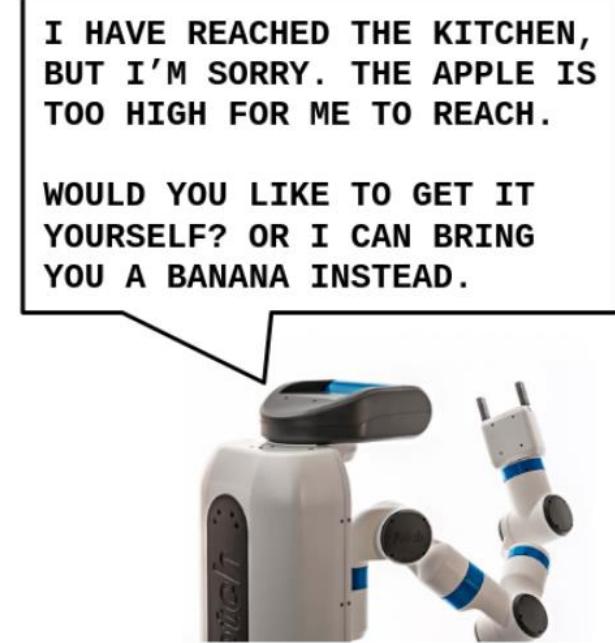
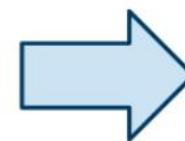
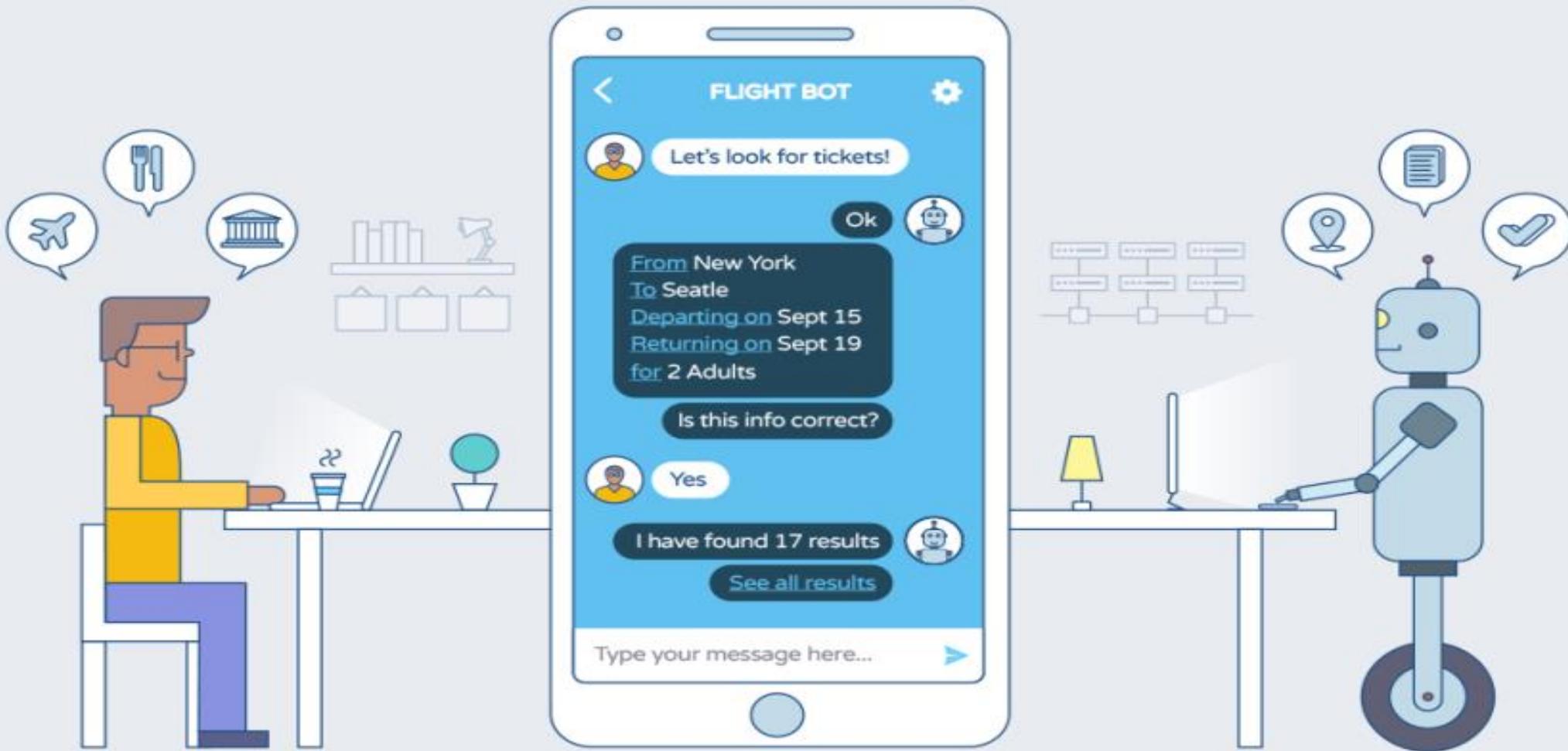


DSE 3155

Natural Language Processing





An Application of NLP

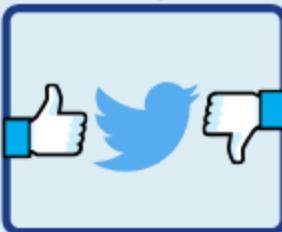
Information
Retrieval



Machine
Translation



Sentiment
Analysis

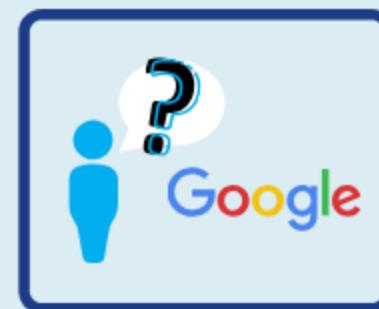


Information
Extraction



Natural Language Processing (NLP)

Question
Answering



NLP

- Natural Language Processing or NLP is a field of Artificial Intelligence that gives the machines the ability to read, understand and derive meaning from human languages
- In particular, concerned with programming computers to fruitfully process large natural language corpora
- Also known as Computational Linguistics (CL), Human Language Technology (HLT), Natural Language Engineering (NLE)
- A natural language is just like a human language which we use for communication like English, Hindi,..etc.
- Language processing problem can be divided as:
 - – Processing written text, semantic and syntactic knowledge of the language.
 - – Processing spoken language

NLP for Machines

- Analyze, understand and generate human languages just like humans do
- Applying computational techniques to language domain
- To explain linguistic theories, to use the theories to build systems that can be of social use
- Started off as a branch of Artificial Intelligence
- Borrows from Linguistics, Psycholinguistics, Cognitive Science & Statistics
- Make computers learn our language rather than we learn theirs

Challenges in NLP

- Text is the largest repository of human knowledge and is growing quickly.
- It is not an easy task to teach a person or a computer, a natural language.
- The problem is Syntax (rules governing the way in which the words are arranged) & understanding context.
- Computers traditionally require humans to “speak” to them in a programming language that is clear, unambiguous and highly structured.
- Human speech is not always precise.

Components of NLP

Natural Language Understanding

- Taking some spoken/typed sentence and working out what it means
- It helps the machine to understand and analyze human natural language

Natural Language Generation

- Taking some formal representation of what you want to say and working out a way to express it in a natural (human) language (e.g., English)
- It is a translator that converts the computerized data into natural language representation.



Natural Language Processing

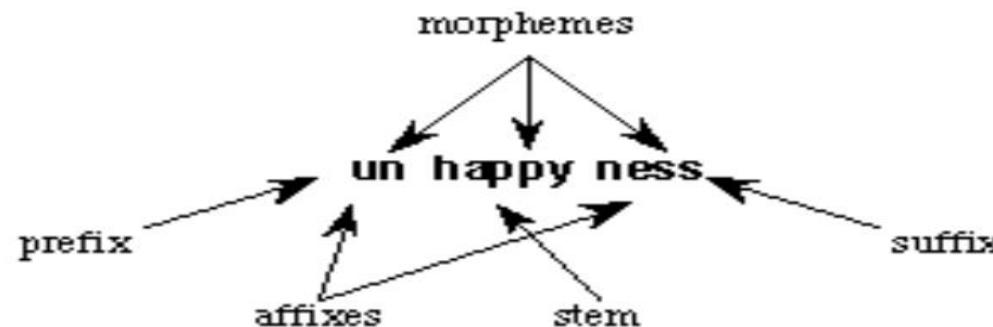
Figure :Components of NLP

Linguistics and Language

- Linguistics is the science of language
- Its study includes:
 - Pronunciation of different speakers which refers to **phonetics**
 - Sounds which refers to **phonology**
 - Study of meaningful components of word formation refers to **morphology**
 - Structural relationship between the words refers to **syntax**
 - Meaning refers to **semantics**
 - How language is used to accomplish task refers to **pragmatics**.
 - Linguistic units larger than a single utterance refers to **Discourse**.

Morphological and Lexical Analysis

- Morphology is the study of the structure and formation of words.
- One of the widespread task here is lemmatizing or stemming which is used in many web search engines. Ex: dog-dogs, run-ran, bus-buses etc.
- Its most important unit is the Morpheme, which is defined as the "minimal unit of meaning".
- Consider a word like: "unhappiness". This has three parts: un means "not", while ness means "being in a state or condition" and happy is a free morpheme because it can appear on its own.



Morphological and Lexical Analysis

- Lexical Analysis involves identifying and analyzing the structure of words.
- Lexicon of a language means the collection of words and phrases in a language.
- Lexical analysis is dividing the whole chunk of text into paragraphs, sentences, and words

Syntactic Analysis

- Syntax concerns the proper ordering of words and its affect on meaning
- This involves analysis of the words in a sentence to depict the grammatical structure of the sentence.
- The words are transformed into structure that shows how the words are related to each other.
- In other words, Syntactic Analysis exploit the results of Morphological analysis to build a structural description of the sentence.

Syntactic Analysis

sentence -> noun_phrase, verb_phrase

noun_phrase -> noun

noun_phrase -> determiner, noun

verb_phrase -> verb, noun_phrase

verb_phrase -> verb

noun -> [mary]

noun -> [apple]

verb -> [ate]

determiner -> [the]

Syntactic Analysis

- Main problems on this level are:
 - part of speech tagging (POS tagging)
 - chunking or detecting syntactic categories (verb, noun phrases)
 - sentence assembling (constructing syntax tree)
- Parsing is the process of assigning structural descriptions to sequences of words in a natural language.

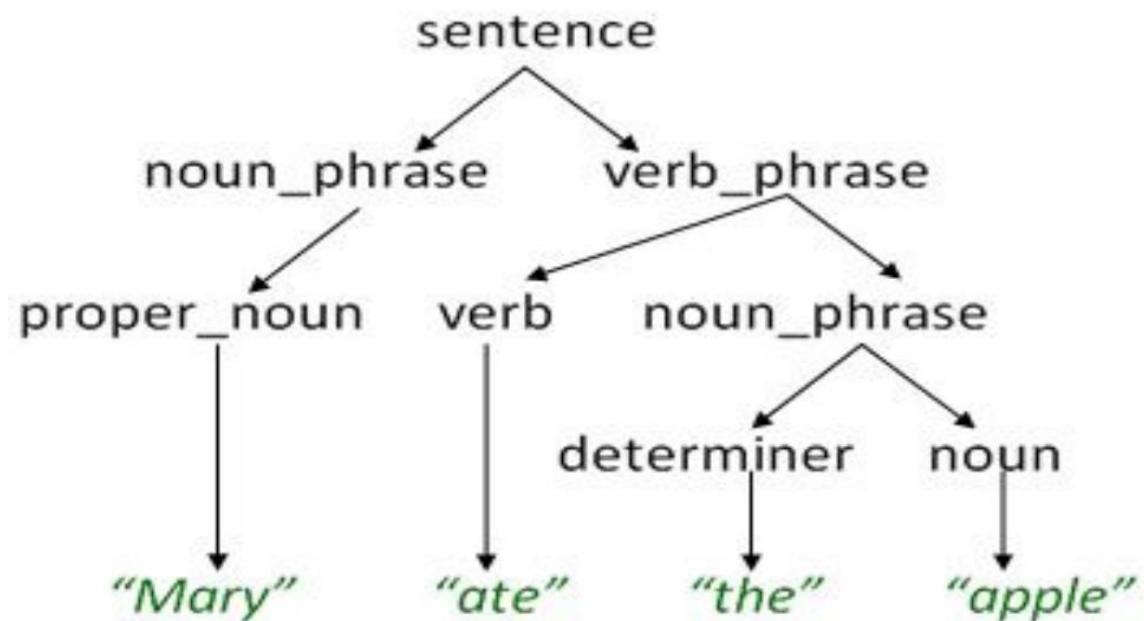
Top-down Parsing

- In this kind of parsing, 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. The main disadvantage of recursive descent parsing is backtracking.

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.

Syntactic Analysis



Semantic Analysis

- Semantics concerns the (literal) meaning of words, phrases, and sentences.
- This abstracts the dictionary meaning or the exact meaning from context.
- The structures which are created by the syntactic analyzer are assigned meaning
- E.g.. “colorless blue car”. This would be rejected by the analyzer as colorless blue do not make any sense together.

Basically one needs to complete morphological and syntactical analysis before trying to solve any semantic problem.

Discourse Integration

- The meaning of any single sentence depends upon the sentences that precedes it and also invokes the meaning of the sentences that follow it.
- In addition, it also brings about the meaning of immediately succeeding sentence

Example:

1. Bill had a red balloon. John wanted it.

“ It” refers to red balloon and it should be identified like that only .

Pragmatic Analysis

- Pragmatics concerns the overall communicative and social context and its effect on interpretation.
- It means abstracting or deriving the purposeful use of the language in situations.
- Importantly those aspects of language which require world knowledge
- The main focus is on what was said is reinterpreted on what it actually means
- E.g. “Close the window?” should have been interpreted as a request rather than an order
“Do you know what time it is?” should be interpreted as a request to be told the time

Ambiguity

- Resolving ambiguity is the task performed in all the steps of NLP
- Language is ambiguous: one word, one phrase, or one sentence can mean different things depending on the context
- Ambiguity at multiple levels
 - Word senses: bank (finance or river ?)
 - Part of speech: chair (noun or verb ?)
 - Syntactic structure: I can see a man with a telescope
 - Multiple: I made her duck

Ambiguity

Example: I made her duck

Five different meanings this sentence could have:

- i. I cooked waterfowl for her
- ii. I cooked waterfowl belonging to her
- iii. I created the (plaster?) duck she owns
- iv. I caused her to quickly lower her head or body
- v. I waved my magic wand and turned her into undifferentiated waterfowl

Ambiguity

Ambiguities arises due to :

- Words duck and her are morphologically or syntactically ambiguous in their parts-of-speech (duck can be verb or noun and her can be pronoun or possessive pronoun)
- Word make is syntactically ambiguous:
 - word make can be transitive (ii) , or it can be ditransitive (v) , or it can take direct object or verb

Resolving Ambiguities

Lexical disambiguation: Resolution of part-of-speech and word sense ambiguities

- Deciding whether duck is a verb or a noun can be solved by part-of-speech tagging.
- Deciding whether make means create or cook is solved by word sense disambiguation

Syntactic disambiguation: represents sentences that can be parsed in multiple syntactical forms

- Deciding whether her and duck are part of same entity or are different entity.

Models and Algorithms

- **Well known Models:**

State Machines, Formal rule systems, logic, Probability theory, machine learning tools

- **Well known Algorithms:**

State space search and Dynamic Algorithms

- State Machines: are formal models that consists of states, transition among states and an input representation.

- Variations of basic model are deterministic and non-deterministic finite state automata, finite-state transducers (can write to output devices), weighted automata, Markov models(also called as Markov models), hidden Markov models (Probabilistic component)

Models and Algorithms

- **Formal rule systems:** Regular grammars, regular relations, context-free grammars, feature-augmented grammars.
- State machines and Formal rule systems are used when dealing with the knowledge of phonology, morphology and syntax.
- Algorithms associated with state machines and formal rule systems involve a search through a space of states representing hypothesis about an input. Example: Depth first search, breadth first search

Logic:

Examples : First order logic (Predicate calculus), semantic networks, conceptual dependency

Models and Algorithms

- **Probability theory :**
 - All the methods (state machines, formal rule systems, logic) can be augmented with probabilities.
 - Used to solve many kinds of Ambiguity problems.
- **Machine Learning Models :**
 - Automatically learn the various representations (automata, rule systems, search heuristics, classifiers)
 - Used as a powerful modeling technique where good casual models or algorithms are not available

END

Unit 2

Finite State Automata and Regular Expression

Introduction

Regular expressions

- Are a formal notation for characterizing the text sequence.
- Regular expressions are an algebraic way to describe languages.
- Are mainly used in web search examples, word processors.

Finite automata

- Are formal (or abstract) machines for recognizing patterns.
- Are mathematical device used to implement regular expressions.
- These machines are used extensively in compilers and text editors, which must recognize patterns in the input

Regular Expressions

- A formal language for specifying text strings
- How can we search for any of these?
 - woodchuck
 - woodchucks
 - Woodchuck
 - Woodchucks

Regular Expressions

- Regular expressions are a very powerful tool to do string matching and processing
- RE search requires pattern and corpus of text.
- Allows you to do things like:
 - Match a string that starts with a lowercase letter, then is followed by 2 numbers and ends with “ing” or “ion”
 - Replace all occurrences of one or more spaces with a single space
 - Split up a string based on whitespace or periods or commas or ...

