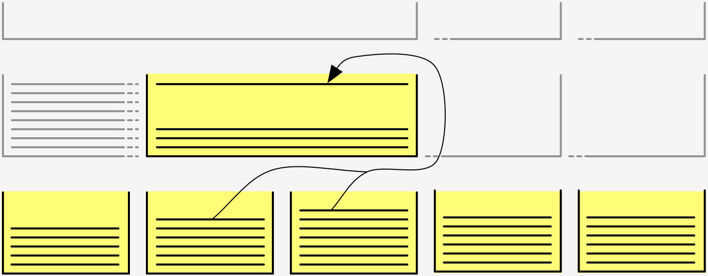
Statistical Machine Translation

-tree-based models (cont.)-

Artem Sokolov
Computerlinguistik
Universität Heidelberg
Sommersemester 2015

material from P. Koehn, S. Riezler, D. Altshuler

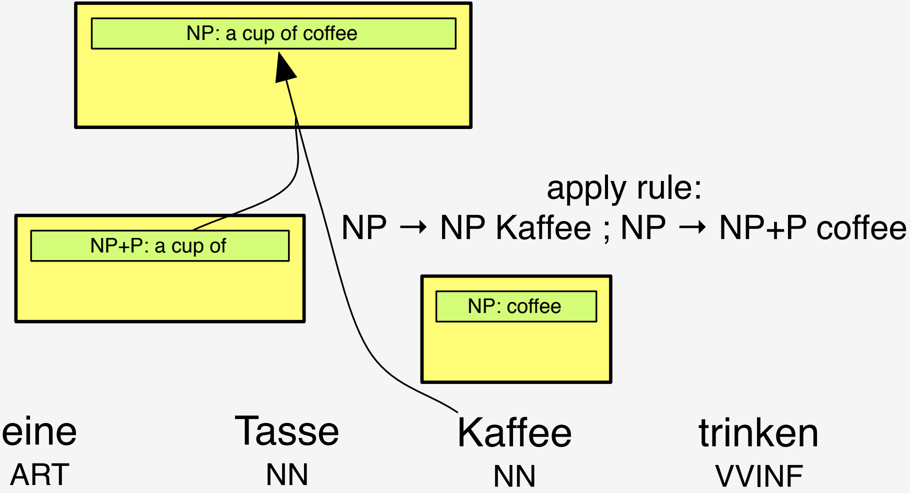
- For each span, a stack of (partial) translations is maintained
- Bottom-up: a higher stack is filled, once underlying stacks are complete



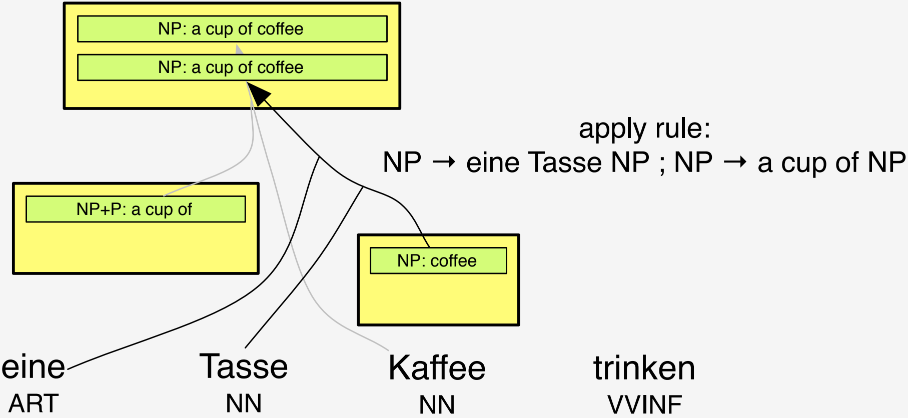
Input: Foreign sentence $\mathbf{f} = f_1, ...f_{l_f}$, with syntax tree
Output: English translation **e**

for all spans [start,end] (bottom up) **do**
  for all sequences s of hypotheses and words in span [start,end] **do**
    for all rules r **do**
      if rule r applies to chart sequence s **then**
        create new hypothesis c
        add hypothesis c to chart
      end if
    end for
  end for
end for
return English translation **e** from best hypothesis in span [0,l_f]

Applying rule creates new hypothesis



Another hypothesis



Both hypotheses are indistiguishable in future search

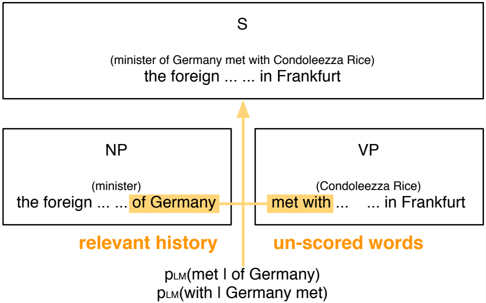
→ can be recombined

Hypotheses have to match in

- span of input words covered
- output constituent label
- first n–1 output words (not properly scored, since they lack context)
- last n–1 output words (still affect scoring of subsequently added words, just like in phrase-based decoding)

(n is the order of the n-gram language model)

When merging hypotheses, internal language model contexts are absorbed



Search space pruning

- recombination
- stack pruning

Algorithmic techniques

- prefix tree
- Earley's parsing algorithm
- cube pruning

- Number of hypotheses in each chart cell explodes

⇒ need to discard bad hypotheses
e.g., keep $n = 100$ best only

- Different stacks for different output constituent labels

⇒ keep at least m different (m=2,3,..)

- Many subspan sequences

  for all sequences s of hypotheses and words in span [start,end]

- Many rules

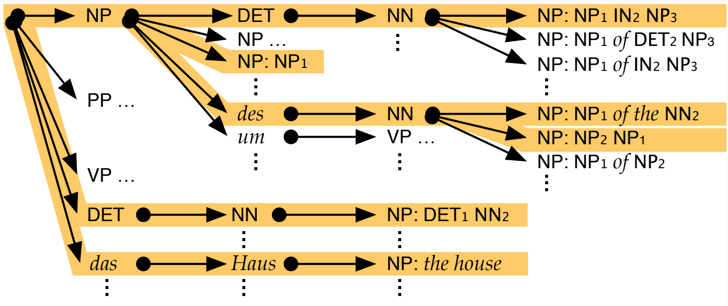
  for all rules r

- Checking if a rule applies not trivial

  rule r applies to chart sequence s

$\Rightarrow$ Unworkable

- Easy:
  - ➡ given a rule
  - ➡ check if and how it can be applied
- But there are too many rules (millions) to check them all
- Instead:
  - ➡ given the underlying chart cells and input words
  - ➡ find which rules apply



Highlighted Rules

$\mathrm{NP} \rightarrow \mathrm{NP}_1\ \mathrm{DET}_2\ \mathrm{NN}_3 \mid \mathrm{NP}_1\ \mathrm{IN}_2\ \mathrm{NN}_3$

$\mathrm{NP} \rightarrow \mathrm{NP}_1 \mid \mathrm{NP}_1$

$\mathrm{NP} \rightarrow \mathrm{NP}_1$ des $\mathrm{NN}_2 \mid \mathrm{NP}_1$ of the NN_2

$\mathrm{NP} \rightarrow \mathrm{NP}_1$ des $\mathrm{NN}_2 \mid \mathrm{NP}_2\ \mathrm{NP}_1$

