
Advanced Natural Language Processing

Lecture 12

Parsing

Philipp Koehn
(based on slides by Mark Steedman)

19 October 2011



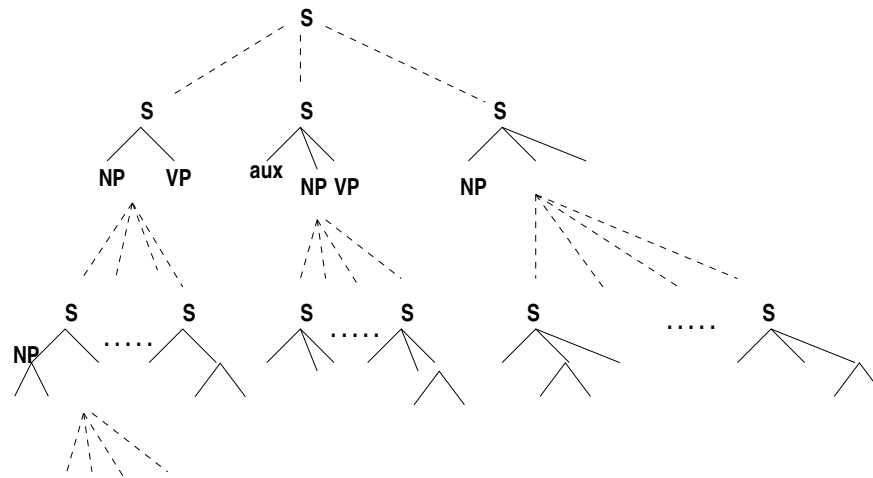
Parsers

A **parser** is an algorithm that computes a structure for an input string given a grammar. All parsers have two fundamental properties:

- **Directionality**: the sequence in which the structures are constructed (e.g., top-down or bottom-up).
- **Search strategy**: the order in which the search space of possible analysis is explored (e.g., depth-first, breadth-first).

Search Strategies

Schematic view of the search space:



In **depth-first search**, the parser explores one branch of the search space at a time. If this branch is a dead-end, it needs to **backtrack**.

In **breadth-first search**, the parser explores all possible branches in parallel (often impossible due to memory requirements).

Global and Local Ambiguity

A string can have more than one structural analysis (called **global ambiguity**) for one or both of two reasons:

- Grammatical rules allow for different attachment options;
- Lexical rules that allow a word to be in more than one word class.

Within a single analysis, some sub-strings can be analyzed in more than one way (called **local ambiguity**), even if not all these sub-string analyses are compatible with some global analysis of the entire string.

Local ambiguity is very common in Natural Languages.

Complexity

Depth-first parsing strategies demonstrate other problems with “parsing as search”:

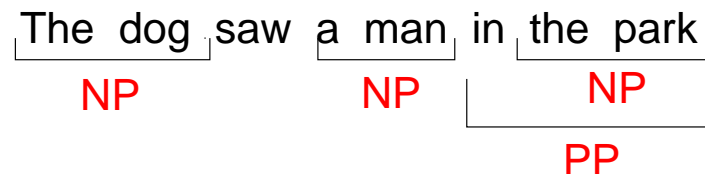
1. **Structural ambiguity** in the grammar and **lexical ambiguity** in the words can lead the parser down a wrong path.
2. So the same sub-tree may be built several times: whenever a path fails, the parser undoes its work, backtracks and starts again.

The complexity of this **blind backtracking** is exponential in the worst case because of repeated **re-analysis** of the same sub-string.

Here we will look at handling ambiguity via **Chart parsing**.

Dynamic Programming

With a CFG, a parser should be able to avoid re-analyzing sub-strings because the analysis of any sub-string is **independent** of the rest of the parse.



The parser's exploration of its search space can exploit this independence if the parser uses **dynamic programming**.

Dynamic programming is the basis for all **chart parsing** algorithms.

Parsing as Dynamic Programming

- Given a problem, Dynamic Programming systematically fills a table of solutions to sub-problems (**memoization**).
- Once solutions to all sub-problems have been accumulated, DP solves the overall problem by composing them..

For parsing, sub-problems are analyses of sub-strings, which are memoized in a **chart** (aka **well-formed substring table**, WFST). Each analysis corresponds to:

- a complete **constituent** (sub-tree), indexed by the start and end of the sub-string that it covers;
- **or** a **hypothesis** about what complete constituent might be found, indexed by the start and end of the sub-strings that support it.

