Unit-II

Syntax

- Parsing Natural Languages
- Tree Banks: A Data Driven Approach to Syntax
- Representation of Syntactic Structure
- Parsing Algorithms
- Models for Ambiguity Resolution in Parsing
- Multilingual Issues

Syntax-Introduction

- Parsing uncovers the hidden structure of linguistic input.
- In Natural Language Applications the predicate structure of sentences can be useful.
- In NLP the syntactic analysis of the input can vary from:
  - Very low level- POS tagging.
  - Very high level- recovering a structural analysis that identifies the dependency between predicates and arguments in the sentence.
- The major problem in parsing natural language is the problem of ambiguity.

Parsing Natural Languages

- Let us look at the following spoken sentences:
  - He wanted to go for a drive in movie.
  - He wanted to go for a drive in the country.
- There is a natural pause between drive and in in the second sentence.
- This gap reflects an underlying hidden structure to the sentence.
- Parsing provides a structural description that identifies such a break in the intonation.

Parsing Natural Languages

- Let us look at another sentence:
  - The cat who lives dangerously had nine lives.
- A text-to-speech system needs to know that the first instance of the word lives is a verb and the second instance a noun.
- This is an instance of POS tagging problem.
- Another important application where parsing is important is text summarization.

Parsing Natural Languages

- Let us look at examples for summarization:
  - Beyond the basic level, the operations of the three products vary widely.
- The above sentence can be summarized as follows:
  - The operations of the products vary.
- To do this task we first parse the first sentence.

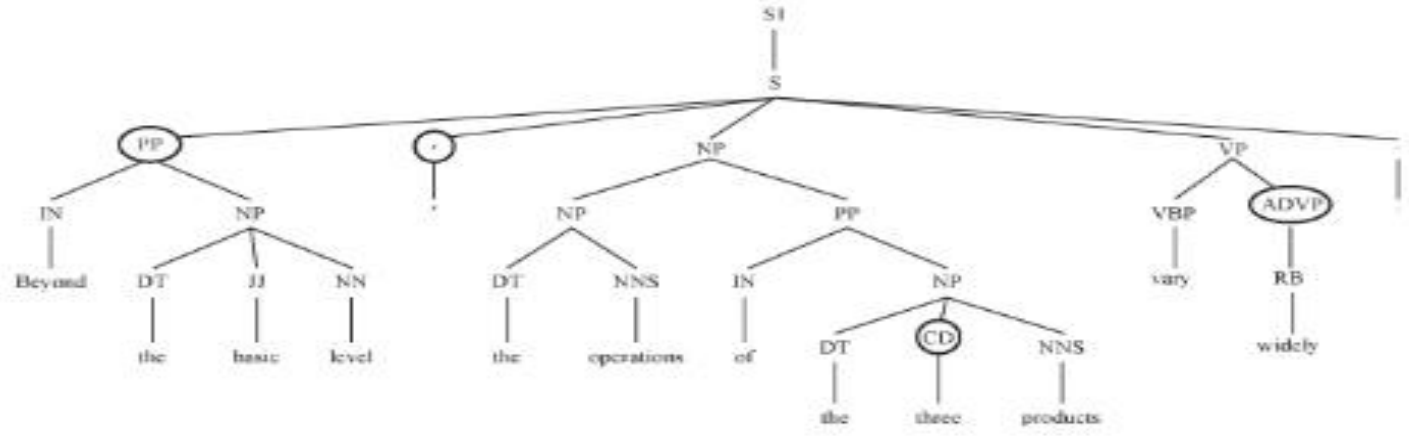


Figure 3–1. Parser output for sentence

Parsing Natural Languages

- Deleting the circled constituents PP, CD and ADVP in the previous diagram results in the short sentence.
- Let us look at another example:
  - Open borders imply increasing racial fragmentation in EUROPEAN COUNTRIES.
- In the above example the capitalized phrase can be replaced with other phrases without changing the meaning of the sentence.
  - Open borders imply increasing racial fragmentation in *the countries of Europe*.
  - Open borders imply increasing racial fragmentation in *European states.*
  - Open borders imply increasing racial fragmentation in *Europe.*
  - Open borders imply increasing racial fragmentation in *European Nations*.
  - Open borders imply increasing racial fragmentation in *the European countries.*

Parsing Natural Languages

- In NLP syntactic parsing is used in many applications like:
  - Statistical Machine Translation
  - Information extraction from text collections
  - Language summarization
  - Producing entity grids for language generation
  - Error correction in text
  - Knowledge acquisition from language

Treebanks: A Data-Driven Approach to Syntax

- Parsing recovers information that is not explicit in the input sentence.
- Parsers require some additional knowledge beyond the input sentence that should be produced as output.
- We can write down the rules of the syntax of a sentence as a CFG.
- Here we have a CFG which represents a simple grammar of transitive verbs in English (verbs that have a subject and object noun phrase (NP), plus modifiers of verb phrases (VP) in the form of prepositional phrases (PP)).