$\mathrm{NP} \rightarrow \mathrm{DET}_1\ \mathrm{NN}_2 \mid \mathrm{DET}_1\ \mathrm{NN}_2$

$\mathrm{NP} \rightarrow$ das Haus | the house

- CFGs are ubiquitous for describing (syntactic) structure in NLP
- parsing algorithms are core of NL analysis systems
- recognition vs. parsing:
  - ➡ recognition - deciding the membership in the language
  - ➡ parsing - recognition + producing a parse tree for it
- parsing has more time complexity than recognition
- an input may have exponentially many parses

- one of the earliest recognition and parsing algorithms
- standard CKY can only recognize languages defined by CFGs in Chomsky Normal Form (CNF).
- based on a dynamic programming

- considers every possible consecutive subsequence of letters and sets $K \in T[i,j]$ if the sequence of letters starting from i to j can be generated from the non-terminal K
- once it has considered sequences of length 1, it goes on to sequences of length 2, and so on
- for subsequences of length 2 and greater, it considers every possible partition of the subsequence into two halves, and checks to see if there is some production $A \to BC$ such that B matches the 1st half and C matches the 2nd half. If so, it records A as matching the whole subsequence
- once completed, the sentence is recognized by the grammar if the entire string is matched by the start symbol

- any portion of the input string spanning i to j can be split at k, and structure can then be built using sub-solutions spanning i to k and sub-solutions spanning k to j
- solution to problem $[i, j]$ can constructed from solution to sub problem $[i, k]$ and solution to sub problem $[k, j]$

CKY Algorithm for Deciding CFL

Consider the grammar G given by:

$S \rightarrow \varepsilon \mid AB \mid XB$

$T \rightarrow AB \mid XB$

$X \rightarrow AT$

$A \rightarrow a$

$B \rightarrow b$

CKY Algorithm for Deciding CFL

Now look at *aaabbb* :

$S \rightarrow \varepsilon \mid AB \mid XB$

$T \rightarrow AB \mid XB$

$X \rightarrow AT$

$A \rightarrow a$

$B \rightarrow b$

a	*a*	*a*	*b*	*b*	*b*

CKY Algorithm for Deciding CFL

1) Write variables for all length 1 substrings.

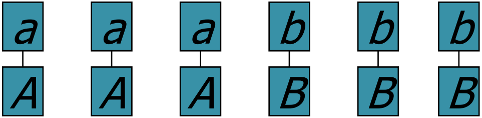
$S \rightarrow \varepsilon \mid AB \mid XB$

$T \rightarrow AB \mid XB$

$X \rightarrow AT$

$A \rightarrow a$

$B \rightarrow b$



CKY Algorithm for Deciding CFL

2) Write variables for all length 2 substrings.

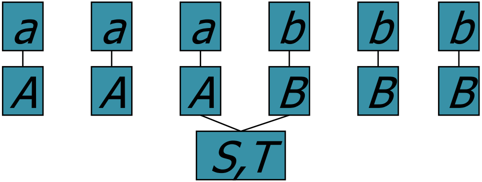
$S \rightarrow \varepsilon \mid AB \mid XB$

$T \rightarrow AB \mid XB$

$X \rightarrow AT$

$A \rightarrow a$

$B \rightarrow b$



CKY Algorithm for Deciding CFL

3) Write variables for all length 3 substrings.

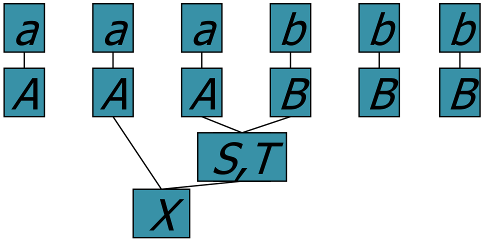
$S \rightarrow \varepsilon \mid AB \mid XB$

$T \rightarrow AB \mid XB$

$X \rightarrow AT$

$A \rightarrow a$

$B \rightarrow b$



CKY Algorithm for Deciding CFL

4) Write variables for all length 4 substrings.

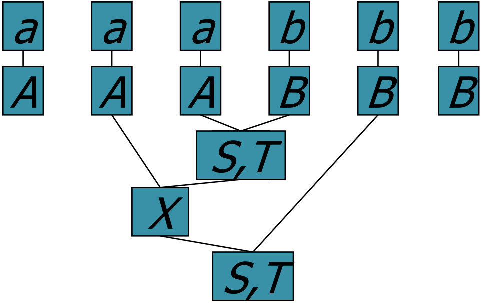
$S \rightarrow \varepsilon \mid AB \mid XB$

$T \rightarrow AB \mid XB$

$X \rightarrow AT$

$A \rightarrow a$

$B \rightarrow b$



CKY Algorithm for Deciding CFL

5) Write variables for all length 5 substrings.

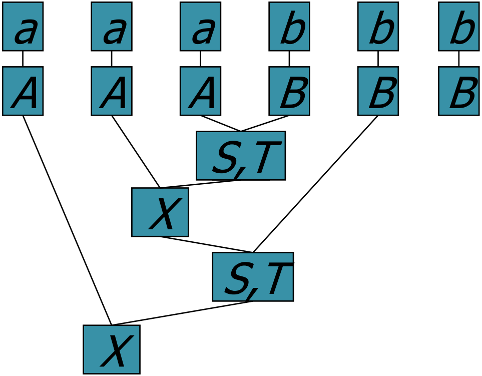
$S \rightarrow \varepsilon \mid AB \mid XB$

$T \rightarrow AB \mid XB$

$X \rightarrow AT$

$A \rightarrow a$

$B \rightarrow b$



CKY Algorithm for Deciding CFL

6) Write variables for all length 6 substrings.

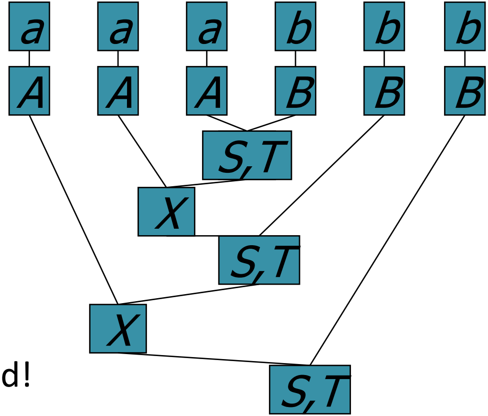
$S \rightarrow \varepsilon \mid AB \mid XB$

$T \rightarrow AB \mid XB$

$X \rightarrow AT$

$A \rightarrow a$

$B \rightarrow b$



S is included so *aaabbb* accepted!