Depicting a WFST/Chart

A well-formed substring table (aka chart) can be depicted as either a **matrix** or a **graph**. Both contain the same information.

When a **WFST** (aka **chart**) is depicted as a matrix:

- Rows and columns of the matrix correspond to the start and end positions of a span (ie, starting **right before** the first word, ending **right after** the final one);
- A cell in the matrix corresponds to the sub-string that starts at the row index and ends at the column index. It can contain information about the **type** of constituent (or constituents) that span(s) the substring, pointers to its sub-constituents, and/or **predictions** about what constituents might follow the substring.

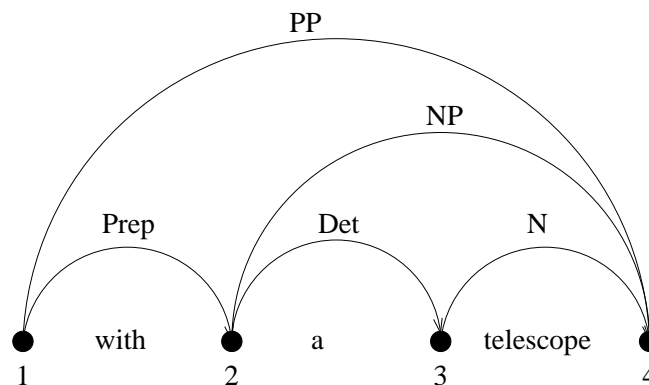
Depicting a WFST as a Matrix

	1	2	3	4	5	6
0	V					
1		Prep		PP		
2			Det	NP		
3				N		
4						
5						

0 *See* 1 *with* 2 *a* 3 *telescope* 4 *in* 5 *hand* 6

Depicting a WFST as a Graph

- Here, nodes/vertices represent positions in the text string, starting **before** the first word, ending **after** the final word.
- arcs/edges connect vertices at the start and the end of a span to represent a particular substring. Edges can be labelled with the same information as in a cell in the matrix representation.



Algorithms for Chart Parsing

Important members of the chart parsing family include:

- the CKY algorithm, which memoizes only constituents;
- three algorithms that memoize both constituents and predictions:
 - a bottom-up chart parser
 - a top-down chart parser
 - the Earley algorithm

CKY Algorithm

CKY (Cocke, Kasami, Younger) is an algorithm for recognizing constituents and recording them in the chart (WFST).

CKY was originally defined for Chomsky Normal Form ($A \rightarrow BC$; $A \rightarrow a$), and later generalized by Gray and Harrison (1972); Harrison (1978).

We can enter constituent A in cell (i, j) if there is a rule $A \rightarrow B$ and B is found in cell (i, j) , or if $A \rightarrow B C$ and B is found in cell (i, k) and C is found in cell (k, j) .

Proceeding systematically, CKY guarantees that the parser only looks for rules that use a constituent from i to j **after** it has processed all the constituents that end at i . This avoids anything being missed.

Chart Parsing with the CKY Algorithm

Let $\text{Close}(X) = \{B \mid B \rightarrow^* A, \text{ using unary productions, and } A \in X\}$

$\text{BUILD_CKY_CHART}(t, [w_1, \dots, w_n])$

for $j \leftarrow 1$ **to** n

do

$t(j-1, j) \leftarrow \text{Close}(\{w_j\})$

for $k \leftarrow 1$ **to** n

for $j \leftarrow k$ **to** n

for $m \leftarrow 1$ **to** $k-1$

do

$t(j-k, j) \leftarrow t(j-k, j) \cup \text{Close}(\{A \mid A \rightarrow B C$
for some $B \in t(j-k, j-m)$ and $C \in t(j-m, j)\})$

This algorithm is complete and performs recognition in time $O(n^3)$.

Visualizing the Chart

Grammatical rules

S → NP VP

NP → Det Nom

NP → Nom

Nom → N SRel

Nom → N

VP → TV NP

VP → IV PP

VP → IV

PP → Prep NP

SRel → Relpro VP

Lexical rules

Det → a | the (determiner)

N → fish | frogs | soup (noun)

Prep → in | for (preposition)

TV → saw | ate (transitive verb)

IV → fish | swim (intransitive verb)

Relpro → that (relative pronoun)

Nom: nominal (the part of the NP after the determiner, if any).