Treebanks: A Data-Driven Approach to Syntax

```
S -> NP VP
NP -> 'John' | 'pockets' | D N | NP PP
VP -> V NP | VP PP
V -> 'bought'
D -> 'a'
N -> 'shirt'
PP -> P NP
P -> 'with'
```

The above CFG can produce the syntax analysis of a sentence like:
John bought a shirt with pockets

Treebanks: A Data-Driven Approach to Syntax

- Parsing the previous sentence gives us two possible derivations.

```
(S (NP John)
  (VP (VP (V bought)
        (NP (D a)
          (N shirt)))
      (PP (P with)
        (NP pockets))))
```

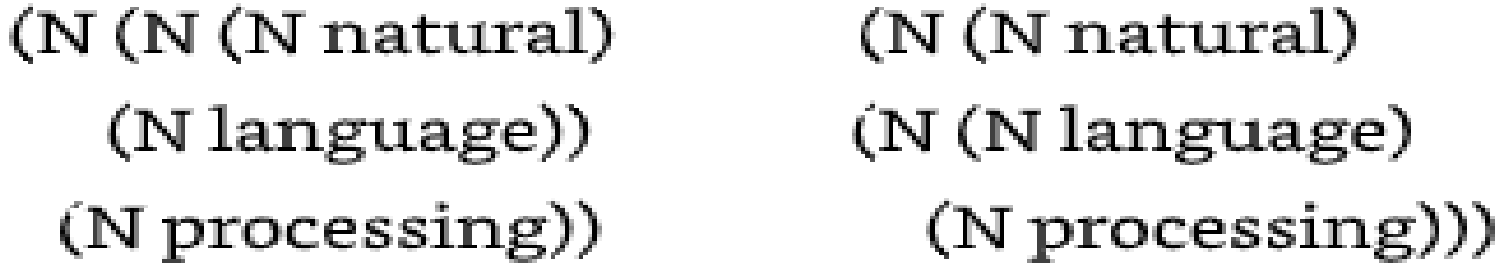
```
(S (NP John)
      (VP (V bought)
    (NP (NP (D a)
          (N shirt))
        (PP (P with)
            (NP pockets)))))
```

Treebanks: A Data-Driven Approach to Syntax

- Writing a CFG for the syntactic analysis of natural language is problematic.
- A simple list of rules does not consider interactions between different components in the grammar.
- Listing all possible syntactic constructions in a language is a difficult task.
- It is difficult to exhaustively list lexical properties of words. This is a typical knowledge acquisition problem.
- One more problem is that the rules interact with each other in combinatorially explosive ways.

Treebanks: A Data-Driven Approach to Syntax

- Let us look at an example of noun phrases as a binary branching tree.
- N -> NN (Recursive Rule)
- N -> ‘natural’|’language’|’processing’|’book’
- For the input ‘natural language processing’ the recursive rules produce two ambiguous parses.



Treebanks: A Data-Driven Approach to Syntax

- For CFGs it can be proved that the number of parsers obtained by using the recursive rule n times is the Catalan number (1,1,2,5,14,42, 132, 429, 1430, 4862,….) of n:

$$\mathrm{Cat}(n) = \frac{1}{n+1}\begin{pmatrix} 2n \\ n \end{pmatrix}$$

- For the input “natural language processing book” only one out of the five parsers obtained using the above CFG is correct:

```
(N (N (N (N (N natural)
            (N language))
         (N processing))
      (N book))
```

Treebanks: A Data-Driven Approach to Syntax

- This is the second knowledge acquisition problem- We need to know not only the rules but also which analysis is most plausible for a given input sentence.
- The construction of a **tree bank** is a data driven approach to syntax analysis that allows us to address both the knowledge acquisition bottlenecks in one stroke.
- A treebank is a collection of sentences where each sentence is provided a complete syntax analysis.
- The syntax analysis for each sentence has been judged by a human expert.

Treebanks: A Data-Driven Approach to Syntax

- A set of annotation guidelines is written before the annotation process to ensure a consistent scheme of annotation throughout the tree bank.
- No set of syntactic rules are provided by a treebank.
- No exhaustive set of rules are assumed to exist even though assumptions about syntax are implicit in a treebank.
- The consistency of syntax analysis in a treebank is measured using interannotator agreement by having approximately 10% overlapped material annotated by more than one annotator.
- Treebanks provide annotations of syntactic structure for a large sample of sentences.

Treebanks: A Data-Driven Approach to Syntax

- A supervised machine learning method can be used to train the parser.
- Treebanks solve the first knowledge acquisition problem of finding the grammar underlying the syntax analysis because the analysis is directly given instead of a grammar.
- The second problem of knowledge acquisition is also solved by treebanks.
- Each sentence in a treebank has ben given its most plausible syntactic analysis.
- Supervised learning algorithms can be used to learn a scoring function over all possible syntax analyses.

Treebanks: A Data-Driven Approach to Syntax

- For real time data the parser uses the scoring function to return the syntax analysis that has the highest score.
- Two main approaches to syntax analysis that are used to construct treebanks are:
  - Dependency graphs
  - Phase structure trees
- These two representations are very closely connected to each other and under some assumptions one can be converted to the other.
- Dependency analysis is used for free word order languages like Indian languages.
- Phrase structure analysis is used to provide additional information about long distance dependencies for languages like English and French.

Treebanks: A Data-Driven Approach to Syntax

- In the discussion to follow we examine three main components for building a parser:
- **The representation of syntactic structure**- it involves the use of a varying amount of linguistic knowledge to build a treebank.
- **The training and decoding algorithms**- they deal with the potentially exponential search space.
- **Methods to model ambiguity**- provides a way to rank parses to recover the most likely parse.

