



UPPSALA
UNIVERSITET

The CKY algorithm part I: Recognition

Syntactic analysis (5LN455)

2014-11-17

Sara Stymne

Department of Linguistics and Philology

Mostly based on slides from Marco Kuhlmann





UPPSALA
UNIVERSITET

Recap: Parsing



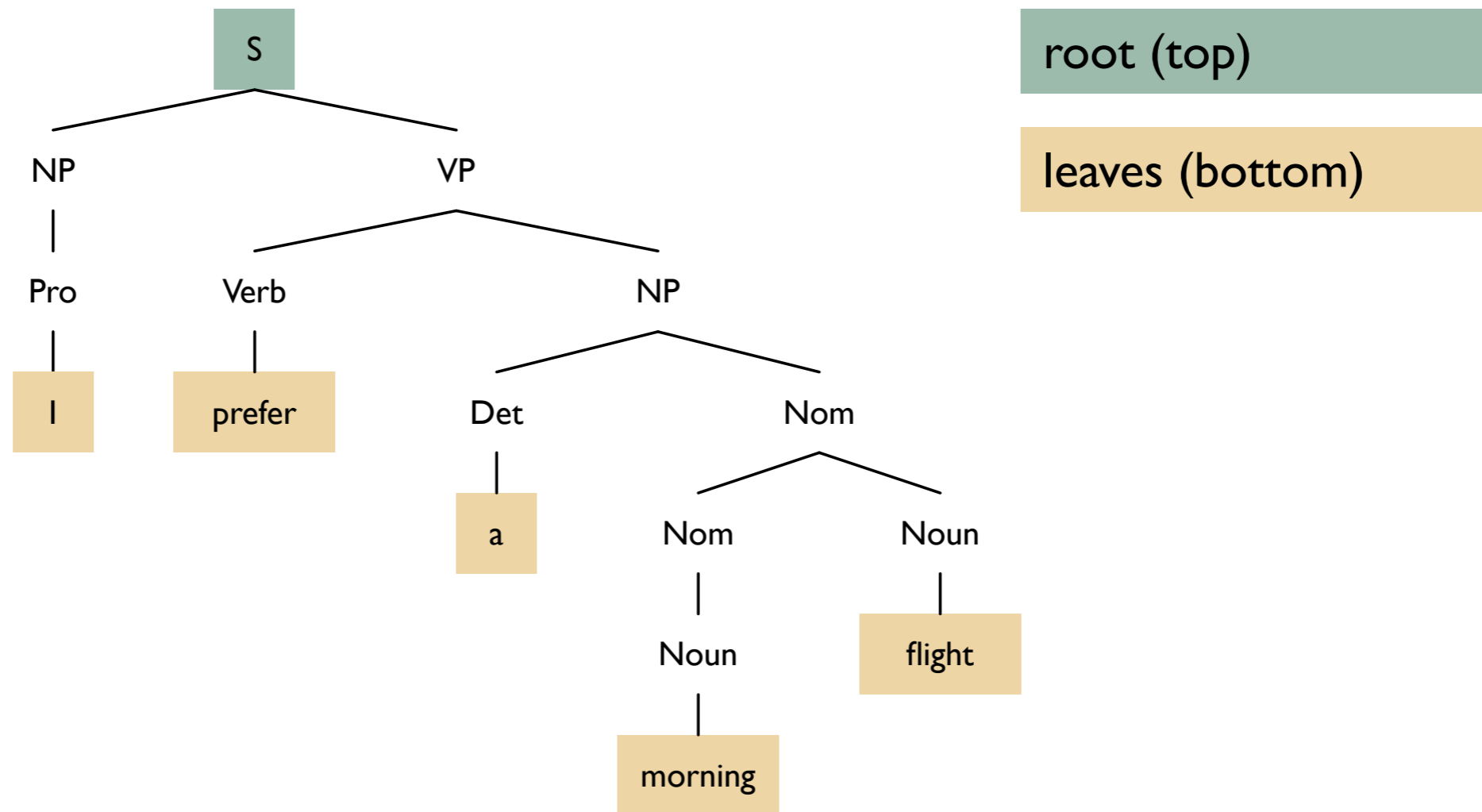
UPPSALA
UNIVERSITET

Parsing

The automatic analysis of a sentence
with respect to its syntactic structure.