SRel: subject relative clause, as in the frogs that ate fish.

Visualizing the Chart

	1	2	3	4
0				
1				
2				
3				
	the	frogs	ate	fish

Visualizing the Chart (0,1)

	1	2	3	4
0	det			
1				
2				
3				
	the	frogs	ate	fish

Unary branching rules: $\text{det} \rightarrow \text{the}$

Visualizing the Chart (1,2)

	1	2	3	4
0	det			
1		n nom np		
2				
3				
	the	frogs	ate	fish

Unary branching rules: $N \rightarrow \text{frogs}$, $\text{Nom} \rightarrow N$, $\text{NP} \rightarrow \text{Nom}$

Visualizing the Chart (2,3)

	1	2	3	4
0	det			
1		n nom np		
2			tv	
3				
	the	frogs	ate	fish

Unary branching rules: $tv \rightarrow ate$

Visualizing the Chart (3,4)

	1	2	3	4
0	det			
1		n nom np		
2			tv	
3				n nom np iv vp
	the	frogs	ate	fish

Unary branching rules: $N \rightarrow \text{frogs}$, $Nom \rightarrow N$, $NP \rightarrow Nom$, $iv \rightarrow \text{fish}$, $vp \rightarrow iv$

Visualizing the Chart (0,2)

	1	2	3	4
0	det	np		
1		n nom np		
2			tv	
3				n nom np iv vp
	the	frogs	ate	fish

Binary branching rule: NP \rightarrow Det Nom (0,1) & (1,2) \rightsquigarrow (0,2)

Visualizing the Chart (1,3)

	1	2	3	4
0	det	np		
1		n nom np		
2			tv	
3				n nom np iv vp
	the	frogs	ate	fish

(1,2) & (2,3) ↗

Visualizing the Chart (2,4)

	1	2	3	4
0	det	np		
1		n nom np		
2			tv	vp
3				n nom np iv vp
	the	frogs	ate	fish

Binary branching rule: $VP \rightarrow TV NP$ $(2,3) \ \& \ (3,4) \rightsquigarrow (2,4)$

Visualizing the Chart (1,4)

	1	2	3	4
0	det	np		
1		n nom np		s
2			tv	vp
3				n nom np iv vp
	the	frogs	ate	fish

Binary branching rule: $S \rightarrow NP VP$ $(1,2) \ \& \ (2,4) \rightsquigarrow (1,4)$
 $(1,3) \ \& \ (3,4) \not\rightsquigarrow$

Visualizing the Chart (0,4)

	1	2	3	4
0	det	np		s
1		n nom np		s
2			tv	vp
3				n nom np iv vp
	the	frogs	ate	fish

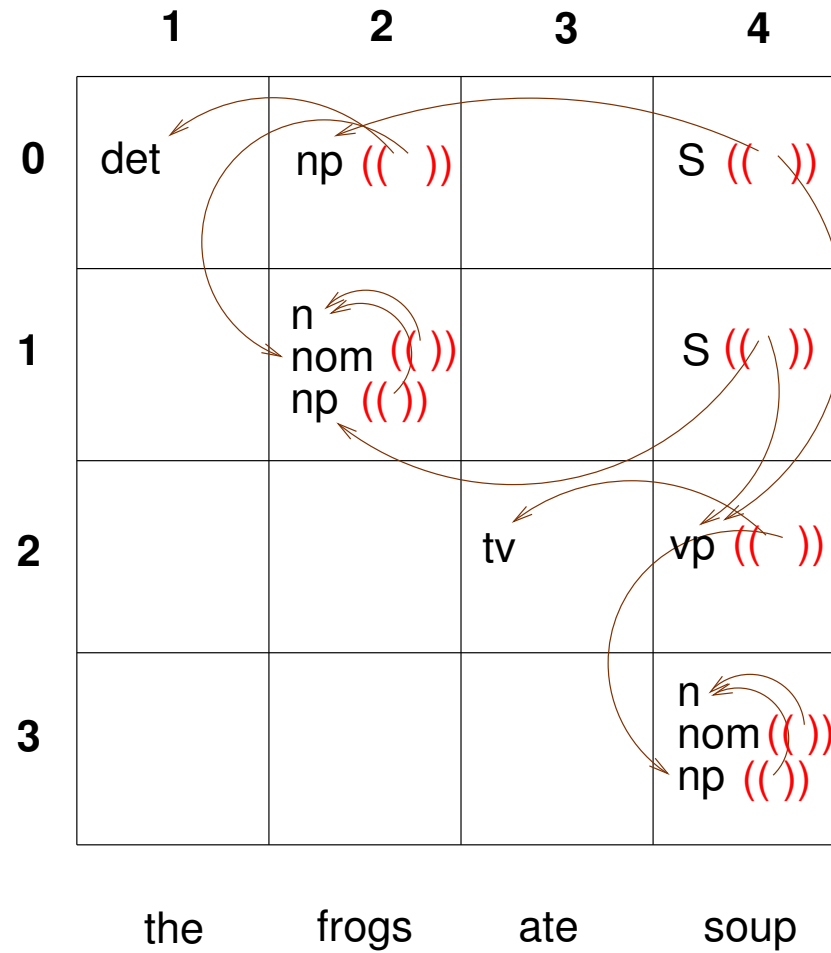
Binary branching rule: $S \rightarrow NP VP$ $(0,1) \& (1,4) \not\rightarrow$ $(0,2) \& (2,4) \rightsquigarrow (0,4)$ $(0,3) \& (3,4) \not\rightarrow$