Representation of Syntactic Structure

- Syntax Analysis using dependency graphs
- Syntax Analysis using phrase structure trees

Syntax Analysis using Dependency Graphs

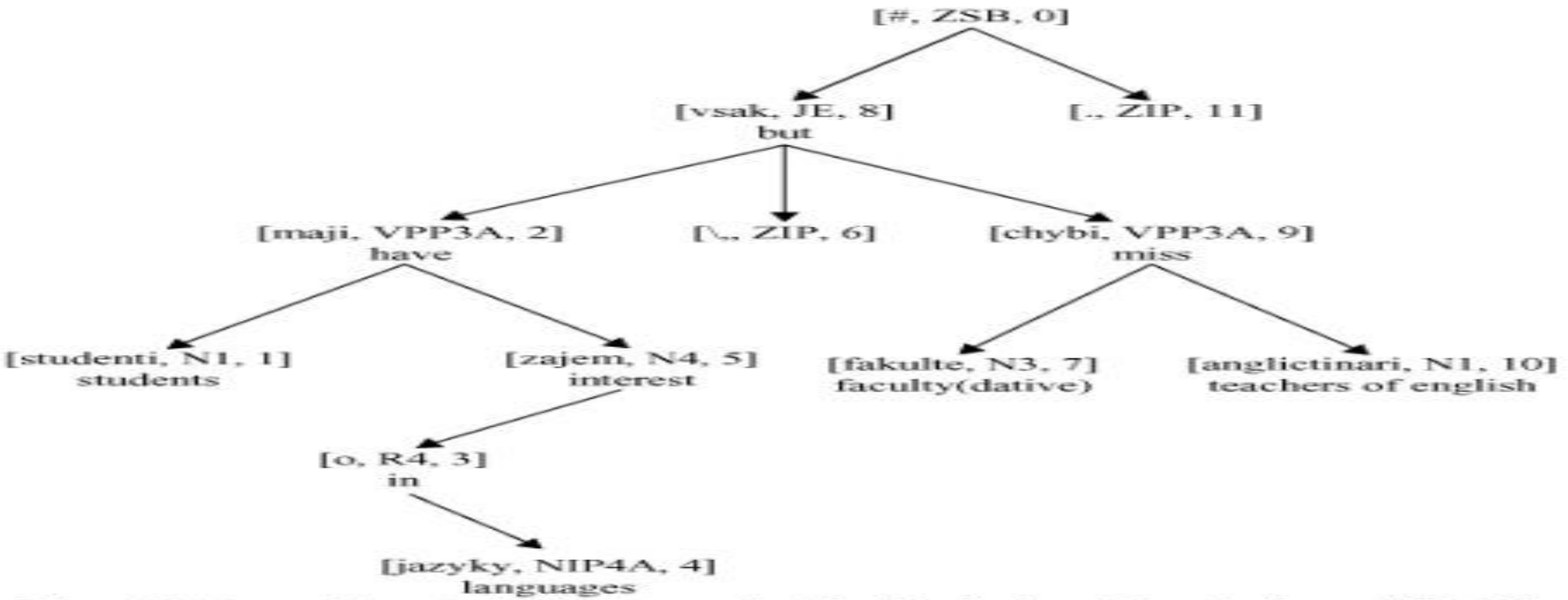
- In dependency graphs the head of a phrase is connected with the dependents in that phrase using directed connections.
- The head-dependent relationship can be semantic (head-modifier) or syntactic (head-specifier).
- The main difference between dependency graphs and phrase structure trees is that dependency analysis make minimal assumptions about syntactic structure.
- Dependency graphs treat the words in the input sentence as the only vertices in the graph which are linked together by directed arcs representing syntactic dependencies.

Syntax Analysis using Dependency Graphs

- One typical definition of dependency graph is as follows:
  - In dependency syntactic parsing the task is to derive a syntactic structure for an input sentence by identifying the syntactic head of each word in the sentence.
  - The nodes are the words of the input sentence and the arcs are the binary relations from head to dependent.
  - It is often assumed that all words except one have a syntactic head.
  - It means that the graph will be a tree with the single independent node as the root.
  - In labeled dependency parsing the parser assigns a specific type to each dependency relation holding between head word and dependent word.
- In the current discussion we will be discussing about dependency trees only where each word depends on exactly one parent either another word or a dummy symbol.

Syntax Analysis using Dependency Graphs

- In dependency trees the 0 index is used to indicate the root symbol and the directed arcs are drawn from head word to the dependent word.



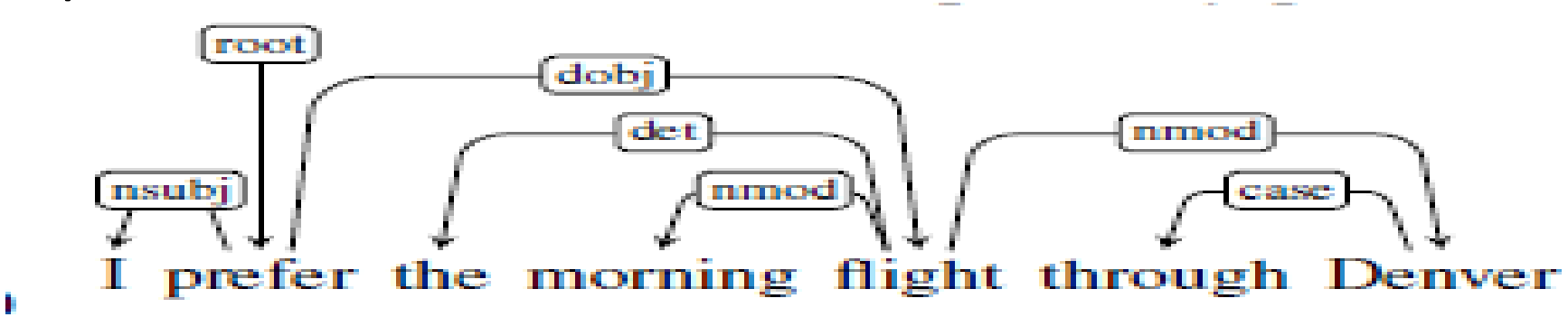
Syntax Analysis using Dependency Graphs

- In the figure in the previous slide [fakulte,N3,7] is the seventh word in the sentence with POS tag N3 and it has dative case.
- Here is a textual representation of a labeled dependency tree:

Index	Word	Part of Speech	Head	Label
1	They	PRP	2	SBJ
2	persuaded	VBD	0	ROOT
3	Mr.	NNP	4	NMOD
4	Trotter	NNP	2	IOBJ
5	to	TO	6	VMOD
6	take	VB	2	OBJ
7	it	PRP	6	OBJ
8	back	RB	6	PRT
9	.	.	2	P

Syntax Analysis using Dependency Graphs

- An important notion in dependency analysis is the notion of **projectivity**.
- A projective dependency tree is one where we put the words in a linear order based on the sentence with the root symbol in the first position.
- The dependency arcs are then drawn above the words without any crossing dependencies.
- Example:



Syntax Analysis using Dependency Graphs

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 Selected dependency relations from the Universal Dependency set. (de Marneffe et al., 2014)

Syntax Analysis using Dependency Graphs

- Let us look at an example where a sentence contains an extra position to the right of a noun phrase modifier phrase which requires a crossing dependency.

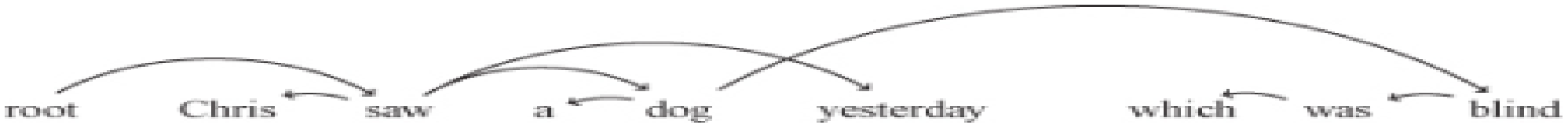


Figure 3–3. An unlabeled nonprojective dependency tree with a crossing dependency

Syntax Analysis using Dependency Graphs

- English has very few cases in a treebank that needs such a non projective analysis.
- In languages like Czech, Turkish, Telugu the number of non productive dependencies are much higher.
- Let us look at a multilingual comparison of crossing dependencies across a few languages:

	Ar	Ba	Ca	Ch	Cz	En	Gr	Hu	It	Tu
% deps	0.4	2.9	0.1	0.0	1.9	0.3	1.1	2.9	0.5	5.5
% sents	10.1	26.2	2.9	0.0	23.2	6.7	20.3	26.4	7.4	33.3

- Ar=Arabic;Ba=Basque;Ca=Catalan;Ch=Chinese;Cz=Czech;En=English; Gr=Greek;Hu=Hungarian;It=Italian;Tu=Turkish

Syntax Analysis using Dependency Graphs

- Dependency graphs in treebanks do not explicitly distinguish between projective and non-projective dependency tree analyses.
- Parsing algorithms are sometimes forced to distinguish between projective and non-projective dependencies.
- Let us try to setup dependency links in a CFG.

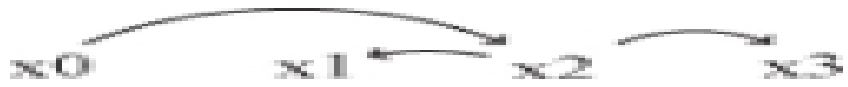
Syntax Analysis using Dependency Graphs

```
X0_2 -> X0_1* X2_1
X0_1 -> x0*
X2_1 -> X1_1 X2_2*
X1_1 -> x1*
X2_2 -> X2_3* X3_1
X2_3 -> x2*
X3_1 -> x3*
```

- In the CFG the terminal symbols are x0,x1,x2,x3.
- The asterisk picks out a single symbol in the right hand side of each rule that specifies the dependency link.
- We can look at the asterisk as either a separate annotation on the non-terminal or simply as a new nonterminal in the probabilistic context-free grammar(PCFG).

Syntax Analysis using Dependency Graphs

- The dependency tree equivalent to the preceding CFG is as follows:



- If we can convert a dependency tree into an equivalent CFG then the dependency tree must be projective.

Syntax Analysis using Dependency Graphs

- In a CFG converted from a dependency tree we have only the following three types of rules:
  - One type of rule to introduce the terminal symbol
  - Two rules where Y is dependent on X or vice-versa.
- The head word of X or Y can be traced by following the asterisk symbol.

$Z \rightarrow X^* Y$

$Z \rightarrow X Y^*$

$A \rightarrow a^*$

Syntax Analysis using Dependency Graphs

- Now let us look at an example non-projective dependency tree:

- When we convert this dependency tree to a CFG with * notation it can only capture the fact that X3 depends on X2 or X1 depends on X3.

```
X2_3 -> X1_1 X2_2*
X1_1 -> x1
X2_2 -> X2_1* X3_1
X2_1 -> x2
X3_1 -> x3
```

```
X2_3 -> X1_1 X3_2*
X1_1 -> x1
X3_2 -> X2_1 X3_1*
X2_1 -> x2
X3_1 -> x3
```

Syntax Analysis using Dependency Graphs

- There is no CFG that can capture the non-projective dependency.
- Projective dependency can be defined as follows:
  - For each word in the sentence its descendants form a contiguous substring of the sentence.
- Non-Projectivity can be defined as follows:
  - A non-projective dependency means that there is a word in the sentence such that its descendants do not form a contiguous substring of the sentence.
- In non-projective dependency there is a non-terminal Z such that Z derives spans(x_i,x_k) and (x_{k+p},x_j) for some p>0.