The CKY Algorithm

function CKY (word w, grammar P) returns table

for i ← from 1 to LENGTH(w) do

  table[i-1, i] ← {A | A → w_i ∈ P }

for j ← from 2 to LENGTH(w) do

  for i ← from j-2 down to 0 do

    for k ← i + 1 to j – 1 do

      table[i,j] ← table[i,j] ∪ {A | A → BC ∈ P,

        B ∈ table[i,k], C ∈ table[k,j] }

If the start symbol *S* ∈ table[0,n] then w ∈ L(G)

CKY Algorithm for Deciding CFL

The table chart used by the algorithm:

j / i	1 a	2 a	3 a	4 b	5 b	6 b
0						
1						
2						
3						
4						
5						

CKY Algorithm for Deciding CFL

1. Variables for length 1 substrings.

i \ j	1 a	2 a	3 a	4 b	5 b	6 b
0	A					
1		A				
2			A			
3				B		
4					B	
5						B

CKY Algorithm for Deciding CFL

2. Variables for length 2 substrings.

j / i	1 a	2 a	3 a	4 b	5 b	6 b
0	A	-				
1		A	-			
2			A	S,T		
3				B	-	
4					B	-
5						B

CKY Algorithm for Deciding CFL

3. Variables for length 3 substrings.

i \ j	1 a	2 a	3 a	4 b	5 b	6 b
0	A	-	-			
1		A	-	X		
2			A	S,T	-	
3				B	-	-
4					B	-
5						B

CKY Algorithm for Deciding CFL

4. Variables for length 4 substrings.

i \ j	1 a	2 a	3 a	4 b	5 b	6 b
0	A	-	-	-		
1		A	-	X	S,T	
2			A	S,T	-	-
3				B	-	-
4					B	-
5						B

CKY Algorithm for Deciding CFL

5. Variables for length 5 substrings.

i \ j	1 a	2 a	3 a	4 b	5 b	6 b
0	A	-	-	-	X	
1		A	-	X	S,T	-
2			A	S,T	-	-
3				B	-	-
4					B	-
5						B

CKY Algorithm for Deciding CFL

6. Variables for *aaabbb*. ACCEPTED!

i \ j	1 a	2 a	3 a	4 b	5 b	6 b
0	A	-	-	-	X	S,T
1		A	-	X	S,T	-
2			A	S,T	-	-
3				B	-	-
4					B	-
5						B

CKY Space and Time Complexity

Time complexity:

- Three nested “for” loop each one of O(n) size.
- Lookup for *r* =|N| pair rules at each step.

Time complexity – $O(r^2n^3) = O(n^3)$

Space complexity:

- A three dimensions table at size n*n*r **or**
- A n*n table with lists up to size of *r*

Space complexity – $O(rn^2) = O(n^2)$

- doesnt require the grammar to be in CNF.
  - ➡ grammar intended to reflect actual structure of language
  - ➡ conversion to CNF completely destroys the parse structure
- efficiency:
  - ➡ usually moves left-to-right (prefix trees!)
  - ➡ faster than $O(n^3)$ for many grammars
  - ➡ uses a parse table as CKY, so can backtrack

- **dotted rule**
  - ➡ a partially constructed constituent, w/ the dot indicating what has been found and what is still predicted
  - ➡ generated from ordinary grammar rules (no CNF!)
- maintains a **set of states**, for each position in the input

The Dotted Rules

With dotted rules, an entry in the chart records:

- Which rule has been used in the analysis
- Which part of the rule has already been found (left of the dot).
- Which part is still predicted to be found and will combine into a complete parse (right of the dot).
- the start and end position of the material left of the dot.

<u>Example:</u> $A \rightarrow X_1 X_2 \ldots \bullet C \ldots X_m$

Parsing Operations

The Earley algorithm has three main operations:

Predictor: an incomplete entry looks for a symbol to the right of its dot. if there is no matching symbol in the chart, one is predicted by adding all matching rules with an initial dot.

Scanner: an incomplete entry looks for a symbol to the right of the dot. this prediction is compared to the input, and a complete entry is added to the chart if it matches.

Completer: a complete edge is combined with an incomplete entry that is looking for it to form another complete entry.

Parsing Operations

- **Predictor:** If state $[A \rightarrow X_1 ... \bullet C ... X_m, j] \in S_i$ then for every rule of the form $C \rightarrow Y_1 ... Y_k$, add to S_i the state $[C \rightarrow \bullet Y_1 ... Y_k, i]$
- **Scanner:** If state $[A \rightarrow X_1 ... \bullet a ... X_m, j] \in S_i$ and the next input word is $x_{i+1} = a$, then add to S_{i+1} the state $[A \rightarrow X_1 ... a \bullet ... X_m, j]$
- **Completer:** If state $[A \rightarrow X_1 ... X_m \bullet, j] \in S_i$ then for every state in S_j of form $[B \rightarrow X_1 ... \bullet A ... X_k, l]$, add to S_i the state $[B \rightarrow X_1 ... A \bullet ... X_k, l]$

The Earley Recognition Algorithm

The Main Algorithm: parsing input $w=w_1w_2 \ldots w_n$

1. S_0 = {[S → • P (0)]}
2. For $0 \le i \le n$ do:

   Process each item $s \in S_i$ in order by applying to it a *single* applicable operation among:

   (a) Predictor (adds new items to S_i)

   (b) Completer (adds new items to S_i)

   (c) Scanner (adds new items to S_{i+1})
3. If $S_{i+1} = \emptyset$ Reject the input.
4. If $i = n$ and [S→P• (0)] $\in S_n$ then Accept the input.

Earley Algorithm Example

Consider the following grammar for arithmetic expressions:

S → P (the start rule)

P → P + M

P → M

M → M * T

M → T

T → number

With the input: 2 + 3 * 4

Earley Algorithm Example

Sequence(0) • 2 + 3 * 4

(1) S → • P (0)   # start rule

Earley Algorithm Example

<u>Sequence(0)</u> • 2 + 3 * 4

(1)	S → • P (0)	# start rule
(2)	P → • P + M (0)	# predict from (1)
(3)	P → • M (0)	# predict from (1)

Earley Algorithm Example

<u>Sequence(0)</u> • 2 + 3 * 4

(1) S → • P (0) # start rule
(2) P → • P + M (0) # predict from (1)
(3) P → • M (0) # predict from (1)
(4) M → • M * T (0) # predict from (3)
(5) M → • T (0) # predict from (3)

Earley Algorithm Example

<u>Sequence(0)</u> • 2 + 3 * 4

(1)	S → • P (0)	# start rule
(2)	P → • P + M (0)	# predict from (1)
(3)	P → • M (0)	# predict from (1)
(4)	M → • M * T (0)	# predict from (3)
(5)	M → • T (0)	# predict from (3)
(6)	T → • number (0)	# predict from (5)

Earley Algorithm Example

<u>Sequence(1)</u> 2 • + 3 * 4

(1) T → number • (0) # scan from S(0)(6)

Earley Algorithm Example

Sequence(1) 2 • + 3 * 4

(1) T → number • (0) # scan from S(0)(6)

(2) M → T • (0) # complete from S(0)(5)

Earley Algorithm Example

<u>Sequence(1)</u> 2 • + 3 * 4

(1) T → number • (0) # scan from S(0)(6)
(2) M → T • (0) # complete from S(0)(5)
(3) M → M • * T (0) # complete from S(0)(4)
(4) P → M • (0) # complete from S(0)(3)

Earley Algorithm Example

<u>Sequence(1)</u> 2 • + 3 * 4

(1)	T → number • (0)	# scan from S(0)(6)
(2)	M → T • (0)	# complete from S(0)(5)
(3)	M → M • * T (0)	# complete from S(0)(4)
(4)	P → M • (0)	# complete from S(0)(3)
(5)	P → P • + M (0)	# complete from S(0)(2)
(6)	S → P • (0)	# complete from S(0)(1)

Earley Algorithm Example

Sequence(2) 2 + • 3 * 4

(1) P → P + • M (0) # scan from S(1)(5)

Earley Algorithm Example

<u>Sequence(2)</u> 2 + • 3 * 4

(1) P → P + • M (0) # scan from S(1)(5)
(2) M → • M * T (2) # predict from (1)
(3) M → • T (2) # predict from (1)

