Syntactic Analysis

Unit-III

Syed Rameem Zahra
(Assistant Professor)
Department of CSE, NSUT

Syntactic Analysis

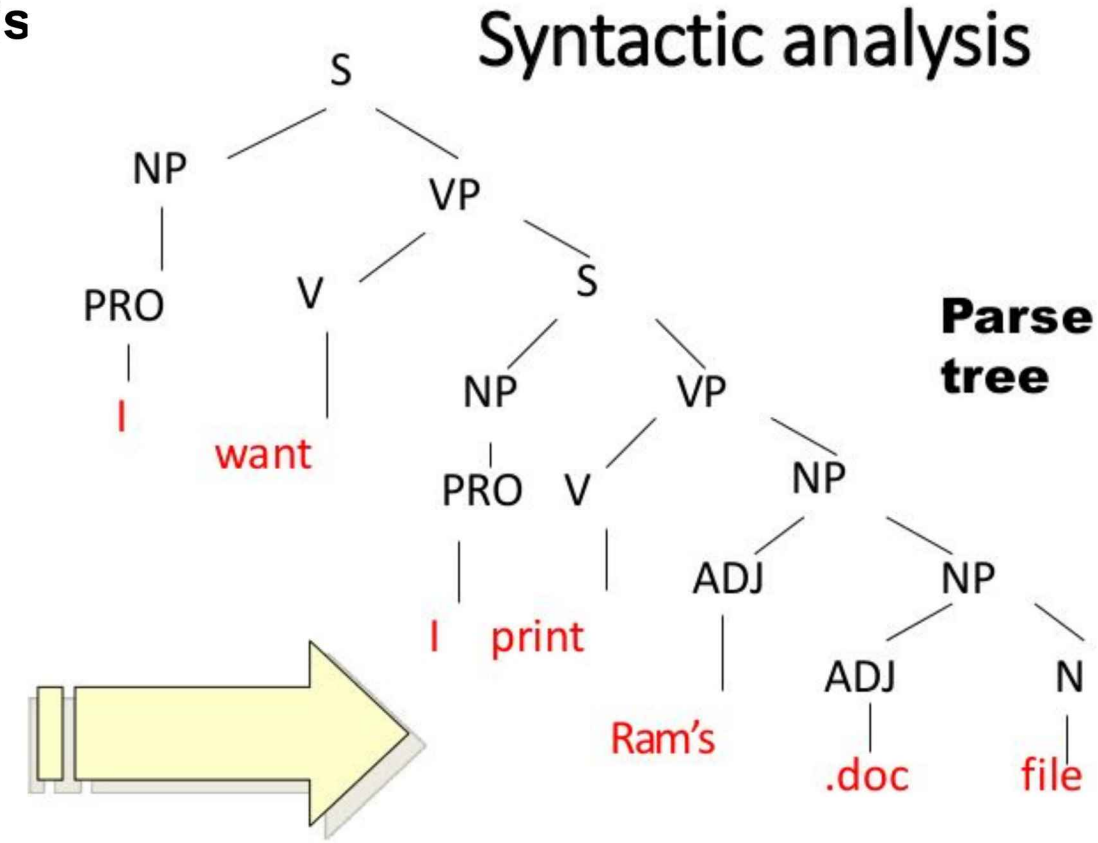
- Syntactic analysis or parsing or syntax analysis is the **third** phase of NLP.
- The purpose of this phase is to draw exact meaning, or you can say **dictionary meaning from the text**.
- **Syntax analysis checks the text for meaningfulness comparing to the rules of formal grammar**.
- For example, the sentence like “hot ice-cream” would be rejected by semantic analyzer.
- This method examines the structure of a sentence and performs detailed analysis of the sentence.
- In order to perform this, the system is expected to have thorough knowledge of the grammar of the language.
- The basic unit of any language is sentence, made up of group of words, having their own meanings and linked together to present an idea or thought.
- Apart from having meanings, words fall under categories called parts of speech: nouns, pronoun, adjectives, verb, adverb, prepositions, conjunction and interjections.

Syntactic Analysis

- In English language, a sentence S is made up of a noun phrase (**NP**) and a verb phrase (**VP**), i.e. **S=NP+VP**
- The given noun phrase (NP) normally can have an article or delimiter (**D**) or an adjective (**ADJ**) and the noun (N), i.e. **NP=D+ADJ+N**
- Also a noun phrase may have a prepositional phrase(PP) which has a preposition (**P**), a delimiter (**D**) and the noun (**N**),i.e. **PP=D+P+N**
- The verb phrase (**VP**) has a verb (**V**) and the object of the verb. The object of the verb may be a noun (**N**) and its determiner, i.e. **VP=V+N+D**
- These are some of the rules of the English grammar that helps one to construct a small parser for NLP.
- In this sense, 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’.

Example

- **Surface form: I want to print Ram's .doc file**
- **Morphological analysis**
  - **Stems:**
    - I (pronoun)
    - want (verb)
    - to (prep)
    - print (verb)
    - Ram (noun)
    - 's (possessive)
    - .doc (adj)
    - file (noun)
    - file (verb)



Parse Tree

- It may be defined as the graphical depiction of a derivation.
- The start symbol of derivation serves as the root of the parse tree.
- In every parse tree, the leaf nodes are terminals and interior nodes are non-terminals.
- A property of parse tree is that in-order traversal will produce the original input string.

The role of a parser

- Parsing (also known as syntax analysis) can be defined as a process of analyzing a text which contains a sequence of tokens, to determine its grammatical structure with respect to a given grammar.
- The main roles of the parse include:
  - To report any syntax error.
  - To recover from commonly occurring error so that the processing of the remainder of program can be continued.
  - To create parse tree.
  - To create symbol table: A symbol table stores information about scope and binding information about names, information about instances of various entities such as variable and function names, classes, objects, etc., and can be implemented in one of the following ways:
    - Linear (sorted or unsorted) list
    - Binary Search Tree
    - Hash table
  - To produce intermediate representations (IR). IR is designed to be conducive for further processing, such as optimization and translation

Types of Parsing

- **Top-down Parsing:**
  - In this kind of parsing, where the parser starts constructing the parse tree from the start symbol and then tries to transform the start symbol to the input.
  - The most common form of top-down parsing uses recursive procedure to process the input.
- **Bottom-up Parsing:**
  - In this kind of parsing, the parser starts with the input symbol and tries to construct the parser tree up to the start symbol.