From CKY Recognizer to CKY Parser

We cannot tell from the CKY chart as specified, the syntactic analysis of the input string.

We just have a chart **recognizer**, a way of determining whether a string belongs to the language generated by the grammar.

Changing this to a **parser** requires recording which existing constituents were combined to make each new constituent. This requires another field to record the one or more ways in which a constituent spanning (i,j) can be made from constituents spanning (i,k) and (k,j) .



Adding Prediction to the Chart: Chart entries

Like all Dynamic Programming algorithms, CKY avoids redundant work by **memo-izing** all the constituents it finds.

What it doesn't record is any justification for a chart entry – why it was built.

So if we have two VP rules:

$$VP \rightarrow V NP$$
$$VP \rightarrow VP PP$$

and the input string

The boy opened the box on the floor

Chart entries

We don't know which production the VP arc [2, 8] represents (here using a graph representation of the chart). Is it $VP \rightarrow V NP$?

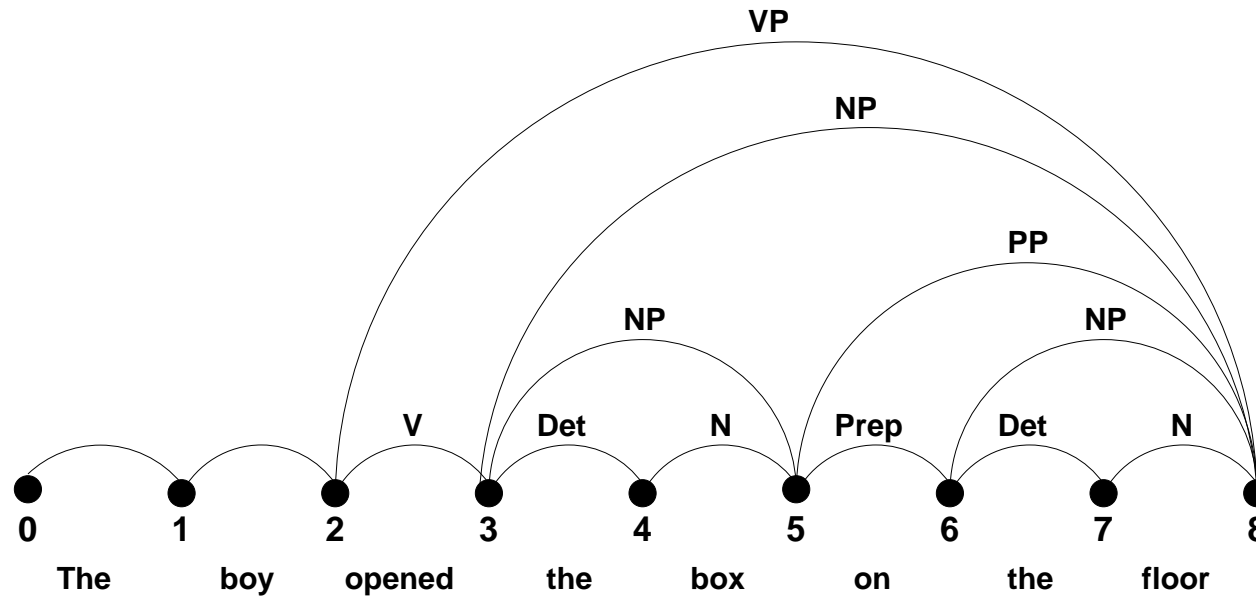


Chart entries

We don't know which production the VP arc [2, 8] represents (here using a graph representation of the chart). Or $VP \rightarrow VP PP$?