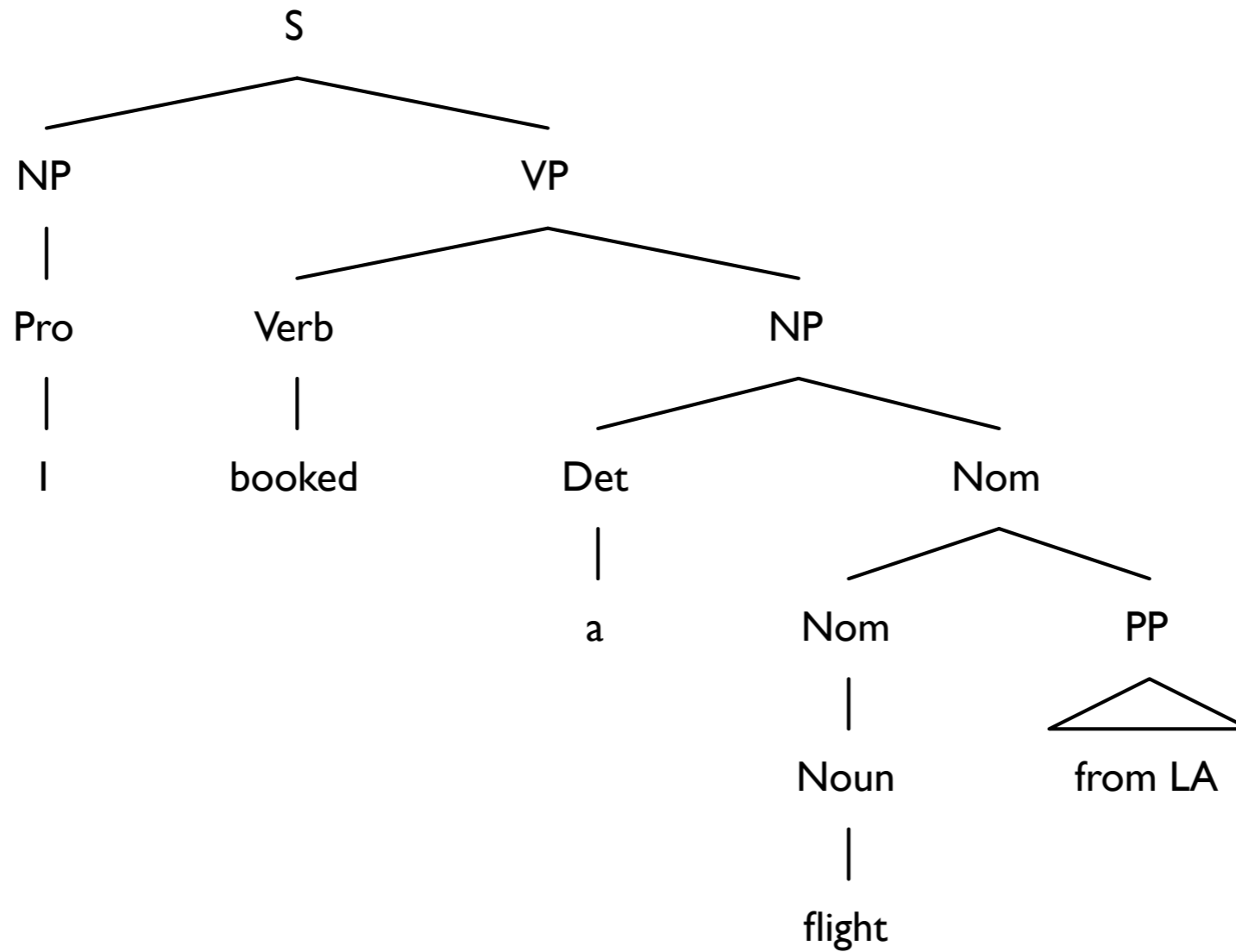


Phrase structure trees



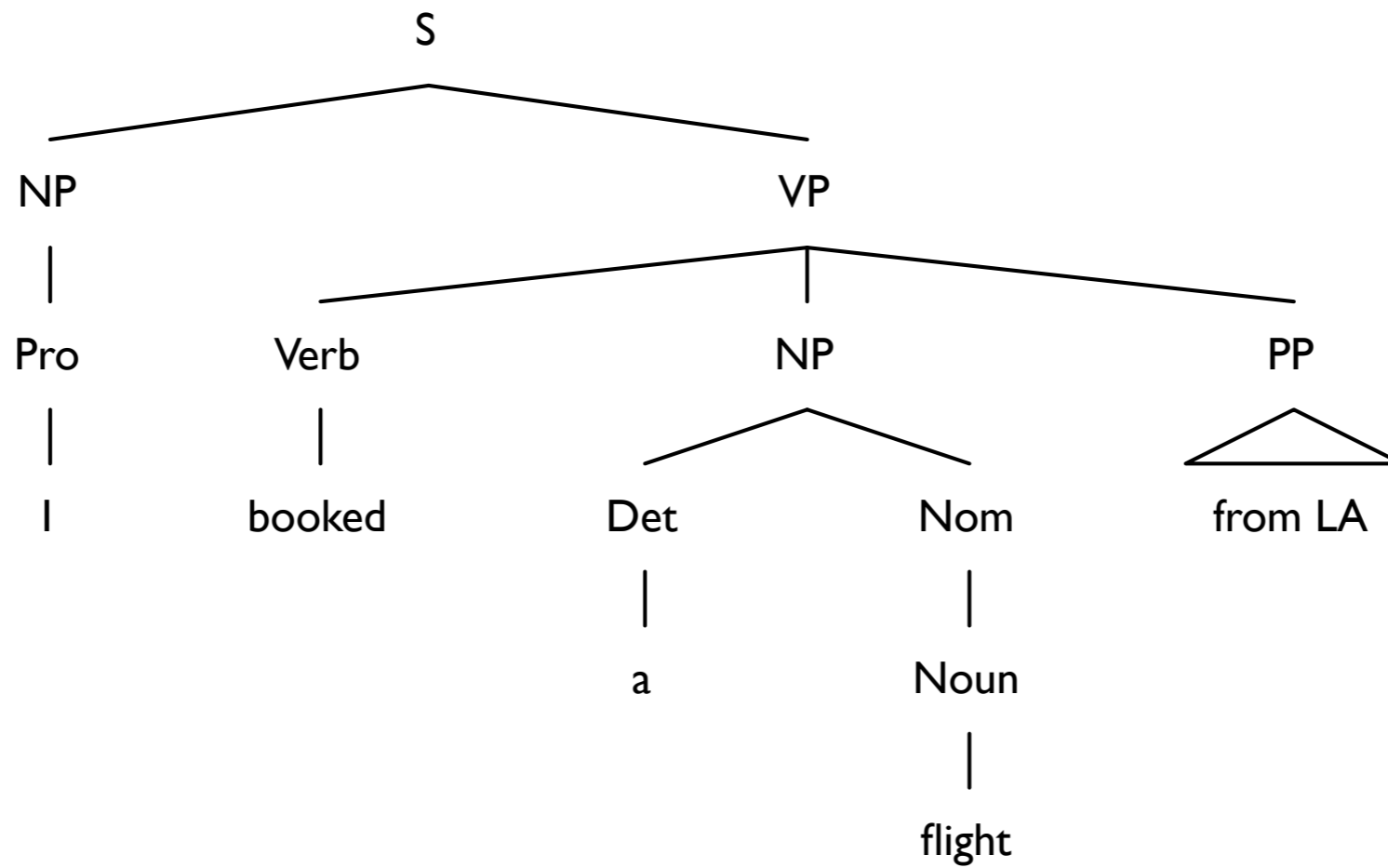


Ambiguity





Ambiguity





Parsing as search

- **Parsing as search:**
search through all possible parse trees
for a given sentence
- **bottom–up:**
build parse trees starting at the leaves
- **top–down:**
build parse trees starting at the root node



Overview of the CKY algorithm

- The CKY algorithm is an efficient bottom-up parsing algorithm for context-free grammars.
- It was discovered at least three (!) times and named after Cocke, Kasami, and Younger.
- It is one of the most important and most used parsing algorithms.



Applications

The CKY algorithm can be used to compute many interesting things.

Here we use it to solve the following tasks:

- **Recognition:**

Is there any parse tree at all?

- **Probabilistic parsing:**

What is the most probable parse tree?



Restrictions

- The CKY algorithm as we present it here can only handle rules that are at most binary:
 $C \rightarrow w_i$, $C \rightarrow C_1$, $C \rightarrow C_1 C_2$.
- This restriction is not a problem theoretically, but requires preprocessing (binarization) and postprocessing (debinarization).
- A parsing algorithm that does away with this restriction is Earley's algorithm (Lecture 5 and J&M 13.4.2).



Restrictions - details

- The CKY algorithm originally handles grammars in CNF (Chomsky normal form):
 $C \rightarrow w_i, C \rightarrow C_1 C_2, (S \rightarrow \varepsilon)$
- ε is normally not used in natural language grammars
- We also allow unit productions, $C \rightarrow C_1$
 - Extended CNF
 - Easy to integrate into CNF, easier grammar conversions



