21CSE356T
NATURAL LANGUAGE PROCESSING
UNIT-2
SYNTAX ANALYSIS

- Context Free Grammars
- Grammar Rules for English
- Top-Down Parsing
- Bottom-Up Parsing
- Ambiguity
- CKY Parsing
- Dependency Parsing
- Earley Parsing
- Probabilistic Context-Free Grammars

SYNTAX ANALYSIS

Syntax analysis, also called **parsing**, is a critical aspect of Natural Language Processing (NLP). It involves analyzing the grammatical structure of sentences to understand their meaning. Syntax analysis builds a **parse tree** or **syntax tree** by identifying the roles of individual words and how they relate to one another within a sentence.

Syntactic analysis or parsing may be defined as the process of analyzing the strings of symbols in natural language conforming to the rules of formal grammar. The origin of the word '**parsing'** is from Latin word '**pars'** which means '**part'**.

Key Concepts in Syntax Analysis

1. **Parts of Speech Tagging (POS):** Assigning grammatical tags to words, such as noun, verb, adjective, etc.
    - Example: "The cat sleeps" → The/DT cat/NN sleeps/VBZ
2. **Parsing:** Constructing a syntactic structure for a sentence according to a given grammar (often represented in forms like Context-Free Grammar).
    - **Dependency Parsing:** Focuses on the relationships between words (e.g., subject-verb, verb-object).
    - **Constituency Parsing:** Breaks sentences into sub-phrases (constituents) like noun phrases and verb phrases.
3. **Grammar Types:**
    - **Context-Free Grammar (CFG):** Rules define how words and phrases can combine. Example: S → NP VP (A sentence is a noun phrase followed by a verb phrase).
      robabilistic CFG (PCFG): Adds probabilities to grammar rules to handle ambiguities.
4. **Ambiguity:**
    - **Syntactic Ambiguity:** Sentences that can be parsed in multiple ways. Example: "I saw the man with a telescope."
    - Resolving ambiguity is a critical challenge in syntax analysis.
5. **Applications:**
    - **Information Extraction:** Identifying entities, relations, or events in text.
    - **Machine Translation:** Understanding sentence structure improves translations.

- **Question Answering Systems:** Parsing helps identify what is being asked and how to answer.

Syntax Analysis Techniques

1. **Rule-Based Parsing:** Uses explicit grammar rules to parse sentences.
   - Examples: Earley Parser, CYK Parser.
2. **Statistical Parsing:** Learns grammatical structures from labeled data using statistical models.
   - Examples: Stanford Parser, spaCy.
3. **Neural Parsing:** Uses deep learning models to learn syntactic structures directly from data.
   - Examples: BERT-based dependency parsers.
4. **Lexicalized Parsing:** Incorporates word-level information (lexicon) into parsing decisions.

CONTEXT FREE GRAMMAR

Context free grammars are also called ***Phrase-Structure Grammars***, and the formalism is equivalent to ***Backus-Naur Form, or BNF***.

A context-free grammar consists of a **set of rules or productions**, each of which expresses the ways that **symbols of the language** can be grouped and ordered together, and a **lexicon of words and symbol**s. For example:

$$\begin{aligned} NP &\rightarrow Det\ Nominal \\ NP &\rightarrow ProperNoun \\ Nominal &\rightarrow Noun \mid Nominal\ Noun \end{aligned}$$

Context-free rules can be hierarchically embedded, so we can combine the previous rules with others, like the following, that express facts about the lexicon:

$$\begin{aligned} Det &\rightarrow a \\ Det &\rightarrow the \\ Noun &\rightarrow flight \end{aligned}$$

The symbols that are used in a CFG are divided into two classes.

- **Terminal Symbol:** The symbols that correspond to words in the language ***("the", "nightclub") are called terminal symbols***; the ***lexicon*** is the set of rules that introduce these terminal symbols.
- **Non-terminal Symbol:** The symbols that express abstractions over these terminals are called ***non-terminals***.

In each context-free rule, the item to the right of the arrow (→) is an ordered list of one or more terminals and non-terminals; to the left of the arrow is a single non-terminal symbol expressing some cluster or generalization.

A CFG can be thought of in two ways:

- as a device for generating sentences
- as a device for assigning a structure to a given sentence

So starting from the symbol:	*NP*
we can use our first rule to rewrite *NP* as:	*Det Nominal*
and then rewrite *Nominal* as:	*Det Noun*
and finally rewrite these parts-of-speech as:	*a flight*

This sequence of rule expansions is called a ***derivation*** of the string of words. It is common to represent a derivation by a **parse tree**.

The **formal language** defined by a CFG is the set of strings that are derivable from the designated start symbol.

Figure 12.1 A parse tree for "a flight".

Grammar Rules			Examples
S	→	*NP VP*	I + want a morning flight
NP	→	*Pronoun*	I
	|	*Proper-Noun*	Los Angeles
	|	*Det Nominal*	a + flight
Nominal	→	*Nominal Noun*	morning + flight
	|	*Noun*	flights
VP	→	*Verb*	do
	|	*Verb NP*	want + a flight
	|	*Verb NP PP*	leave + Boston + in the morning
	|	*Verb PP*	leaving + on Thursday
PP	→	*Preposition NP*	from + Los Angeles

Figure 12.3 The grammar for $\mathcal{L}_0$, with example phrases for each rule.

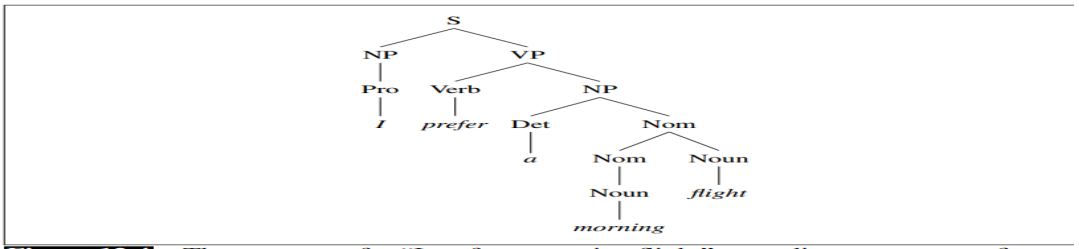


Figure 12.4 The parse tree for "I prefer a morning flight" according to grammar $\mathcal{L}_0$.

Sentences (strings of words) that can be derived by a grammar are in the formal language defined by that grammar, and are called **grammatical sentences**. Sentences that cannot be derived by a given formal grammar are not in the language defined by that grammar and are referred to as **ungrammatical**.

Formal Definition of Context-Free Grammar

A context-free grammar G is defined by four parameters: **N, Σ, R, S (technically this is a "4-tuple").**

Capital letters like A, B, and S	Non-terminals
S	The start symbol
Lower-case Greek letters like α, β, and γ	Strings drawn from $(\Sigma \cup N)^*$
Lower-case Roman letters like u, v, and w	Strings of terminals

GRAMMAR RULES FOR ENGLISH

1. Sentences
2. Noun phrases
   - Determiners
   - Nominals
   - Agreement
   - Post modifiers
3. Verb phrases
   - Subcategorization

1. **Sentence Types**

- Declaratives: *A plane left.*
  $S \rightarrow NP\ VP$
- Imperatives: *Leave!*
  $S \rightarrow VP$
- Yes-No Questions: *Did the plane leave?*
  $S \rightarrow Aux\ NP\ VP$
- WH Questions: *When did the plane leave?*
  $S \rightarrow WH\text{-}NP\ Aux\ NP\ VP$

2. **Noun Phrases**

- Most of the complexity of English noun phrases is hidden in this rule.