Syntax Analysis using Dependency Graphs

- It means there must be a rule Z->PQ where P derives (x_1,x_k) and Q derives (x_{k+p},x_j).
- By the definition of CFG this is valid if p=0 because P and Q must be continuous substrings.
- Hence non-projective dependency can not be converted to a CFG.

Syntax Analysis using Phrase Structure Trees

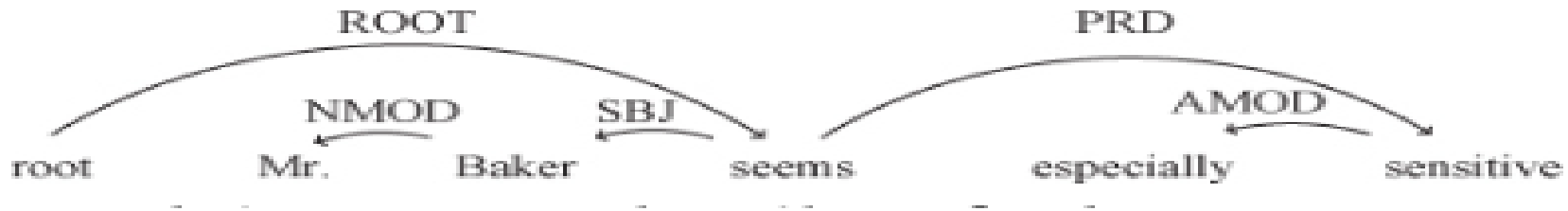
- A phrase structure syntax analysis can be viewed as implicitly having a predicate argument structure associated with it.
- Now let us look at an example sentence:
  - Mr. Baker seems especially sensitive

```
(S (NP-SBJ (NNP Mr.)
           (NNP Baker))
   (VP (VBZ seems)
       (ADJP-PRD (RB especially)
                 (JJ sensitive))))
```

```
Predicate-argument structure:
seems((especially(sensitive))(Mr. Baker))
```

Syntax Analysis using Phrase Structure Trees

- The same sentence gets the following dependency tree analysis.



- We can find some similarity between Phase Structure Trees and Dependency Trees.
- We now look at some examples of Phase Structure Analysis in tree banks and see how null elements are used to localize certain predicate-argument dependencies in the tree structure.

Syntax Analysis using Phrase Structure Trees

- In this example we can see that an NP dominates a trace *T* which is a null element (same as ϵ).
- The empty trace has an index and is associated with the WHNP (WH-noun phrase) constituent with the same index.
- This co-indexing allows us to infer the predicate-argument structure shown in the tree.

```
(SBARQ (WHNP-1 What)
    (SQ is (NP-SBJ Tim)
        (VP eating (NP *T*-1)))
    ?)

Predicate-argument structure:
eat(Tim, what)
```

Syntax Analysis using Phrase Structure Trees

- In this example “the ball” is actually not the logical subject of the predicate.
- It has been displaced due to the passive construction.
- “Chris” the actual subject is marked as LGS (Logical subjects in passives) enabling the recovery of predicate argument structure for the sentence.

```
(S (NP-SBJ-1 The ball)
  (VP was (VP thrown)
        (NP *-1)
        (PP by (NP-LGS Chris))))

Predicate-argument structure:
throw(Chris, the ball)
```

Syntax Analysis using Phrase Structure Trees

- In this example we can see that different syntactic phenomena are combined in the corpus.
- Both the analysis are combined to provide the predicate argument structure in such cases.

```
(SBARQ (WHNP-1 Who)
       (SQ was (NP-SBJ-2 *T*-1)
               (VP believed (S (NP-SBJ-3 *-2)
                               (VP to (VP have
                                          (VP been
                                              (VP shot
                                                  (NP *-3)))))))))
       ?)

Predicate-argument structure:
believe(*someone*, shoot(*someone*, who))
```

Syntax Analysis using Phrase Structure Trees

- Here we see a pair of examples to show how null elements are used to annotate the presence of a subject for a predicate even if it is not explicit in the sentence.
- In the first case the analysis marks the missing subject for “take back” as the object of the verb *persuaded*.
- In the second case the missing subject for “take back ” is the subject of the verb *promised.*

Syntax Analysis using Phrase Structure Trees

```
(S (NP-SBJ (PRP They))
  (VP (VP (VBD persuaded)
        (NP-1 (NNP Mr.)
          (NNP Trotter))
        (S (NP-SBJ (-NONE- *-1))
          (VP (TO to)
            (VP (VB take)
              (NP (PRP it))
              (PRT (RB back)))))))))
```

Predicate argument structure:

persuade(they, Mr. Trotter, take_back(Mr. Trotter, it))

```
(S (NP-SBJ-1 (PRP They))
  (VP (VP (VBD promised)
        (NP (NNP Mr.)
          (NNP Trotter))
        (S (NP-SBJ (-NONE- *-1))
          (VP (TO to)
            (VP (VB take)
              (NP (PRP it))
              (PRT (RB back)))))))))
```

Predicate argument structure:

promise(they, Mr. Trotter, take_back(they, it))