Derivation

- In order to get the input string, we need a sequence of production rules.
- Derivation is a set of production rules.
- During parsing, we need to decide the non-terminal, which is to be replaced along with deciding the production rule with the help of which the non-terminal will be replaced.
- Types of Derivation:
  - **Left-most Derivation**
    - In the left-most derivation, the sentential form of an input is scanned and replaced from the left to the right. The sentential form in this case is called the left-sentential form.
  - **Right-most Derivation**
    - In the right-most derivation, the sentential form of an input is scanned and replaced from right to left. The sentential form in this case is called the right-sentential form.

Need of grammars

- The parse tree breaks down the sentence into structured parts so that the computer can easily understand and process it.
- In order for the parsing algorithm to construct this parse tree, a set of rules, which describe what tree structures are legal, need to be constructed.

Context free grammars

- It is the grammar that consists rules with a single symbol on the left-hand side of the rewrite rules.
- A context free grammar consists of:
  - a set of non-terminal symbols **N**
  - a set of terminal symbols **Σ** (disjoint from N)
  - a set of productions, **P**, each of the form $A \rightarrow \alpha$, where A is a non-terminal and α is a string of symbols from the infinite set of strings $(\Sigma \cup N)$
  - a designated start symbol **S**

Example1

- **The bird pecks the grains**

Making Rules:

Articles (DET) – a | an | the

Nouns – bird | birds | grain | grains

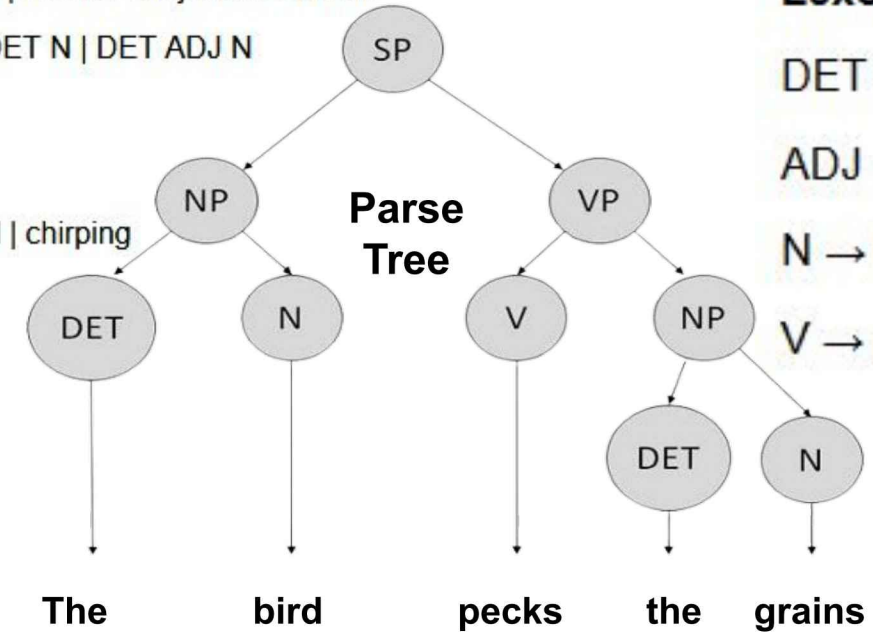
Noun Phrase (NP) – Article + Noun | Article + Adjective + Noun

= DET N | DET ADJ N

Verbs – pecks | pecking | pecked

Verb Phrase (VP) – NP V | V NP

Adjectives (ADJ) – beautiful | small | chirping



Parse Tree

CFG Productions:

S → NP VP

NP → DET N | DET ADJ N

VP → V NP

Lexocon –

DET → a | the

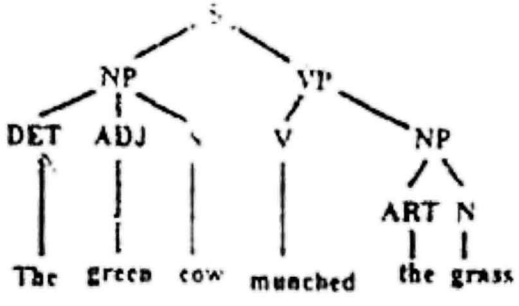
ADJ → beautiful | perching

N → bird | birds | grain | grains

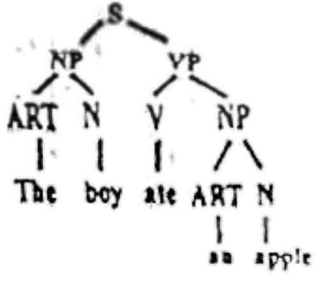
V → peck | pecks | pecking

Other Examples

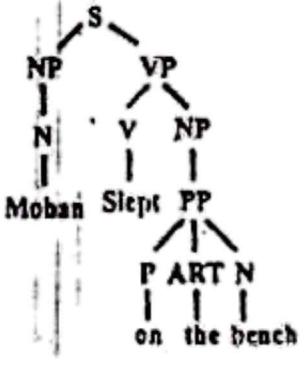
S	→	NP	VP			
S	→	DET	ADJ	N	VP	
S	→	ART	ADJ	cow	VP	
S	→	The	green	cow	V	NP
S	→	The	green	cow	munched	ART
S	→	The	green	cow	munched	the grass



S	→	NP	VP			
S	→	ART	N	VP		
S	→	The	boy	VP		
S	→	The	boy	V	NP	
S	→	The	boy	ate	NP	
S	→	The	boy	ate	ART	N
S	→	The	boy	ate	an	N
S	→	The	boy	ate	an	apple



S	→	NP	VP				
S	→	NP	V	NP			
S	→	N	V	PP			
S	→	N	V	P	ART	N	
		↓	↓	↓	↓	↓	
S	→	Mohan	Slept	on	the	bench	



Ambiguity in Grammar

- A grammar is said to be ambiguous if there exists more than one leftmost derivation or more than one rightmost derivation or more than one parse tree for the given input string.
- Example: below grammar can generate 2 parse trees by leftmost derivation for the string "3 * 2 + 5"
  - $E \rightarrow I$
  - $E \rightarrow E + E$
  - $E \rightarrow E * E$
  - $E \rightarrow (E)$
  - $I \rightarrow \varepsilon \mid 0 \mid 1 \mid 2 \mid ... \mid 9$