Earley Algorithm Example

Sequence(2) 2 + • 3 * 4

(1) P → P + • M (0) # scan from S(1)(5)
(2) M → • M * T (2) # predict from (1)
(3) M → • T (2) # predict from (1)
(4) T → • number (2) # predict from (3)

Earley Algorithm Example

Sequence(3) 2 + 3 • * 4

(1) T → number • (2)   # scan from S(2)(4)

Earley Algorithm Example

<u>Sequence(3)</u> 2 + 3 • * 4

(1) T → number • (2) # scan from S(2)(4)

(2) M → T • (2) # complete from S(2)(3)

Earley Algorithm Example

<u>Sequence(3)</u> 2 + 3 • * 4

(1) T → number • (2) # scan from S(2)(4)
(2) M → T • (2) # complete from S(2)(3)
(3) M → M • * T (2) # complete from S(2)(2)
(4) P → P + M • (0) # complete from S(2)(1)

Earley Algorithm Example

<u>Sequence(3)</u> 2 + 3 • * 4

(1)	T → number • (2)	# scan from S(2)(4)
(2)	M → T • (2)	# complete from S(2)(3)
(3)	M → M • * T (2)	# complete from S(2)(2)
(4)	P → P + M • (0)	# complete from S(2)(1)
(5)	P → P • + M (0)	# complete from S(0)(2)
(6)	S → P • (0)	# complete from S(0)(1)

Earley Algorithm Example

<u>Sequence(4)</u> 2 + 3 * • 4

(1) M → M * • T (2) # scan from S(3)(3)

Earley Algorithm Example

<u>Sequence(4)</u> 2 + 3 * • 4

(1) M → M * • T (2) # scan from S(3)(3)

(2) T → • number (4) # predict from (1)

Earley Algorithm Example

<u>Sequence(5)</u> 2 + 3 * 4 •

(1) T → number • (4) # scan from S(4)(2)

Earley Algorithm Example

<u>Sequence(5)</u> 2 + 3 * 4 •

(1) T → number • (4) # scan from S(4)(2)

(2) M → M * T • (2) # complete from S(4)(1)

Earley Algorithm Example

<u>Sequence(5)</u> 2 + 3 * 4 •

(1) T → number • (4) # scan from S(4)(2)
(2) M → M * T • (2) # complete from S(4)(1)
(3) M → M • * T (2) # complete from S(2)(2)
(4) P → P + M • (0) # complete from S(2)(1)

Earley Algorithm Example

<u>Sequence(5)</u> 2 + 3 * 4 •

(1) T → number • (4) # scan from S(4)(2)
(2) M → M * T • (2) # complete from S(4)(1)
(3) M → M • * T (2) # complete from S(2)(2)
(4) P → P + M • (0) # complete from S(2)(1)
(5) P → P • + M (0) # complete from S(0)(2)
(6) S → P • (0) # complete from S(0)(1)

Earley Algorithm Example

<u>Sequence(5)</u> 2 + 3 * 4 •

(1)	T → number • (4)	# scan from S(4)(2)
(2)	M → M * T • (2)	# complete from S(4)(1)
(3)	M → M • * T (2)	# complete from S(2)(2)
(4)	P → P + M • (0)	# complete from S(2)(1)
(5)	P → P • + M (0)	# complete from S(0)(2)
(6)	S → P • (0)	# complete from S(0)(1)

The state S→P• (0) represents a completed parse.

Finding the parse tree

Seq 0 • 2 + 3 * 4	Seq 1 2 • + 3 * 4	Seq 2 2 + • 3 * 4	Seq 3 2 + 3 • * 4	Seq 4 2 + 3 * • 4	Seq 5 2 + 3 * 4 •
S → • P (0)	T → '2' • (0)	P → P + • M (0)	T → '3' • (2)	M → M * • T (2)	T → '4' • (4)
P → • P + M (0)	M → T • (0)	M → • M * T (2)	M → T • (2)	T → • num (4)	M → M * T • (2)
P → • M (0)	M → M • * T (0)	M → • T (2)	M → M • * T (2)		M → M • * T (2)
M → • M * T (0)	P → M • (0)	T → • num (2)	P → P + M • (0)		P → P + M • (0)
M → • T (0)	P → P • + M (0)		P → P • + M (0)		P → P • + M (0)
T → • num (0)	S → P • (0)		S → P • (0)		S → P • (0)

Finding the parse tree

Seq 0 • 2 + 3 * 4	Seq 1 2 • + 3 * 4	Seq 2 2 + • 3 * 4	Seq 3 2 + 3 • * 4	Seq 4 2 + 3 * • 4	Seq 5 2 + 3 * 4 •
S → • P (0)	T → '2' • (0)	P → P + • M (0)	T → '3' • (2)	M → M * • T (2)	**T → '4' • (4)**
P → • P + M (0)	M → T • (0)	M → • M * T (2)	M → T • (2)	T → • num (4)	**M → M * T • (2)**
P → • M (0)	M → M • * T (0)	M → • T (2)	M → M • * T (2)		M → M • * T (2)
M → • M * T (0)	P → M • (0)	T → • num (2)	P → P + M • (0)		**P → P + M • (0)**
M → • T (0)	P → P • + M (0)		P → P • + M (0)		P → P • + M (0)
T → • num (0)	S → P • (0)		S → P • (0)		**S → P • (0)**

Finding the parse tree

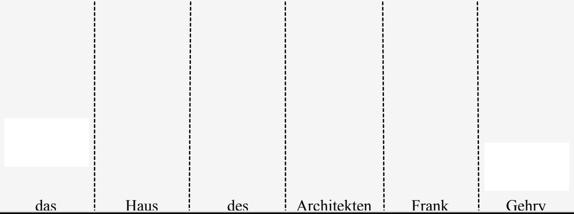
Seq 0 • 2 + 3 * 4	Seq 1 2 • + 3 * 4	Seq 2 2 + • 3 * 4	Seq 3 2 + 3 • * 4	Seq 4 2 + 3 * • 4	Seq 5 2 + 3 * 4 •
S → • P (0)	T → ‘2’ • (0)	P → P + • M (0)	**T → ‘3’ • (2)**	M → M * • T (2)	**T → ‘4’ • (4)**
P → • P + M (0)	M → T • (0)	M → • M * T (2)	**M → T • (2)**	T → • num (4)	**M → M * T • (2)**
P → • M (0)	M → M • * T (0)	M → • T (2)	M → M • * T (2)		M → M • * T (2)
M → • M * T (0)	P → M • (0)	T → • num (2)	P → P + M • (0)		**P → P + M • (0)**
M → • T (0)	P → P • + M (0)		P → P • + M (0)		P → P • + M (0)
T → • num (0)	S → P • (0)		S → P • (0)		**S → P • (0)**