Terminology of Languages:

- *An alphabet is a finite, non-empty set of symbols*
- We use the symbol Σ (sigma) to denote an alphabet
- Examples:
 - Binary: $\Sigma = \{0,1\}$
 - All lower case letters: $\Sigma = \{a,b,c,..z\}$
 - Alphanumeric: $\Sigma = \{a-z, A-Z, 0-9\}$

Terminology of Languages

- A string or word is a finite sequence of symbols chosen from Σ
- Empty string is ε (or “epsilon”)
- Length of a string w , denoted by “ $|w|$ ”, is equal to the number of (non- ε) characters in the string
 - E.g., $x = 010100$ $|x| = 6$
 - $x = 01 \varepsilon 0 \varepsilon 1 \varepsilon 00 \varepsilon$ $|x| = ?$
- xy = concatenation of two strings x and y

Terminology of Languages

L is said to be a language over alphabet Σ , only if $L \subseteq \Sigma^*$

→ this is because Σ^* is the set of all strings (of all possible length including 0) over the given alphabet Σ

Examples:

1. Let L be the language of all strings consisting of n 0's followed by n 1's:

$$L = \{\epsilon, 01, 0011, 000111, \dots\}$$

2. Let L be the language of all strings of with equal number of 0's and 1's:

$$L = \{\epsilon, 01, 10, 0011, 1100, 0101, 1010, 1001, \dots\}$$

Examples:

- $L_1 = \{a,b,c,d\}$ $L_2 = \{1,2\}$
- $L_1 L_2 = \{a1,a2,b1,b2,c1,c2,d1,d2\}$
- $L_1 \cup L_2 = \{a,b,c,d,1,2\}$
- L_1^3 = all strings with length three (using a,b,c,d)
- L_1^* = all strings using letters a,b,c,d and empty string
- L_1^+ = doesn't include the empty string

Regular Expressions: Literals

- Basic regular expression consists of a single literal character
- We can put any string in a regular expression
 - `/test/` - matches any string that has “test” in it
 - `/this class/` - matches any string that has “this class” in it
 - `/Test/`. case sensitive: matches any string that has “Test” in it

Regular Expressions: Meta characters

[]	A set of characters (character class)
\	Signals a special sequence (can also be used to escape special characters)
.	Any character (except newline character)
^	Starts with (anchor)
\$	Ends with (anchor)
*	Zero or more occurrences
+	One or more occurrences
?	Zero or one occurrence
{}	Exactly the specified number of occurrences
	Either or
\b	Word boundary (anchor)
\B	Non boundary (anchor)

Anchors are special characters that anchor regular expressions to particular places in string.

Operator precedence hierarchy

Parenthesis)
Counters	* + ? {}
Sequences and anchors	the ^my end\$
Disjunction	

Regular Expressions: Character Classes

➤ A set of characters to match:

- put in brackets: []
- [abc] matches a single character a or b or c

➤ Can use - to represent ranges

- [a-z] is equivalent to [abcdefghijklmnopqrstuvwxyz]
- [A-D] is equivalent to [ABCD]
- [0-9] is equivalent to [0123456789]

Pattern	Matches
[wW]oodchuck	Woodchuck, woodchuck
[1234567890]	Any digit

Pattern	Matches	
[A-Z]	An upper case letter	Drenched Blossoms
[a-z]	A lower case letter	my beans were impatient
[0-9]	A single digit	Chapter 1: Down the Rabbit Hole

- **What would the following match?**

- a) /[Tt]est/
- any string with “Test” or “test” in it

- b) /[0-9][0-9][0-9][0-9]/
matches any four digits, e.g. a year

- c)/[aeiou][0-9]/
- matches a1 or e4

Regular Expressions: Character Classes

- Can also specify a set NOT to match:
- ^ means all characters EXCEPT those specified
- Carat means negation only when written first in []
- [^a] all characters except 'a'
- [^0-9] all characters except numbers
- [^A-Z] not an upper case letter

Pattern	Matches	
[^A-Z]	Not an upper case letter	O <u>y</u> fn pripetchik
[^Ss]	Neither ‘S’ nor ‘s’	I have no exquisite reason”
a^b	The pattern a carat b	Look up <u>a^b</u> now

Regular Expressions: Advanced Operators

Aliases for common set of operators:

RE	Expansion	Match	Example Patterns
\d	[0-9]	any digit	Party_of_5
\D	[^0-9]	any non-digit	Blue_moon
\w	[a-zA-Z0-9_]	any alphanumeric or underscore	Daiyu
\W	[^\w]	a non-alphanumeric	!!!
\s	[\r\t\n\f]	whitespace (space, tab)	
\S	[^\s]	Non-whitespace	in_Concord

Regular expression operators for counting

RE	Match
*	zero or more occurrences of the previous char or expression
+	one or more occurrences of the previous char or expression
?	exactly zero or one occurrence of the previous char or expression
{n}	<i>n</i> occurrences of the previous char or expression
{n, m}	from <i>n</i> to <i>m</i> occurrences of the previous char or expression
{n, }	at least <i>n</i> occurrences of the previous char or expression

What would the following match?

- `/19\d\d/`
- would match any 4 digits starting with 19
- `\s\s/`
- matches anything with two adjacent whitespace characters
- `\s[aeiou]..\s/`
- any three letter word that starts with a vowel

Regular Expressions

- * matches zero or more of the preceding character
- $/ba^*d/$
- matches any string with:
- bd bad baad baaad
- $/A.*A/$
- matches any string starts and ends with A
- + matches one or more of the preceding character
- $/ba+d/$
- matches any string with:
- bad baad baaad baaaad

Regular Expressions

- ? zero or 1 occurrence of the preceding
- `/fights?/`
- matches any string with “fight” or “fights” in it
- {n,m} matches n to m inclusive
- `/ba{3,4}d/`
- matches any string with
- baaad
- baaaad

Pattern	Matches	
colou?r	Optional previous char	<u>color</u> <u>colour</u>
oo*h!	0 or more of previous char	<u>oh!</u> <u>ooh!</u> <u>oooh!</u> <u>ooooh!</u>
o+h!	1 or more of previous char	<u>oh!</u> <u>ooh!</u> <u>oooh!</u> <u>ooooh!</u>
baa+		<u>baa</u> <u>baaa</u> <u>baaaa</u> <u>baaaaa</u>
beg.n		<u>begin</u> <u>begun</u> <u>begun</u> <u>beg3n</u>

Regular Expressions:beginning and end

- \wedge marks the beginning of the line
- $\$$ marks the end of the line
- $/test/$ test can occur anywhere
- $/^test/$ must start with test
- $/test$/$ must end with test
- $/^test$/$ must be exactly test

Pattern	Matches
$\wedge [A-Z]$	Palo Alto
$\wedge [^A-Za-z]$	1 "Hello"
$\backslash . \$$	The end.
$. \$$	The end? The end!

Question Hour

- What is the regular expression if we wanted to match:
 - This is very interesting
 - This is very very interesting
 - This is very very very interesting
- A. /This is very+ interesting/
- B. /This is (very)+interesting/
- Repetition operators only apply to a single character.
 - Use parentheses to group a string of characters.

Regular Expressions: More Disjunction

- The pipe | for disjunction

Pattern	Matches
groundhog woodchuck	woodchuck
yours mine	yours
a b c	= [abc]
[gG] roundhog [Ww] oodchuck	Woodchuck

Introduction to Finite-State-Automata

Finite Automata

- Theory of automata is a theoretical branch of computer science and mathematics.
- It is the study of abstract machines and the computation problems that can be solved using these machines. The abstract machine is called the **automata**.
- **Automata** is a abstract machine which takes some string as input and this input goes through a finite number of states and may enter in the final state.
- An automaton with a finite number of states is called a **Finite automaton**.
- This automaton consists of **states** and **transitions**.
- The **State** is represented by **circles**, and the **Transitions** is represented by **arrows**.

Finite Automata

- Finite automata are used to recognize patterns.
- It takes the string of symbol as input and changes its state accordingly. When the desired symbol is found, then the transition occurs.
- At the time of transition, the automata can either move to the next state or stay in the same state.
- Finite automata have two possible final states, **Accept state** or **Reject state**
- When the input string is processed successfully, and the automata reached its final state, then it will be accepted.

Formal Definition of FA

- A finite automaton is a collection of 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where:
- Q : finite set of states
- Σ : finite set of the input symbol
- q_0 : initial state
- F : final state
- $\Delta(q,i)$: Transition function

Transition Diagram

- A **transition diagram or state transition diagram** is a directed graph which can be constructed as follows:
- There is a node for each state in Q , which is represented by the circle.
- There is a directed edge from node q to node p labeled a if $\delta(q, a) = p$
- In the start state, there is an arrow with no source.
- Accepting states or final states are indicating by a double circle.

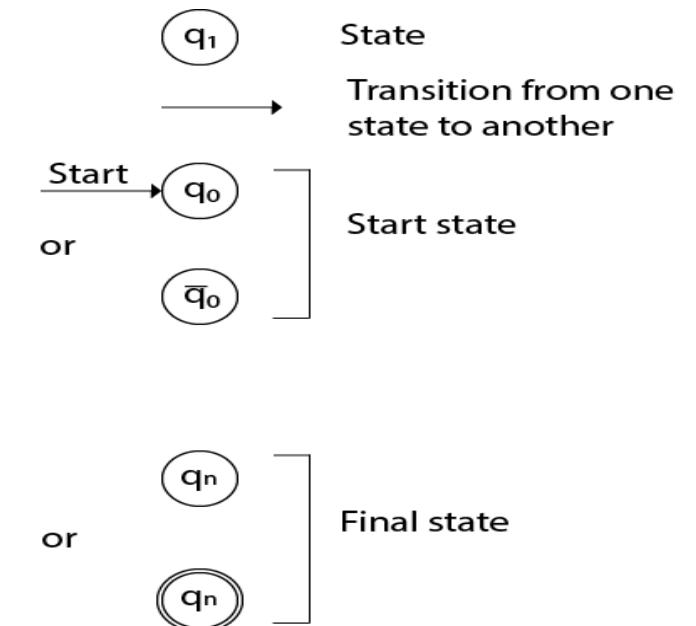
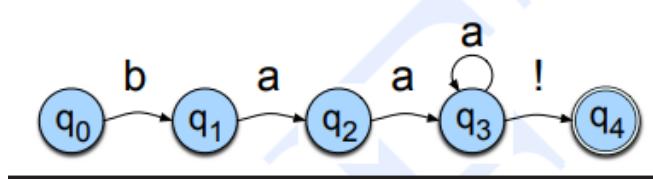


Fig:- Notations

Figure 1:Notations

FSA and State transition table



	Input		
State	b	a	!
0	1	0	0
1	0	2	0
2	0	3	0
3	0	3	4
4:	0	0	0

Types of Automata

- There are two types of finite automata:
 1. DFA (Deterministic Finite Automata)
 2. NFA (Non-deterministic Finite Automata)

1. DFA

DFA refers to **deterministic finite automata**. Deterministic refers to the uniqueness of the computation. In the DFA, the machine goes to one state only for a particular input character. DFA does not accept the null move.

2. NFA

NFA stands for non-deterministic finite automata. It is used to transmit any number of states for a particular input. It can accept the null move.

There can be multiple final states in both DFA and NFA.

DFA

- In DFA, there is only one path for specific input from the current state to the next state
- DFA does not accept the null move, i.e., the DFA cannot change state without any input character.
- DFA can contain multiple final states. It is used in Lexical Analysis in Compiler.
- In the following diagram, we can see that from state q_0 for **input a**, there is only one path which is going to q_1 . Similarly, from q_0 , there is only one path for input b going to q_2 .

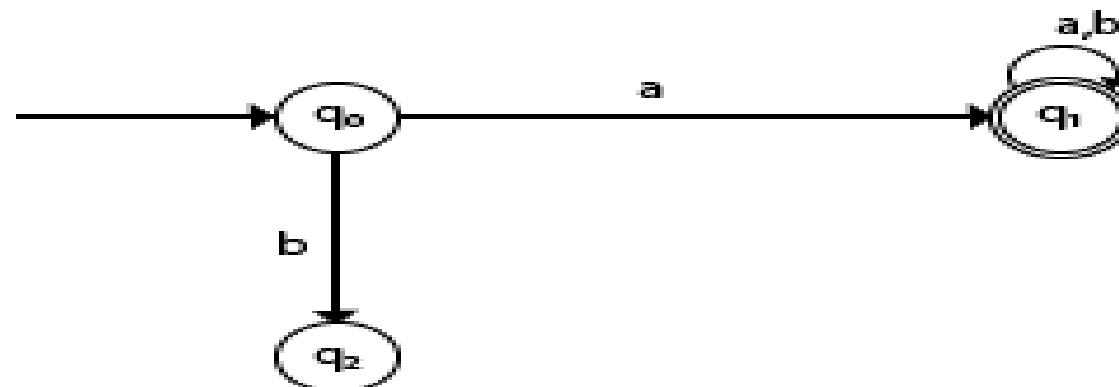


Fig:2 DFA

Graphical Representation of DFA

- A DFA can be represented by digraphs called state diagram. In which:
 1. The state is represented by vertices.
 2. The arc labeled with an input character show the transitions.
 3. The initial state is marked with an incoming arrow.
 4. The final state is denoted by a double circle.

Tabular Representation of DFA

- **Transition Table**
- The transition table is basically a tabular representation of the transition function. It takes two arguments (a state and a symbol) and returns a state (the "next state")
- A transition table is represented by the following things:
 - Columns correspond to input symbols
 - Rows correspond to states
 - Entries correspond to the next state
 - The start state is denoted by an arrow with no source
 - The accept state is denoted by a star

Example for DFA

Example 1: Construct a DFA with $\Sigma = \{0, 1\}$ that accepts all strings starting with 1.

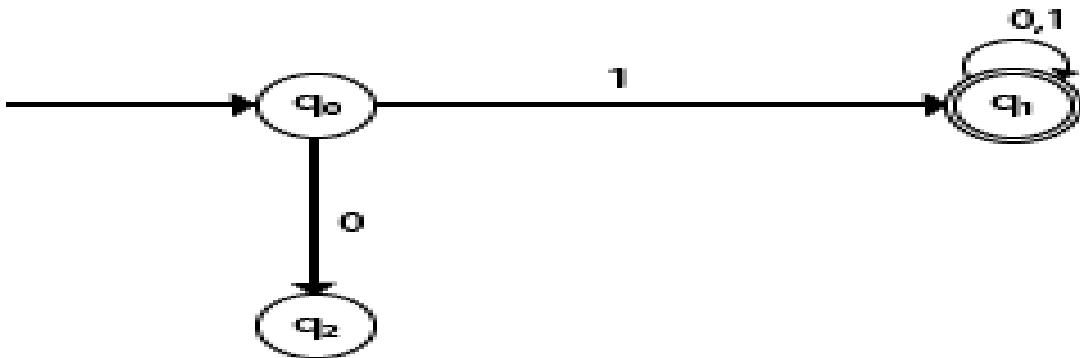


Fig: Transition diagram

- In the diagram, machine initially is in start state **q0** then on receiving input 1 the machine changes its state to **q1**
- From **q0** on receiving 0, the machine changes its state to **qd**, which is the **dead /trap state** (usually not represented)
- From **q1** on receiving input 0, 1 the machine changes its state to **q1**, which is the final state
- The possible input strings that can be generated are 10, 11, 110, 101, 111....., that means all string starts with 1

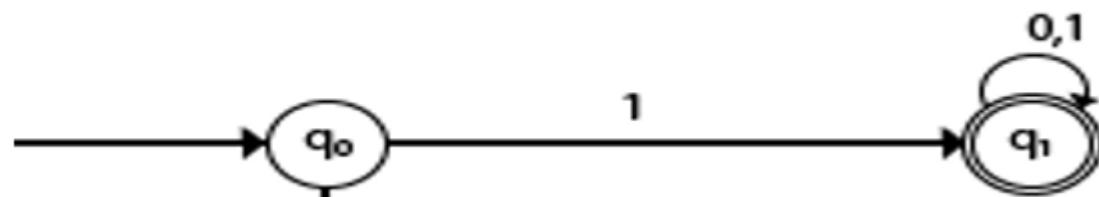


Fig: DFA without Dead State
Figure 3 and 4

Example for DFA

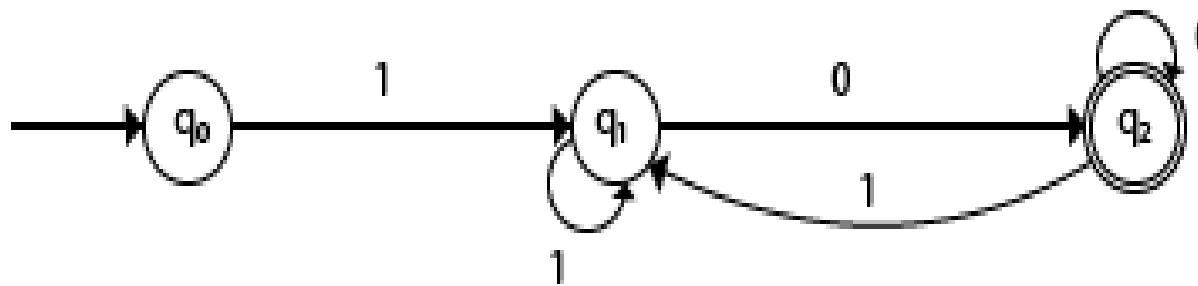
- Transition table for Example 1:

Present State	Next state for Input 0	Next State of Input 1
$\rightarrow q_0$	-	q_1
$*q_1$	q_1	q_1

Figure 5: Transition table for example 1

Example for DFA

- Example 2: Design a DFA with $\Sigma = \{0, 1\}$ accepts those string which starts with 1 and ends with 0



- In state q_1 , if we read 1, we will be in state q_1 , but if we read 0 at state q_1 , we will reach to state q_2 which is the final state
- In state q_2 , if we read either 0 or 1, we will go to q_2 state or q_1 state respectively
- Note that if the input ends with 0, it will be in the final state.