Unambiguous Grammar

- To convert ambiguous grammar to unambiguous grammar, we will apply the following rules:
  - If the **left associative operators** (+, -, *, /) are used in the production rule, then apply left recursion in the production rule. Left recursion means that the leftmost symbol on the right side is the same as the non-terminal on the left side. For example, **X → Xa**
  - If the **right associative operates** (^) is used in the production rule then apply right recursion in the production rule. Right recursion means that the rightmost symbol on the left side is the same as the non-terminal on the right side. For example, **X → aX**

Chomsky's Normal Form (CNF)

- A CFG is in CNF if all production rules satisfy one of the following conditions:
  - Start symbol generating ε. For example, S → ε.
  - A non-terminal generating two non-terminals. For example, C → AB.
  - A non-terminal generating a terminal. For example, C → a.
- For example:
  - G1 = {S → AB, S → c, A → a, B → b} => **In CNF**
  - G2 = {S → aA, A → a, B → c} => **Not in CNF**
- Converting a CFG to CNF:
  - S → a | aA | B
  - A → aBB | ε
  - B → Aa | b

S → a | XA | AX | b
A → RB
B → AX | b | a
X → a
R → XB

Step 1: Eliminate start symbol from the RHS. If the start symbol T is at the right-hand side of any production, create a new production as: S1 → S
Step 2: In the grammar, remove the null, unit and useless productions.
Step 3: Eliminate terminals from the RHS of the production if they exist with other non-terminals or terminals.
Step 4: Eliminate RHS with more than two non-terminals.

Removal of Null Productions:
Step 1 − Find out nullable non-terminal variables which derive ε.
Step 2 − For each production A → B, construct all productions A → X where X is obtained from 'B' by removing one or multiple non-terminals from Step 1.
Step 3 − Combine the original productions with the result of step 2 and remove ε - productions.

Removal of Unit Productions:
Step 1 − To remove A → B, add production A → x to the grammar rule whenever B → x occurs in the grammar. [x ∈ Terminal, x can be Null]
Step 2 − Delete A → B from the grammar.

To remove **useless productions** , we first find all the variables which will never lead to a terminal string such as variable 'B', We then remove all the productions in which variable 'B' occurs.

Greibach Normal Form (GNF)

- A CFG is in GNF if all the production rules satisfy one of the following conditions:
  - A start symbol generating ε. For example, S → ε.
  - A non-terminal generating a terminal. For example, A → a.
  - A non-terminal generating a terminal which is followed by any number of non-terminals. For example, S → aASB.
- For example:
  - G1 = {S → aAB | aB, A → aA| a, B → bB | b} => **In GNF**
  - G2 = {S → aAB | aB, A → aA | ε, B → bB | ε} => **Not In GNF**
- Converting CFG to GNF:
  - S → XB | AA
  - A → a | SA
  - B → b
  - X → a

S → aB | aCA | aBACA | aA | aBAA
A → aC | aBAC | a | aBA
C → aCAC | aBACAC | aAC | aBAAC
C → aCA | aBACA | aA | aBAA
B → b
X → a

Steps for converting CFG into GNF
Step 1 − Convert the grammar into CNF.
Step 2 − If the grammar consists of left recursion, eliminate it.
Step 3 − In the grammar, convert the given production rule into GNF form.

Probabilistic CFGs

- Augments each rule in P with a conditional probability:
  - $A \rightarrow \beta$ [p]
- where **p** is the probability that the non-terminal **A** will be expanded to the sequence **β**.
- Often referred to as:
  - P ($A \rightarrow \beta$) or
  - P ($A \rightarrow \beta | A$).

Example

$S \rightarrow NP\ VP$	[.80]	$Det \rightarrow that$ [.05] | the [.80] | a [.15]	
$S \rightarrow Aux\ NP\ VP$	[.15]	$Noun \rightarrow book$	[.10]
$S \rightarrow VP$	[.05]	$Noun \rightarrow flights$	[.50]
$NP \rightarrow Det\ Nom$	[.20]	$Noun \rightarrow meal$	[.40]
$NP \rightarrow Proper\text{-}Noun$	[.35]	$Verb \rightarrow book$	[.30]
$NP \rightarrow Nom$	[.05]	$Verb \rightarrow include$	[.30]
$NP \rightarrow Pronoun$	[.40]	$Verb \rightarrow want$	[.40]
$Nom \rightarrow Noun$	[.75]	$Aux \rightarrow can$	[.40]
$Nom \rightarrow Noun\ Nom$	[.20]	$Aux \rightarrow does$	[.30]
$Nom \rightarrow Proper\text{-}Noun\ Nom$	[.05]	$Aux \rightarrow do$	[.30]
$VP \rightarrow Verb$	[.55]	$Proper\text{-}Noun \rightarrow TWA$	[.40]
$VP \rightarrow Verb\ NP$	[.40]	$Proper\text{-}Noun \rightarrow Denver$	[.40] .60
$VP \rightarrow Verb\ NP\ NP$	[.05]	$Pronoun \rightarrow you$ [.40] | I [.60]	

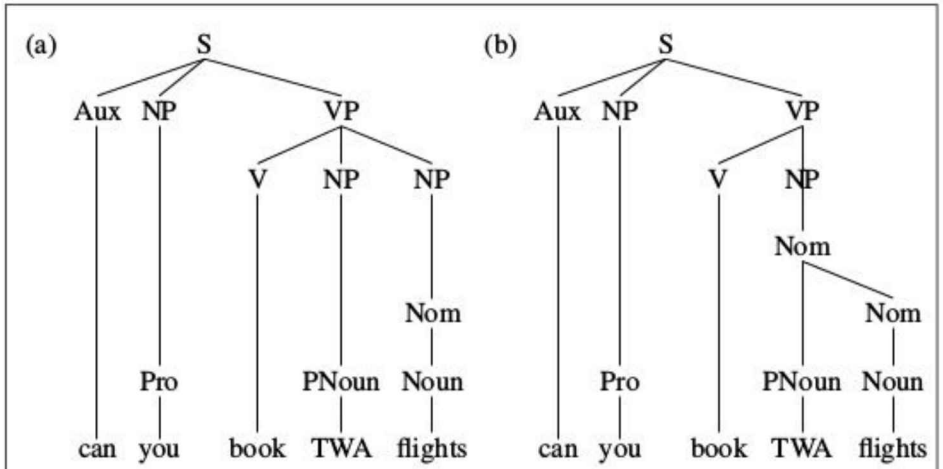
Why are PCFGs useful?