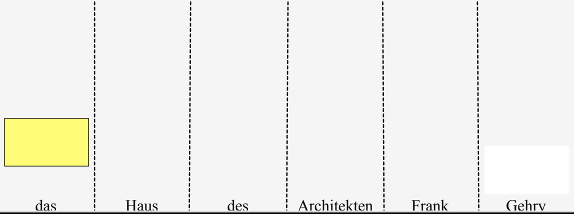
Finding the parse tree

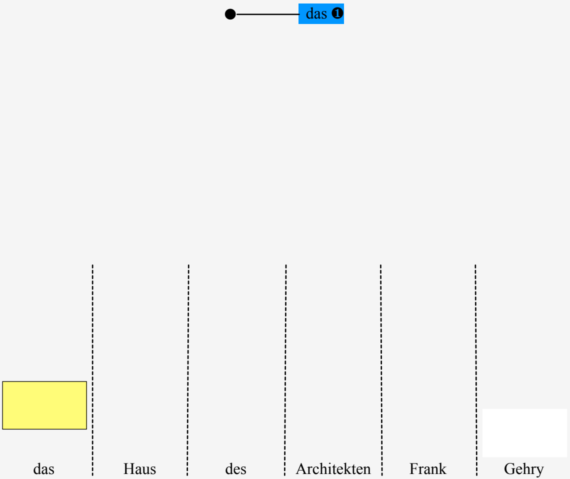
Seq 0 • 2 + 3 * 4	Seq 1 2 • + 3 * 4	Seq 2 2 + • 3 * 4	Seq 3 2 + 3 • * 4	Seq 4 2 + 3 * • 4	Seq 5 2 + 3 * 4 •
S → • P (0)	**T → ‘2’ • (0)**	P → P + • M (0)	**T → ‘3’ • (2)**	M → M * • T (2)	**T → ‘4’ • (4)**
P → • P + M (0)	**M → T • (0)**	M → • M * T (2)	**M → T • (2)**	T → • num (4)	**M → M * T • (2)**
P → • M (0)	M → M • * T (0)	M → • T (2)	M → M • * T (2)		M → M • * T (2)
M → • M * T (0)	**P → M • (0)**	T → • num (2)	P → P + M • (0)		**P → P + M • (0)**
M → • T (0)	P → P • + M (0)		P → P • + M (0)		P → P • + M (0)
T → • num (0)	S → P • (0)		S → P • (0)		**S → P • (0)**

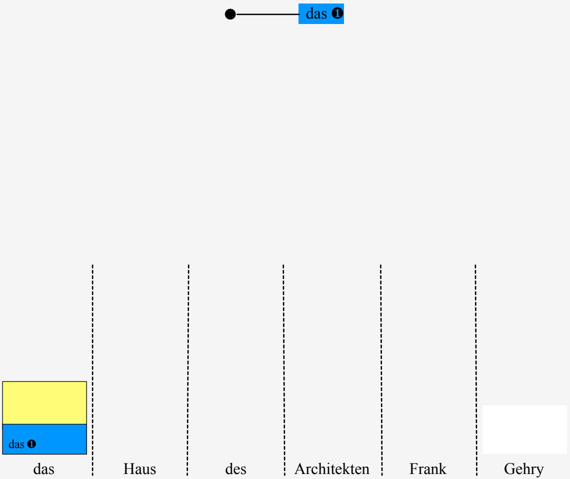
```
Input: Foreign sentence f = f_1,...f_{l_f}, with syntax tree
Output: English translation e
for i=0 .. length(f)-1 do // initialize chart
  store pointer to initial node in prefix tree in span [i,i]
end for
for l=1..l_f do // build chart from the bottom up
 for start=0 .. l_f - l do // beginning of span
   end = start+l
   for midpoint=start .. end-1 do
     for all dotted rules d in span [start,midpoint] do
       for all distinct head node nonterminals or input words h covering
       span [midpoint+1,end] do
         if extension d→ h exists in prefix tree then
           d_new = d→ h
           for all complete rules at d_new do
             apply rules
             store chart entries in span [start,end]
           end for
           if extension exist for d_new then
             store d_new in span [start,end] // new dotted rule
           end if
         end if
       end for
     end for
   end for
 end for
end for
return English translation e from best chart entry in span [0,l_f]
```