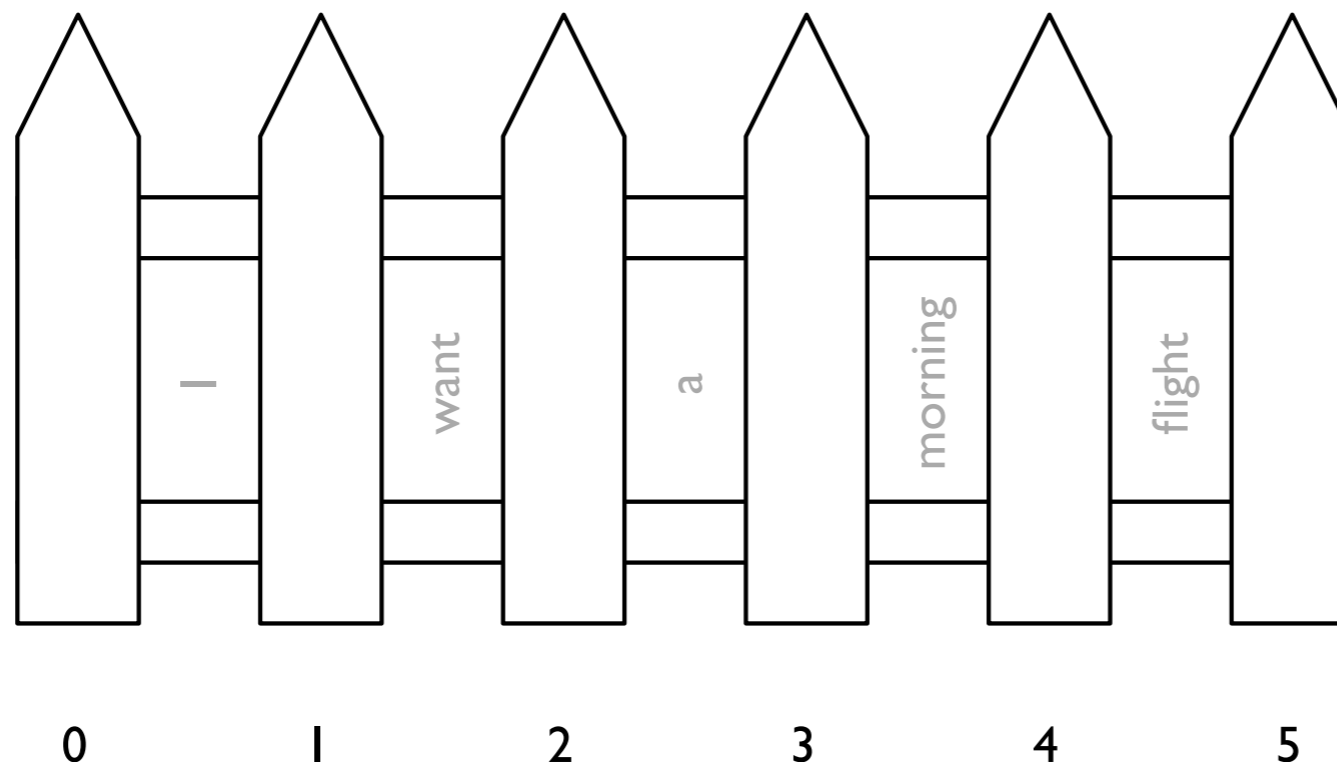
Conventions

- We are given a context-free grammar G and a sequence of word tokens $w = w_1 \dots w_n$.
- We want to compute parse trees of w according to the rules of G .
- We write S for the start symbol of G .



Fencepost positions

We view the sequence w as a fence with n holes,
one hole for each token w_i ,
and we number the fenceposts from 0 till n .





Structure

- Is there any parse tree at all?
- What is the most probable parse tree?



UPPSALA
UNIVERSITET

Recognition



Recognizer

A computer program that can answer the question

Is there any parse tree at all

for the sequence w according to the grammar G ?

is called a **recognizer**.

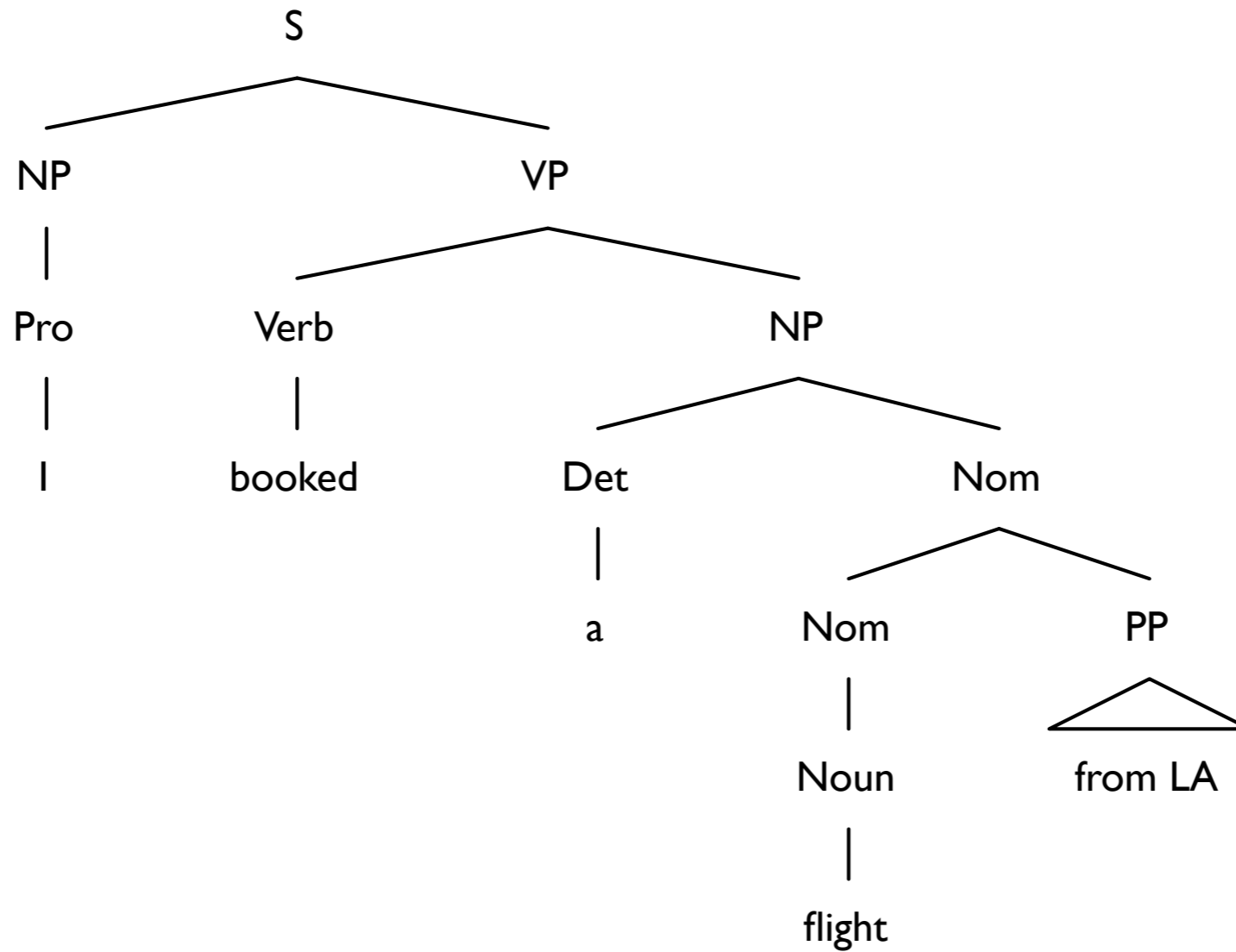
In practical applications one also wants

a concrete parse tree, not only an answer

to the question whether such a parse tree exists.



Parse trees





Preterminal rules and inner rules

- **preterminal rules:**

rules that rewrite a part-of-speech tag to a token, i.e. rules of the form $C \rightarrow w_i$.