- Assigns a probability to each parse tree T
- Useful in disambiguation
  - Choose the most likely parse
  - Computing the probability of a parse
- Useful in language modeling tasks.
- Where do the probabilities come from?

from a ***treebank***:

$$P(\alpha \rightarrow \beta|\alpha) = \text{Count}(\alpha \rightarrow \beta)/\text{Count}(\alpha)$$

Example



Rules			P
S	→	Aux NP VP	.15
NP	→	Pro	.40
VP	→	V NP NP	.05
NP	→	Nom	.05
NP	→	PNoun	.35
Nom	→	Noun	.75
Aux	→	Can	.40
NP	→	Pro	.40
Pro	→	you	.40
Verb	→	book	.30
PNoun	→	TWA	.40
Noun	→	flights	.50

Rules			P
S	→	Aux NP VP	.15
NP	→	Pro	.40
VP	→	V NP	.40
NP	→	Nom	.05
Nom	→	PNoun Nom	.05
Nom	→	Noun	.75
Aux	→	Can	.40
NP	→	Pro	.40
Pro	→	you	.40
Verb	→	book	.30
Pnoun	→	TWA	.40
Noun	→	flights	.50

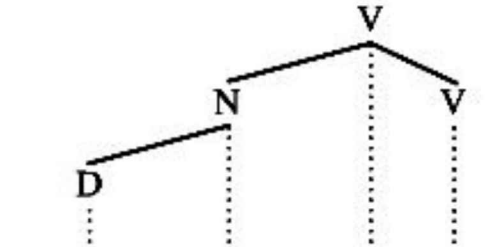
Dependency Grammar

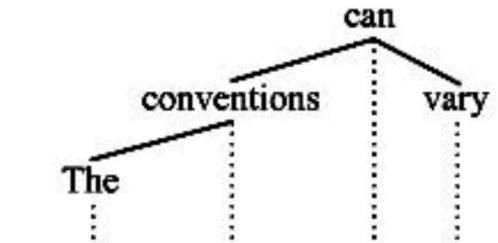
- Dependency grammar (DG) is a class of modern grammatical theories that are all based on the **dependency relation** (as opposed to the constituency relation of phrase structure).
- Dependency is the notion that linguistic units, e.g. words, are connected to each other by directed links.
- The **(finite) verb** is taken to be the structural **center** of clause structure. All other syntactic units (words) are either directly or indirectly connected to the verb in terms of the directed links, which are called dependencies.
- A dependency structure is determined by the relation between a word (a head) and its dependents.

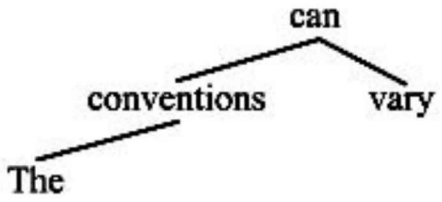
Dependency grammars

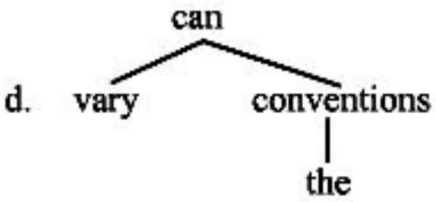
- The following frameworks are dependency-based:
  - Algebraic syntax
  - Operator grammar
  - Link grammar
  - Functional generative description
  - Lexicase
  - Meaning–text theory
  - Word grammar
  - Extensible dependency grammar
  - Universal Dependencies

Dependency grammars representations

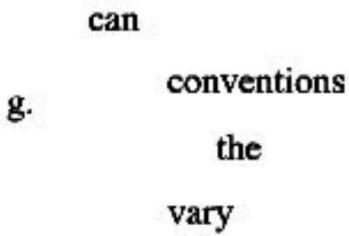








f. [[The] conventions] can [vary].



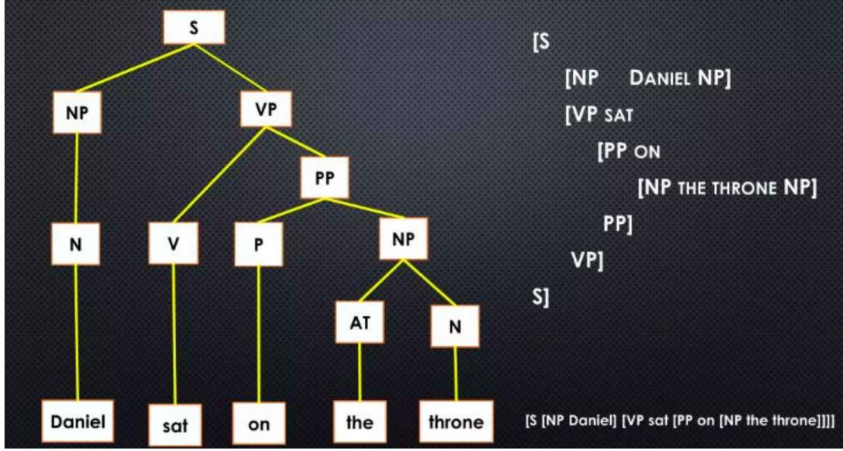
Types of dependencies

- Syntactic dependencies.
- Semantic dependencies
- Morphological dependencies
- Prosodic dependencies

Where does the grammar come from?