Finding Applicable Rules in Prefix Tree

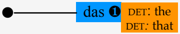


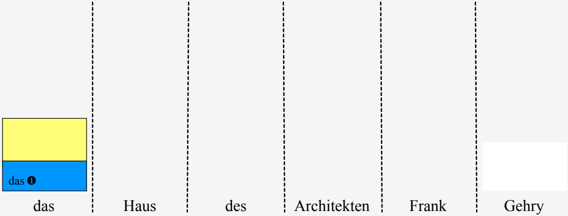

das
Haus
des
Architekten
Frank
Gehry


das ❶
das
Haus
des
Architekten
Frank
Gehry

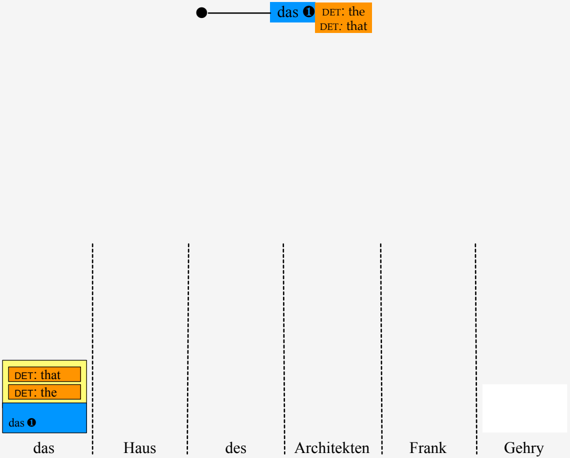

das ❶
das ❶
das
Haus
des
Architekten
Frank
Gehry

Checking if Dotted Rule has Translations

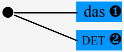


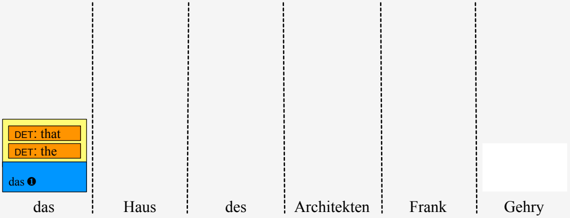


Applying the Translation Rules

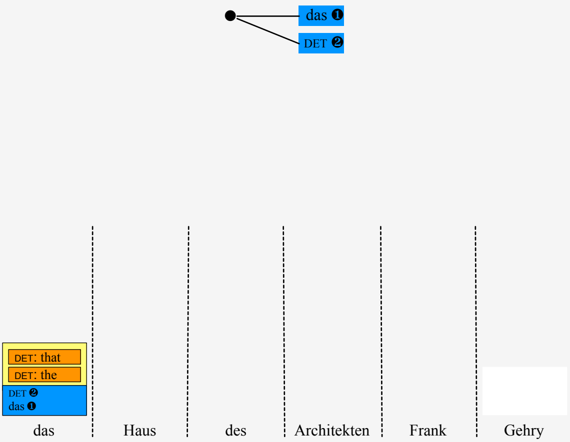


Looking up Constituent Label in Prefix Tree

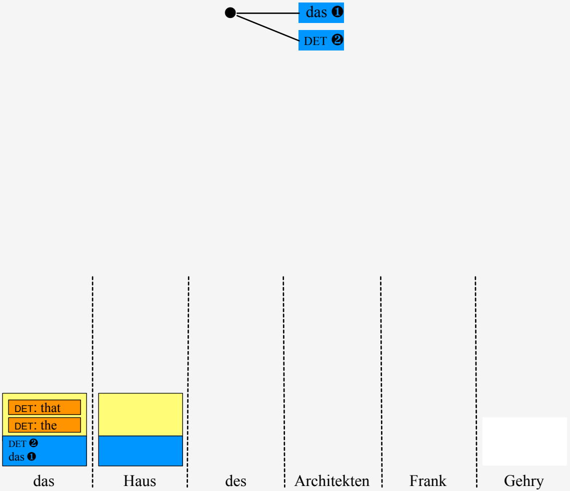




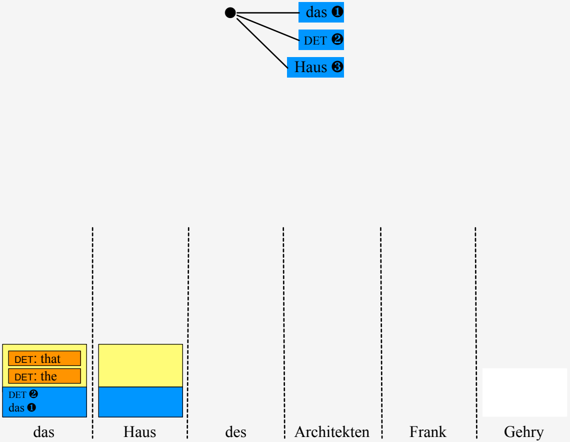
Add to Span's List of Dotted Rules



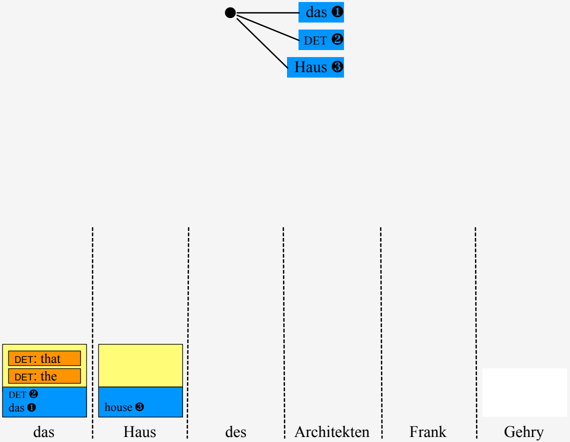
Moving on to the Next Cell



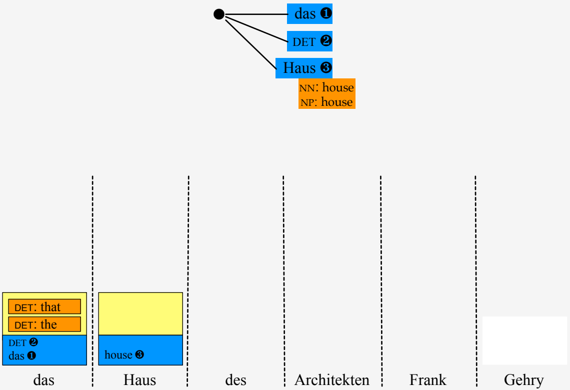
Looking up Rules in the Prefix Tree

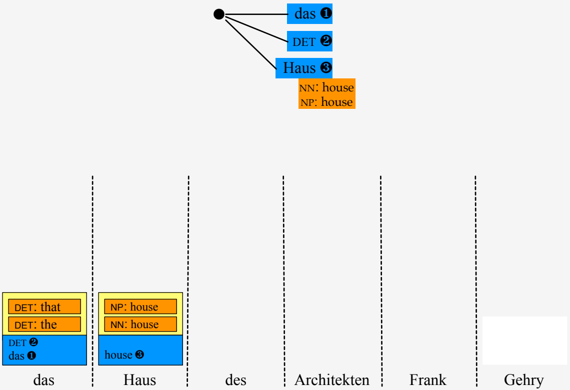


Taking Note of the Dotted Rule

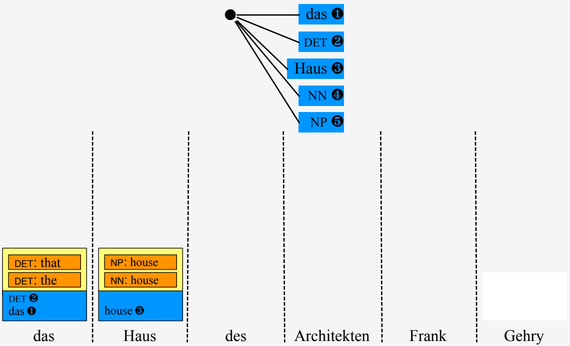


Checking if Dotted Rule has Translations

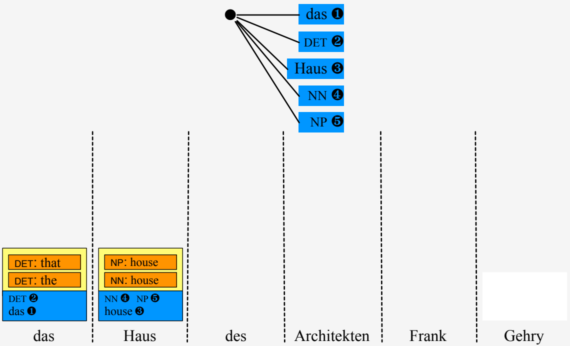


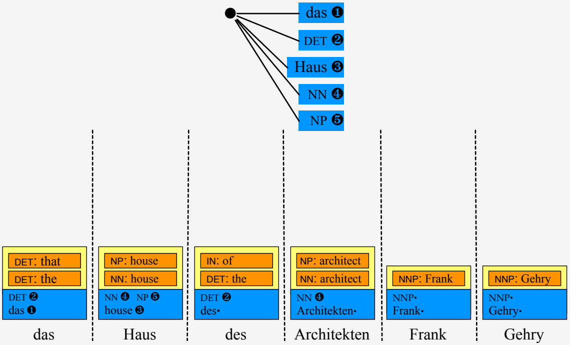

das ❶
DET ❷
Haus ❸
NN: house
NP: house
DET: that
DET: the
DET ❷
das ❶
NP: house
NN: house
house ❸
das
Haus
des
Architekten
Frank
Gehry

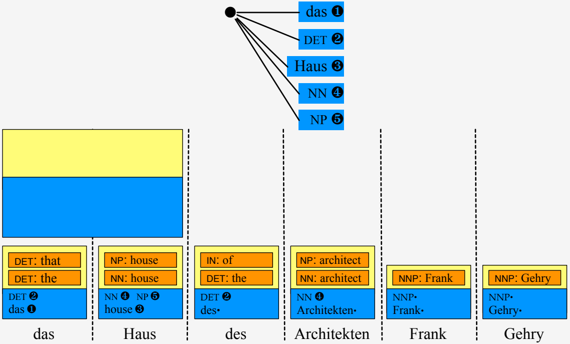
Looking up Constituent Label in Prefix Tree



Add to Span's List of Dotted Rules



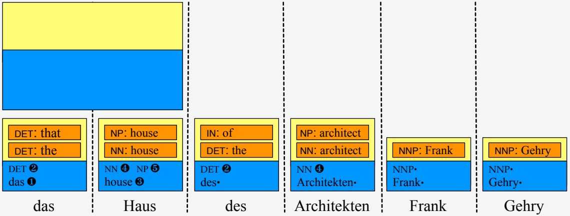

das ❶
DET ❷
Haus ❸
NN ❹
NP ❺
DET: that
DET: the
DET ❷
das ❶
NP: house
NN: house
NN ❹ NP ❺
house ❸
IN: of
DET: the
DET ❷
des•
NP: architect
NN: architect
NN ❹
Architekten•
NNP: Frank
NNP•
Frank•
NNP: Gehry
NNP•
Gehry•
das
Haus
des
Architekten
Frank
Gehry