Determiners: *the, this, a, an*, etc.

Nominals:

- Quantifiers, cardinals, ordinals...
  - Three cars
- Adjectives and Aps
  - large cars
- Ordering constraints
  - Three large cars
  - ?large three cars

Postmodifiers:

- Three kinds
  - Prepositional phrases
    - From Seattle
  - Non-finite clauses
    - Arriving before noon
  - Relative clauses
    - That serve breakfast

Agreement:

- By *agreement*, we have in mind constraints that hold among various constituents that take part in a rule or set of rules
- For example, in English, determiners and the head nouns in NPs have to agree in their number.

This flight	*This flights
Those flights	*Those flight

3. **Verb Phrases**

- English *VP*s consist of a head verb along with 0 or more following constituents which we'll call *arguments*.

$VP \rightarrow Verb$ disappear
$VP \rightarrow Verb\ NP$ prefer a morning flight
$VP \rightarrow Verb\ NP\ PP$ leave Boston in the morning
$VP \rightarrow Verb\ PP$ leaving on Thursday

PARSING TECHNIQUE

For FSAs, for example, the parser is searching through the space of all possible paths through the automaton. In syntactic parsing, the parser can be viewed as searching through the space of all possible parse trees to find the correct parse tree for the sentence.

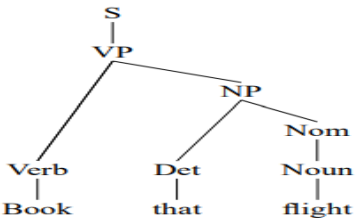


Figure 10.1 The correct parse tree for the sentence *Book that flight* according to the grammar in Figure 10.2.

The goal of a parsing search is to find all trees whose root is the start symbol S, which cover exactly the words in the input.

$S \rightarrow NP\ VP$	$Det \rightarrow that \mid this \mid a$
$S \rightarrow Aux\ NP\ VP$	$Noun \rightarrow book \mid flight \mid meal \mid money$
$S \rightarrow VP$	$Verb \rightarrow book \mid include \mid prefer$
$NP \rightarrow Det\ Nominal$	$Aux \rightarrow does$
$Nominal \rightarrow Noun$	
$Nominal \rightarrow Noun\ Nominal$	$Prep \rightarrow from \mid to \mid on$
$NP \rightarrow Proper\text{-}Noun$	$Proper\text{-}Noun \rightarrow Houston \mid TWA$
$VP \rightarrow Verb$	
$VP \rightarrow Verb\ NP$	$Nominal \rightarrow Nominal\ PP$

Figure 10.2 A miniature English grammar and lexicon.

Two search strategies underlying most parsers: ***top-down or goal-directed search and bottom-up or data-directed search.***

TOP-DOWN PARSING

A top-down parser searches for a parse tree by trying to build from the root node S down to the leaves. The algorithm starts by assuming the input can be derived by the designated start symbol S. The next step is to find the tops of all trees which can start with S, by looking for all the grammar rules with S on the left-hand side.

```
function TOP-DOWN-PARSE(input, grammar) returns a parse tree
  agenda ← (Initial S tree, Beginning of input)
  current-search-state ← POP(agenda)
  loop
    if SUCCESSFUL-PARSE?(current-search-state) then
      return TREE(current-search-state)
    else
      if CAT(NODE-TO-EXPAND(current-search-state)) is a POS then
        if CAT(node-to-expand)
              ⊂
            POS(CURRENT-INPUT(current-search-state)) then
          PUSH(APPLY-LEXICAL-RULE(current-search-state), agenda)
        else
          return reject
      else
        PUSH(APPLY-RULES(current-search-state, grammar), agenda)
    if agenda is empty then
      return reject
    else
      current-search-state ← NEXT(agenda)
  end
```

Figure 10.6 A top-down, depth-first left-to-right parser.

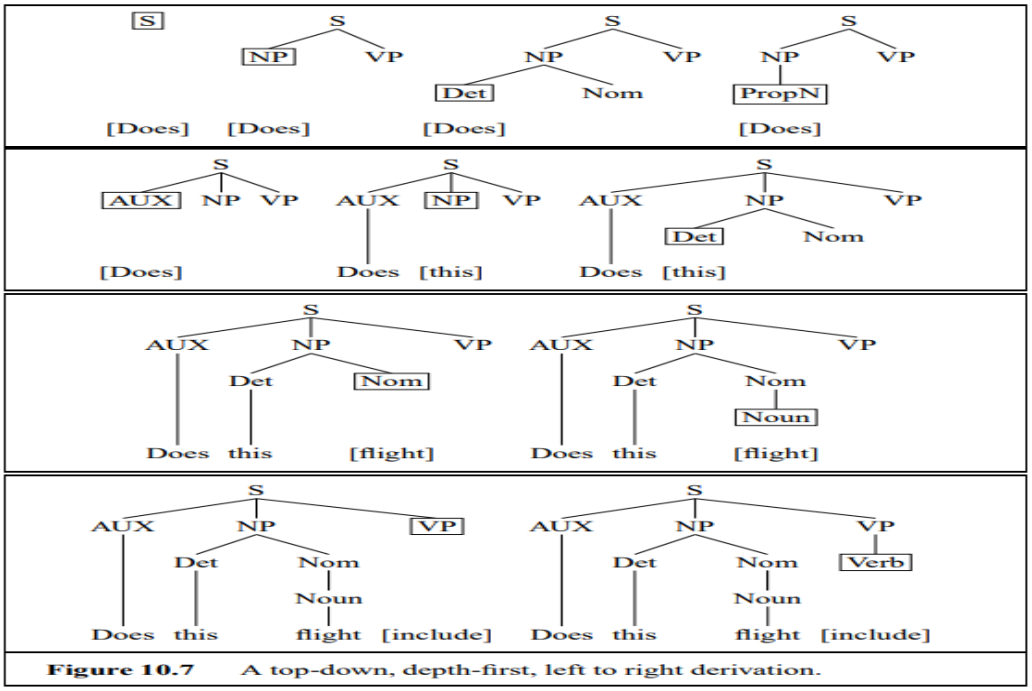


Figure 10.7 A top-down, depth-first, left to right derivation.

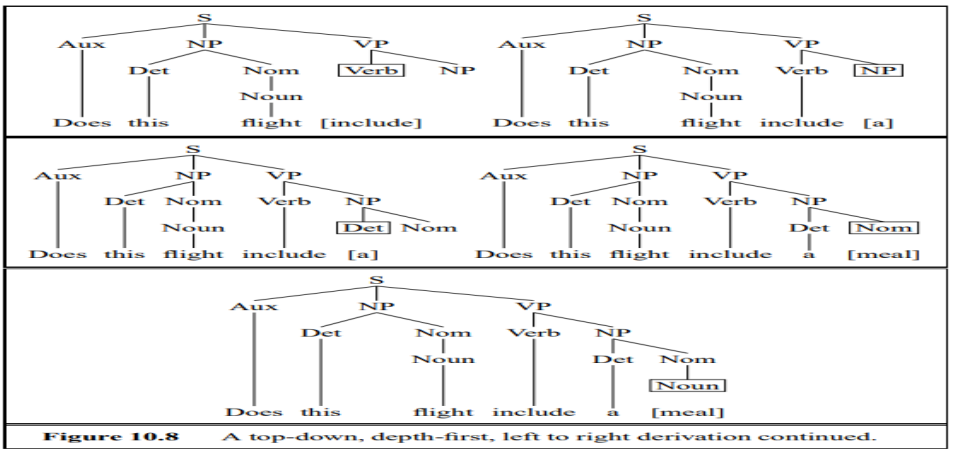


Figure 10.8 A top-down, depth-first, left to right derivation continued.