- Developed **manually**
- From a **treebank**

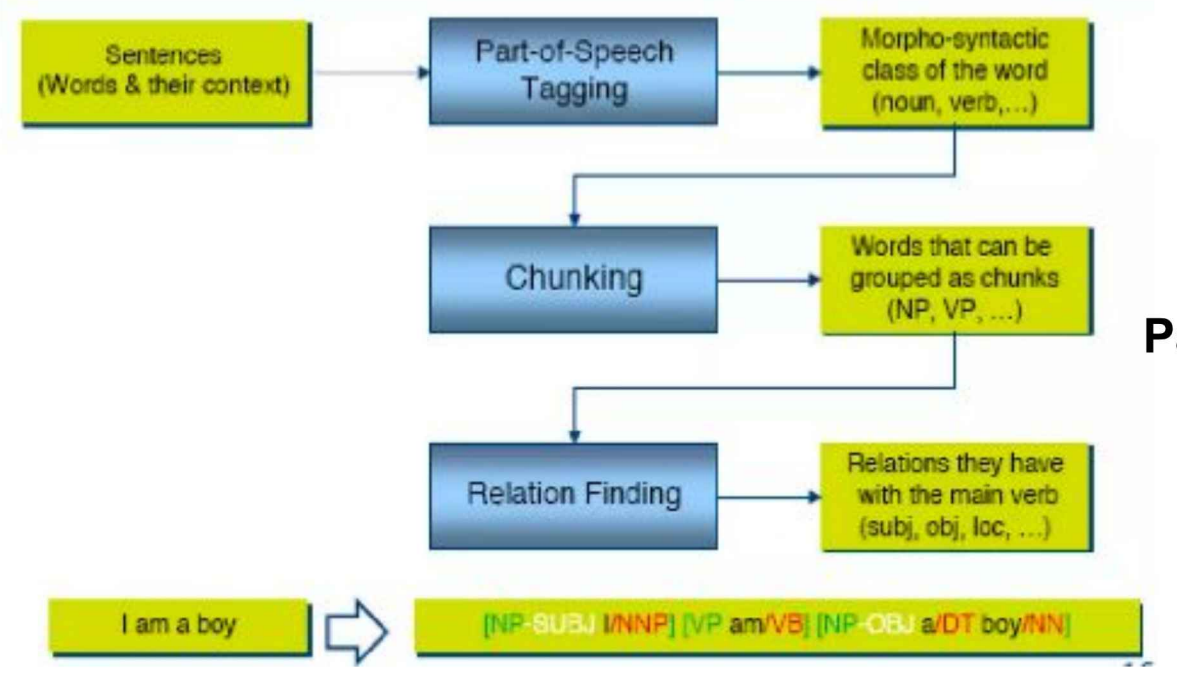
Treebanks



- How are they created?
  - Parse the collection with an automatic parser
  - Manually correct each parse as necessary.
- Requires detailed annotation guidelines that provide
  - a POS tagset
  - a grammar
  - instructions for how to deal with particular grammatical constructions.
- **Treebank Grammars:**
- Treebanks implicitly define a grammar.
- Simply take the local rules that make up the sub-trees in all the trees in the collection and you have a grammar.
- Not complete, but if you have decent size corpus, you'll have a grammar with decent coverage.
- Tend to be very flat due to the fact that they tend to avoid recursion.
- For example, the **Penn Treebank** has 4500 different rules for VPs.

Shallow Parsing

- **Shallow syntactic parsing** (also called "chunking") typically identifies noun, verb, preposition phrases, and so forth in a sentence, while **deep syntactic parsing** produces full parse trees, in which the syntactic function (e.g., Part of Speech, or POS) of each word or phrase is tagged with a short label



Typical Shallow Parser Architecture

Parsing Context Free Grammars

- Apply **dynamic programming**…
  - to find any tree that matches the data
  - (can be generalized to find the “most likely” parse also...)
- Algorithm which uses dynamic programming for parsing is “**CKY** - Cocke Kasami Younger” algorithm, which recognize languages defined by CFGs in CNF.

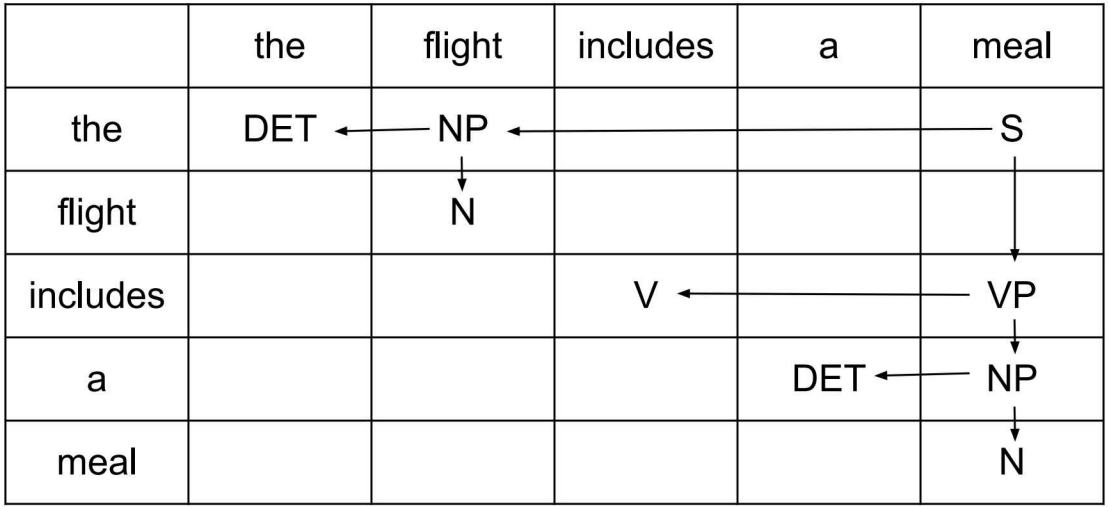
CYK Algorithm

- Let w be the n length string to be parsed and G represent the set of rules in our grammar with start state S.
- Construct a table DP for size n × n.
- If w = e (empty string) and S -> e is a rule in G then we accept the string else we reject.
- For i = 1 to n:
- For each variable A:
- We check if A -> b is a rule and b = w_i for some i:
- If so, we place A in cell (i, i) of our table.
- For l = 2 to n:
- For i = 1 to n-l+1:
- j = i+l-1
- For k = i to j-1:
- For each rule A -> BC:
- We check if (i, k) cell contains B and (k + 1, j) cell contains C:
- If so, we put A in cell (i, j) of our table.
- We check if S is in (1, n):
- If so, we accept the string
- Else, we reject.

Time Complexity – $O(n^3.|G|)$
Space Complexity – $O(n^2)$
$|G|$ is the number of rules in the given grammar.

Example1: CYK for CFG:

- Given a grammar G in CNF be:
  - S -> NP VP
  - NP -> DET N
  - VP -> V NP
  - V -> includes
  - DET -> the | a
  - N -> meal | flight
- Check if “the flight includes a meal” is in L(G)?
- We observe that S is in the cell (1, 5), Hence, the given string belongs to L(G).