Syntax Analysis using Phrase Structure Trees

- The dependency analysis for “persuaded” and “promised” do not make such a distinction.
- The dependency analysis for the two sentences is as follows:

```
1 They      PRP 2 SBJ
2 persuaded VBD 0 ROOT
3 Mr.       NNP 4 NMOD
4 Trotter   NNP 2 IOBJ
5 to        TO  6 VMOD
6 take      VB  2 OBJ
7 it        PRP 6 OBJ
8 back      RB  6 PRT
9 .         .   2 P
```

```
1 They      PRP 2 SBJ
2 promised  VBD 0 ROOT
3 Mr.       NNP 4 NMOD
4 Trotter   NNP 2 IOBJ
5 to        TO  6 VMOD
6 take      VB  2 OBJ
7 it        PRP 6 OBJ
8 back      RB  6 PRT
9 .         .   2 P
```

Syntax Analysis using Phrase Structure Trees

- Most statistical parsers trained using Phrase Structure treebanks ignore these differences.
- Here we look at one example from the Chinese treebank which uses IP instead of S.
- This is a move from transformational grammar-based phrase structure to government-binding (GB) based phrase structure.
- It is very difficult to take a CFG-based parser initially developed for English parsing and adapt it to Chinese parsing by training it on Chinese phrase structure treebank.

Syntax Analysis using Phrase Structure Trees

```
(IP (NP-SBJ (NP (NN 结售/settlement and sale)
                (NN 制度/system))
            (CC 和/and)
            (NP (CP (WHNP-2 (-NONE- *OP*))
                    (CP (IP (NP-SBJ (-NONE- *T*-2))
                            (VP (VA 新/new)))
                        (DEC 的)))
                (NP (NN 核销/verification and cancellation)
                    (NN 制度/system))))
    (VP (PP-LOC (P 在/in)
                (NP-PN (NR 西藏/Tibet)))
        (ADVP (AD 全面/fully))
        (VP (VV 实施/operating))))

English translation:
A (foreign exchange) settlement and sale system and a verification and
cancellation system that is newly created is fully operational in Tibet.
```

Parsing Algorithms

- Introduction to Parsing Algorithms
- Shift-Reduce Parsing
- Hypergraphs and Chart Parsing
- Minimum Spanning Trees and Dependency Parsing

Parsing Algorithms

- Given an input sentence a parser produces an output analysis of that sentence.
- The analysis will be consistent with the tree-bank used to train the parser.
- Tree-banks do not need to have an explicit grammar.
- Here for better understanding we assume the existence of a CFG.

Parsing Algorithms

- Let us look at an example of a CFG that is used to derive strings such as “a and b or c” from the start symbol N:

```
N -> N 'and' N
N -> N 'or' N
N -> 'a' | 'b' | 'c'
```

```
N
=> N ‘or’ N
=> N ‘or c’
=> N ‘and’ N ‘or c’
=> N ‘and b or c’
=> ‘a and b or c’
```

- An ımportant concept of parsing is a **derivation**.
- In the derivation each line is called **sentential form**.
- The method of derivation used here is called **rightmost derivation**.

Parsing Algorithms

- An interesting property of rightmost derivation is revealed if we arrange the derivation in reverse order.

```
'a and b or c'
=> N 'and b or c'       # use rule N -> a
=> N 'and' N 'or c'     # use rule N -> b
=> N 'or c'             # use rule N -> N and N
=> N 'or' N             # use rule N -> c
=> N                    # use rule N -> N or N
```

```
(N (N (N a)
   and
   (N b))
or
(N c))
```

- The above sequence corresponds to the construction of the above parse tree from left to right, one symbol at a time.