PROBLEMS WITH THE BASIC TOP-DOWN PARSER

- Left-recursion
- Ambiguity
- Inefficient reparsing of subtrees

Left-Recursion

A-> Aβ, where the first constituent of the right-hand side is identical to the left-hand side.

$$NP \rightarrow Det\ Nominal$$
$$Det \rightarrow NP\ 's$$

Some of the immediately left recursive rules that make frequent appearances in grammars of English.

$$NP \rightarrow NP\ PP$$
$$VP \rightarrow VP\ PP$$
$$S \rightarrow S\ and\ S$$

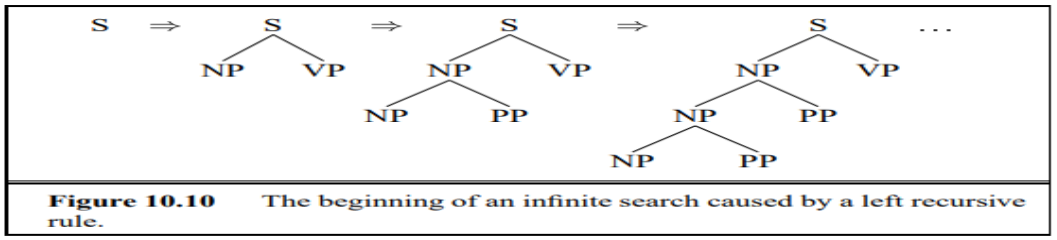


Figure 10.10 The beginning of an infinite search caused by a left recursive rule.

Ambiguity

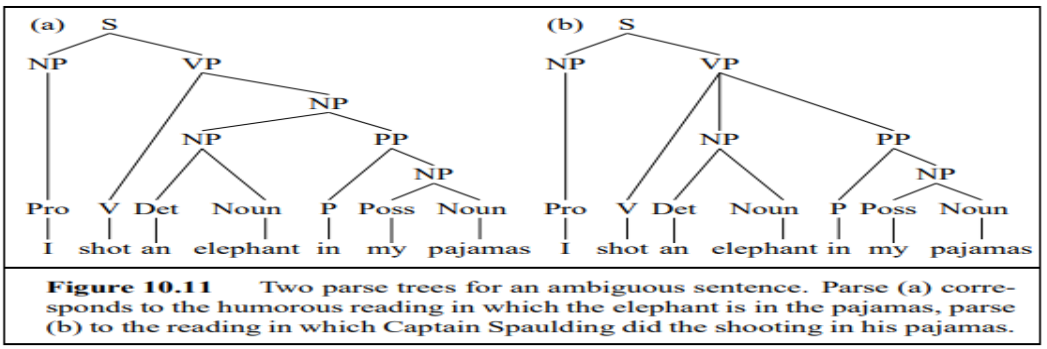


Figure 10.11 Two parse trees for an ambiguous sentence. Parse (a) corresponds to the humorous reading in which the elephant is in the pajamas, parse (b) to the reading in which Captain Spaulding did the shooting in his pajamas.

Structural ambiguity occurs when the grammar assigns more than one possible parse to a sentence and comes in many forms. Three particularly common kinds of ambiguity are ***attachment ambiguity, coordination ambiguity, and noun-phrase bracketing ambiguity.***

- A sentence has an **attachment ambiguity** if a particular constituent can be attached to the parse tree at more than one place.
- **Coordination ambiguity**, in which there are different sets of phrases that can be conjoined by a conjunction like and. For example, the phrase old men and women can be bracketed [old [men and women]], referring to old men and old women, or as [old men] and [women], in which case it is only the men who are old.
- **Noun phrase** could be parsed [nationwide [television and radio]] or [[nationwide television] and radio].

Inefficient reparsing of subtrees

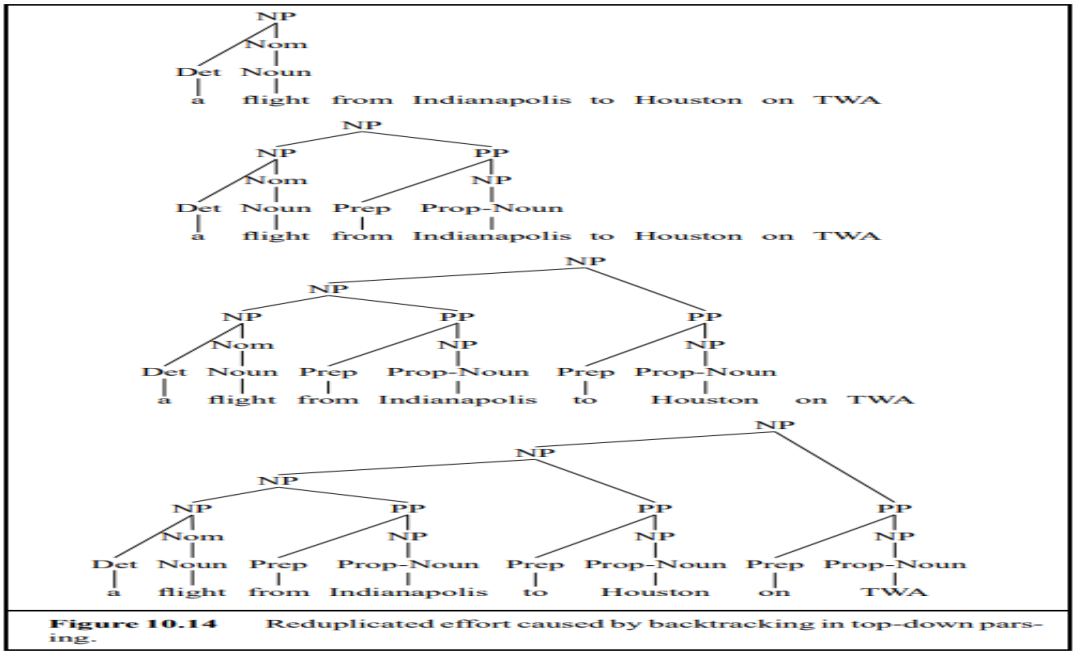


Figure 10.14 Reduplicated effort caused by backtracking in top-down parsing.

BOTTOM-UP PARSING

Bottom-up parsing is the earliest known parsing algorithm (it was first suggested by Yngve (1955)), and is used in the shift-reduce parsers common for computer languages.