Example2:

a 1	pilot 2	likes 3	flying 4	planes 5
DT	NP	-	-	S S
	NN	-	-	-
		VBZ	-	VP VP
			JJ VBG	NP VP
				NNS

$S \rightarrow NP\ VP$
$VP \rightarrow VBG\ NNS$
$VP \rightarrow VBZ\ VP$
$VP \rightarrow VBZ\ NP$
$NP \rightarrow DT\ NN$
$NP \rightarrow JJ\ NNS$
$DT \rightarrow a$
$NN \rightarrow pilot$
$VBZ \rightarrow likes$
$VBG \rightarrow flying$
$JJ \rightarrow flying$
$NNS \rightarrow planes$

Example3: Probabilistic CYK for CFG:

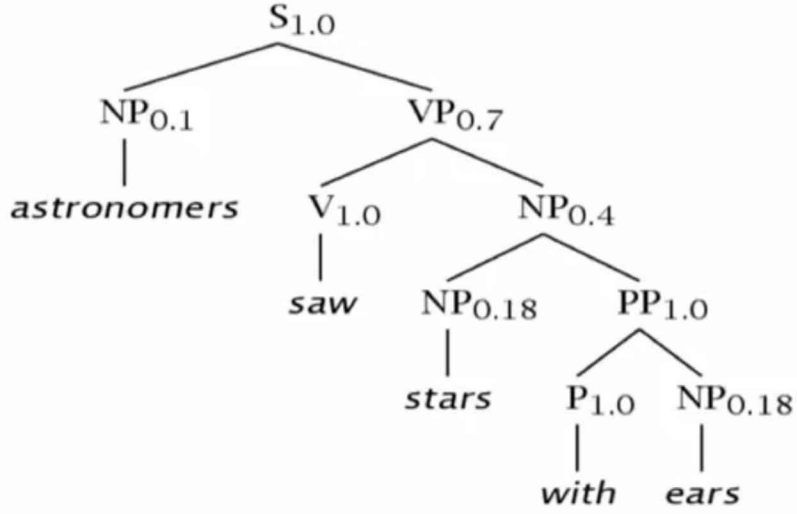
- Given a grammar G in CNF be:
  - S -> NP VP [0.80]
  - NP -> DET N [0.30]
  - VP -> V NP [0.2]
  - V -> includes [0.05]
  - DET -> the | a [0.4, 0.4]
  - N -> meal | flight [0.01, 0.02]
- Check if “the flight includes a meal” is in L(G) and find the probability of occurrence?
- We observe that S is in the cell (1, 5), Hence, the given string belongs to L(G) with a probability of 0.00000002304.

	the	flight	includes	a	meal
the	DET 0.4	NP 0.0024 (0.3*0.4*0.02)			S 0.00000 002304
flight		N 0.02			
includes			V 0.05		VP 0.000012
a				DET 0.4	NP 0.0012
meal					N 0.01

Example4:

S	→	NP VP	1.0	NP	→	NP PP	0.4
VP	→	V NP	0.7	NP	→	*astronomers*	0.1
VP	→	VP PP	0.3	NP	→	*ears*	0.18
PP	→	P NP	1.0	NP	→	*saw*	0.04
P	→	*with*	1.0	NP	→	*stars*	0.18
V	→	*saw*	1.0	NP	→	*telescope*	0.1

t_1:



w_{15} = *astronomers saw stars with ears*

$P(t_1) = 1.0 * 0.1 * 0.7 * 1.0 * 0.4 * 0.18 * 1.0 * 1.0 * 0.18$

$= 0.0009072$

$P(t_2) = 1.0 * 0.1 * 0.3 * 0.7 * 1.0 * 0.18 * 1.0 * 1.0 * 0.18$

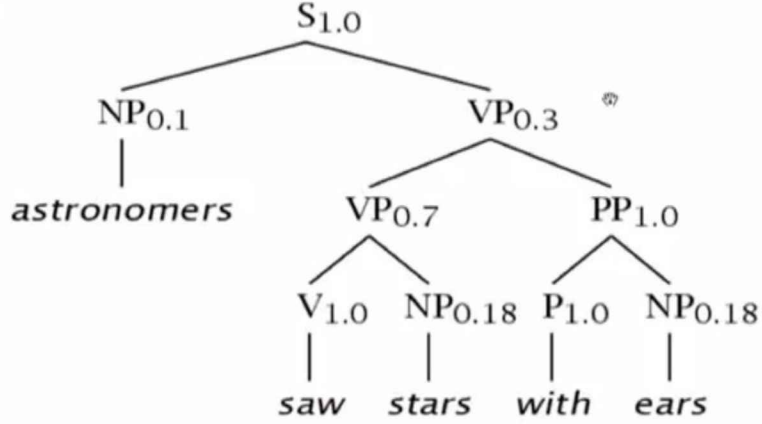
$= 0.0006804$

$P(w_{15}) = P(t_1) + P(t_2)$

$= 0.0009072 + 0.0006804$

$= 0.0015876$

t_2:



Weaknesses of PCFGs as Parsing Models

- Lack of sensitivity to lexical information; and
- Lack of sensitivity to structural preferences.

Lexicalized Probabilistic Context Free Grammar

N is a set of non-terminal symbols

Σ is a set of terminal symbols

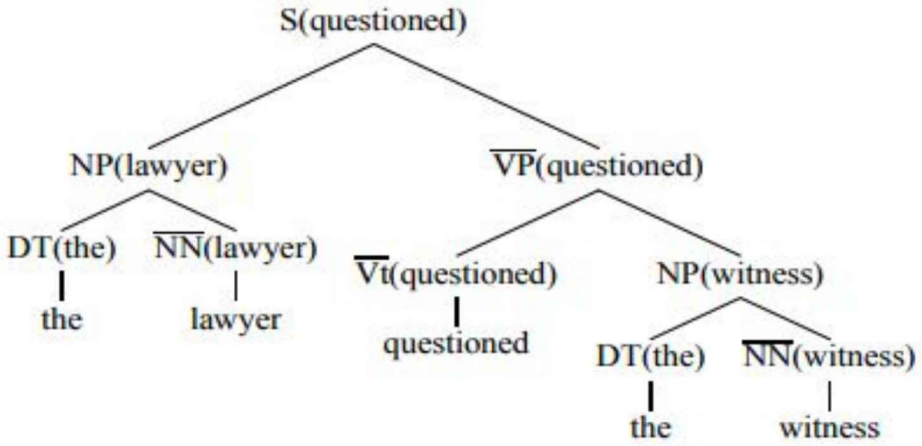
R is a set of rules which take one of three forms:

- $X(h) \rightarrow_1 Y_1(h)\ Y_2(w)$ for $X \in N$, and $Y_1, Y_2 \in N$, and $h, w \in \Sigma$
- $X(h) \rightarrow_2 Y_1(w)\ Y_2(h)$ for $X \in N$, and $Y_1, Y_2 \in N$, and $h, w \in \Sigma$
- $X(h) \rightarrow h$ for $X \in N$, and $h \in \Sigma$

$S \in N$ is a distinguished start symbol

using $\rightarrow_1$ to specify that the left child shares the lexical item with the parent, and $\rightarrow_2$ to specify that the right child shares the lexical item with the parent.