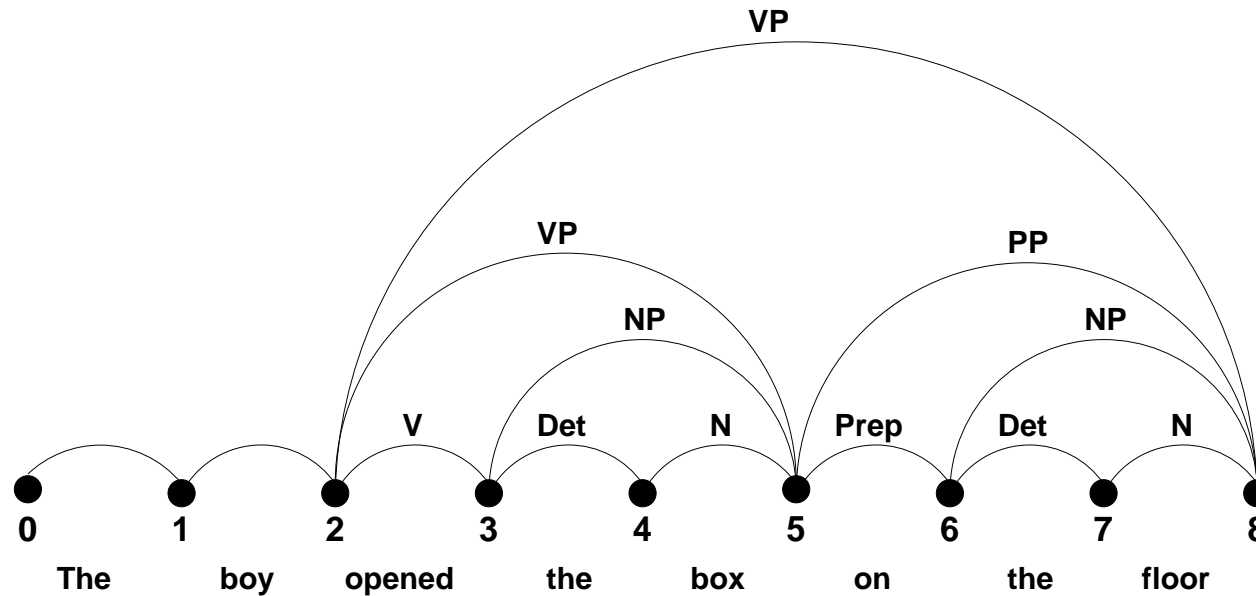


Chart entries

If the entire **production** were recorded, rather than just its LHS (ie, the constituent that it analyses), then we'd know.