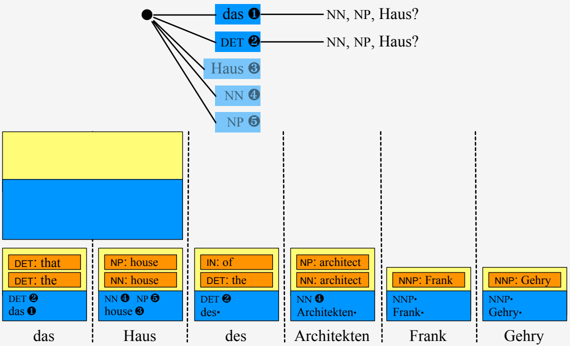

das ❶
DET ❷
Haus ❸
NN ❹
NP ❺
DET: that
DET: the
DET ❷
das ❶
NP: house
NN: house
NN ❹ NP ❺
house ❸
IN: of
DET: the
DET ❷
des•
NP: architect
NN: architect
NN ❹
Architekten•
NNP: Frank
NNP•
Frank•
NNP: Gehry
NNP•
Gehry•
das
Haus
des
Architekten
Frank
Gehry

Cannot consume multiple words at once

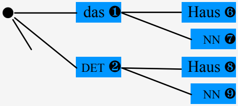
All rules are extensions of existing dotted rules

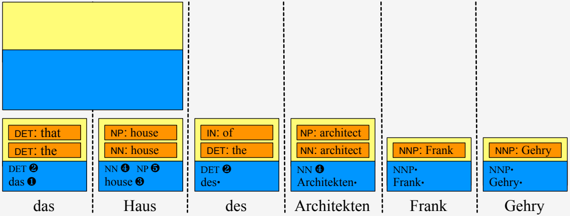
Here: only extensions of span over *das* possible

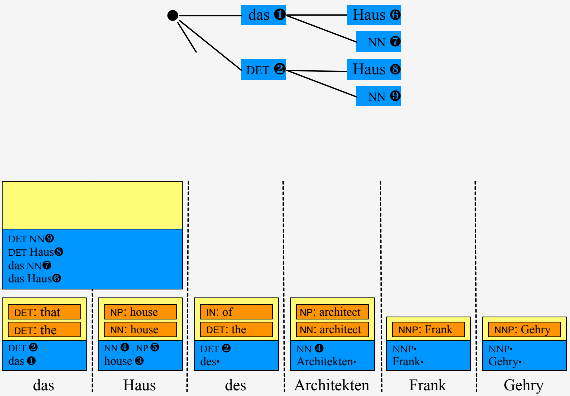



das ❶
NN, NP, Haus?
DET ❷
NN, NP, Haus?
Haus ❸
NN ❹
NP ❺
DET: that
DET: the
DET ❷
das ❶
NP: house
NN: house
NN ❹ NP ❺
house ❸
IN: of
DET: the
DET ❷
des•
NP: architect
NN: architect
NN ❹
Architekten•
NNP: Frank
NNP•
Frank•
NNP: Gehry
NNP•
Gehry•
das
Haus
des
Architekten
Frank
Gehry

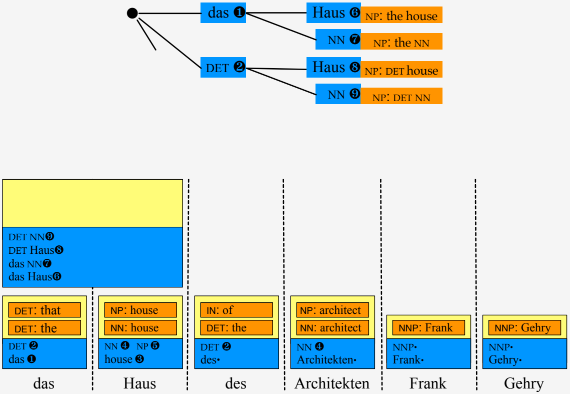
Looking up Rules in the Prefix Tree

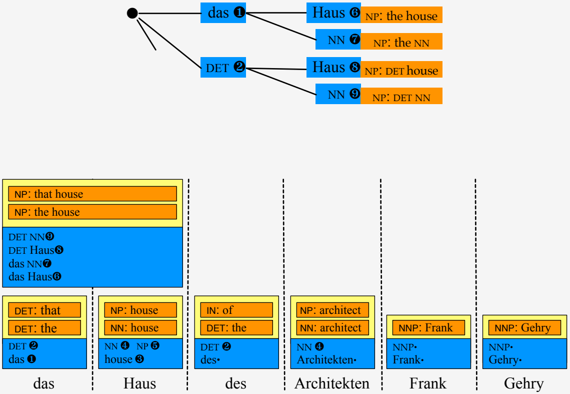




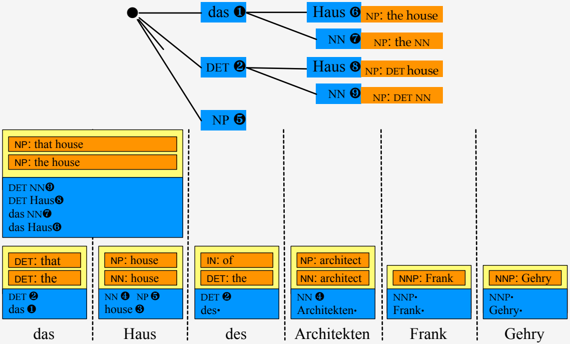

das ❶
Haus ❻
NN ❼
DET ❷
Haus ❽
NN ❾
DET NN❾
DET Haus❽
das NN❼
das Haus❻
DET: that
DET: the
DET ❷
das ❶
NP: house
NN: house
NN ❹ NP ❺
house ❸
IN: of
DET: the
DET ❷
des•
NP: architect
NN: architect
NN ❹
Architekten•
NNP: Frank
NNP•
Frank•
NNP: Gehry
NNP•
Gehry•
das
Haus
des
Architekten
Frank
Gehry

Checking if Dotted Rules have Translations

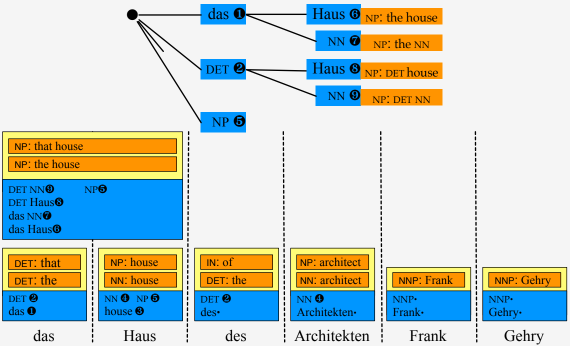



das ❶
Haus ❻
NP: the house
NN ❼
NP: the NN
DET ❷
Haus ❽
NP: DET house
NN ❾
NP: DET NN
NP: that house
NP: the house
DET NN❾
DET Haus❽
das NN❼
das Haus❻
DET: that
DET: the
DET ❷
das ❶
NP: house
NN: house
NN ❹ NP ❺
house ❸
IN: of
DET: the
DET ❷
des•
NP: architect
NN: architect
NN ❹
Architekten•
NNP: Frank
NNP•
Frank•
NNP: Gehry
NNP•
Gehry•
das
Haus
des
Architekten
Frank
Gehry