Example: Lexicalized Probabilistic Context Free Grammar



$$
\begin{aligned}
&\text{S(questioned)} \rightarrow_2 \text{NP(lawyer) VP(questioned)}\\
&\text{NP(lawyer)} \rightarrow_2 \text{DT(the) NN(lawyer)}\\
&\text{DT(the)} \rightarrow \text{the}\\
&\text{NN(lawyer)} \rightarrow \text{lawyer}\\
&\text{VP(questioned)} \rightarrow_1 \text{Vt(questioned) NP(witness)}\\
&\text{NP(witness)} \rightarrow_2 \text{DT(the) NN(witness)}\\
&\text{DT(the)} \rightarrow \text{the}\\
&\text{NN(witness)} \rightarrow \text{witness}
\end{aligned}
$$

- The model looks very similar to regular PCFGs, where the probability of a tree is calculated as a product of terms, one for each rule in the tree.
- One difference is that we have the γ(S, questioned) term for the root of the tree: this term can be interpreted as the probability of choosing the nonterminal S(questioned) at the root of the tree.

$$
\begin{aligned}
&\gamma(\text{S, questioned})\\
&\times q(\text{S(questioned)} \rightarrow_2 \text{NP(lawyer) VP(questioned)})\\
&\times q(\text{NP(lawyer)} \rightarrow_2 \text{DT(the) NN(lawyer)})\\
&\times q(\text{DT(the)} \rightarrow \text{the})\\
&\times q(\text{NN(lawyer)} \rightarrow \text{lawyer})\\
&\times q(\text{VP(questioned)} \rightarrow_1 \text{Vt(questioned) NP(witness)})\\
&\times q(\text{NP(witness)} \rightarrow_2 \text{DT(the) NN(witness)})\\
&\times q(\text{DT(the)} \rightarrow \text{the})\\
&\times q(\text{NN(witness)} \rightarrow \text{witness})
\end{aligned}
$$

Problems with CFGs

- We know that CFGs cannot handle certain things which are available in natural languages.
- In particular, CFGs cannot handle very well:
  - agreement
  - subcategorization
- We will look at a constraint-based representation schema which will allow us to represent fine-grained information such as:
  - number/person agreement
  - subcategorization
  - semantic categories like mass/count

Agreement Problem

- What is the problem with the following CFG rules:

  S → NP VP

  NP → Det NOMINAL

  NP → Pronoun

- *Answer*: Since these rules do not enforce number and person agreement constraints, they over-generate and allow the following constructs:

  * They sleeps

  * He sleep

  * A dogs

  * These dog

An Awkward Solution to Agreement Problem

- One way to handle the agreement phenomena in a strictly context-free approach is to encode the constraints into the non-terminal categories and then into CFG rules.
- For example, our grammar will be:

  S → SgS | PlS
  SgS → SgNP SgVP
  PlS → PlNP PlVP
  SgNP → SgDet SgNOMINAL
  SgNP → SgPronoun
  PlNP → PlDet PlNOMINAL
  PlNP → PlPronoun

- This solution will explode the number of non-terminals and rules.
- The resulting grammar will not be a clean grammar.

Subcategorization Problem

- What is the problem with the following CFG rules:

  VP → Verb

  VP → Verb NP

- *Answer*: Since these rules do not enforce subcategorization constraints, they over-generate and allow the following constructs:

  * They take

  * They sleep a glass

An Awkward Solution to Subcategorization Problem

- Again, one way to handle the subcategorization phenomena in a strictly context-free approach is to encode the constraints into the non-terminal categories and then into CFG rules.
- For example, our grammar will be:

VP → IntransVP | TransVP
IntransVP → IntransVerb
TransVP → TransVerb NP

- This solution will again explode the number of non-terminals and rules.
- Remember that we may have almost 100 subcategorizations for English verbs.
- The resulting grammar will not be a clean grammar.

A Better Solution

- A better solution for agreement and subcategorization problems is to treat terminals and non-terminals as complex objects with associated properties (called **features**) that can be manipulated.
- So, we may code rules as follows: (not CF rules anymore)

S → NP VP *Only if the number of the NP is equal to the number of the VP.*

- Where *number of* are **features** of NP and VP, and they are manipulated (they are checked to see whether they are equal or not) by the rule above.

Feature Structures

- We can encode the properties associated with grammatical constituents (terminals and non-terminals) by using **Feature Structures**.
- A **feature structure** is a set of **feature-value** pairs.
  - A **feature** is an atomic symbol.
  - A **value** is either an atomic value or another feature structure.
- A feature structure can be illustrated by a matrix-like diagram (called **attribute-value matrix**).

$$\begin{bmatrix} Feature-1 & Value-1 \\ Feature-2 & Value-2 \\ . & \\ Feature-n & Value-n \end{bmatrix}$$

Example - Feature Structures

$$[NUMBER \quad SG]$$

$$\begin{bmatrix} NUMBER & SG \\ PERSON & 3 \end{bmatrix}$$

$$\begin{bmatrix} CAT & NP \\ NUMBER & SG \\ PERSON & 3 \end{bmatrix}$$

$$\begin{bmatrix} CAT & NP \\ AGREEMENT & \begin{bmatrix} NUMBER & SG \\ PERSON & 3 \end{bmatrix} \end{bmatrix}$$

Reentrant Feature Structures

- We will allow multiple features in a feature structure to share the same values.
- They share the same structures not just that they have same value.

$$\begin{bmatrix} CAT & S \\ HEAD & \begin{bmatrix} AGREEMENT & (1)\begin{bmatrix} NUMBER & SG \\ PERSON & 3 \end{bmatrix} \\ SUBJECT & [AGREEMENT \quad (1)] \end{bmatrix} \end{bmatrix}$$

Feature Path

- A **feature path** is a list of features through a feature structure leading to a particular value.
- For example,

  <HEAD AGREEMENT NUMBER> leads to SG

  <HEAD SUBJECT AGREEMENT PERSON> leads to 3

- We will use feature paths in the constraints of the rules.