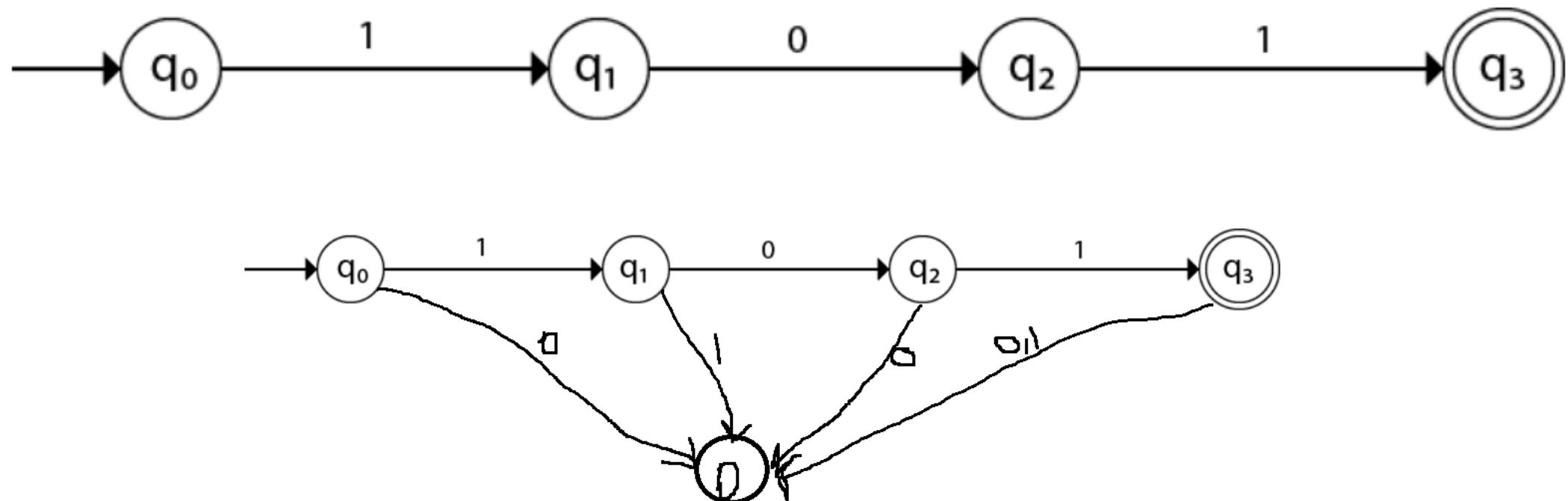
Example for DFA

- Transition Table for Example 2:

Present State	Next state for Input 0	Next State of Input 1
$\rightarrow q_0$	-	q_1
q_1	q_2	q_1
$*q_2$	q_2	q_1

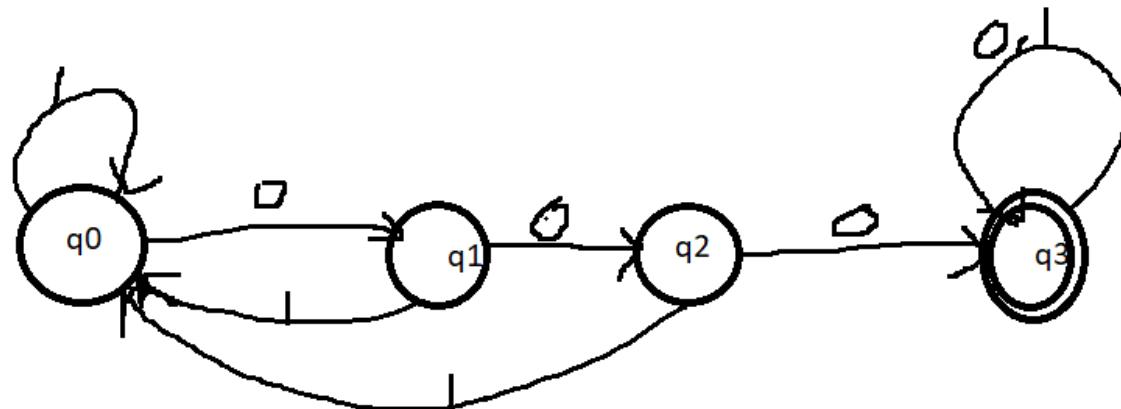
Example for DFA

- Example 3: Design a DFA with $\Sigma = \{0, 1\}$ accepts the only input 101



Example for DFA

- Example 4: Design DFA with $\Sigma = \{0, 1\}$ accepts the set of all strings with three consecutive 0's.



NFA

- NFA stands for non-deterministic finite automata. It is easy to construct an NFA than DFA for a given regular language.
- The finite automata are called NFA when there exist many paths for specific input from the current state to the next state
- Every NFA is not DFA, but each NFA can be translated into DFA
- NFA is defined in the same way as DFA but with the following two exceptions, it contains multiple next states, and it contains ϵ transition
- The transition without consuming an input symbol are called ϵ transitions

NFA

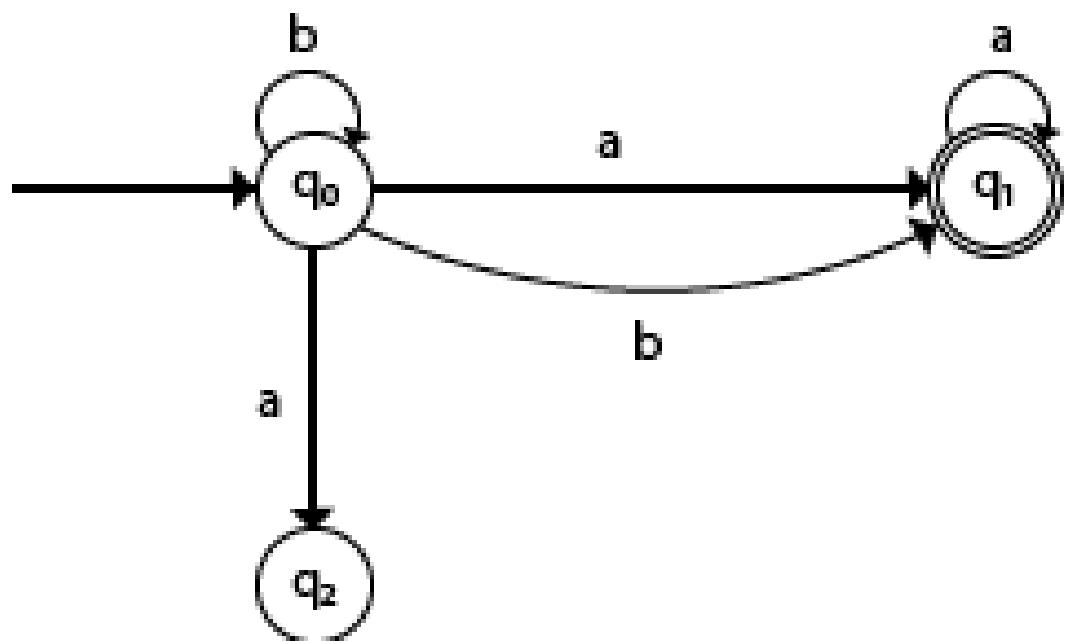
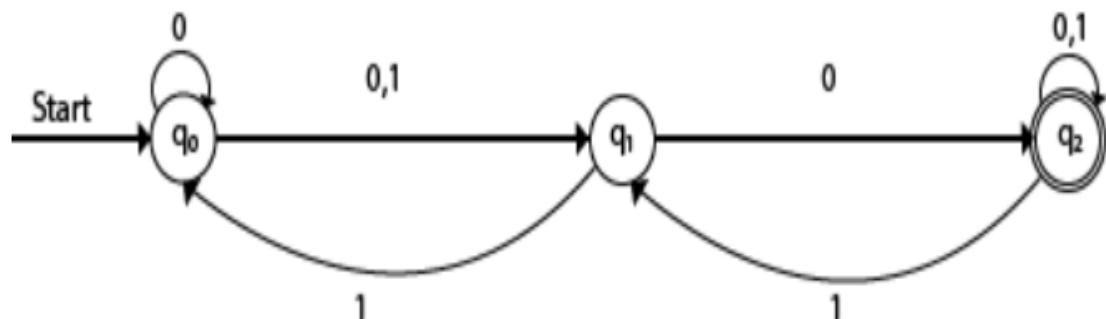


Fig:- NDFA

- In the figure, we can see that from state q_0 for input a , there are two next states q_1 and q_2 , similarly, from q_0 for input b , the next states are q_0 and q_1
- Thus it is not fixed or determined that with a particular input where to go next
- Hence this FA is called non-deterministic finite automata.

NFA

Example 1



Present State	Next state for Input 0	Next State of Input 1
$\rightarrow q_0$	q_0, q_1	q_1
q_1	q_2	q_0
$*q_2$	q_2	q_1, q_2

NFA

- Example 2: NFA with $\Sigma = \{0, 1\}$ accepts all strings begin with 01

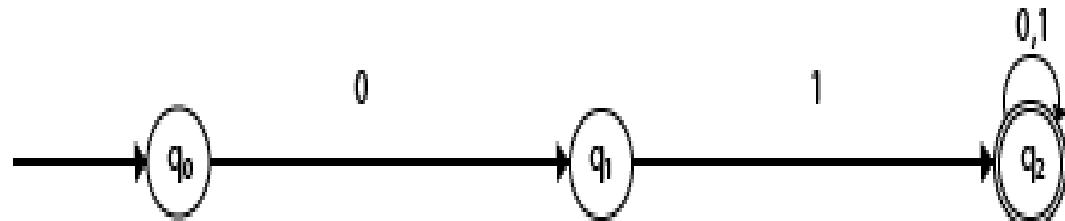
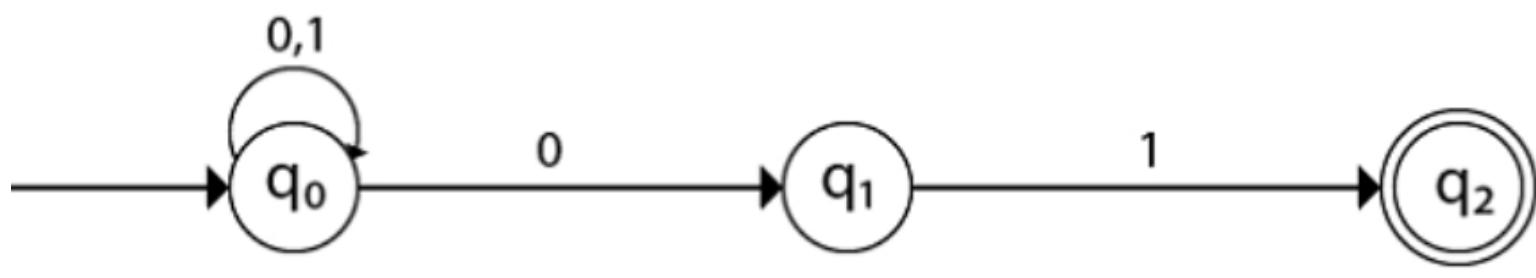


Fig: NFA

Present State	Next state for Input 0	Next State of Input 1
$\rightarrow q_0$	q_1	ϵ
q_1	ϵ	q_2
$*q_2$	q_2	q_2

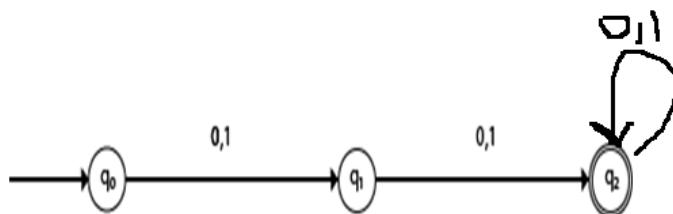
NFA

- Construct an NFA with $\Sigma = \{0, 1\}$ which accepts all string ending with 01.



NFA

- Example 3:NFA with $\Sigma = \{0, 1\}$ and accept all string of length at least 2.



Present State	Next state for Input 0	Next State of Input 1
$\rightarrow q_0$	q_1	q_1
q_1	q_2	q_2
$*q_2$	ϵ	ϵ

NFA

- Example 4: Design an NFA with $\Sigma = \{0, 1\}$ accepts all string in which the third symbol from the right end is always 0



Question Hour

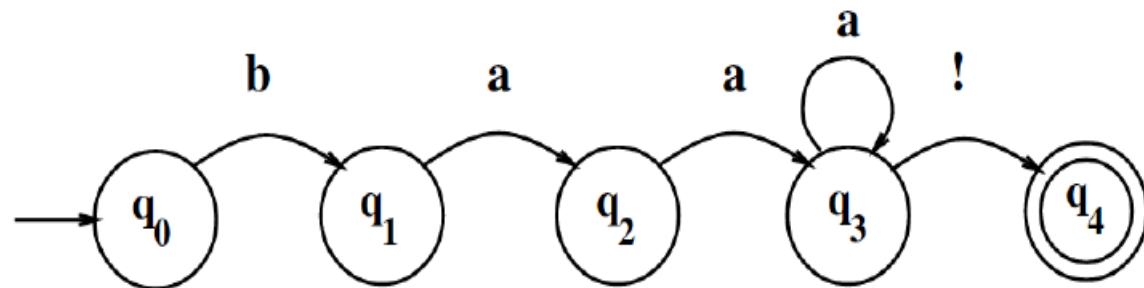
Sheep Language can be any string from the following language
 $L=\{baa!, baaa!, baaaa!, \dots\}$

- Specify the regular expression
- Draw DFA and NFA for L

Question Hour

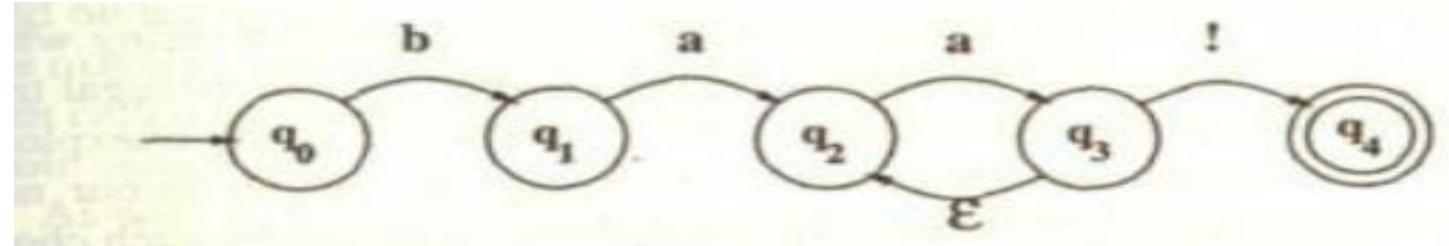
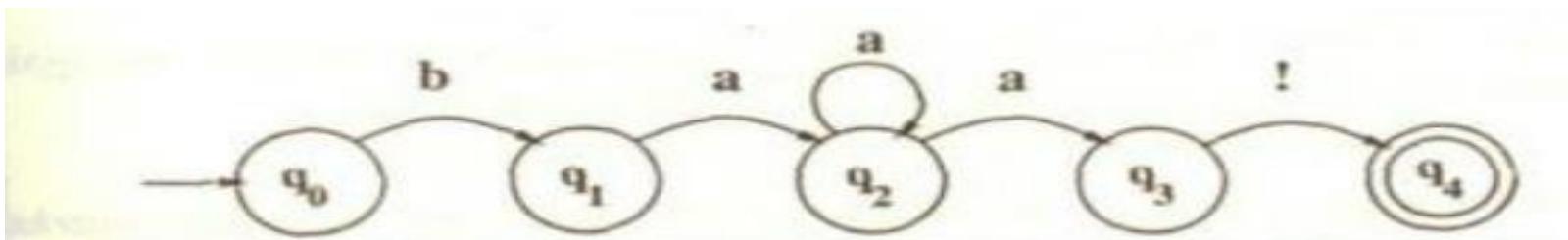
Regular expression can be specified in two ways for the sheep language: $/baaa^*/$ or $/baa^+*/$

DFA



Question Hour

NFA



Unit 3

Morphology and Finite-State Transducers

Morphology

- Study of Words
 - Their **internal structure**

washing → wash + -ing

- How they are **formed?**

bat	→	bats	⋮	rat	→	rats
write	→	writer	⋮	browse	→	browser

- Morphology tries to formulate rules

What is Morphology

- **Morph** = form or shape, **ology** = study of
- Morphology is the study of the way words are built up from smaller meaning-bearing units, **morphemes**.
- A morpheme is often defined as the **minimal meaning-bearing unit in a language**.

Example:

Singular	Plural
Fox	Foxes
Peccary	Peccaries
Goose	Geese
Fish	Fish

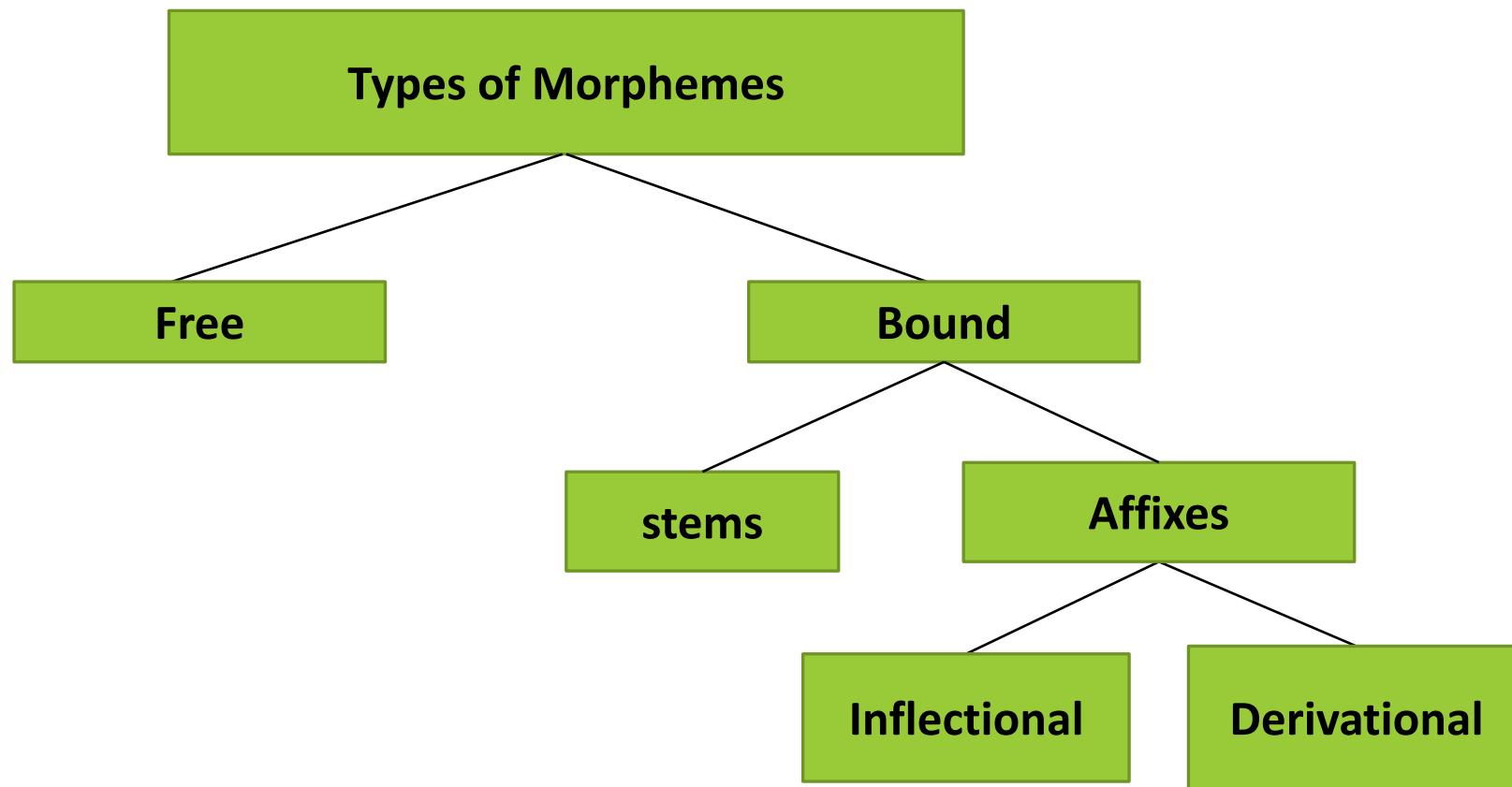
- The word **fox** consists of a single morpheme (the morpheme fox)
- The word **cats** consists of two: the morpheme **cat** and the morpheme **-s**

Morphological Parsing

*The problem of recognizing that **foxes** break down into two morphemes **fox** and **-es** is called morphological parsing*

- **Parsing** means taking an input and producing some sort of structure for it
- In information retrieval domain, the similar problem of mapping from **foxes** to **fox** is called **stemming**
- It takes two kinds of knowledge to correctly search for singulars and plurals of these forms:
 - **Spelling rules** : tells that English words ending in **-y** are pluralized by changing the **-y** to **-i** and adding an **-es**.
 - **Morphological rules** : tells that **fish** has null plural, and the plural of **goose** is formed by changing the vowel.