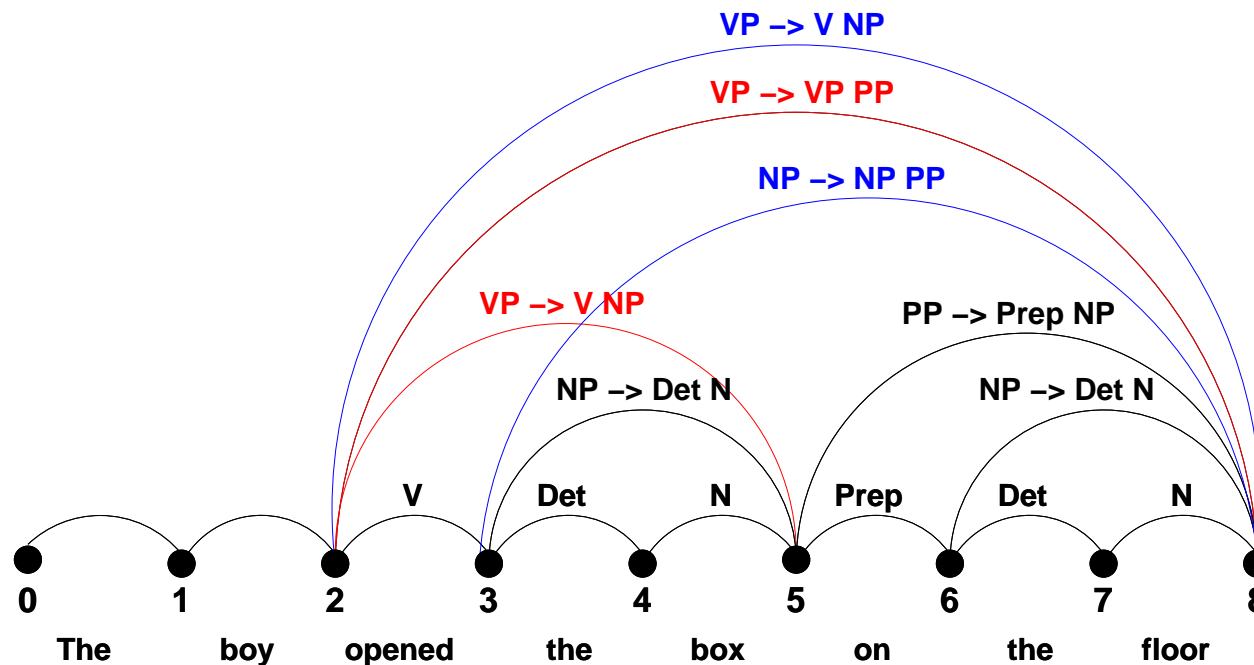


Chart entries

We are not limited to recording **complete productions** (ie, ones whose entire RHS have been recognized): We could also consider recording **incomplete productions** (ie, ones for which there may so far be only partial evidence).

Incomplete productions (aka **incomplete constituents**) are effectively **predictions** about what might come next and what will be learned from finding it.

Incomplete constituents can be represented using an extended form of production rule called a **dotted rule**.

The **dot** indicates how much of the RHS has already been found. The rest is a prediction of what is to come.

Incomplete constituent is a useful concept: What else have you seen that indicates this?

Dotted Rules: Examples

The grammar rule $VP \rightarrow V NP$ yields the following dotted rules:

$VP \rightarrow \cdot V NP$ **incomplete edge**
 $VP \rightarrow V \cdot NP$ **incomplete edge**
 $VP \rightarrow V NP \cdot$ **complete edge**

Dotted Rules

Just as we can record in the chart the production rules for complete constituents, we can record the dotted rules for incomplete constituents, noting

- the production rule used in the analysis, along with the start and end position of the material already found;
- the part of the rule already found (to the left of the dot), and the part still predicted to be found (to the right of the dot).

Thus the chart could record for the substring \dots_5 *on* $_6$ *the* $_7$ *floor* $_8$

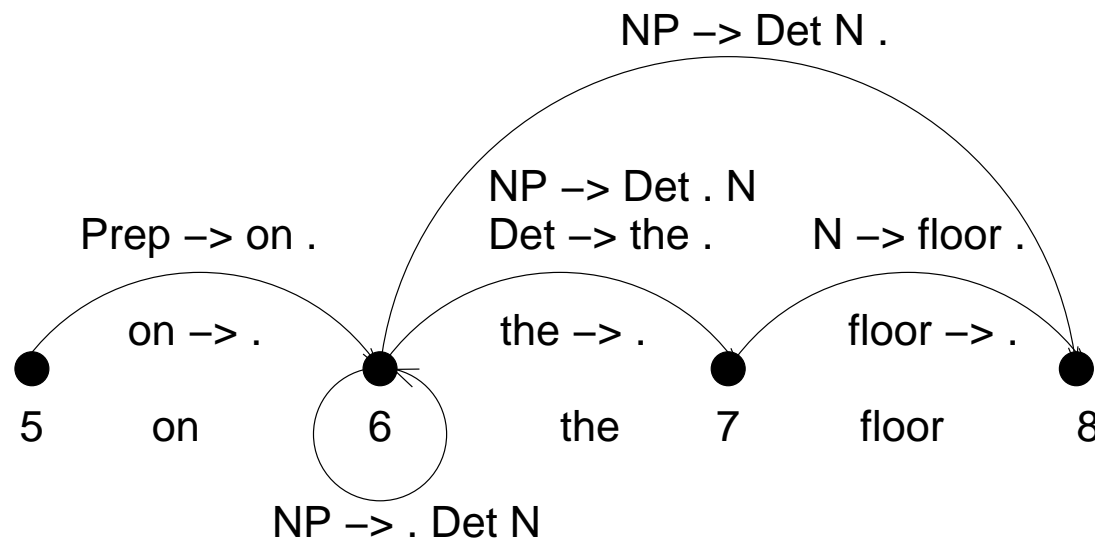
$NP \rightarrow \text{Det} . N, [6, 7]$

indicating the word between 6 and 7 is spanned by Det, and an N is predicted next. If found, we get an NP starting at 6.