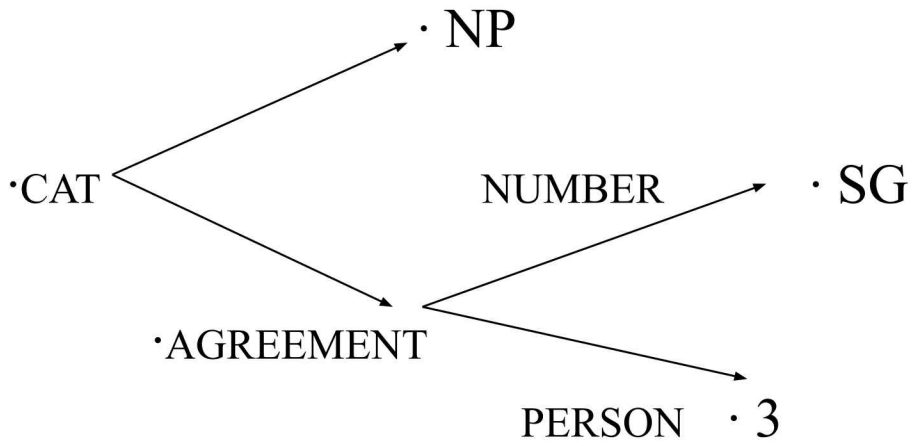
  S → NP VP

  <NP AGREEMENT> = <VP AGREEMENT>

DAG Representation of Feature Structures

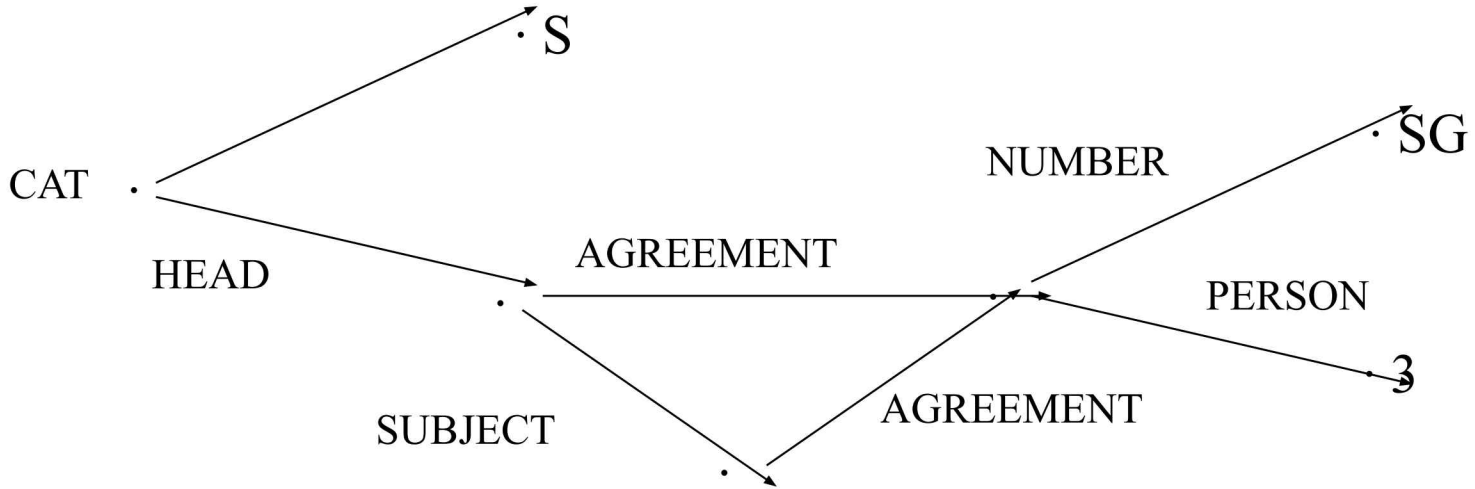
- A feature structure can also be represented by using a DAG (directed acyclic graph).

$$\begin{bmatrix} CAT & NP \\ AGREEMENT & \begin{bmatrix} NUMBER & SG \\ PERSON & 3 \end{bmatrix} \end{bmatrix}$$



DAG of A Reentrant Feature Structure

$$\begin{bmatrix} CAT & S \\ HEAD & \begin{bmatrix} AGREEMENT & (1)\begin{bmatrix} NUMBER & SG \\ PERSON & 3 \end{bmatrix} \\ SUBJECT & [AGREEMENT \quad (1)] \end{bmatrix} \end{bmatrix}$$



Unification of Feature Structures

- By the unification of feature structures, we will:
  - Check the compatibility of two feature structures.
  - Merge the information in two feature structures.
- The result of a unification operation of two feature structures can be:
  - **unifiable** -- they will merge into a single feature structure
  - **fails** -- if two feature structures are not compatible.

- We will look at how does this unification process perform the above tasks.

Unification Example

- We say that two feature structures can be unified if two feature structures that make them up are compatible.

$$[NUMBER \quad SG] \sqcup [NUMBER \quad SG] = [NUMBER \quad SG]$$

succeeds

$$[NUMBER \quad SG] \sqcup [NUMBER \quad PL]$$

fails

Unification Operator

Unification Example (cont.)

The unification process can bind an undefined value to a value, or can merge the information in two feature structures.

$$[NUMBER \quad SG] \sqcup [NUMBER \quad []] = [NUMBER \quad SG]$$

$$[NUMBER \quad SG] \sqcup [PERSON \quad 3] = \begin{bmatrix} NUMBER & SG \\ PERSON & 3 \end{bmatrix}$$

Unification Example -- Complex Structures

$$\begin{bmatrix} AGREEMENT & (1) \\ SUBJECT & [AGREEMENT \quad (1)] \end{bmatrix}$$

$$\begin{bmatrix} SUBJECT & \begin{bmatrix} AGREEMENT & \begin{bmatrix} PERSON & 3 \\ NUMBER & SG \end{bmatrix} \end{bmatrix} \end{bmatrix}$$

$$= \begin{bmatrix} AGREEMENT & (1) \\ SUBJECT & \begin{bmatrix} AGREEMENT & (1) \begin{bmatrix} PERSON & 3 \\ NUMBER & SG \end{bmatrix} \end{bmatrix} \end{bmatrix}$$