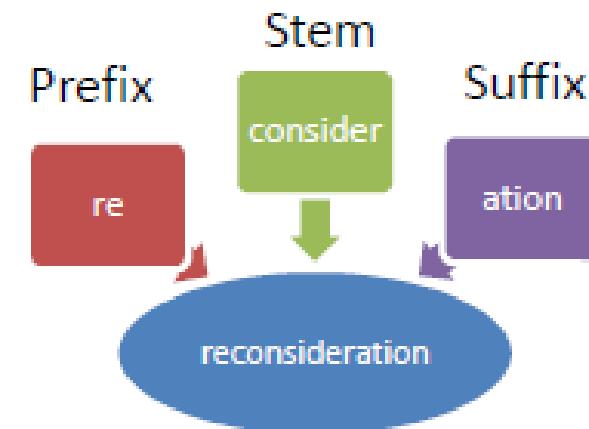
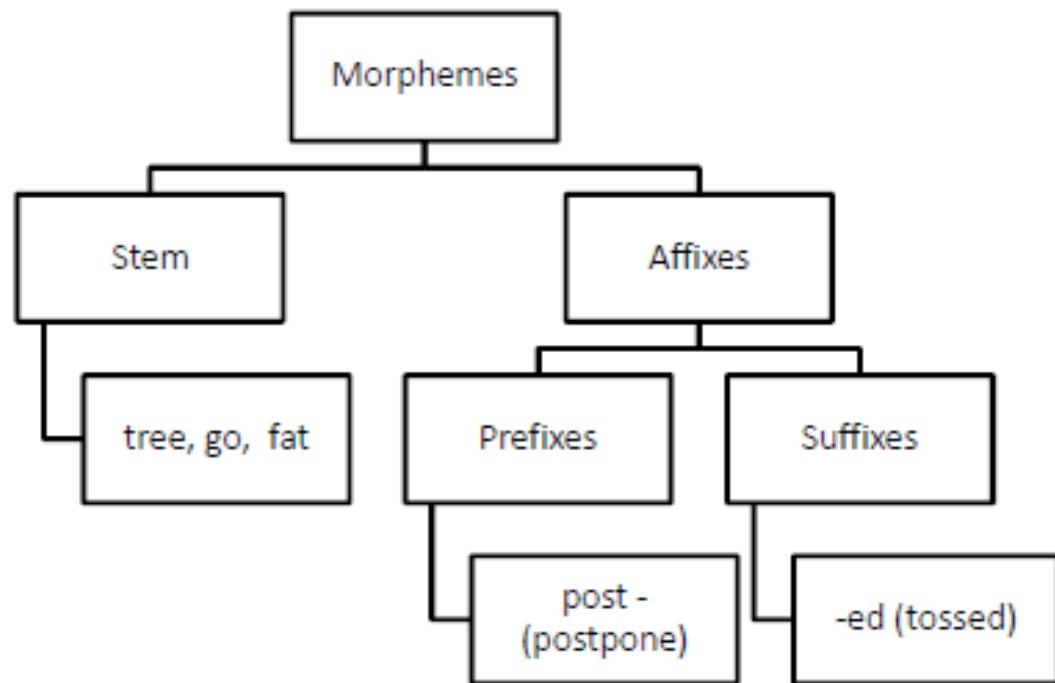
Types of morphemes



Survey of English Morphology

- Two broad classes of morphemes
 - **Stems** – is the “*main*” morpheme of the word, supplying the main meaning
 - **Affixes** – add “*additional*” meanings of various kinds
 - **Prefixes** – precede the stem Eg: the word **unbuckle** composed of **buckle** and prefix **-un**
 - **Suffixes** – follow the stem Eg: the word **eats** composed of stem **eat** and suffix **-s**
 - **Circumfixes** – do both prefix and suffix operation Ex: the word **enlighten** has **en** as prefix and suffix
 - **Infixes** – inserted inside the stem and generally found in plural forms in English. Eg: plural form of **cupful** is **cupsful** and **spoonful** is **spoonsful**.

Survey of English Morphology



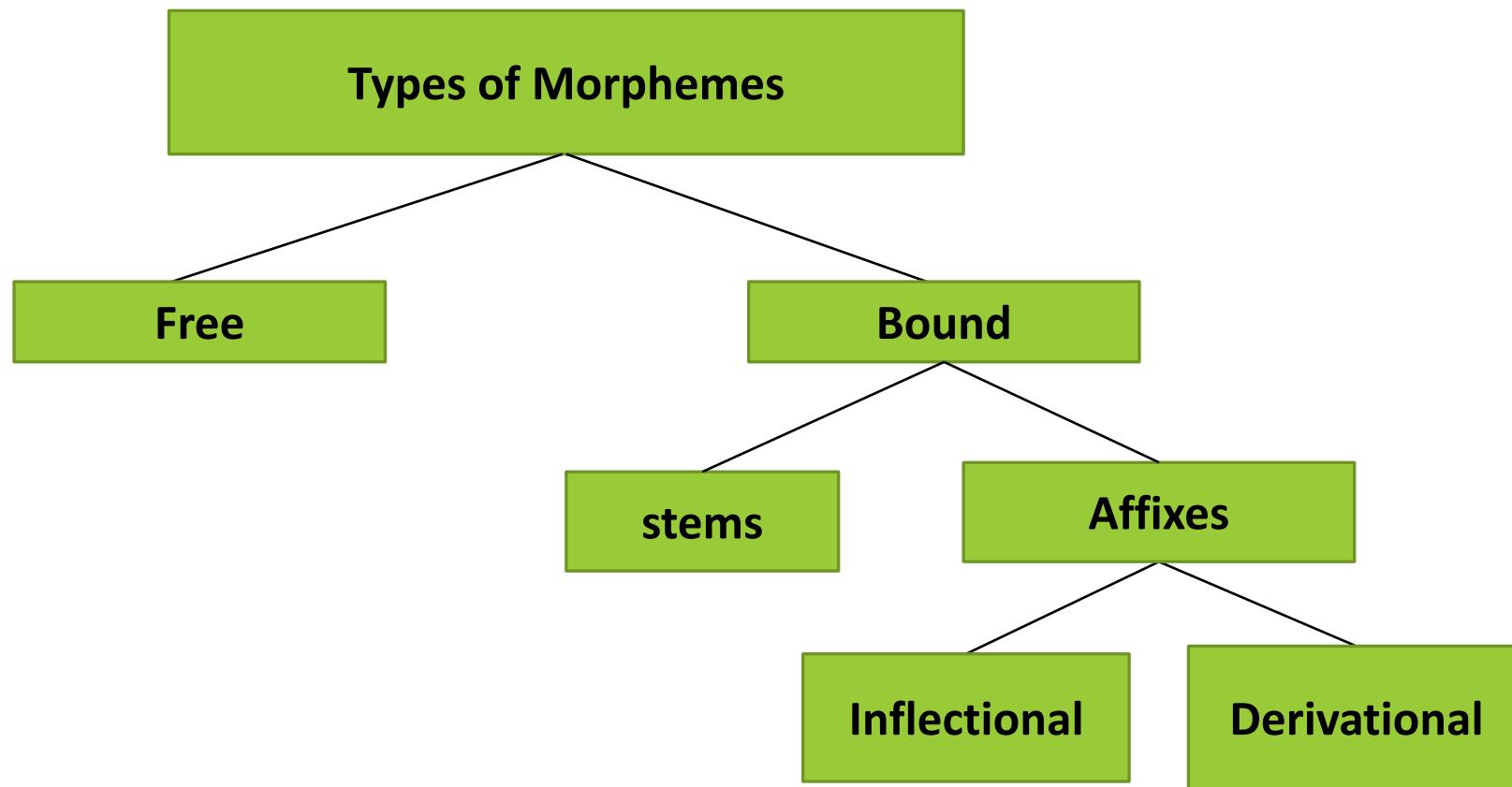
Survey of English Morphology - Another classification of morphology

- **Concatenative morphology** – word is composed of a number of morphemes concatenated together or two morphemes are ordered one after another.
i.e. **affixation** and **compounding**
 - **Affixation**: involves attachment of morphemes to stems. (Suffix, prefix, circumfixes, infix)
 - Example: multiple affixations: **anti- inter- govern -ment -al-ist** . Here govern is root or stem.
 - **Compounding**: New words formed by combining two stems. Can be formed with many parts of speech.
 - Example: **noun-noun : horse-shoe,**
noun-verb : trouble-shoot,
adjective- verb : high-jump,
adjective-adjective: bitter - sweet

Survey of English Morphology - Another classification of morphology Contd.

- **Non-Concatenative morphology (templatic morphology)**- morphemes are combined in more complex ways
 - **Reduplication** : process that involves taking part of the base and attaching it as an affix; description involves how much is copied.
 - Examples: *bang-bang* : sound of a gun when firing, *bye-bye* : goodbye
 - **Internal modification (vowel modification)** : grammatical opposition expressed via a vowel alternation
 - Examples: *sing – sang – sung- song, begin-began, goose-geese , bind- bound*
 - **Consonant modification** : changes made for other than vowel
 - Examples: *belief-believe, grief-grieve*
 - **Mixed modification** : present/past , verb/noun
 - Examples: *catch-caught, seek-sought, live-life, defence-defend, bent-bend*

Types of morphemes



Ways to form words from Morphemes

Two broad classes:

- I. **Inflectional Morphology** : the combination of a word stem with a grammatical morpheme usually resulting in a word of the same class as the original stem

- II. **Derivational Morphology** : the combination of a word stem with a grammatical morpheme usually resulting in a word of a different class, often with a meaning hard to predict exactly

Inflectional Morphology

- In English, only nouns, verbs, and sometimes adjectives can be inflected. The number of affixes is small
- English nouns have only two kinds of inflection:
 - An affix that marks plural
 - An affix that marks possessive
- **Regular nouns** are nouns that can be converted into their plural form by simply adding “-s” and “-es” to their end, whereas **irregular nouns** are nouns that do not follow a standard rule in converting plurals.

An affix marking plural

cat(-s)	finch(-es),
ibis(-es),	box(-es)
thrush(-es)	butterfly(-lies)
ox (oxen) [irregular nouns]	waltz(-es)
mouse (mice) [irregular nouns]	

An affix that marks possessive (realized by apostrophe's)

Regular singular noun	Llama's
Plural nouns not ending in -s	Children's
Regular plural nouns	Llamas'
Names ending in -s or -z	Euripides' , comedies'

Inflectional morphology Contd.

Morphological forms of Irregular verbs

Irregular verbs are those that have some more or less peculiar forms of inflection.

- Often have five different forms, but can have as many as eight (eg. Verb **be**) or as few as three (eg. **cut** or **hit**).

Morphological Form Classes	Irregularly Inflected Verbs		
Stem	eat	catch	cut
-s form	eats	catches	cuts
-ing participle	eating	catching	cutting
Past form	ate	caught	cut
-ed participle	eaten	caught	cut

Derivational Morphology

- **Nominalization in English:** formation of new nouns, often from verbs or adjectives.

Suffix	Base Verb/Adjective	Derived Noun
-ation	computerize (V)	computerization
-ee	appoint (V)	appointee
-er	kill (V)	killer
-ness	fuzzy (A)	fuzziness

- Adjectives can also be derived from nouns and verbs.

Suffix	Base Noun/Verb	Derived Adjective
-al	computation (V)	computational
-able	embrace (V)	embraceable
-less	clue (N)	clueless

Derivational Morphology

- Derivation in English is more complex than inflection because
 - It is generally less productive
 - A nominalizing affix like *-ation* cannot be added to every verb. Thus, cannot be said as *eatation*, *spellation*
 - There are subtle and complex meaning differences among nominalizing suffixes.

Finite-State Morphological Parsing

- Aim is to take the input forms like those in first column and produce output forms like those in second column
- Second column contains stem of each word and asserted morphological features. (asserted features specify additional information of the stem)
 - Example: *+N for Noun, +SG for singular, +PL for plural.*

Input	Morphological Parsed Output
cats	cat + N + PL
cat	cat + N + SG
cities	city + N + PL
geese	goose + N + PL
goose	(goose + N + SG)
merging	(merge + V + PRES-PART)
caught	(catch + V + PAST-PART) or (catch + V + PAST)

Finite-State Morphological Parsing

To build a morphological parser, the following are needed:

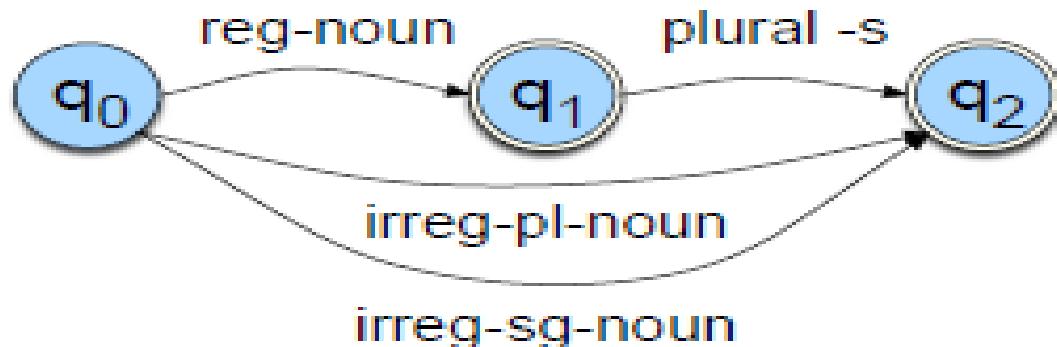
- i. **Lexicon:** the list of stems and affixes, together with basic information about them
Eg., Noun stem or Verb stem, etc.
- ii. **Morphotactics:** the model of morpheme ordering that explains which classes of morphemes can follow other classes of morphemes inside a word.
E.g., the rule that English plural morpheme follows the noun rather than preceding it.
- iii. **Orthographic rules:** spelling rules are used to model the changes that occur in a word, usually when two morphemes combine
E.g., the **y→ie** spelling rule changes **city + -s** to **cities** instead of **citys**.

Lexicon and Morphotactics

- A lexicon is a repository for words.
 - The simplest one would consist of an explicit list of every word of the language. i.e. including abbreviations and proper names.
 - Example: a, AAA, AA, Aachen, aardvark, aba, abaca,
- Computational lexicons are usually structured with
 - a list of each of the stems and
 - Affixes of the language together with a representation of morphotactics telling us how they can fit together.
- The most common way of modeling morphotactics is the **finite-state automaton**.

Lexicon and Morphotactics Contd...

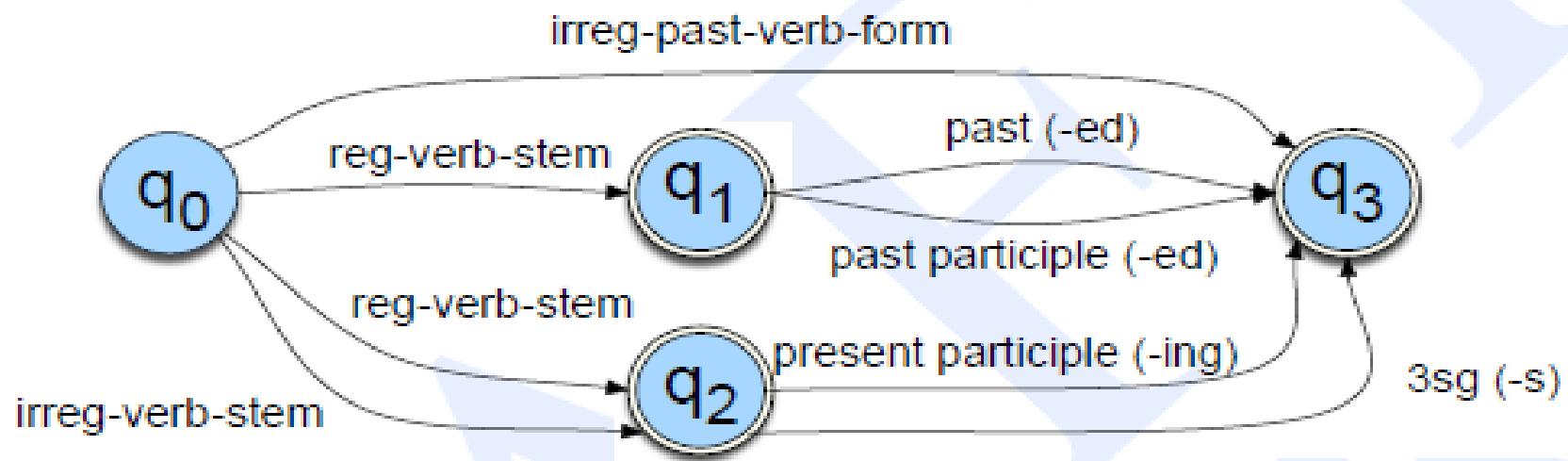
A finite-state automaton for English nominal inflection



reg-noun	irreg-pl-noun	irreg-sg-noun	plural
fox	geese	goose	-s
cat	sheep	sheep	
aardvark	mice	mouse	

Lexicon and Morphotactics Contd.