Looking up Constituent Label in Prefix Tree



Add to Span's List of Dotted Rules



- Complexity $O(rn^3)$ with sentence length n and size of dotted rule list r
  - ➡ may introduce maximum size for spans that do not start at beginning
  - ➡ may limit size of dotted rule list (very arbitrary)
- Does the list of dotted rules explode?
- Yes, if there are many rules with neighboring target-side non-terminals
  - ➡ such rules apply in many places
  - ➡ rules with words are much more restricted

- Some rules may apply in too many ways
- Neighboring input non-terminals

$$\text{VP} \rightarrow \textit{gibt}\ \text{X}_1\ \text{X}_2 \mid \textit{gives}\ \text{NP}_2\ \textit{to}\ \text{NP}_1$$

  - ➡ non-terminals may match many different pairs of spans
  - ➡ especially a problem for hierarchical models (no constituent label restrictions)
  - ➡ may be okay for syntax-models

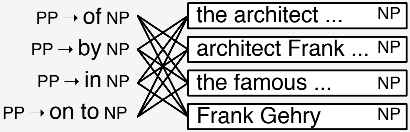
- Three neighboring input non-terminals

$$\text{VP} \rightarrow \textit{trifft}\ \text{X}_1\ \text{X}_2\ \text{X}_3\ \textit{heute} \mid \textit{meets}\ \text{NP}_1\ \textit{today}\ \text{PP}_2\ \text{PP}_3$$

  - ➡ will get out of hand even for syntax models

- We know which rules apply
- We know where they apply (each non-terminal tied to a span)
- But there are still many choices
  - ➡ many possible translations
  - ➡ each non-terminal may match multiple hypotheses
  - → number choices exponential with number of non-terminals

Found applicable rules $\mathrm{PP} \rightarrow$ *des* X | ... NP ...

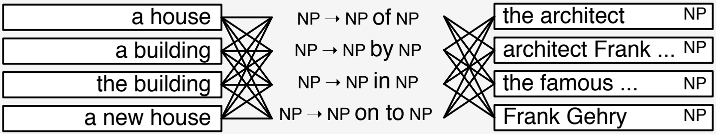


- Non-terminal will be filled any of h underlying matching hypotheses
- Choice of t lexical translations

⇒ Complexity $O(ht)$

(note: we may not group rules by target constituent label, so a rule $\mathrm{NP} \rightarrow$ *des* X | *the* NP would also be considered here as well)

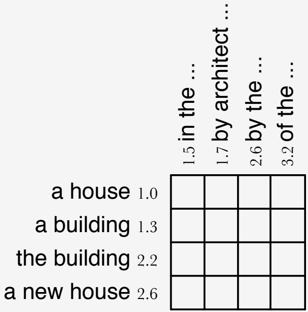
Found applicable rule $\text{NP} \rightarrow \text{X}_1$ des X_2 | NP_1 ... NP_2



- Two non-terminal will be filled any of h underlying matching hypotheses each
- Choice of t lexical translations

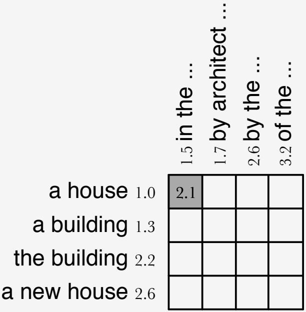
$\Rightarrow$ Complexity $O(h^2 t)$ — a three-dimensional "cube" of choices

(note: rules may also reorder differently)



Arrange all the choices in a “cube”

(here: a square, generally a orthotope, also called a hyperrectangle)



- Hypotheses created in cube: (0,0)

Add ("Pop") Hypothesis to Chart Cell

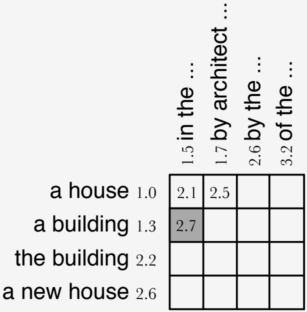
	1.5 in the ...	1.7 by architect ...	2.6 by the ...	3.2 of the ...
a house 1.0	2.1			
a building 1.3				
the building 2.2				
a new house 2.6				

- Hypotheses created in cube: ϵ
- Hypotheses in chart cell stack: (0,0)

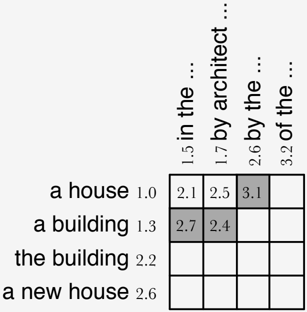
Create Neighboring Hypotheses

	1.5 in the ...	1.7 by architect ...	2.6 by the ...	3.2 of the ...
a house 1.0	2.1	2.5		
a building 1.3	2.7			
the building 2.2				
a new house 2.6				

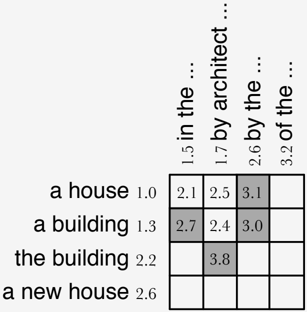
- Hypotheses created in cube: (0,1), (1,0)
- Hypotheses in chart cell stack: (0,0)



- Hypotheses created in cube: (0,1)
- Hypotheses in chart cell stack: (0,0), (1,0)



- Hypotheses created in cube: (0,1), (1,1), (2,0)
- Hypotheses in chart cell stack: (0,0), (1,0)



- Hypotheses created in cube: (0,1), (1,2), (2,1), (2,0)
- Hypotheses in chart cell stack: (0,0), (1,0), (1,1)

- Several groups of rules will apply to a given span
- Each of them will have a cube
- We can create a queue of cubes

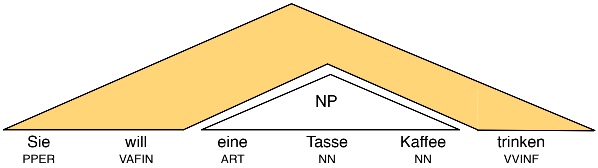
⇒ Always pop off the most promising hypothesis, regardless of cube

- May have separate queues for different target constituent labels

```
for all spans (bottom up) do
    extend dotted rules
    for all dotted rules do
        find group of applicable rules
        create a cube for it
        create first hypothesis in cube
        place cube in queue
    end for
    for specified number of pops do
        pop off best hypothesis of any cube in queue
        add it to the chart cell
        create its neighbors
    end for
    extend dotted rules over constituent labels
end for
```

- First stage: decoding without a language model (-LM decoding)
  - ➡ may be done exhaustively
  - ➡ optionably prune out low scoring hypotheses
- Second stage: add language model
  - ➡ limited to packed chart obtained in first stage
- Note: essentially, we do two-stage decoding for each span at a time

- Which spans should be more emphasized in search?
- Initial decoding stage can provide outside cost estimates



- Use min/max language model costs to obtain admissible heuristic (or at least something that will guide search better)

- Synchronous context free grammars
- Extracting rules from a syntactically parsed parallel corpus
- Bottom-up decoding
- Chart organization: dynamic programming, stacks, pruning
- Prefix tree for rules
- Dotted rules
- Cube pruning