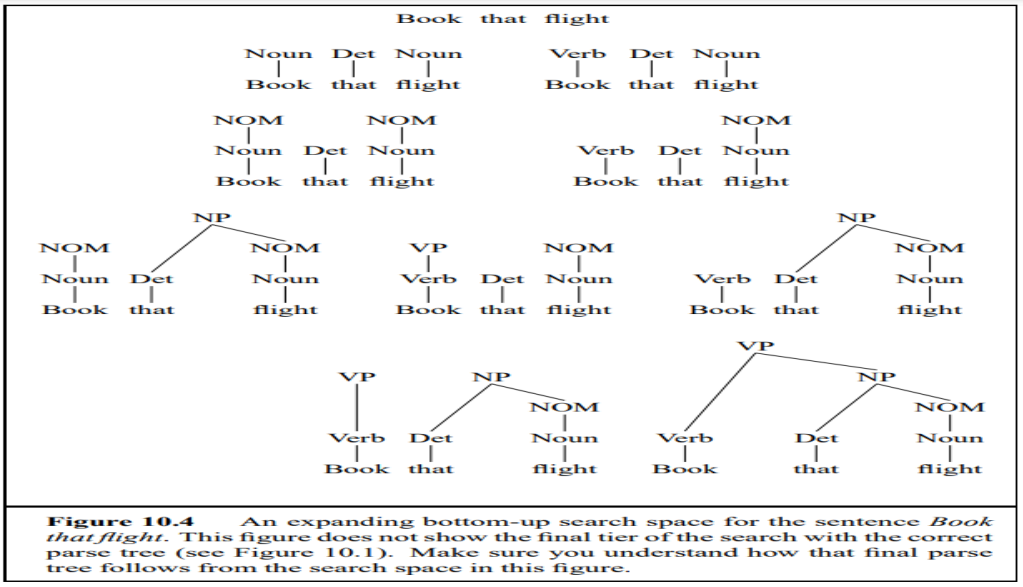


Figure 10.4 An expanding bottom-up search space for the sentence *Book that flight*. This figure does not show the final tier of the search with the correct parse tree (see Figure 10.1). Make sure you understand how that final parse tree follows from the search space in this figure.

DYNAMIC PROGRAMMING PARSING METHODS: CKY PARSING

Dynamic programming provides a powerful framework for **addressing the problems caused by ambiguity in grammars.** Recall that a dynamic programming approach systematically **fills in a table of solutions to sub-problems**. **Cocke-KasamiYounger (CKY)** algorithm, the most widely used dynamic-programming based approach to parsing.

- **Conversion to Chomsky Normal Form**

The CKY algorithm requires grammars to first be in Chomsky Normal Form (CNF). Grammars in CNF are restricted to rules of the form **A → B C or A → w.**

The process of converting a generic CFG into one represented in CNF.

Assuming we're dealing with an €-free grammar, there are three situations we need to address in any generic grammar:

- **Rules that mix terminals with non-terminals on the right-hand side**: The remedy for rules that mix terminals and non-terminals is to simply introduce a new dummy non-terminal that covers only the original terminal. For example, a rule for an infinitive verb phrase such as **INF-VP → to VP** would be replaced by the two rules **INF-VP → TO VP** and **TO → to.**
- **Rules that have a single non-terminal on the right-hand side:** Rules with a single non-terminal on the right are called **unit productions**.
- **Rules in which the length of the right-hand side is greater than 2:** Rules with right-hand sides longer than 2 are normalized through the introduction of new non-terminals that spread the longer sequences over several new rules. Formally, if we have a rule like

  A → B C γ

we replace the leftmost pair of non-terminals with a new non-terminal and introduce a new production, resulting in the following new rules:

A → X1 γ

X1 → B C

The rule **S → Aux NP VP** would be replaced by the two rules **S → X1 VP and X1 → Aux NP**.

The entire conversion process can be summarized as follows:

1. Copy all conforming rules to the new grammar unchanged.

2. Convert terminals within rules to dummy non-terminals.

3. Convert unit productions.

4. Make all rules binary and add them to new grammar.

$\mathcal{L}_1$ Grammar	$\mathcal{L}_1$ in CNF
$S \rightarrow NP\ VP$	$S \rightarrow NP\ VP$
$S \rightarrow Aux\ NP\ VP$	$S \rightarrow X1\ VP$
	$X1 \rightarrow Aux\ NP$
$S \rightarrow VP$	$S \rightarrow book \mid include \mid prefer$
	$S \rightarrow Verb\ NP$
	$S \rightarrow X2\ PP$
	$S \rightarrow Verb\ PP$
	$S \rightarrow VP\ PP$
$NP \rightarrow Pronoun$	$NP \rightarrow I \mid she \mid me$
$NP \rightarrow Proper\text{-}Noun$	$NP \rightarrow TWA \mid Houston$
$NP \rightarrow Det\ Nominal$	$NP \rightarrow Det\ Nominal$
$Nominal \rightarrow Noun$	$Nominal \rightarrow book \mid flight \mid meal \mid money$
$Nominal \rightarrow Nominal\ Noun$	$Nominal \rightarrow Nominal\ Noun$
$Nominal \rightarrow Nominal\ PP$	$Nominal \rightarrow Nominal\ PP$
$VP \rightarrow Verb$	$VP \rightarrow book \mid include \mid prefer$
$VP \rightarrow Verb\ NP$	$VP \rightarrow Verb\ NP$
$VP \rightarrow Verb\ NP\ PP$	$VP \rightarrow X2\ PP$
	$X2 \rightarrow Verb\ NP$
$VP \rightarrow Verb\ PP$	$VP \rightarrow Verb\ PP$
$VP \rightarrow VP\ PP$	$VP \rightarrow VP\ PP$
$PP \rightarrow Preposition\ NP$	$PP \rightarrow Preposition\ NP$

Figure 13.3 $\mathcal{L}_1$ Grammar and its conversion to CNF. Note that although they aren't shown here, all the original lexical entries from $\mathcal{L}_1$ carry over unchanged as well.

CKY Recognition

A two-dimensional matrix can be used to encode the structure of an entire tree. For a sentence of length n, we will work with the upper-triangular portion of an **(n+1)×(n+1)** matrix. Each cell **[i, j]** in this matrix contains the set of non-terminals that represent all the constituents that span positions i through j of the input.

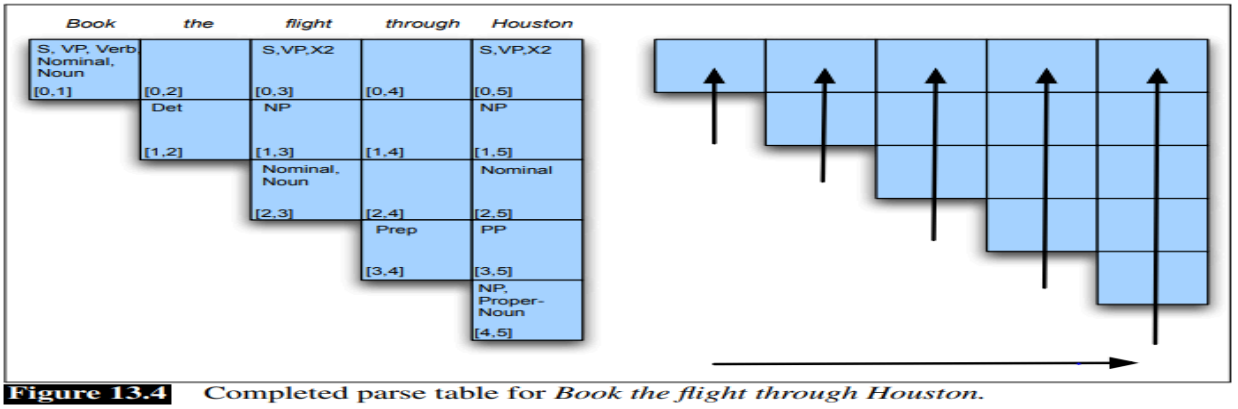


Figure 13.4 Completed parse table for *Book the flight through Houston.*

```
function CKY-PARSE(words, grammar) returns table

  for j ← from 1 to LENGTH(words) do
      for all {A | A → words[j] ∈ grammar}
              table[j − 1, j] ← table[j − 1, j] ∪ A
      for i ← from j − 2 down to 0 do
          for k ← i + 1 to j − 1 do
              for all {A | A → BC ∈ grammar and B ∈ table[i, k] and C ∈ table[k, j]}
                      table[i,j] ← table[i,j] ∪ A
```

Figure 13.5 The CKY algorithm.