Pro \rightarrow I, Verb \rightarrow booked, Noun \rightarrow flight

- **inner rules:**

rules that rewrite a syntactic category to other categories: $C \rightarrow C_1 C_2$, $C \rightarrow C_1$.

S \rightarrow NP VP, NP \rightarrow Det Nom, NP \rightarrow Pro



UPPSALA
UNIVERSITET

Recognition

Recognizing small trees

w_i



UPPSALA
UNIVERSITET

Recognition

Recognizing small trees

$$C \rightarrow w_i$$

w_i



UPPSALA
UNIVERSITET

Recognition

Recognizing small trees

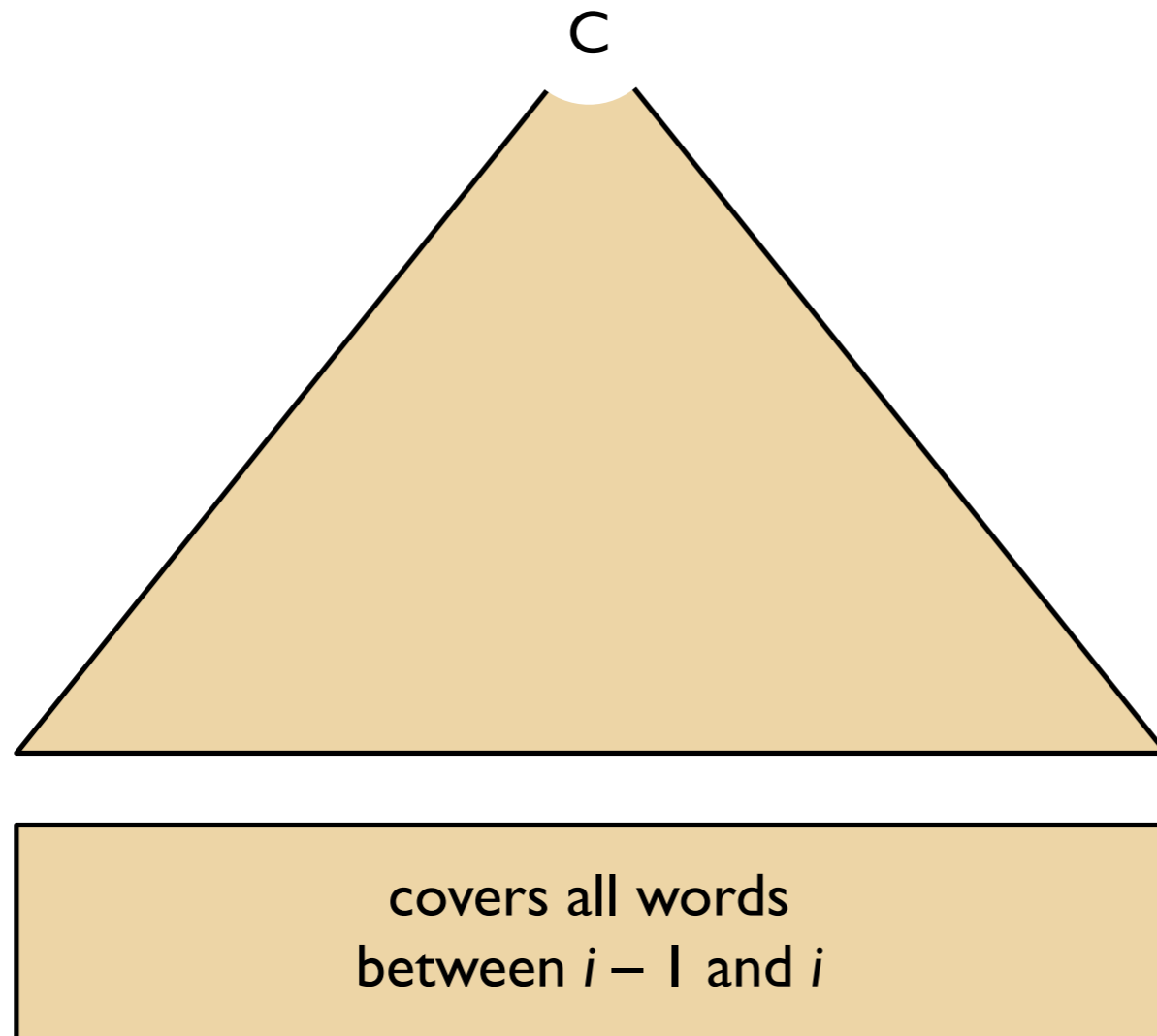




UPPSALA
UNIVERSITET

Recognition

Recognizing small trees

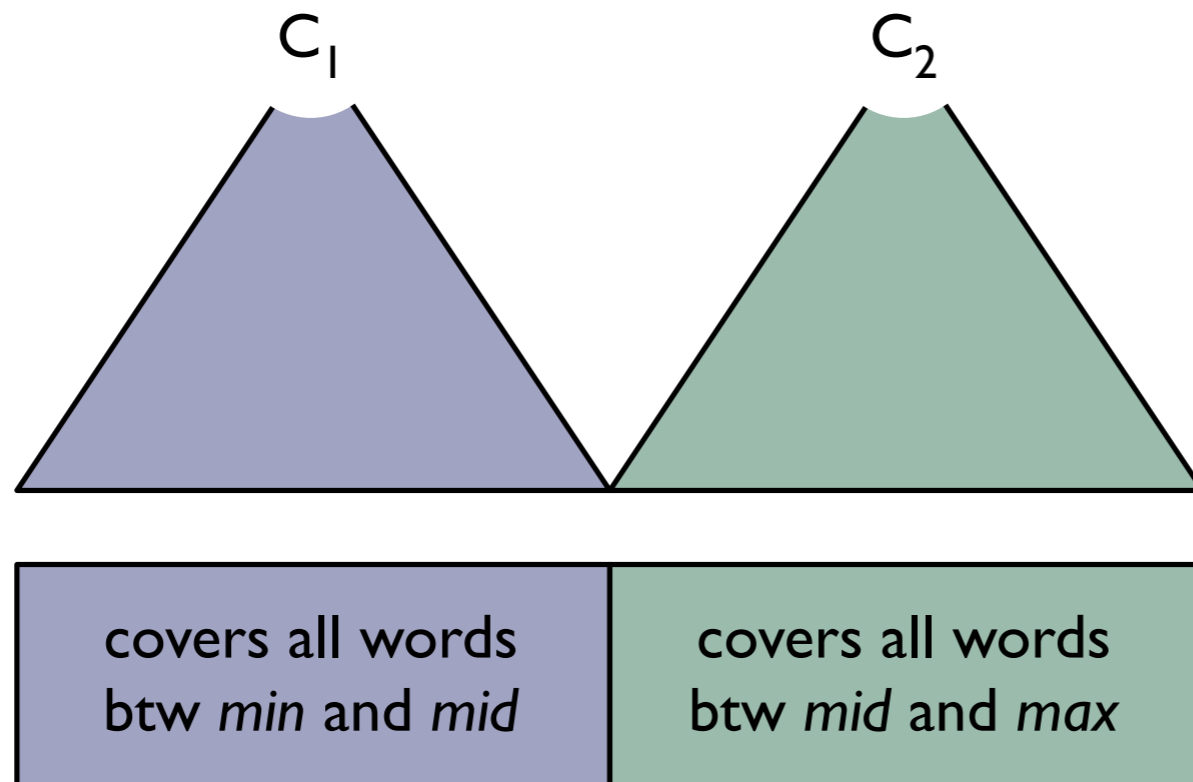




UPPSALA
UNIVERSITET

Recognition

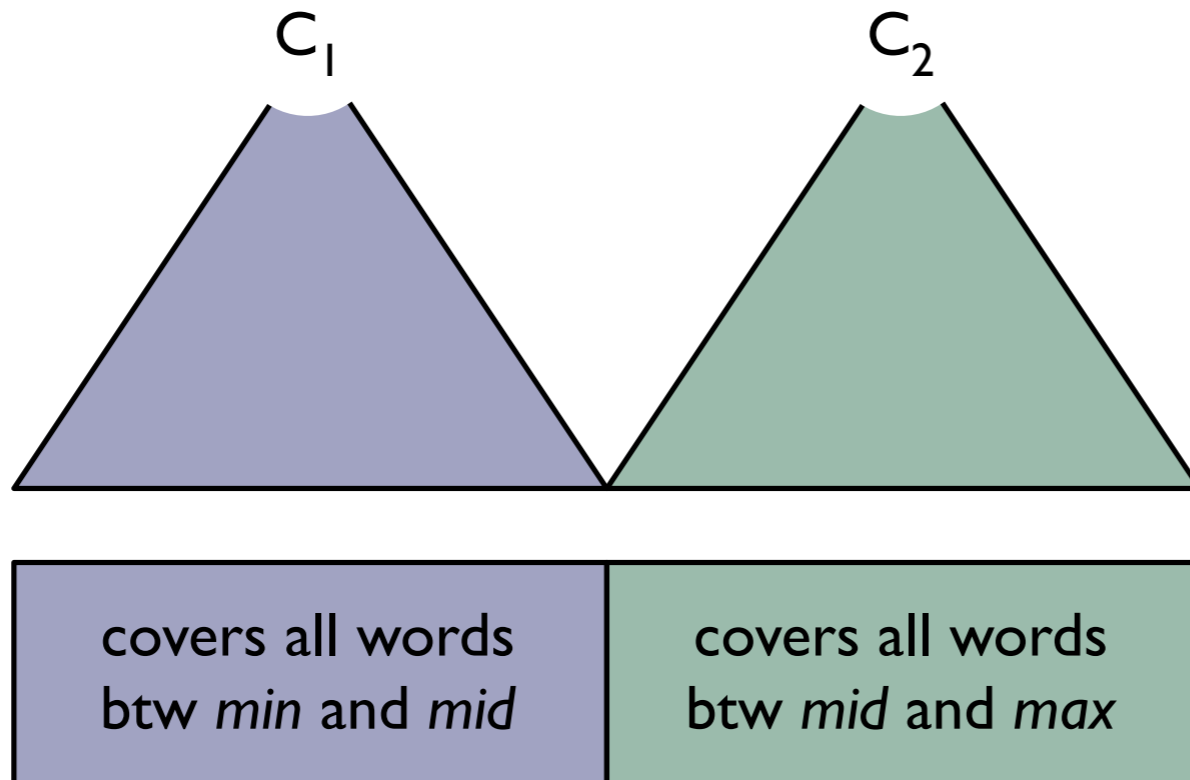
Recognizing big trees





Recognizing big trees

$$C \rightarrow C_1 C_2$$

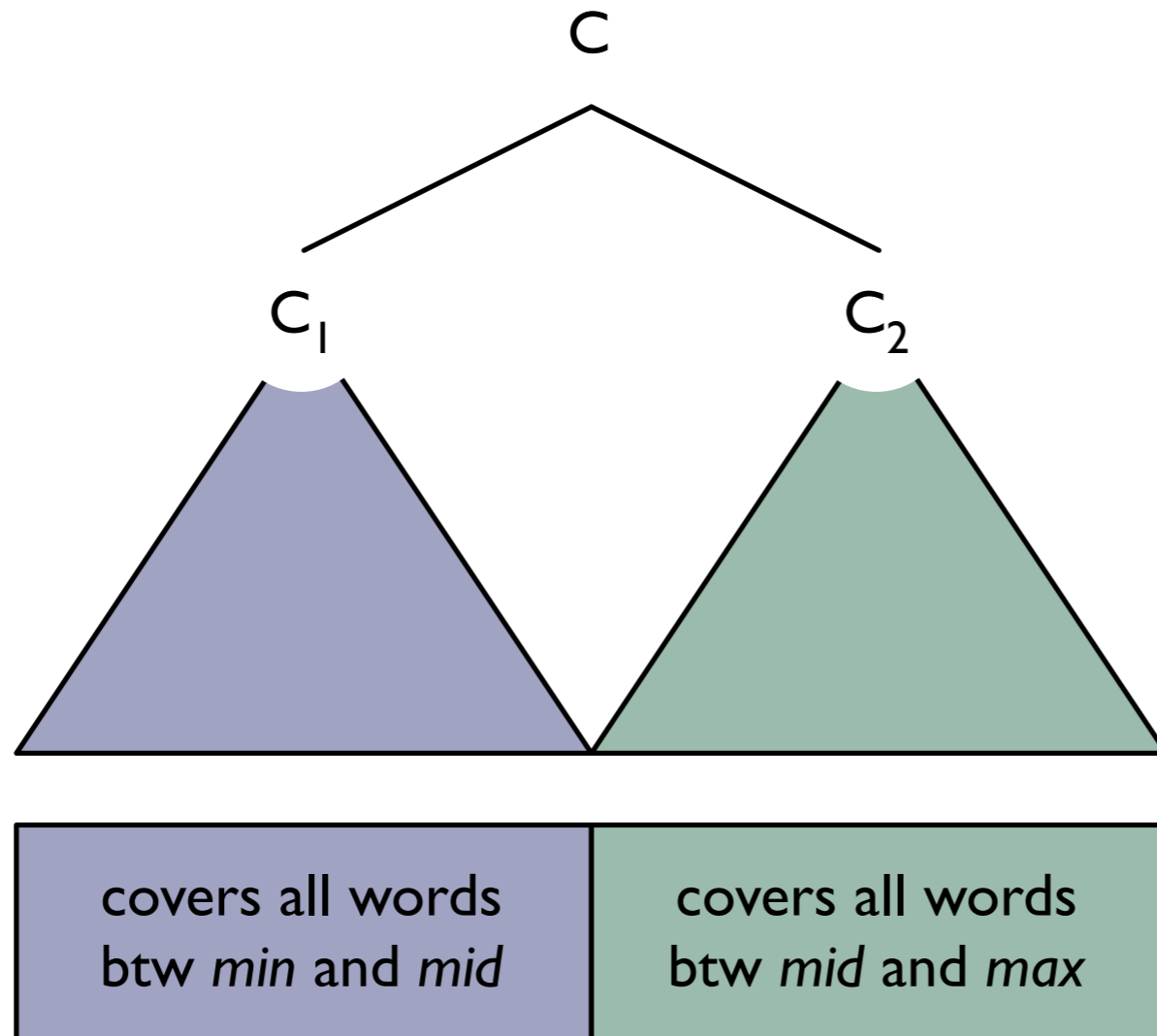




UPPSALA
UNIVERSITET

Recognition

Recognizing big trees

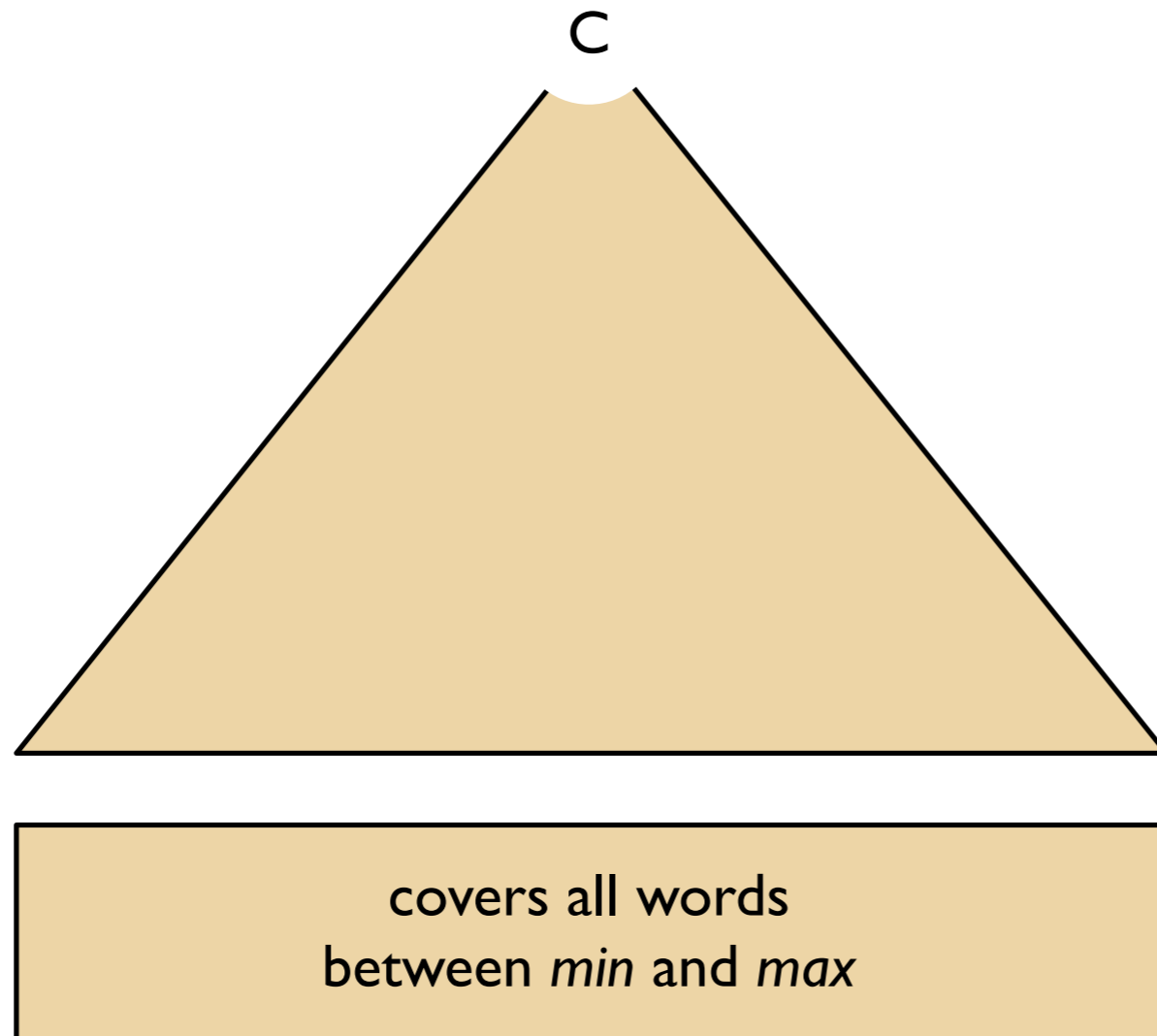




UPPSALA
UNIVERSITET

Recognition

Recognizing big trees