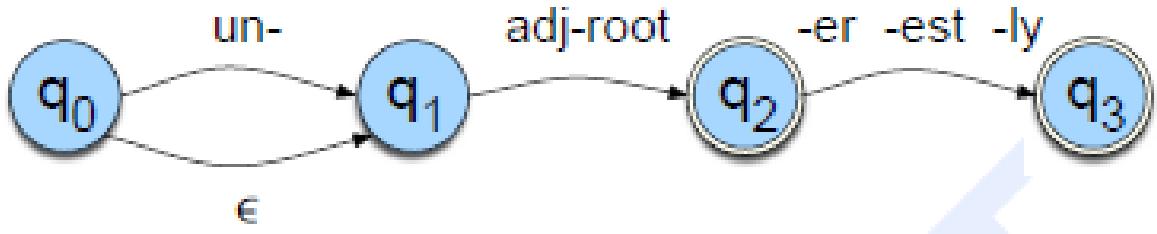
A finite-state automaton for English verbal inflection



reg-verb-stem	irreg-verb-stem	irreg-past-verb	past	past-part	pres-part	3sg
walk	cut	caught	-ed	-ed	-ing	-s
fry	speak	ate				
talk	sing	eaten				
impeach		sang				

Lexicon and Morphotactics Contd.

- English derivational morphology is more complex than English inflectional morphology, and so automata of modeling English derivation tends to be quite complex.
 - Some are even based on Context free Grammers
- A small part of morphosyntax of English adjectives is shown:



An FSA for a fragment of English adjective morphology: #1

big, bigger, biggest

cool, cooler, coolest, coolly

red, redder, reddest

clear, clearer, clearest, clearly, unclear, unclearly

happy, happier, happiest, happily

unhappy, unhappier, unhappiest, unhappily

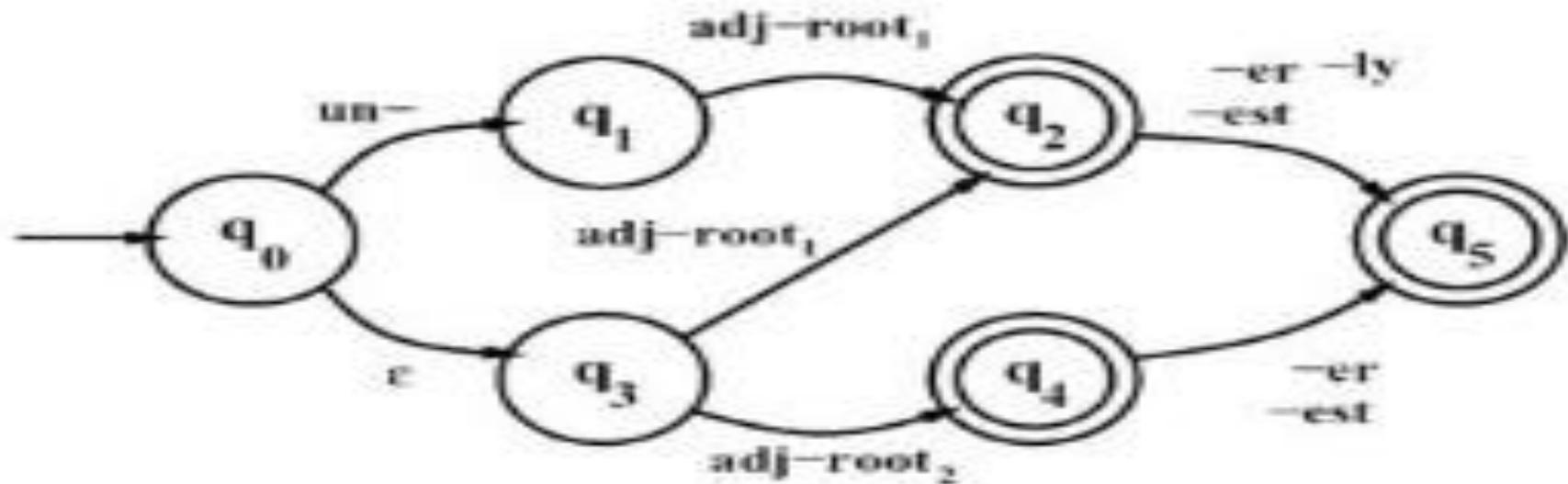
real, unreal, really

Lexicon and Morphotactics Contd.

- The FSA#1 recognizes all the listed adjectives, and ungrammatical forms like unbig, redly, and realest.
- Need to setup classes of roots and specify which can occur with which suffixes. Thus #1 is revised to become #2.
 - adj-root₁ would include adjectives that can occur with un- and -ly (clear, happy)
 - adj-root₂ will include adjectives that can't occur with un- and -ly (big, red)

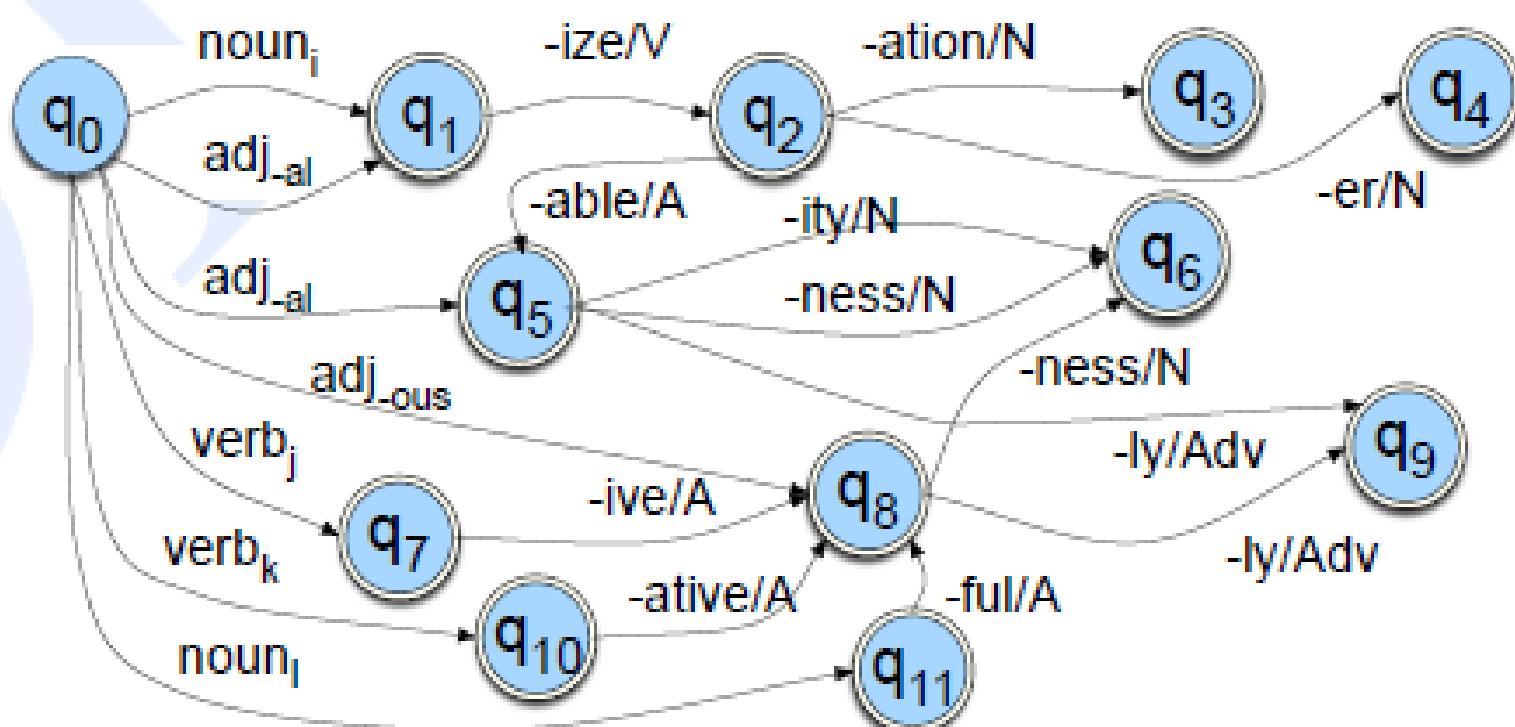
An FSA for a fragment

of English adjective Morphology #2



Lexicon and Morphotactics Contd.

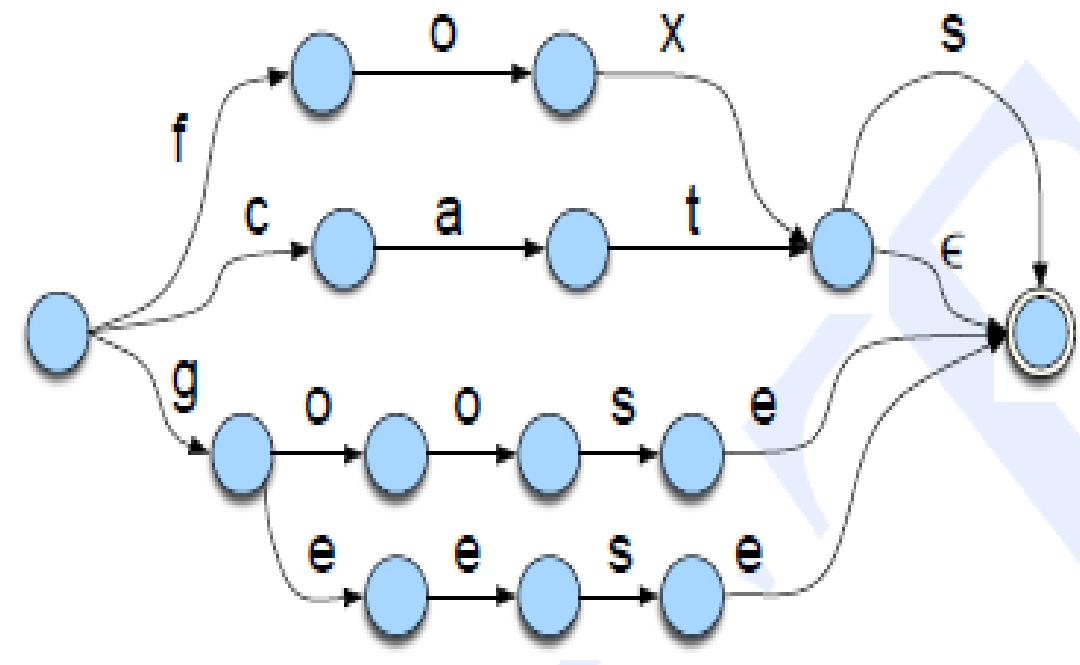
An FSA for English nominal and verbal derivational morphology



- Verb ending with **-ize** can be followed by nominalizing suffix **-ation**
- For the word **fossilize** , the word **fossilization** can be predicted by following the states q_0, q_1, q_2 .

Lexicon and Morphotactics Contd.

- FSAs can be used to solve the problem of **morphological recognition**.
- **Morphological recognition:** determining whether an input string of letters makes up a legitimate English word or not.
- This is done by taking morphotactic FSAs, and plugging in each “**sub-lexicon**” into FSA. (i.e. expand each arc (eg. the **reg-noun-stem** arc)with all the morphemes that make up the set of **reg-noun-stem**.)
- The resulting FSA can then be defined at the level of the individual user.



Compiled FSA for a few English nouns
with their inflections

Finite State Transducers

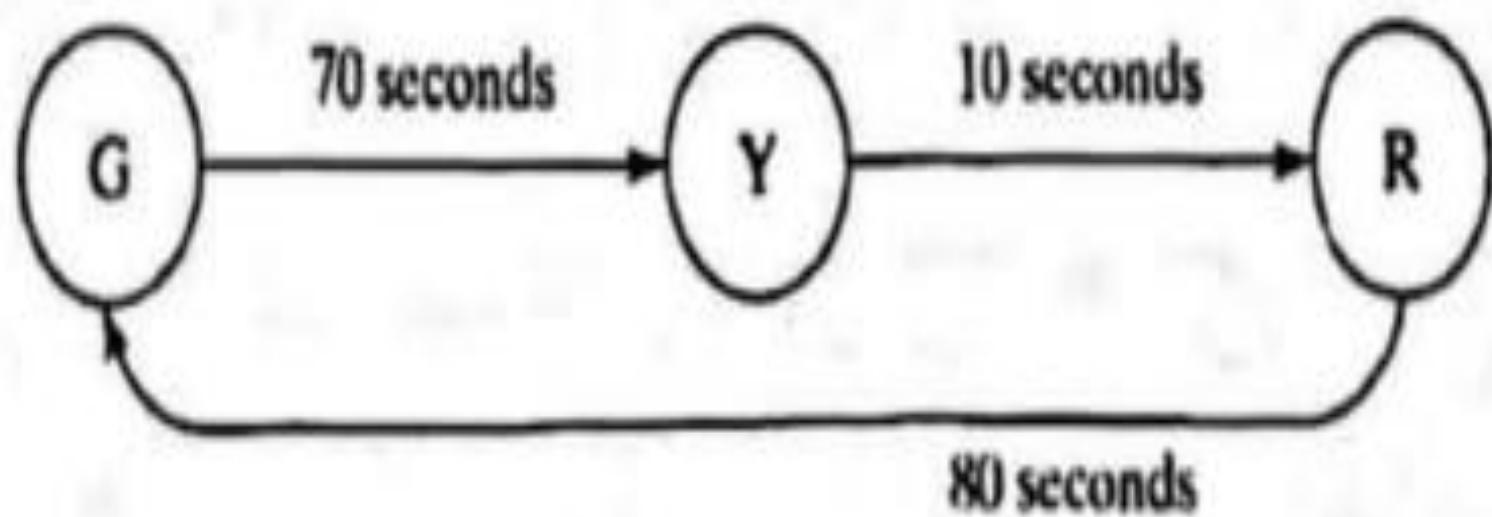
Finite State Transducers are loops that simply run forever, processing inputs

- An **automaton** that produces outputs based on current input and/or previous state is called a **transducer**
- Transducers can be of two types:
 - **Moore Machine** -The output depends only on the current state
 - **Mealy Machine**- The output depends both on the current state and the current input

Finite State Transducers

Moore Machine: The output depends only on the current state.

Example: Traffic light

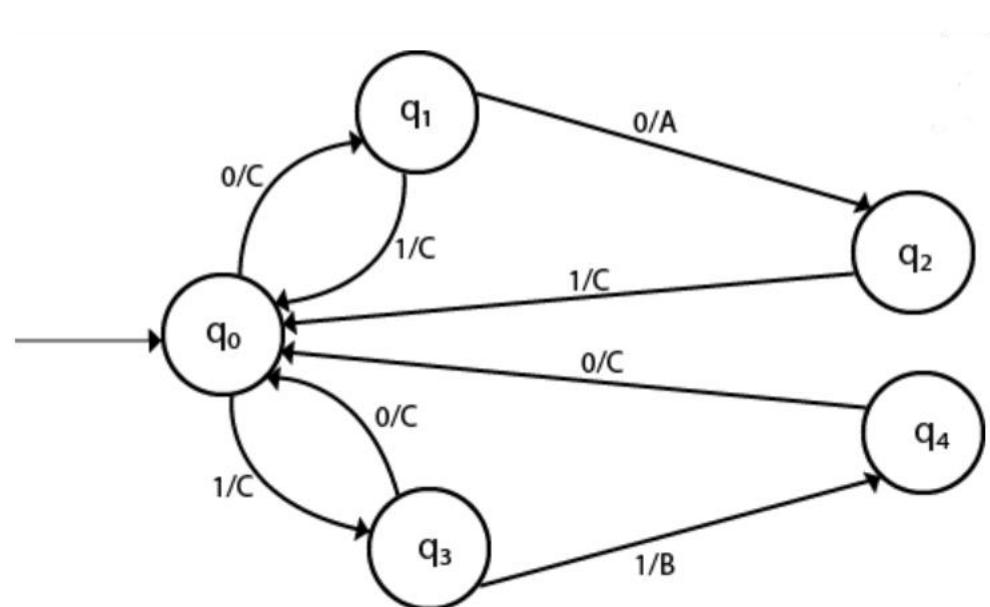


Finite-State Transducers

2. **Mealy Machine:** The output depends both on the current state and the current input

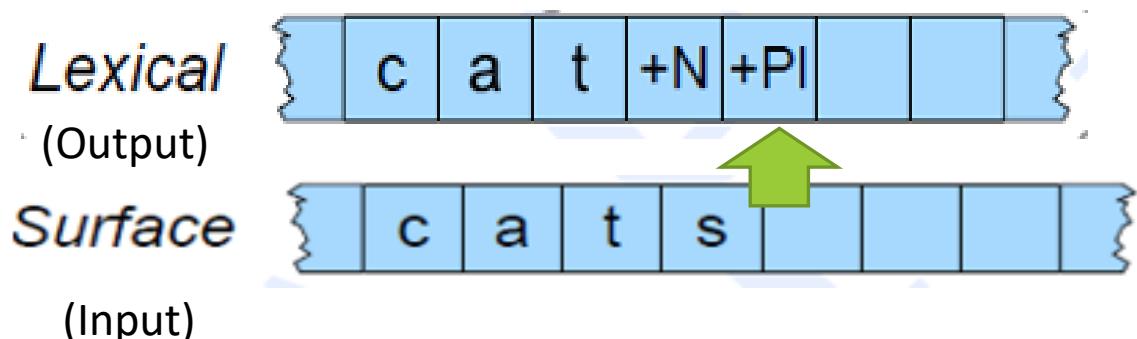
I. Generating parity bits (adds odd parity bit after every four bits)

II. Design a mealy machine that scans sequence of input of 0 and 1 and generates output 'A' if the input string terminates in 00, output 'B' if the string terminates in 11, and output 'C' otherwise



Morphological Parsing with Finite-State Transducers

- Given the input, for example, ***cats***, we would like to produce ***cat +N +PL***.
- Two-level morphology**, by Koskenniemi (1983)
 - The lexical level** represents a simple concatenation of morphemes making up a word
 - The surface level** which represents the actual spelling of the final word.
- Morphological parsing is implemented by building mapping rules that maps letter sequences like ***cats*** on the surface level into morpheme and features sequence like ***cat +N +PL*** on the lexical level.