Parsing Algorithms

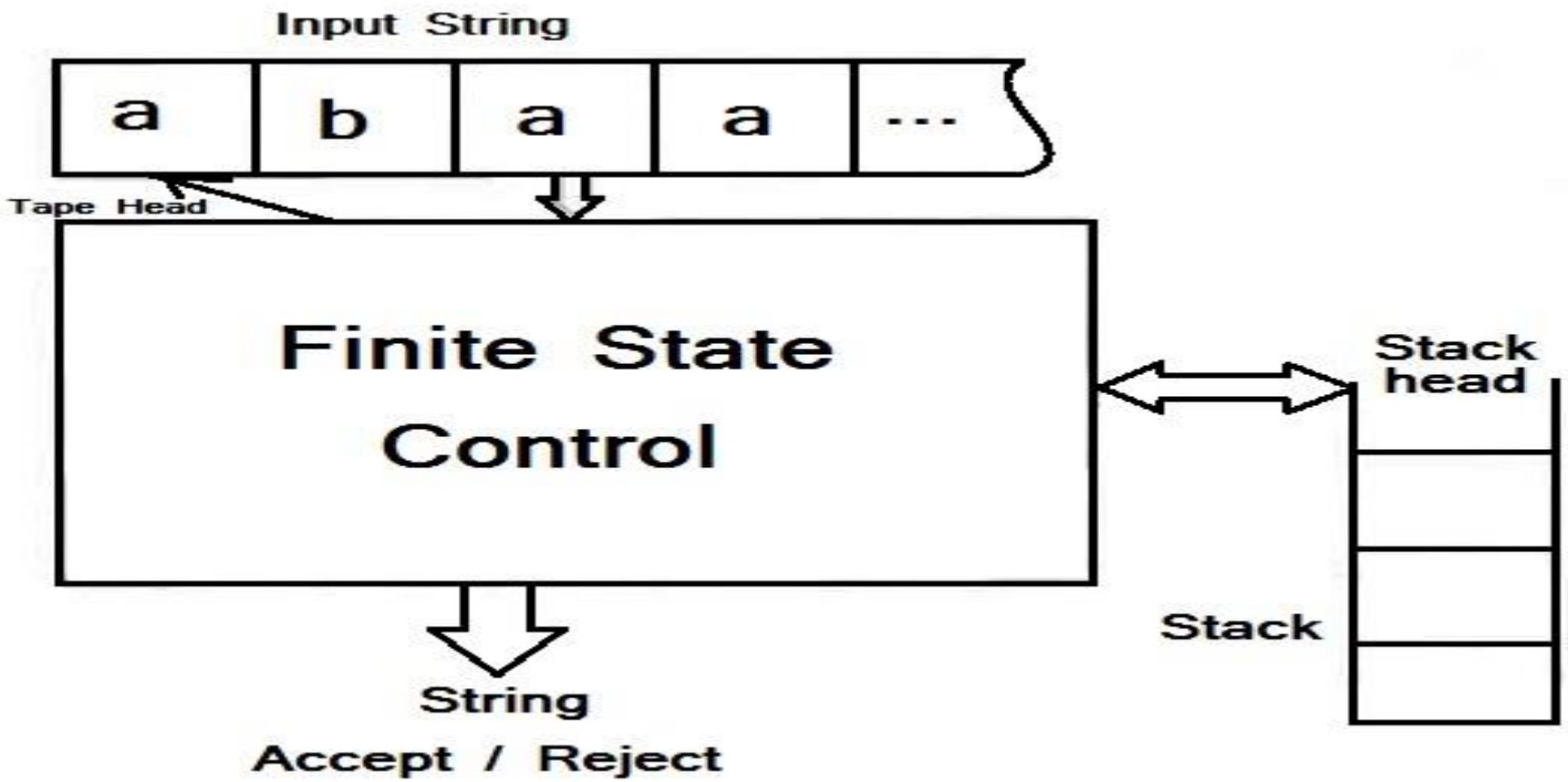
- There can be many parse trees for the given input string.
- Another rightmost derivation for the given input string is as follows:

```
(N (N a)
   and
   (N (N b)
      or
      (N c)))

‘a and b or c’
=> N ‘and b or c’          # use rule N -> a
=> N ‘and’ N ‘or c’        # use rule N -> b
=> N ‘and’ N ‘or’ N        # use rule N -> c
=> N ‘and’ N               # use rule N -> N or N
=> N                       # use rule N -> N and N
```

Shift-Reduce Parsing

- Every CFG has an automaton equivalent to it called the pushdown automaton.



Shift-Reduce Parsing

- The shift-reduce parsing algorithm is defined as follows:

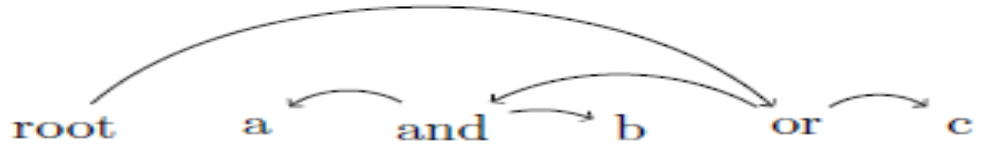
1. Start with an empty stack and the buffer contains the input string.
2. Exit with success if the top of the stack contains the start symbol of the grammar and if the buffer is empty.
3. Choose between the following two steps (if the choice is ambiguous, choose one based on an oracle):
    - Shift a symbol from the buffer onto the stack.
    - If the top k symbols of the stack are $\alpha_1 \ldots \alpha_k$ which corresponds to the right-hand side of a CFG rule $A \rightarrow \alpha_1 \ldots \alpha_k$ then replace the top k symbols with the left-hand side non-terminal A.
4. Exit with failure if no action can be taken in previous step.
5. Else, go to Step 2.

Shift-Reduce Parsing

- For the example “a and b or c” the parsing steps are s follows:

Parse Tree	Stack	Input	Action
		a and b or c	Init
a	a	and b or c	shift a
(N a)	N	and b or c	reduce `N -> a`
(N a) and	N and	b or c	shift and
(N a) and b	N and b	or c	shift b
(N a) and (N b)	N and N	or c	reduce `N -> b`
(N (N a) and (N b))	N	or c	reduce `N -> a`
(N (N a) and (N b)) or	N or	c	shift or
(N (N a) and (N b)) or c	N or c		shift c
(N (N a) and (N b)) or (N c)	N or N		reduce `N -> c`
(N (N (N a) and (N b)) or (N c))	N		reduce `N -> N or N`
(N (N (N a) and (N b)) or (N c))	N		Accept!

Shift-Reduce Parsing- for dependency parsing

Dependency tree	Stack	Input	Action
root	root	a and b or c	Init
root a	root a	and b or c	shift a
root a and	root a and	b or c	shift and
root a ← and	root and	b or c	a ← and
root a ← and b	root and b	or c	shift b
root a ← and → b	root and	or c	and → b
root a ← and → b or	root and or	c	shift or
root a ← and → b ← or	root or	c	and ← or
root a ← and → b ← or c	root or c		shift c
root a ← and → b ← or → c	root or		or → c
	root		root → or

Hyper Graphs and Chart Parsing

- Shift-reduce parsing allows a linear time parser but requires access to an oracle.
- CFGs in the worst case need backtracking and have a worst case parsing algorithm which run in $O(n^3)$ where n is the size of the input.
- Variants of this algorithm are used in statistical parsers that attempt to search the space of possible parse trees without the limitation of left-to-right parsing.