CKY Parsing

The first change is to augment the entries in the table so that each non-terminal is paired with pointers to the table entries from which it was derived. The second change is to permit multiple versions of the same non-terminal to be entered into the table.

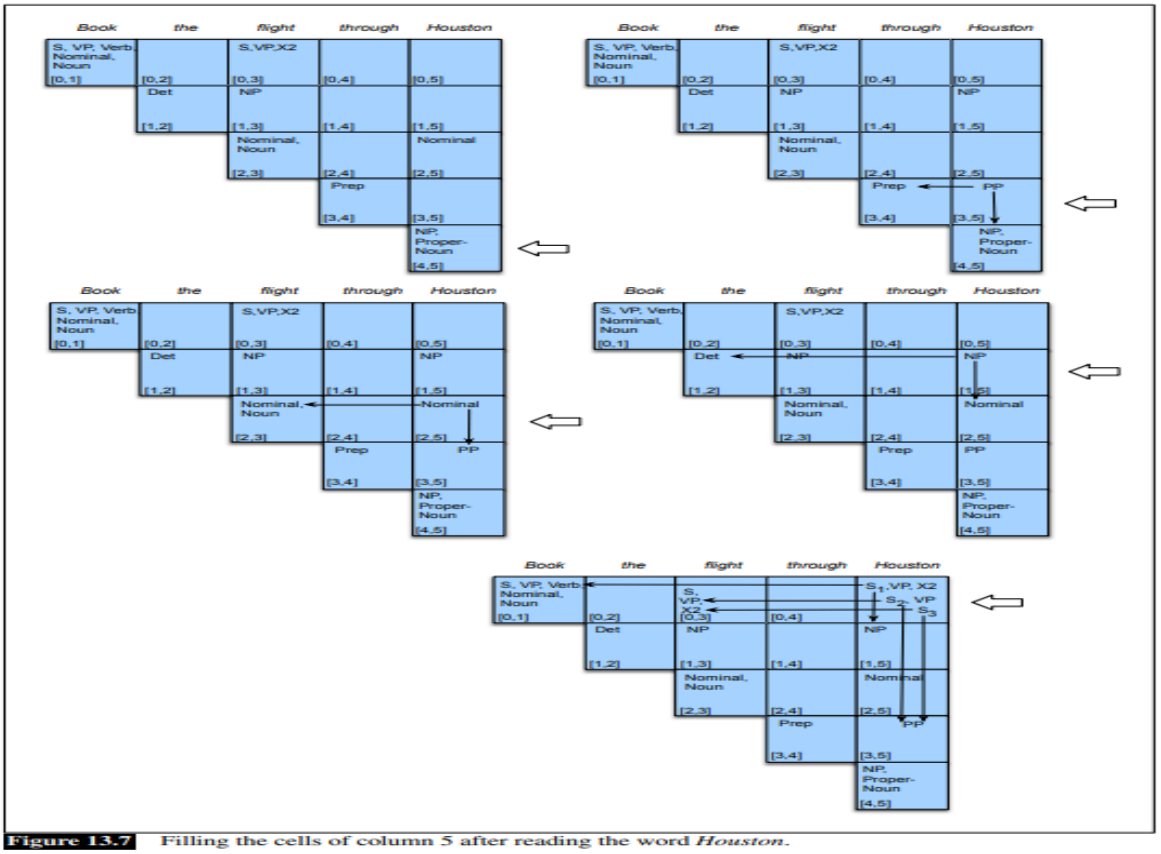
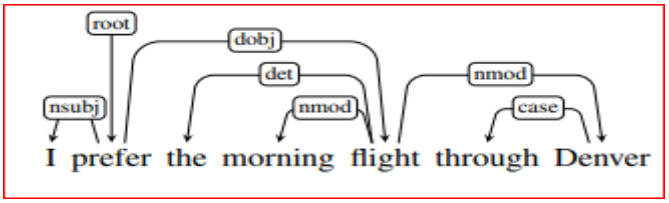


Figure 13.7 Filling the cells of column 5 after reading the word *Houston*.

You tube link: https://youtu.be/SFQ-owZaU_s?si=c5S9dkyjvUNE1akN

DEPENDENCY PARSING

In dependency formalisms, dependency grammars phrasal constituents and phrase-structure rules do not play a direct role. Instead, the syntactic structure of a sentence is described solely in terms of directed binary grammatical relations between the words, as in the following dependency parse:



Typed Dependency: Relations among the words are illustrated above the sentence with directed, labeled arcs from heads to dependents.

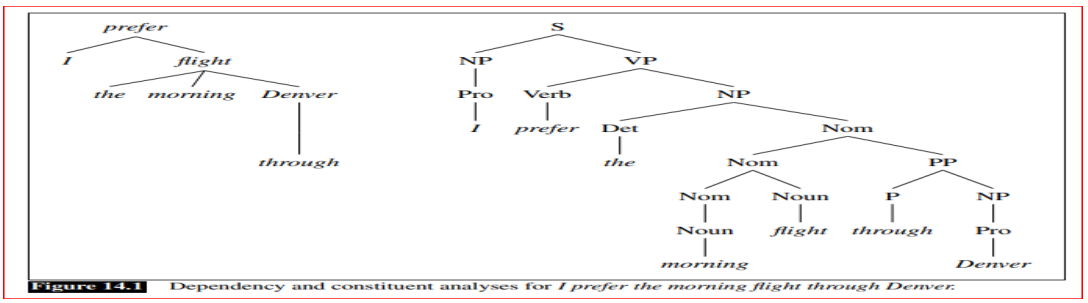


Figure 14.1 Dependency and constituent analyses for *I prefer the morning flight through Denver.*

Free Word Order: Ability to deal with languages that have relatively free word order. For example, word order in *Czech* can be much more flexible than *English*; a grammatical *object* might occur before or after a *location adverbial.*

Dependency Relations

The traditional linguistic notion of grammatical relation provides the basis for the **binary relations** that comprise these dependency structures.
The arguments to these relations consist of a **head** and a **dependent**.
The **head** word of a constituent was the central organizing word of a larger constituent (e.g, the primary noun in a noun phrase, or verb in a verb phrase). The remaining words in the constituent are either direct, or indirect, dependents of their head.
In dependency-based approaches, the **head-dependent relationship** is made explicit by directly linking heads to the words that are immediately dependent on them, bypassing the need for constituent structures.
Universal Dependency Relations (UD): an open community effort to annotate dependencies and other aspects of grammar across more than 100 languages provided by an inventory of 37 dependencies and relations.

Clausal Argument Relations	**Description**
NSUBJ	Nominal subject
DOBJ	Direct object
IOBJ	Indirect object
CCOMP	Clausal complement
XCOMP	Open clausal complement
Nominal Modifier Relations	**Description**
NMOD	Nominal modifier
AMOD	Adjectival modifier
NUMMOD	Numeric modifier
APPOS	Appositional modifier
DET	Determiner
CASE	Prepositions, postpositions and other case markers
Other Notable Relations	**Description**
CONJ	Conjunct
CC	Coordinating conjunction

Figure 14.2 Some of the Universal Dependency relations (de Marneffe et al., 2014).