- Figure shows two levels: lexical and surface level for the word ***cats***.
- Lexical level has the stem word followed by morphological information ***+N +PL*** that tells ***cats*** is a plural noun

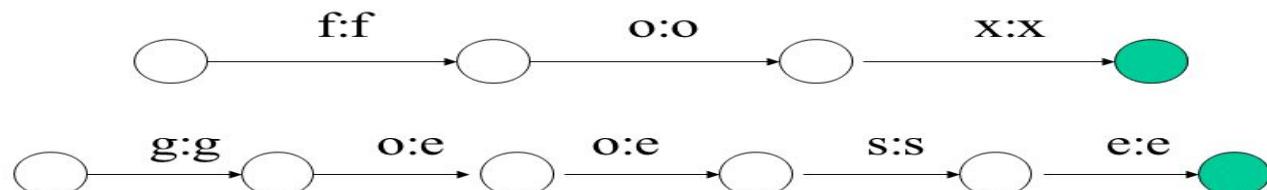
Morphological Parsing with Finite-State Transducers

- The automaton we use for performing the mapping between these two levels is the **Finite-State Transducer or FST**.
- A transducer maps one set of symbols with another set via a **finite automaton**.
- Thus an FST can be seen as a **two-tape automaton which recognizes or generates pairs of strings**.
- The FST has a more general function than an FSA:
 - An FSA defines a **formal language** by defining set of strings, whereas FST defines a relation between sets of strings.

1) FSA **recogniser** for "fox"



2) FST **transducers** for fox/fox; goose/geese



Morphological Parsing with Finite-State Transducers

- Another view of an FST is that it is a machine that reads one string and generates another.

$L_{IN} = \{cat, cats, fox, foxes, \dots\}$

$L_{out} = \{cat +N +SG, cat +N +PL, fox +N +SG, fox +N +PL \dots\}$

$T = \{<cat, cat +N +SG>, <cats, cat +N +PL>, <fox, fox +N +SG>, <foxes, fox +N +PL >\dots\}$

Morphological Parsing with Finite-State Transducers

Four-fold way thinking about transducers

- **FST as recognizer:** a transducer that takes (recognizes) a pair of strings as input and outputs *accept* if the string-pair is in the string-pair language, and a *reject* if it is not.
- **FST as generator:** a machine that outputs pairs of strings of the language. Thus the output is a yes or no, and a pair of output strings.
- **FST as transducer(translator):** A machine that reads a string and outputs another string.
 - Morphological parsing: letter(input), morphemes (output)
- **FST as set relater:** A machine that computes relations between sets.

Morphological Parsing with Finite-State Transducers

- A formal definition of FST (Mealy machine extension to a simple FSA):

- Q : a finite set of **N** states q_0, q_1, \dots, q_N
- Σ : a finite alphabet of complex symbols.

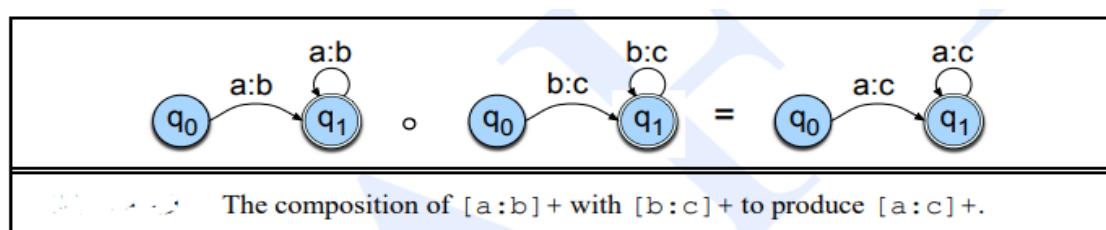
Each complex symbol is composed of an input-output pair $i : o$. I and O may each also include the epsilon symbol ϵ .

- q_0 : the start state
- F : the set of final states, $F \subseteq Q$
- $\delta(q, i:o)$: the transition function or transition matrix between states. Given a state $q \in Q$ and complex symbol $i:o \in \Sigma$, $\delta(q, i:o)$ returns a new state $q' \in Q$. δ is thus a relation from $Q \times \Sigma$ to Q

Morphological Parsing with Finite-State Transducers

- Two useful closure properties of FSTs:

- **Inversion:** Inversion of transducers switches input and output labels. Thus, If T maps from Input alphabet I to output alphabet O , then the inverse of T , T^{-1} maps from O to I .
- Inversion is useful because it makes it easy to convert a **FST-as-parser** into an **FST-as-generator**.
- **Composition:** If T_1 is a transducer from I_1 to O_1 and T_2 a transducer from I_2 to O_2 , then $T_1 \circ T_2$ maps from I_1 to O_2
- Composition is useful because it allows us to take two transducers that run in series and replace them with one complex transducer.
- $T_1 \circ T_2(S) = T_2(T_1(S))$



→

The composition of $[a:b]^+$ with $[b:c]^+$ to produce $[a:c]^+$.

Morphological Parsing with Finite-State Transducers

- Morphological parser maps the surface forms to lexical forms by **cascading** the lexicon with singular or plural automaton.
- **Cascading:** running two automata in series with the output of first feeding the input to the second
- lexicon of stems are represented as FST T_{stems} . Example: dog to ***reg-noun-stem***
- For suffixes: T_{stems} allows the forms to be followed by **@:@** (any feasible pair)
- Alternative to cascading is **composing** the transducer using **composition operator**.
- **Composing** is a way of taking a cascade of transducers with many different levels of inputs and outputs and converting them into single “***two level***” transducer with one input tape and one output tape.
- Composed automaton is $T_{\text{lex}} = T_{\text{num}} \circ T_{\text{stems}}$

Morphological Parsing with Finite-State Transducers

- Regular expressions can be written in the complex alphabet Σ as in FSA
- For two-level morphology, FST is viewed as having two level tapes.
 - **The upper or lexical tape, The lower or surface tape**
- Each symbol $a:b$ in the transducer alphabet Σ expresses how the symbol a from one tape is mapped to the symbol b on the another tape
 - Example: $a:\epsilon$ means a on the upper tape will correspond to nothing in lower tape.
- Generally, symbols map to themselves, in two level morphology. Hence called as **default pairs**. Example: $a:a$ (**referred to this by single letter a**)
- A simple pair of symbols in Σ are called **feasible pairs**. Example: $a:!, a:\epsilon$

Morphological Parsing with Finite-State Transducers

- FSA accepts a language stated over a finite alphabet of single symbols.

Example: alphabet of sheep language $\Sigma=\{b,a,!\}$

- FST accepts a language stated over a pair of symbols

Example: $\Sigma=\{a:a, b:b, !:!, a:!, a: \epsilon, \epsilon:!\}$

- FSAs are isomorphic to regular languages, FSTs are isomorphic to regular relations.

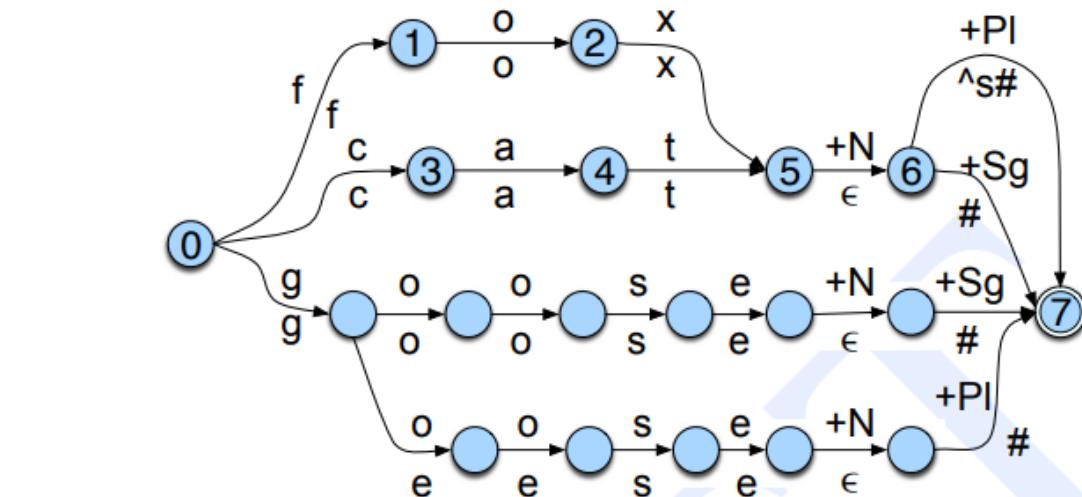
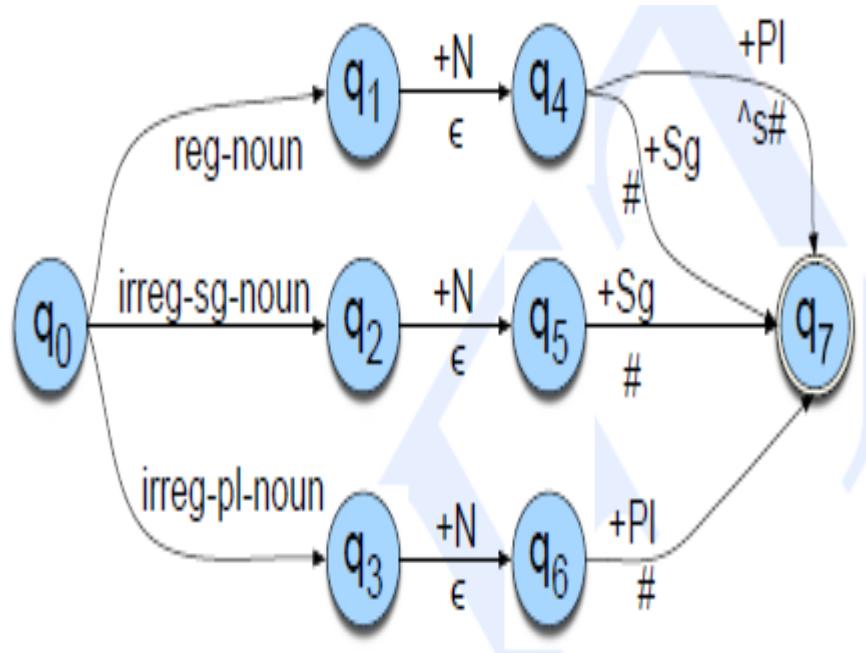
- Regular relations are sets of pairs of strings, an extension of the regular language, which are sets of strings.

Morphological Parsing with Finite-State Transducers

- **Application**

- An FST morphological parser is built for earlier morphotactic FSAs and lexica by adding an extra “*lexical*” tape and appropriate morphological features.
 - Nominal morphological features map to the empty string ϵ or the word/morpheme boundary symbol $\#$ since there is no segment corresponding to them on the output tape.

Morphological Parsing with Finite-State Transducers



A fleshed-out English nominal inflection FST T_{lex} , expanded from T_{num} by replacing the three arcs with individual word stems (only a few sample word stems are shown).

Morphological Parsing with Finite-State Transducers

- For morphological noun parser, all the individual regular and irregular noun stems are augmented.
 - Lexicon of the transducer is updated so that regular plurals like **geese** will parse into correct stem **goose +N +PL**.
Example: surface **geese** maps to underlying **goose**, the new lexical entry will be “**g:g o:e o:e s:s e:e**”.
 - Regular forms are simpler. Two level entry for fox will be “**f:f o:o x:x**”, but relying on orthographic convention that **f** stands for **f:f**, **o** stands for **o:o** etc. it can be referred to as “**fox**” and **goose** as “**g o:e o:e s e**”.

Morphological Parsing with Finite-State Transducers

- Maps plural nouns into stem plus the morphological marker **+PL**, and singular nouns into stem plus **+SG**.

- Thus, surface **cats** will map to **cat +N +PL** as :
c:c a:a t:t +N: ε +PL: ^ s #

- Symbol **^** indicates morpheme boundary
- Symbol **#** indicates word boundary.

reg-noun	irreg-pl-noun	irreg-sg-noun
fox	g o: e o: e s e	goose
cat	sheep	sheep
aardvark	m o i u e s c e	mouse

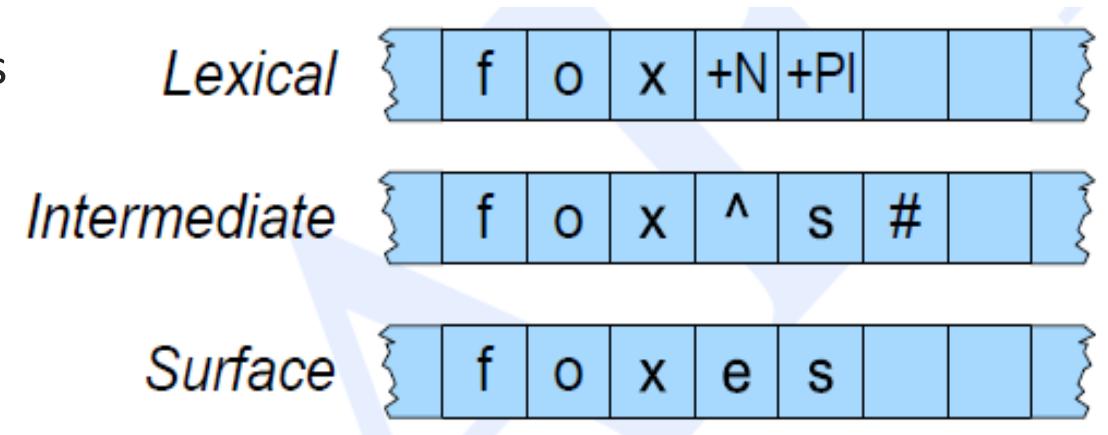
Orthographic rules and Finite-State Transducers

- Concatenating the morphemes can work to parse the words like “**dog**”, “**cat**”.
- But this simple method does not work. *Example:* “**foxes**” is to be parsed into lexicons “**fox +N +PL**” etc.
- This requires introduction of spelling rules (also called orthographic rules).
- Some of the spelling rules:

Name	Description of Rule	Example
Consonant doubling	1-letter consonant doubled before <i>-ing/-ed</i>	beg/begging
E deletion	Silent e dropped before <i>-ing</i> and <i>-ed</i>	make/making
E insertion	e added after <i>-s,-z,-x,-ch, -sh</i> before <i>-s</i>	watch/watches
Y replacement	<i>-y</i> changes to <i>-ie</i> before <i>-s</i> , <i>-i</i> before <i>-ed</i>	try/tries
K insertion	verbs ending with <i>vowel + -c</i> add <i>-k</i>	panic/panicked

Orthographic rules and Finite-State Transducers

- To account for the spelling rules, another tape is introduced, called intermediate tape.
- Intermediate tape:** produces the output slightly modified, thus going from 2-level to 3-level morphology
- Between each level of tapes is a **two-level transducer**.
 - Lexical transducer** - between lexical and intermediate levels
 - E-insertion spelling rule** - between the intermediate and surface levels
- E-insertion spelling rule inserts an **e** on the surface tape when the intermediate tape has a morpheme boundary **^** followed by the morpheme **-s**



Orthographic rules and Finite-State Transducers

- The E-insertion rule states that : “*insert an e on the surface tape just when the lexical tape has a morpheme ending in x (or z, etc) and the next morpheme is -s*”.

Formalization of the rule :

$$\epsilon \rightarrow e / \left\{ \begin{array}{l} x \\ s \\ z \end{array} \right\} \wedge _ s\#$$

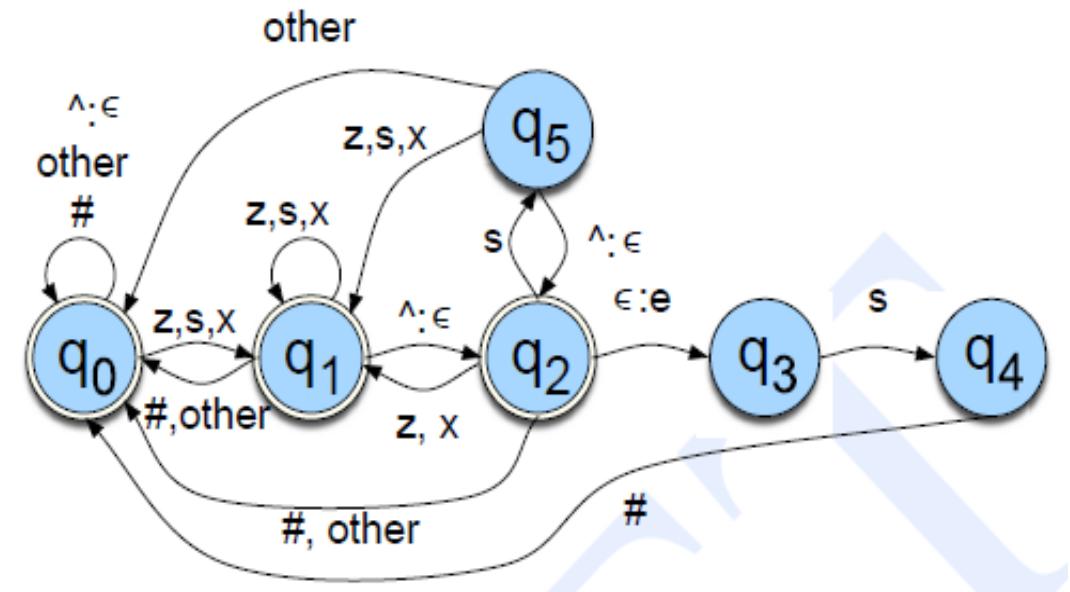
Meaning: Insert e after a morpheme-final x, s, or z, and before the morpheme s

This follows the rule notation of Chomsky and Halle(1968); $(a \rightarrow b/c _ d$ means “*rewrite a as b when it occurs between c and d*”)

- ϵ is the **empty transition**, replacing it means inserting something.
- the symbol \wedge indicates morpheme boundary. Boundaries are deleted by including the symbol $\wedge: \epsilon$ in the default pairs of transducer. Thus morpheme boundary markers are deleted on the surface level by default.
- The # symbol marks the word boundary.

