammatical

For instance

*a woman shoots a man a man shoots*

Belong to the language generated by out little grammar

## Recogniser

- A context free **recogniser** is a program which correctly tells us whether of 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 ow 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 strings

```
[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 lets turn this idea into Prolog

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

### Problems

- 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 obstained 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 `[]`

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

## Definite Clause Grammars

- What are DCGs?
- Quite simply, a nice notation for writing grammars that hides the underlying difference list variables

```
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], []).
true
```

- 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

## Formal Language

- A formal language is simpley 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 $a^n b^n$

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

?- s([a, a, a, b, b, b], []).
true
?- s([a, a, a, a, b, b, b], []).
false
```

## Extra Arguments

```
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].
```

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

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

- The DCG ignores some basic facts about English
  - *she* and *he* are *subject pronouns* are 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?

```
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, him], []).
true
?- s([she, shoots, he], []).
false
```

- Recall that the rules `s --> np, vp.` is really syntactic sugar for `s(A, B) :- np(A, C), vp(C, B).`

# Building Parse Trees

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

# Beyond Context Free Languages

- 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 $\mathrm{a}^n\mathrm{b}^n\mathrm{c}^n \backslash \varepsilon$
- This is not context free

```
s(Count) --> as(Count), bs(Count), cs(Count).

as(0) --> [].
as(NewCnt) --> [a], as(Cnt), {NewCnt is Cnt+1}.

cs(0) --> [].
cs(NewCnt) --> [c], cs(Cnt), {NewCnt is Cnt+1}.

cs(0) --> [].
cs(NewCnt) --> [c], cs(Cnt), {NewCnt is Cnt+1}.
```

## Seperating Rules and Lexicon

- One classic application of the extra goals of DCGs in computation linguistics in 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 seperate lexicon