Relation	Examples with *head* and **dependent**
NSUBJ	**United** *canceled* the flight.
OBJ	United *diverted* the **flight** to Reno.
	We *booked* her the first **flight** to Miami.
IOBJ	We *booked* **her** the flight to Miami.
COMPOUND	We took the **morning** *flight*.
NMOD	*flight* to **Houston**.
AMOD	Book the **cheapest** *flight*.
APPOS	*United*, a **unit** of UAL, matched the fares.
DET	**The** *flight* was canceled.
	Which *flight* was delayed?
CONJ	We *flew* to Denver and **drove** to Steamboat.
CC	We flew to Denver **and** *drove* to Steamboat.
CASE	Book the flight **through** *Houston*.

Figure 19.3 Examples of some Universal Dependency relations.

Dependency Formalisms

A dependency structure can be represented as a directed graph G = (V,A), consisting of a set of vertices V, and a set of ordered pairs of vertices A, which we'll call arcs.
That is, a **dependency tree** is a directed graph that satisfies the following constraints:
1. There is a single designated root node that has no incoming arcs.
2. With the exception of the root node, each vertex has exactly one incoming arc.
3. There is a unique path from the root node to each vertex in V

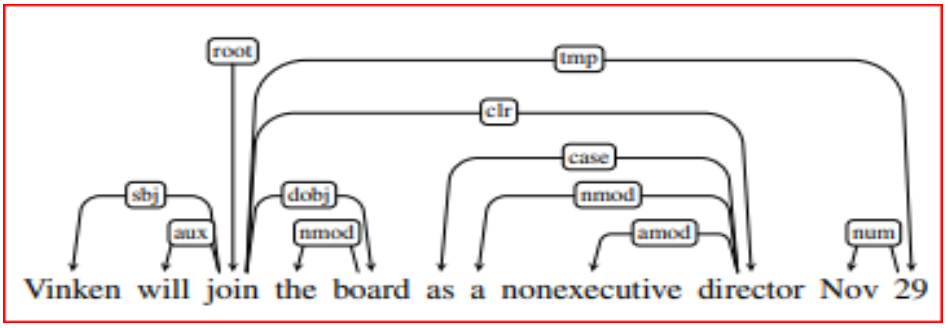
Projectivity

An arc from **head** to a dependent is said to be **projective** if there is a path from **head to every word** that lies between head and dependent in the sentence. A **dependency tree** is said to be **projective,** if all the arcs that make it up projective.

Dependency Treebanks

Here's a simple and effective algorithm from Xia and Palmer (2001):

1. Mark the head child of each node in a phrase structure, using the appropriate head rules.
2. In the dependency structure, make the head of each non-head child depend on the head of the head-child.



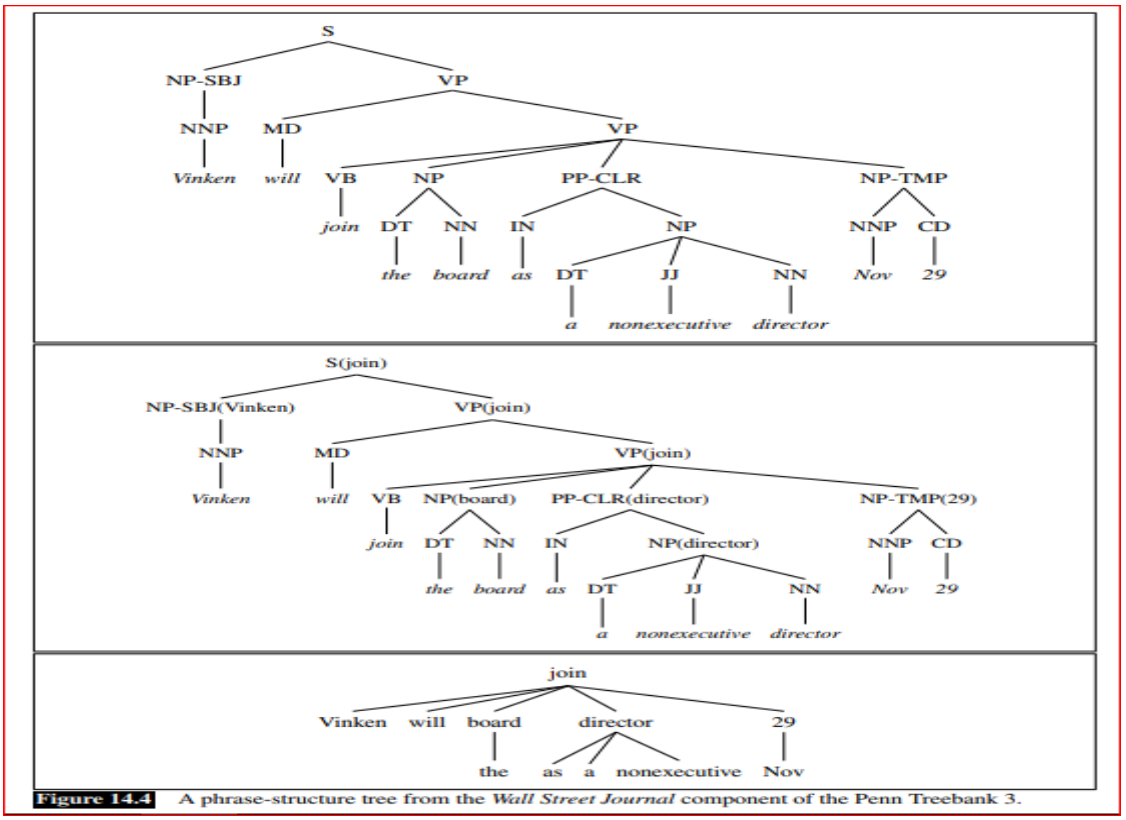


Figure 14.4 A phrase-structure tree from the *Wall Street Journal* component of the Penn Treebank 3.

EARLEY PARSING

Three kinds of problems that afflict standard bottom-up or top-down parsers, even when they have been augmented with filtering and other improvements: **left-recursive rules, ambiguity, and inefficient reparsing of subtrees.**

The **Earley algorithm (Earley, 1970)** uses a dynamic programming approach to efficiently implement a **parallel top-down search**. As with many dynamic programming solutions, this algorithm reduces an apparently exponential-time problem to a polynomial time one by eliminating the repetitive solution of sub-problems inherent in backtracking approaches. In this case, the dynamic programming approach leads to a worst-case behavior of **O (N ^{3}),** where N is the number of words in the input.

The core of the Earley algorithm is a single left-to-right pass that fills an array called a chart that has **N + 1** entries. For each word position in the sentence, the chart contains a list of states representing the partial parse trees that have been generated so far. By the end of the sentence, the chart compactly encodes all the possible parses of the input. Each possible subtree is represented only once and can thus be shared by all the parses that need it. The individual states contained within each chart entry contain three kinds of information: **a subtree corresponding to a single grammar rule, information about the progress made in completing this subtree, and the position of the subtree with respect to the input**. Graphically, we will use a **dot within the right-hand side** of a state's grammar rule to indicate the progress made in recognizing it. The resulting structure is called a **dotted rule**.