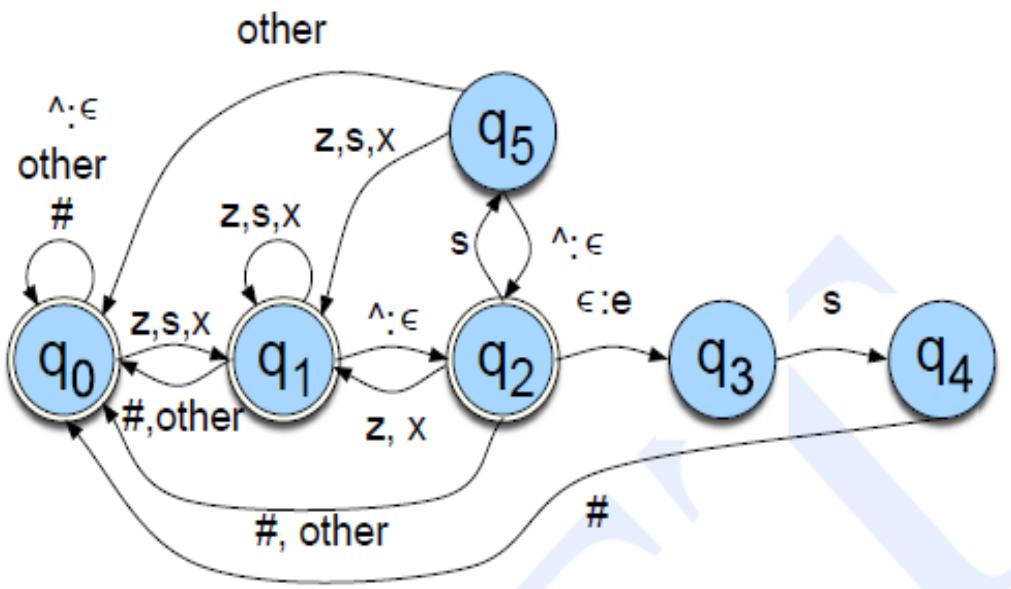
Orthographic rules and Finite-State Transducers

- Building a transducer for any rule is to express only the constraints necessary for that rule, allowing any other strings of symbols to pass through unchanged. (i.e. it ensures only $\epsilon:e$ pair is seen in proper context)
- State q_0 models when seeing only default pairs unrelated to the rule. It is an accepting state.
- State q_1 models having seen z , s , or x and it's an accepting state
- State q_2 models having seen the morpheme boundary. It is also an accepting state
- State q_3 models having seen the **E-insertion**; *it is not an accepting state since the insertion is only allowed if it is followed by the s morpheme and then the end-of-the-word symbol #*



Orthographic rules and Finite-State Transducers

- The **other** symbol is used to pass through any parts of words that don't have a role in E-insertion rule. (*other means “any feasible pair that is not in this transducer”*)

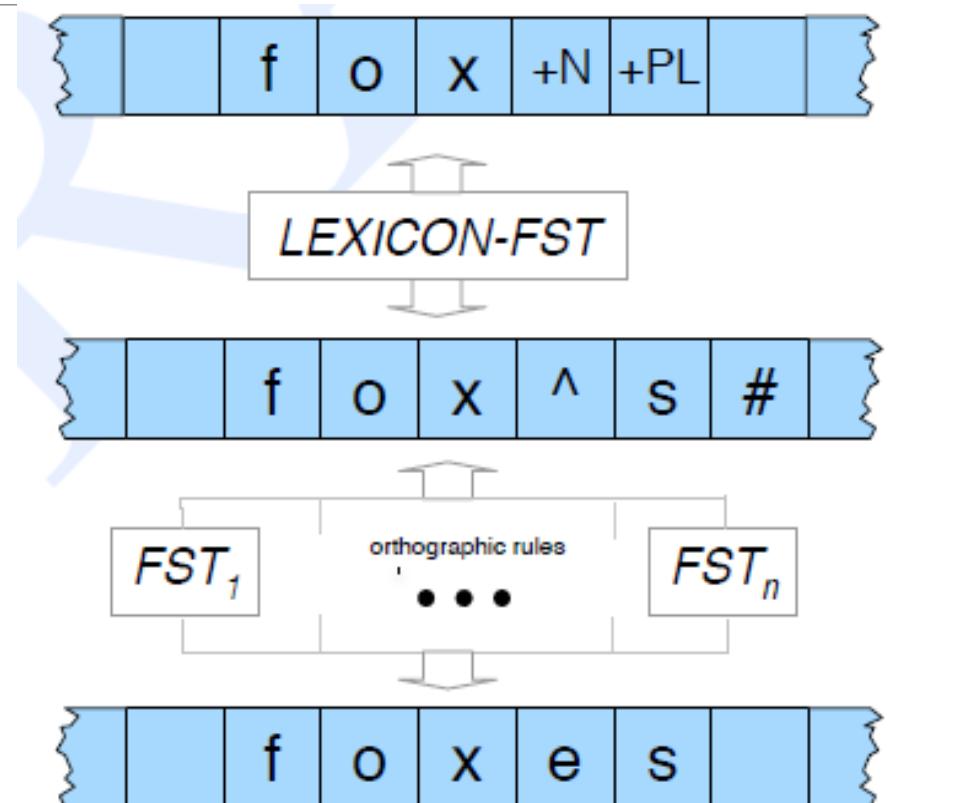


State \ Input	s : s	x : x	z : z	^ : ε	ε : e	#	other
$q_0:$	1	1	1	0	-	0	0
$q_1:$	1	1	1	2	-	0	0
$q_2:$	5	1	1	0	3	0	0
q_3	4	-	-	-	-	-	-
q_4	-	-	-	-	-	0	-
q_5	1	1	1	2	-	-	0

State-transition table for E-insertion rule

Combining FST Lexicon and Rules

- Two-level morphology system used for parsing or generating.
- **Lexicon transducer** – maps between the *lexical level* (stems and morphological features) and an *intermediate level* (represents concatenation of morphemes)
- **Host of transducers** – (*each represents a single spelling rule*) – run in parallel; maps between *intermediate level* and *surface level*
- *Putting the spelling rules in parallel or in series (cascading) is a design choice.*
- The architecture is a **two level cascade of transducers**
- The output from one transducer acts as an input to another transducer
- *The cascade can be run top-down to generate a string, or bottom-up to parse it*



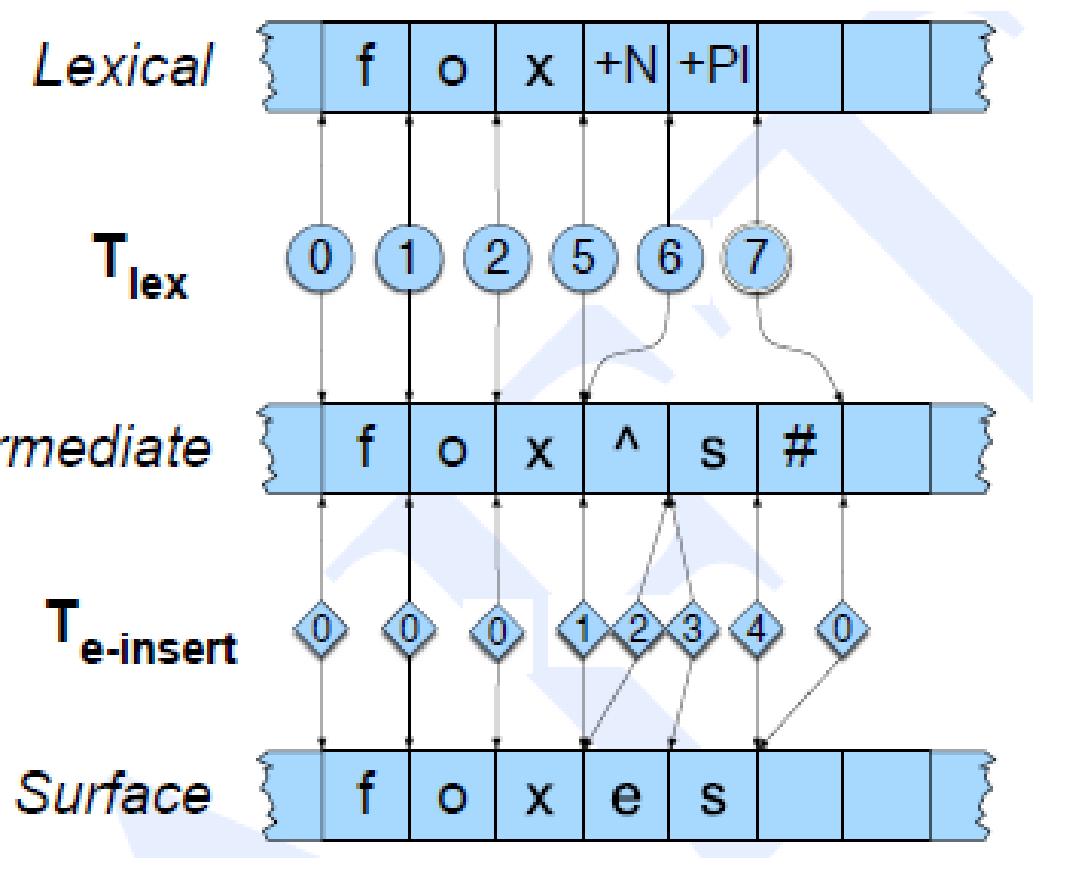
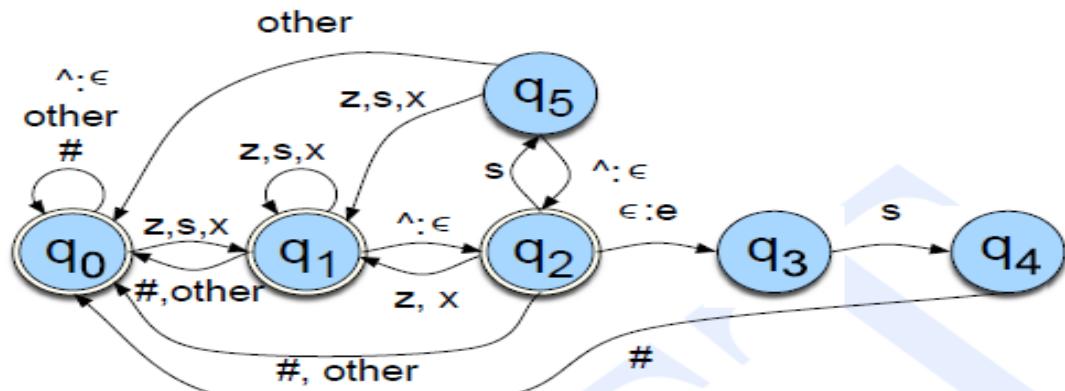
Generating or parsing with FST lexicon and rules

Combining FST Lexicon and Rules

- Figure shows a trace of the system *accepting* the mapping from **fox +N +PL** to **foxes**.

- Generation:** (surface tape from lexical tape)

- Running lexicon transducer, for **fox +N +PL**, it will produce **fox^#** on the intermediate tape.
- Running all possible orthographic transducers to run in parallel, will produce the **foxes** in the surface tape.



Generating or parsing with FST lexicon and rules

Combining FST Lexicon and Rules

- **Parsing:** Complicated because of the problem of “ambiguity”
- Example: **foxes** can be verb and hence lexical parse for **foxes** could be **foxes +V +3SG** or **fox +N +PL**

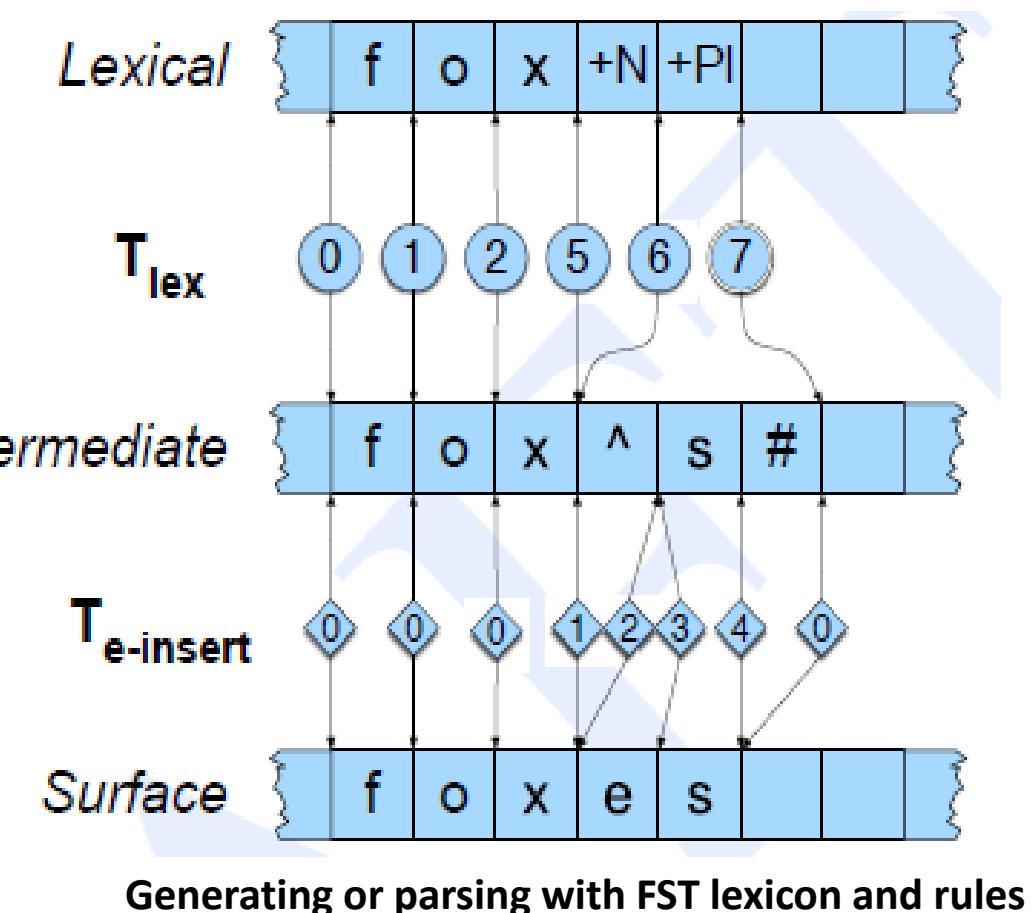
(Can't decide on the correct parse by the transducer)

- **Disambiguating:** require external evidence such as surrounding words

Example: “*I saw two foxes yesterday*”. (*Foxes will be noun*)

“*That trickster foxes me every time!*” (*Foxes will be verb*)

Disambiguation algorithms are used to solve this.
If not both the choices are considered. (Noun and verb)

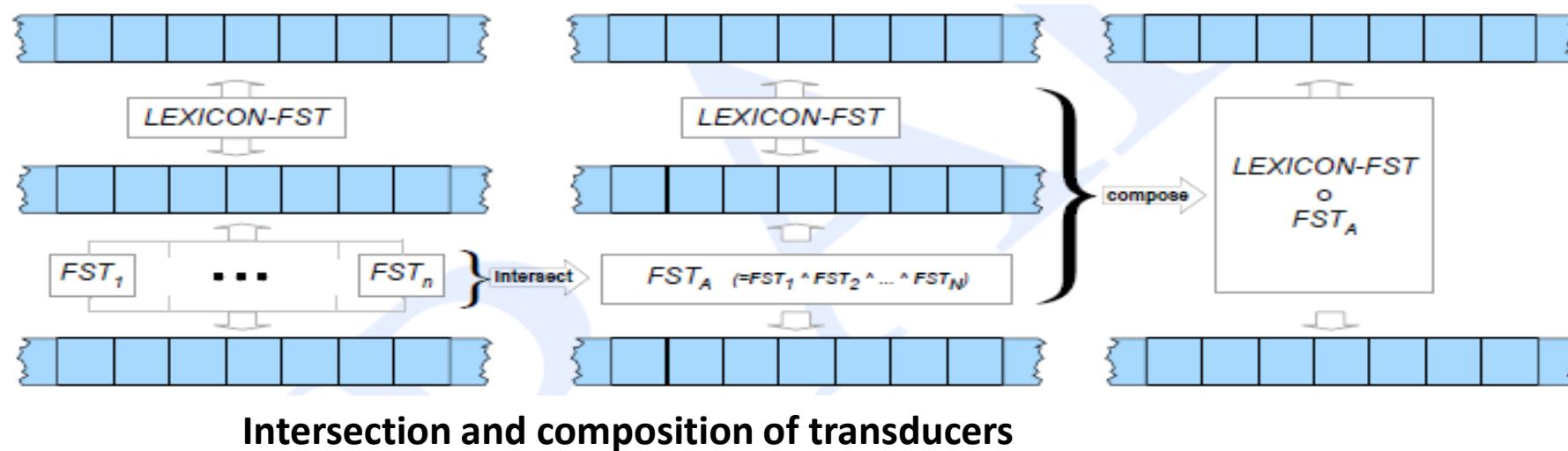


Combining FST Lexicon and Rules

- **Local ambiguity:** occurs during the process of parsing
- Example:
 - Parsing the input verb “*assess*”
 - E-insertion transducer, after seeing the word “*ass*” may propose that, *e* that follows is inserted by spelling rule. Hence, the word “*asses*” would be parsed instead of “*assess*”
 - But *#* symbol is not present after “*asses*” instead another *s* is seen. Then its realized that wrong word is parsed
- Solution: FST-parsing algorithm need to incorporate search algorithms.