Dotted Rules

The extreme dotted rule has the dot at the left end of the RHS, meaning **nothing** on the RHS has yet been found, and everything is predicted.

Example of a chart labelled with dotted rules (in graph representation):



Adding Edges

An **active chart parser** can add an edge for any of three reasons:

1. The **input** can license an edge. In particular, each word w_i in the input licenses the complete edge $[w_i \rightarrow \bullet, (i, i+1)]$;
2. The **grammar** can license an edge. In particular, each grammar production $A \rightarrow \alpha$ licenses a self-loop edge $[A \rightarrow \bullet \alpha, (i, i)]$ for every i , $0 \leq i < n$, where n is the length of the string.
3. The **contents of the current chart** can license an edge.

It rarely makes sense to add **all** licensed edges to a chart, since many of them will not be used in any complete parse.

Fundamental Rule

The strategy of an active chart parser comprises the rules used to decide when to add an edge, along with a specification of when rules should be applied.

Every active chart parser uses what's called the **Fundamental Rule** to combine an incomplete edge with a complete one.

If the chart contains the edges

$$[A \rightarrow \alpha \bullet B \beta, (i, j)] \quad \text{and} \quad [B \rightarrow \gamma \bullet, (j, k)]$$

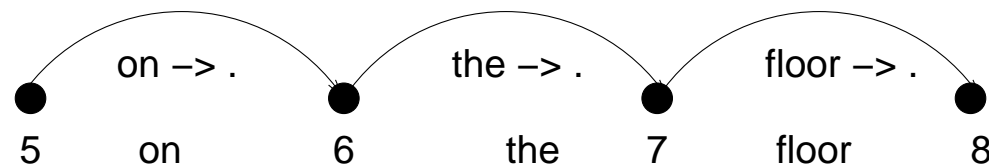
then add the new edge $[A \rightarrow \alpha B \bullet \beta, (i, k)]$
where α , β , and γ are (possibly empty) sequences of terminals or non-terminals.

The dot moves one place right, and the new edge spans the other two.

Bottom-up Active Parsing

As with the CKY Parser, **Bottom-up Active Parsing** starts from the input string, identifying sequences of words and phrases that correspond to the RHS of a grammar production.

Bottom-up Initialization Rule: For every word w_i , add the edge $[w_i \rightarrow \bullet, (i, i + 1)]$;



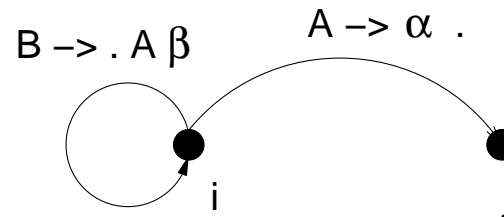
Bottom-up Active Parsing

Bottom-up Predict Rule: If the chart contains the complete edge

$$[A \rightarrow \alpha \bullet, (i, j)]$$

and the grammar contains the production $B \rightarrow A \beta$

then add the self-loop edge $[B \rightarrow \bullet A \beta, (i, i)]$



Bottom-up Active Parsing Strategy

- Create an empty chart spanning the sentence.
- Apply the **Bottom-Up Initialization Rule** to each word.
- Until no more edges are added:
 - Apply the **Bottom-Up Predict Rule** everywhere it applies.
 - Apply the **Fundamental Rule** everywhere it applies.
- Return all of the parse trees corresponding to the parse edges in the chart.

References

- Gray, J. and Harrison, M. A. (1972). On the covering and reduction problems for context-free phrase structure grammars. *Journal of the Association for Computing Machinery*, 19:385–395.
- Harrison, M. (1978). *Introduction to Formal Language Theory*. Addison-Wesley, Reading MA.