A **state's position** with respect to the input will be represented by *two numbers* indicating where the *state begins and where its dot lies*.

Example: Book that flight.

$$\begin{aligned} S &\rightarrow \bullet\ VP,\ [0,0] \\ NP &\rightarrow Det \bullet\ Nominal,\ [1,2] \\ VP &\rightarrow V\ NP \bullet,\ [0,3] \end{aligned}$$

The **first state**, with its dot to the left of its constituent, represents a topdown prediction for this particular kind of S. The first 0 indicates that the constituent predicted by this state should begin at the start of the input; the second 0 reflects the fact that the dot lies at the beginning as well. The **second state**, created at a later stage in the processing of this sentence, indicates that an NP begins at position 1, that a Det has been successfully parsed and that a Nominal is expected next. The **third state**, with its dot to the right of all its two constituents, represents the successful discovery of a tree corresponding to a VP that spans the entire input. These states can also be represented graphically, in which the states of the parse are edges, or arcs, and the chart as a whole is a directed acyclic graph.

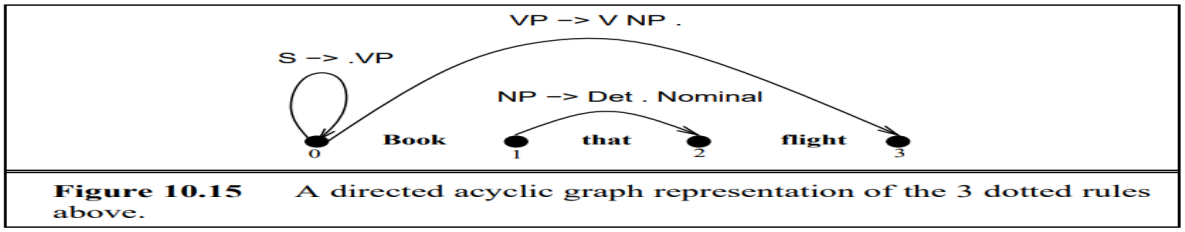


Figure 10.15 A directed acyclic graph representation of the 3 dotted rules above.

The PREDICTOR and the COMPLETER add states to the chart entry being processed, while the SCANNER adds a state to the next chart entry.

- **PREDICTOR**

The job of the PREDICTOR is to create new states representing top-down expectations generated during the parsing process. The PREDICTOR is applied to any state that has a non-terminal to the right of the dot that is not a part-of-speech category.

For example, applying the PREDICTOR to the state $S \rightarrow \bullet\ VP,\ [0,0]$ results in adding the states $VP \rightarrow \bullet\ Verb,\ [0,0]$ and $VP \rightarrow \bullet\ Verb\ NP,\ [0,0]$ to the first chart entry.

- **SCANNER**

When a state has a part-of-speech category to the right of the dot, the SCANNER is called to examine the input and incorporate a state corresponding to the predicted part-of-speech into the chart.

Returning to our example, when the state $VP \rightarrow \bullet Verb\ NP,\ [0,0]$ is processed, the SCANNER consults the current word in the input since the category following the dot is a part-of-speech. The SCANNER then notes that *book* can be a verb, matching the expectation in the current state. This results in the creation of the new state $VP \rightarrow Verb \bullet NP,\ [0,1]$. The new state is then added to the chart entry that *follows* the one currently being processed.

```
function EARLEY-PARSE(words, grammar) returns chart
  ENQUEUE((γ → • S, [0,0]), chart[0])
  for i ← from 0 to LENGTH(words) do
    for each state in chart[i] do
      if INCOMPLETE?(state) and
              NEXT-CAT(state) is not a part of speech then
          PREDICTOR(state)
      elseif INCOMPLETE?(state) and
              NEXT-CAT(state) is a part of speech then
          SCANNER(state)
      else
          COMPLETER(state)
    end
  end
  return(chart)
procedure PREDICTOR((A → α • B β, [i, j]))
    for each (B → γ) in GRAMMAR-RULES-FOR(B, grammar) do
        ENQUEUE((B → • γ, [j, j]), chart[j])
    end
procedure SCANNER((A → α • B β, [i, j]))
    if B ⊂ PARTS-OF-SPEECH(word[j]) then
        ENQUEUE((B → word[j], [j, j + 1]), chart[j+1])
procedure COMPLETER((B → γ •, [j, k]))
    for each (A → α • B β, [i, j]) in chart[j] do
        ENQUEUE((A → α B • β, [i, k]), chart[k])
    end
procedure ENQUEUE(state, chart-entry)
    if state is not already in chart-entry then
        PUSH(state, chart-entry)
    end
```

Figure 10.16 The Earley algorithm.

- **COMPLETER**

The COMPLETER is applied to a state when its dot has reached the right end of the rule.

For example, when the state $NP \rightarrow Det\ Nominal\bullet,\ [1,3]$ is processed, the COMPLETER looks for states ending at 1 expecting an *NP*. In the current example, it will find the state $VP \rightarrow Verb \bullet NP,\ [0,1]$ created by the Scanner. This results in the addition of a new complete state $VP \rightarrow Verb\ NP\bullet, [0,3]$.

Example:

- **Retrieving Parse Trees from a Chart**

Chart[0]

S0	$\gamma \rightarrow \bullet S$	[0,0]	[]	Dummy start state
S1	$S \rightarrow \bullet NP\ VP$	[0,0]	[]	Predictor
S2	$NP \rightarrow \bullet Det\ NOMINAL$	[0,0]	[]	Predictor
S3	$NP \rightarrow \bullet Proper\text{-}Noun$	[0,0]	[]	Predictor
S4	$S \rightarrow \bullet Aux\ NP\ VP$	[0,0]	[]	Predictor
S5	$S \rightarrow \bullet VP$	[0,0]	[]	Predictor
S6	$VP \rightarrow \bullet Verb$	[0,0]	[]	Predictor
S7	$VP \rightarrow \bullet Verb\ NP$	[0,0]	[]	Predictor

Chart[1]

S8	$Verb \rightarrow book \bullet$	[0,1]	[]	Scanner
S9	$VP \rightarrow Verb \bullet$	[0,1]	[S8]	Completer
S10	$S \rightarrow VP \bullet$	[0,1]	[S9]	Completer
S11	$VP \rightarrow Verb \bullet NP$	[0,1]	[S8]	Completer
S12	$NP \rightarrow \bullet Det\ NOMINAL$	[1,1]	[]	Predictor
S13	$NP \rightarrow \bullet Proper\text{-}Noun$	[1,1]	[]	Predictor

Chart[2]

S14	$Det \rightarrow that \bullet$	[1,2]	[]	Scanner
S15	$NP \rightarrow Det \bullet NOMINAL$	[1,2]	[S14]	Completer
S16	$NOMINAL \rightarrow \bullet Noun$	[2,2]	[]	Predictor
S17	$NOMINAL \rightarrow \bullet Noun\ NOMINAL$	[2,2]	[]	Predictor

Chart[3]

S18	$Noun \rightarrow flight \bullet$	[2,3]	[]	Scanner
S19	$NOMINAL \rightarrow Noun \bullet$	[2,3]	[S18]	Completer
S20	$NOMINAL \rightarrow Noun \bullet NOMINAL$	[2,3]	[S18]	Completer
S21	$NP \rightarrow Det\ NOMINAL \bullet$	[1,3]	[S14,S19]	Completer
S22	$VP \rightarrow Verb\ NP \bullet$	[0,3]	[S8,S21]	Completer
S23	$S \rightarrow VP \bullet$	[0,3]	[S22]	Completer
S24	$NOMINAL \rightarrow \bullet Noun$	[3,3]	[]	Predictor
S25	$NOMINAL \rightarrow \bullet Noun\ NOMINAL$	[3,3]	[]	Predictor

Figure 10.18 Sequence of states created in chart while parsing *Book that flight* including structural information.