Hyper Graphs and Chart Parsing

- Our example CFG G is rewritten as new CFG G_c which contains up to two non-terminals on the right hand side.

```
N -> N 'and' N
N -> N 'or' N
N -> 'a' | 'b' | 'c'
```

```
N -> N N^
N^ -> 'and' N
N -> N Nv
Nv -> 'or' N
N -> 'a' | 'b' | 'c'
```

Hyper Graphs and Chart Parsing

- We can specialize the CFG Gc to a particular input string by creating a new CFG that represents all possible parse trees that are valid in grammar G_c for this particular input sentence.
- For the input “a and b or c” the new CFG Cf that represents the forest of parse trees can be constructed.
- Let the input string be broken up into spans 0 a 1 and 2 b 3 or 4 c 5.

Hyper Graphs and Chart Parsing

```
N[O,5] -> N[O,1] N^[1,5]
N[O,3] -> N[O,1] N^[1,3]
N^[1,3] -> ‘and’[1,2] N[2,3]
N^[1,5] -> ‘and’[1,2] N[2,5]
N[O,5] -> N[O,3] Nv[3,5]
N[2,5] -> N[2,3] Nv[3,5]
Nv[3,5] -> ‘or’[3,4] N[4,5]
N[O,1] -> ‘a’[O,1]
N[2,3] -> ‘b’[2,3]
N[4,5] -> ‘c’[4,5]
```

Hyper Graphs and Chart Parsing

- Here a parsing algorithm is defined as taking as input a CFG and an input string and producing a specialized CFG that represents all legal parsers for the input.
- A parser has to create all the valid specialized rules from the start symbol nonterminal that spans the entire string to the leaf nodes that are the input tokens.

Hyper Graphs and Chart Parsing

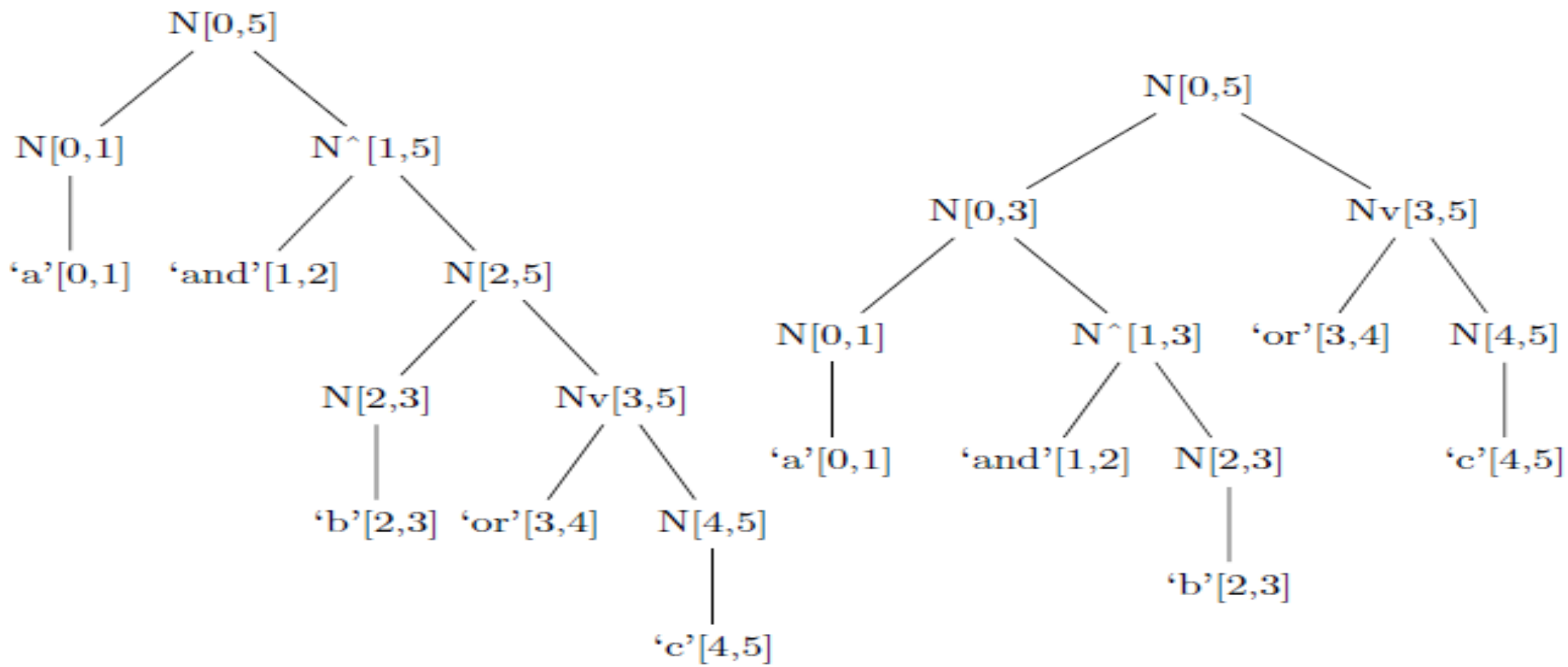


Figure 4: Parse trees embedded in the *specialized* CFG for a particular input string. The nodes with the same label, e.g. *N[0,5]*, *N[0,1]*, *and[1,2]*, *N[2,3]*, and *Nv[3,5]* can be merged to form a hypergraph representation of all parses for the input.

Hyper Graphs and Chart Parsing

- Now let us look at the steps the parser has to take to construct a specialized CFG.
- Let us consider the rules that generate only lexical items:

N[0,1] -> 'a'[0,1]

N[2,3] -> 'b'[2,3]

N[4,5] -> 'c'[4,5]