Questions

- How do we know that we have recognized that the input sequence is grammatical?
- How do we need to extend this reasoning in the presence of unary rules: $C \rightarrow C_1$?



Signatures

- The rules that we have just seen are independent of a parse tree's inner structure.
- The only thing that is important is how the parse tree looks from the 'outside'.
- We call this the **signature** of the parse tree.
- A parse tree with **signature** $[min, max, C]$ is one that covers all words between min and max and whose root node is labeled with C .



Questions

- What is the signature of a parse tree for the complete sentence?
- How many different signatures are there?
- Can you relate the runtime of the parsing algorithm to the number of signatures?



UPPSALA
UNIVERSITET

Implementation



Data structure

- The implementation represents signatures by means of a three-dimensional array *chart*.
- Initially, all entries of *chart* are set to *false*. (This is guaranteed by Java.)
- Whenever we have recognized a parse tree that spans all words between *min* and *max* and whose root node is labeled with *C*, we set the entry *chart*[*min*][*max*][*C*] to *true*.



UPPSALA
UNIVERSITET

Implementation

Preterminal rules

```
for each  $w_i$  from left to right
```

```
  for each preterminal rule  $C \rightarrow w_i$ 
```

```
    chart[i - 1][i][C] = true
```




Binary rules

```
for each max from 2 to n
  for each min from max - 2 down to 0
    for each syntactic category C
      for each binary rule C -> C1 C2
        for each mid from min + 1 to max - 1
          if chart[min][mid][C1] and chart[mid][max][C2] then
            chart[min][max][C] = true
```



Numbering of categories

- In order to use standard arrays, we need to represent syntactic categories by numbers.
- We write m for the number of categories; we number them from 0 till $m - 1$.
- We choose our numbers such that the start symbol S gets the number 0.



Skeleton code

```
// int n = number of words in the sequence  
  
// int m = number of syntactic categories in the grammar  
  
// int s = the (number of the) grammar's start symbol  
  
boolean[][][] chart = new boolean[n + 1][n + 1][m]  
  
// Recognize all parse trees built with with preterminal rules.  
  
// Recognize all parse trees built with inner rules.  
  
return chart[0][n][s]
```



Questions

- In what way is this algorithm bottom–up?
- Why is that property of the algorithm important?
- How do we need to extend the code in order to handle unary rules $C \rightarrow C_1$?



Summary

- The CKY algorithm is an efficient parsing algorithm for context-free grammars.
- Today: Recognizing whether there is any parse tree at all.
- Next time: Probabilistic parsing – computing the most probable parse tree.



Reading

- Recap of the introductory lecture:
J&M chapter 12.1-12.7 and 13.1-13.3
- CKY recognition:
J&M section 13.4.1
- CKY probabilistic parsing
J&M section 14.1-14.2