Chart[1]	S12	$Verb \rightarrow book \bullet$	[0,1]	Scanner
Chart[2]	S23	$Det \rightarrow that \bullet$	[1,2]	Scanner
Chart[3]	S28	$Noun \rightarrow flight \bullet$	[2,3]	Scanner
	S29	$Nominal \rightarrow Noun \bullet$	[2,3]	(S28)
	S30	$NP \rightarrow Det\ Nominal \bullet$	[1,3]	(S23, S29)
	S33	$VP \rightarrow Verb\ NP \bullet$	[0,3]	(S12, S30)
	S36	$S \rightarrow VP \bullet$	[0,3]	(S33)

Figure 13.15 States that participate in the final parse of *Book that flight*, including structural parse information.

You tube link: https://youtu.be/1j6hB3O4hAM?si=WvcIxUXWr5nzI3MU

PROBABILISTIC CONTEXT-FREE GRAMMARS

The simplest approach is to count the number of times each rule is used in a corpus containing parsed sentences and use this to estimate the probability of each rule being used. For instance, consider a category C, where the grammar contains m rules, R_1 R_m, with the left-hand side C. You could estimate the probability of using rule R_j to derive C by the formula

$$\text{PROB}(R_j \mid C) \; Z \; \text{Count (\#times } R_j \text{ used)} \; / \; \text{Œ}_{j=1,m} \text{ (\#times } R_i \text{ used)}$$

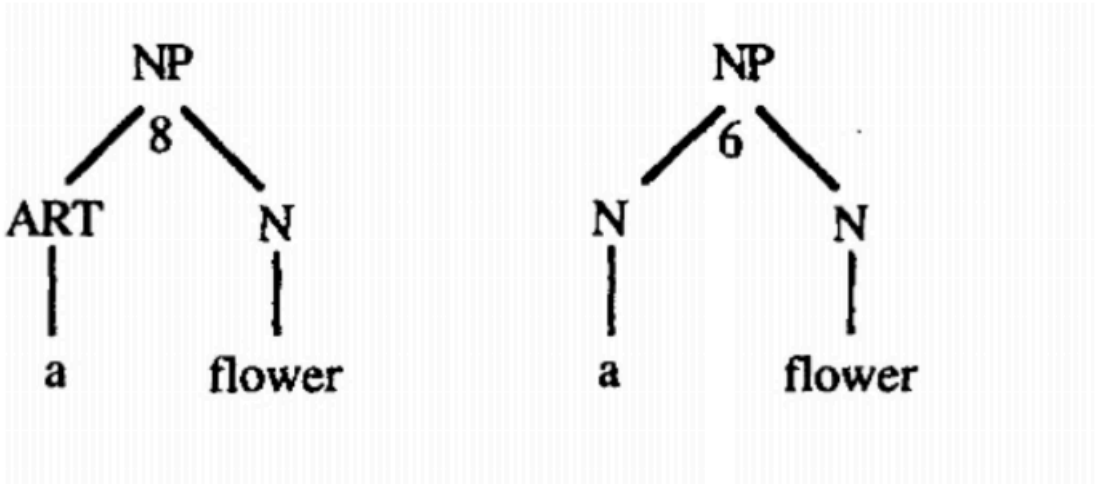
PROB (*the* | ART)	.54	PROB (*a* | ART)	.360
PROB (*flies* | N)	.025	PROB (*a* | N)	.001
PROB (*flies* | V)	.076	PROB (*flower* | N)	063
PROB (*like* | V)	.1	PROB (*flower* | V)	.05
PROB (*like* | P)	.068	PROB (*birds* | N)	.076
PROB (*like* | N)	.012		

Figure 7.6 The lexical-generation probabilities

Rule		Count for LHS	Count for Rule	Probability
1.	S → NP VP	300	300	1
2.	VP → V	300	116	.386
3.	VP → V NP	300	118	.393
4.	VP → V NP PP	300	66	.22
5.	NP → NP PP	1023	241	.24
6.	NP → N N	1023	92	.09
7.	NP → N	1023	141	.14
8.	NP → ART N	1023	558	.55
9.	PP → P NP	307	307	1

Grammar 7.17 A simple probabilistic grammar

With this assumption, a formalism can be developed based on the probability that a constituent **C** generates a sequence of words $w_i, w_{i+1},...,w_j$ written as $w_{i,j}$. This type of probability is called the **inside probability** because it assigns a probability to the word sequence inside the constituent. It is written as **PROB($w_{i,j}$ | C).**



7.18 The two possible ways that *a flower* could be an NP

$$
\begin{aligned}
&PROB(a\ flower \mid \text{NP}) = \\
&\quad PROB(\text{Rule 8} \mid \text{NP}) * PROB(a \mid \text{ART}) * PROB(flower \mid \text{N}) + \\
&\quad PROB(\text{Rule 6} \mid \text{NP}) * PROB(a \mid \text{N}) * \text{PROB}(flower \mid \text{N}) \\
&\quad = .55 * .36 * .06 + .09 * .001 * .06 \\
&\quad = .012
\end{aligned}
$$

$$
\begin{aligned}
&PROB(a\ flower\ blooms \mid \text{S}) = \\
&\quad PROB(\text{Rule 1} \mid \text{S}) * PROB(a\ flower \mid \text{NP}) * PROB(blooms \mid \text{VP}) + \\
&\quad PROB(\text{Rule 1} \mid \text{S}) * PROB(a \mid \text{NP}) * PROB(flower\ blooms \mid \text{VP})
\end{aligned}
$$

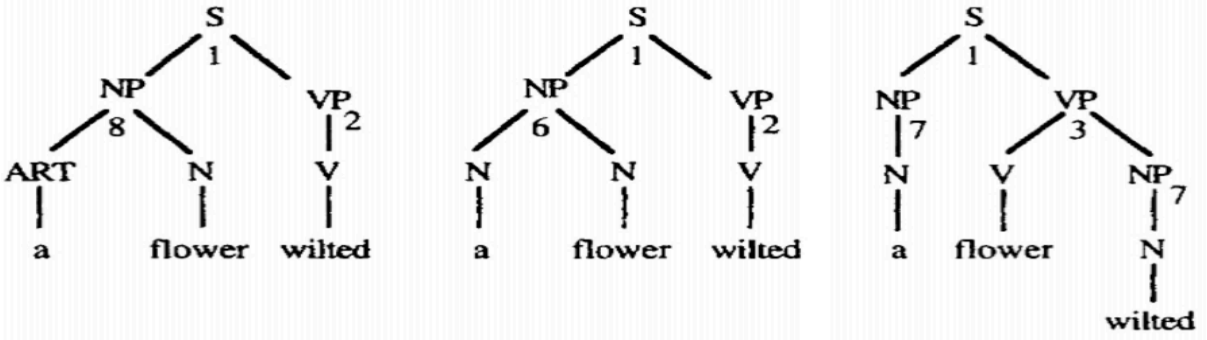


Figure 7.19 The three possible ways to generate *a flower wilted* as an S