Combining FST Lexicon and Rules

- Composing a cascade of transducers in series into a single complex transducer
- Transducers running in parallel can be combined by **automaton intersection**
- **Automaton intersection algorithm takes cartesian product of the states.**
 - i.e. for each state q_i in machine 1 and state q_j in machine 2, we create a new state q_{ij} . Then for any input symbol a , if machine 1 would transition to state q_n and machine 2 would transition to state q_m , we transition to state q_{nm} .



Lexicon-Free FSTs: The Porter Stemmer

- **Information Retrieval (IR) tasks** like web search - a query such as a Boolean combination of relevant **keywords or phrases**, e.g., **(kangaroo)** returns documents that have these words in them.
- A document with the word **kangaroos** might not match the keyword **kangaroo**. Hence, some IR systems first **run a stemmer on the query and document words**.
- Morphological information in IR is used to determine that two words have the **same stem**; the suffixes are thrown away.
- The most widely used such **stemming algorithm** is the **Porter algorithm (1980)** .

Human Morphological Processing

- Modern experimental evidence suggests that both the theories (**full listing and minimum redundancy**) are not completely true
- Instead it states that some kinds of morphological relationships are mentally represented for some words (**inflections and certain derivations**) and **for others, words are being fully listed.**
 - **Example:** derived forms **happiness, happily** are stored separately from their stem **happy**
- But regularly inflected forms (**pouring**) are distinct in the lexicon from their stems (**pour**). This is done by using a **repetition priming experiment**.
 - **Repetition priming** – word is recognized faster if it has been seen before (**primed**)
- **Marslen-Wilson (1994)** found that spoken words can prime their stems, but only if the meaning of the derived form is closely related to the stem
 - **Example:** **Government** primes **govern**, but **department** does not prime **depart**

Human Morphological Processing

Speech errors (slips of the tongue)

- In normal conversation, speakers often mix up the order of the words or initial sounds:

Example:

- *If you break it it'll drop*
- *I don't have time to work to watch television because I have to work*

- Inflectional and derivational affixes can appear separately from their stems:

Example:

- *Easily enoughly (for “easily enough”)*
- *Words of rule formation (for “rules of word formation”)*

This shows that mental lexicon must contain some representation of the morphological structure of these words

Hence, Morphology play a role in the human lexicon

Summary

1. Morphological Parsing
2. Affixes: prefixes, suffixes
3. Survey of English Languages
4. Inflectional and Derivational morphology
5. Finite-state Morphological parsing
6. Lexicon and Morphotactics
7. Finite-state Transducers (FST) – FST Morphological parsing
8. Spelling rules (Orthographic rules) and Finite-state Transducers
9. Combining FST lexicon and spelling rules (composing and intersecting)
10. Lexicon-Free FSTs - (Porter stemmer algorithm)
11. Human Morphology processing

END

UNIT-4

Tokenization, Normalization, Stemming

- **corpus (plural corpora)** - computer-readable corpora collection of text or speech.
 - For example the Brown corpus is a million-word collection of samples from 500 written English texts from different genres (newspaper, fiction, non-fiction, academic, etc.), assembled at Brown University in 1963–64 (Kucera and Francis, 1967).
- **“He stepped out into the hall, was delighted to encounter a water brother”.**
 - This sentence has 13 words if we don’t count punctuation marks as words, 15 if we count punctuation.
 - Whether we treat period (“.”), comma (“,”), and so on as words depends on the task.
 - Punctuation is critical for finding boundaries of things (commas, periods, colons) and for identifying some aspects of meaning
 - part-of-speech tagging or parsing or speech synthesis, punctuations are treated as separate words

- The best way for creating the corpus is to build a datasheet or data statement for each corpus.
- A datasheet specifies properties of a dataset like:
 - **Motivation:** Why was the corpus collected, by whom, and who funded it?
 - **Situation:** When and in what situation was the text written/spoken?
For example, was there a task? Was the language originally spoken conversation, edited text, social media communication, monologue vs. dialogue?
 - **Language variety:** What language (including dialect/region) was the corpus in?
 - **Speaker demographics:** What was, e.g., age or gender of the authors of the text?
 - **Collection process:** How big is the data? If it is a subsample how was it sampled? Was the data collected with consent? How was the data pre-processed, and what metadata is available?
 - **Annotation process:** What are the annotations, what are the demographics of the annotators, how were they trained, how was the data annotated?
 - **Distribution:** Are there copyright or other intellectual property restrictions?

Basic Terminologies

Example: “do uh main- mainly business data processing”

- **Fragment** - The broken-off word **main-** is called a fragment.
- Words like **uh** and **um** are called **fillers or filled pauses**. It depends on the application whether to consider this or not
- Example: **Cat** and **cats**
 - Have same lemma **cat**, but are different **word forms**
 - A **lemma** is a set of lexical forms having the same stem, the same major part-of-speech, and the same word sense.
 - The **wordform** is the full inflected or derived form of the word

- **Types:** are the number of distinct words in a corpus
 - If the set of words in the vocabulary is V , the number of types is the word token vocabulary size $|V|$.
- **Tokens(Word Tokens)** are the total number of running words in the corpora

Corpus	Tokens = N	Types = V
Shakespeare	884 thousand	31 thousand
Brown corpus	1 million	38 thousand
Switchboard telephone conversations	2.4 million	20 thousand
COCA	440 million	2 million
Google N-grams	1 trillion	13 million

Rough numbers of types and tokens for some English language corpora

Herdan's Law or Heaps' Law

- Relationship between the number of types $|V|$ and number of tokens N

$$|V| = kN^\beta$$

where k and β are positive constants, and $0 < \beta < 1$

- The value of β depends on the corpus size and the genre, but generally ranges from 0.4 to 0.6.

Example:

N (TOTAL NUMBER OF WORD)	V (UNIQUE WORDS)
2100	728
4162	1120
6000	1458
8000	1800
10000	2022

$$V(n) = K n^\beta \Rightarrow \log V(n) = \beta \log(n) + K$$

Use, $V(n) = 1458$ and $n = 6000$
 $\Rightarrow \log(1458) = \beta \log(6000) + K \quad \text{--(1)}$

$V(n) = 1800$ and $n = 8000$
 $\Rightarrow \log(1800) = \beta \log(8000) + K \quad \text{--(2)}$

- Solving (1) and (2), we get $\beta = 0.69$ and $K = 3.71$
- For example, for the 12000 total words Heaps' law predicts:
 $3.71 \times 12000^{0.69} = \mathbf{2421}$ unique words
- The actual number of unique words in 12000 total words is **2340**, which is very close to the predicted result

- Another measure of the number of words in the language is the number of lemmas instead of wordform types.
(A lemma is a set of lexical forms having the same stem, the same major part-of-speech, and the same word sense)
- Dictionaries can help in giving lemma counts
- Dictionary entries or boldface forms are a very rough upper bound on the number of lemmas

Text Normalization

- Before almost any natural language processing of a text, the text has to be normalized.
- At least three tasks are commonly applied as part of any normalization process:
 1. Tokenizing (segmenting) words
 2. Normalizing word formats
 3. Segmenting sentences

Word Tokenization

- The task of segmenting running text into words
- Often need to break off punctuation as a separate token;
 - commas are a useful piece of information for parsers, periods help indicate sentence boundaries
- But often want to keep the punctuation that occurs word internally, in examples like m.p.h., Ph.D., AT&T, and cap'n.
- Special characters and numbers will need to be kept in prices (\$45.55) and dates (01/02/06);
 - don't want to segment that price into separate tokens of "45" and "55".
- And there are URLs (<http://www.stanford.edu>), Twitter hashtags (#nlproc), or email addresses (someone@cs.colorado.edu).

- A tokenizer can also be used to expand **clitic** contractions that are marked by apostrophes,
 - for example, converting **what're** to the two tokens **what are**
- A **clitic** is a part of a word that can't stand on its own, and can only occur when it is attached to another word
- Tokenization algorithms may also tokenize multiword expressions like **New York** or **rock 'n' roll** as a single token, which requires a multiword expression dictionary of some sort.
- Tokenization is thus intimately tied up with **named entity recognition**, the task of detecting names, dates, and organizations

- One commonly used tokenization standard is known as the Penn Treebank tokenization standard, used for the parsed corpora (treebanks) released by the Linguistic Data Consortium (LDC), the source of many useful datasets.
- This standard separates out clitics (**doesn't** becomes **does** plus **n't**), keeps hyphenated words together, and separates out all punctuation
- Tokenization needs to be run before any other language processing
- The standard method for tokenization is to use deterministic algorithms based on regular expressions compiled into very efficient finite state automata

- Example of a basic regular expression that can be used to tokenize with the **nltk.regexp tokenize function** of the Python-based Natural Language Toolkit (NLTK) (Bird et al. 2009; <http://www.nltk.org>).

```
>>> text = 'That U.S.A. poster-print costs $12.40...'
>>> pattern = r'''(?x)      # set flag to allow verbose regexps
...     ([A-Z]\.)+        # abbreviations, e.g. U.S.A.
...     | \w+(-\w+)*       # words with optional internal hyphens
...     | \$?\d+(\.\d+)?%?  # currency and percentages, e.g. $12.40, 82%
...     | \.\.\.            # ellipsis
...     | [][],;'"?O:-_-`' # these are separate tokens; includes ], [
...     ''
...
>>> nltk.regexp_tokenize(text, pattern)
['That', 'U.S.A.', 'poster-print', 'costs', '$12.40', '...']
```

- Deterministic algorithms can deal with the ambiguities that arise
 - Example: Apostrophe needs to be tokenized differently when used as a genitive marker (as in the **book's cover**), a quotative as in '**The other class'**, she said, or in clitics like **they're**.

Byte-Pair Encoding for Tokenization

- Instead of defining tokens as words (whether delimited by spaces or more complex algorithms), or as characters, use the data to automatically tell us what the tokens should be.
- Useful in dealing with unknown words, an important problem in language processing
- NLP algorithms often learn some facts about language from one corpus (**a training corpus**) and then use these facts to make decisions about a separate **test corpus** and its language.
 - Thus if our training corpus contains, say the words *low*, *new*, *newer*, but not *lower*, then if the word *lower* appears in our test corpus, our system will not know what to do with it

- Modern tokenizers often automatically induce sets of tokens that include tokens smaller than words, called **subwords**.
- **Subwords** can be arbitrary substrings, or they can be meaning-bearing units like the morphemes **-est** or **-er**. (Morphemes)
- In modern tokenization schemes, most tokens are words, but some tokens are frequently occurring morphemes or other subwords like **-er**.
- Every unseen words like ***lower*** can thus be represented by some sequence of known subword units, such as ***low*** and ***er***

- Most tokenization schemes have two parts:
 - a **token learner**, and a **token segmenter**.
- The ***token learner*** takes a raw *training corpus* (sometimes roughly preseparated into words, for example by whitespace) and induces a vocabulary, a set of tokens.
- The ***token segmenter*** takes a raw test sentence and segments it into the tokens in the vocabulary.
- Three algorithms are widely used:
 - **byte-pair encoding** (Sennrich et al., 2016),
 - **unigram language modeling** (Kudo, 2018), and
 - **WordPiece** (Schuster and Nakajima, 2012);
- There is also a **SentencePiece** library that includes implementations of the first two of the three

Byte-pair encoding Algorithm

Step 1: The BPE token learner begins with a vocabulary that is just the set of all individual characters.

Step 2 : It then examines the training corpus, chooses the two symbols that are most frequently adjacent (say 'A', 'B'), adds a new merged symbol 'AB' to the vocabulary, and replaces every adjacent 'A' 'B' in the corpus with the new 'AB'

Step 3: It continues to count and merge, creating new longer and longer character strings, until k merges have been done creating k novel tokens; k is thus a parameter of the algorithm.

The resulting vocabulary consists of the original set of characters plus k new symbols.

low low low low lowest lowest newer newer newer
newer newer newer wider wider wider new new'

Example: Input corpus of 18 word tokens with counts for each word (the word low appears 5 times, the word newer 6 times, and so on), which would have a starting vocabulary of 11 letters

corpus	vocabulary
5 low _	_ , d, e, i, l, n, o, r, s, t, w
2 lowest _	
6 newer _	
3 wider _	
2 new _	

- The most frequent is the pair e r because it occurs in newer (frequency of 6) and wider (frequency of 3) for a total of 9 occurrences.

- Treating **er** as one symbol, and counted again:

corpus	vocabulary
5 low_	_ , d, e, i, l, n, o, r, s, t, w, er
2 lowest_	
6 newer_	
3 wider_	
2 new_	

- Now the most frequent pair is **er_** , which we merge; system has learned that there should be a token for word-final **er**, represented as **er_** :

corpus	vocabulary
5 low_	_ , d, e, i, l, n, o, r, s, t, w, er, er_
2 lowest_	
6 newer_	
3 wider_	
2 new_	

- Next *n e* (total count of 8) get merged to *ne*:

corpus	vocabulary
5 low _	_d, e, i, l, n, o, r, s, t, w, er, er_, ne
2 lowest _	
6 newer_	
3 wider_	
2 new_	

- If continued, the next merges are:

Merge	Current Vocabulary
(ne, w)	_d, e, i, l, n, o, r, s, t, w, er, er_, ne, new
(l, o)	_d, e, i, l, n, o, r, s, t, w, er, er_, ne, new, lo
(lo, w)	_d, e, i, l, n, o, r, s, t, w, er, er_, ne, new, lo, low
(new, er_)	_d, e, i, l, n, o, r, s, t, w, er, er_, ne, new, lo, low, newer_
(low, _)	_d, e, i, l, n, o, r, s, t, w, er, er_, ne, new, lo, low, newer_, low_

- Once vocabulary is learned, the ***token parser*** is used to tokenize a test sentence. The token parser just runs on the test data the merges that has been learned from the training data, greedily,

- First segment each test sentence or word into characters.
- Then apply the first rule: replace every instance of **e r** in the test corpus with **er**
- Then apply the second rule: replace every instance of **er** in the test corpus with **er_**, and so on.
- By the end, if the test corpus contained the word **n e w e r_**, it would be tokenized as a full word.
- In real algorithms BPE is run with many thousands of merges on a very large input corpus.
- After training :
 - On the test data run each merge learned on training data.
 - Test set “n e w e r_” will be tokenized as a full word
 - Test set “l o w e r_” will be tokenized as “low er_”

Word Normalization, Lemmatization and Stemming

- Word normalization is the task of putting words/tokens in a standard format, choosing a single normal form for words with multiple forms like ***USA*** and ***US*** or ***uh-huh*** and ***uhhuh***.
- **Disadvantage:** the spelling information is lost in the normalization process. (but still useful in many applications)

Case folding

- Case folding is a kind of normalization.
- Mapping everything to lower case means that **Woodchuck** and **woodchuck** are represented identically, which is very helpful for generalization in many tasks, such as information retrieval or speech recognition.
- For sentiment analysis and other text classification tasks, information extraction, and machine translation, by contrast, case can be helpful and case folding is generally not done.
 - This is because maintaining the difference between, for example, **US** the country and **us** the pronoun can outweigh the advantage in generalization that case folding would have provided for other words.

Example text normalization

- Input: “The quick BROWN Fox Jumps OVER the lazy dog.”
- Output: “the quick brown fox jumps over the lazy dog.”

Advantages

- It eliminates case sensitivity, making text data consistent and easier to process.
- It reduces the dimensionality of the data, which can improve the performance of NLP algorithms.

Disadvantages

- It can lead to loss of information, as capitalization can indicate proper nouns or emphasis.

Punctuation Removal

- Punctuation removal is the process of removing special characters and punctuation marks from the text. This technique is useful when working with text data containing many punctuation marks, which can make the text harder to process.

Example text normalization

- Input: “The quick BROWN Fox Jumps OVER the lazy dog!!!”
- Output: “The quick BROWN Fox Jumps OVER the lazy dog”

Advantages

- It removes unnecessary characters, making the text cleaner and easier to process.
- It reduces the dimensionality of the data, which can improve the performance of NLP algorithms.

Disadvantages

- It can lead to loss of information, as punctuation marks can indicate sentiment or emphasis.

Stop Word Removal

- Stop word removal is the process of removing common words with little meaning, such as “the” and “a”. This technique is useful when working with text data containing many stop words, which can make the text harder to process.

Example text normalization

- Input: “The quick BROWN Fox Jumps OVER the lazy dog.”
- Output: “quick BROWN Fox Jumps OVER lazy dog.”

Advantages

- It removes unnecessary words, making the text cleaner and easier to process.
- It reduces the dimensionality of the data, which can improve the performance of NLP algorithms.

Disadvantages

- It can lead to loss of information, as stop words can indicate context or sentiment.

Removing numbers and symbol to normalize the text in NLP

- This technique is useful when working with text data that contain numbers and symbols that are not important for the NLP task.

Example text normalization

- Input: “I have 2 apples and 1 orange #fruits”
- Output: “I have apples and orange fruits”

Advantages

- It removes unnecessary numbers and symbols, making the text cleaner and easier to process.
- It reduces the dimensionality of the data, which can improve the performance of NLP algorithms.

Disadvantages

- It can lead to loss of information, as numbers and symbols can indicate quantities or sentiments.

Removing any remaining non-textual elements to normalize the text in NLP

- Removing any remaining non-textual elements such as HTML tags, URLs, and email addresses This technique is useful when working with text data that contains non-textual elements such as HTML tags, URLs, and email addresses that are not important for the NLP task.

Example text normalization

- Input: “Please visit example.com for more information or contact me at info@example.com”
- Output: “Please visit for more information or contact me at “

Advantages

- It removes unnecessary non-textual elements, making the text cleaner and easier to process.
- It reduces the dimensionality of the data, which can improve the performance of NLP algorithms.

Disadvantages

- It can lead to loss of information, as non-textual elements can indicate context or sentiment.

- Many natural language processing situations one may need two morphologically different forms of a word to behave similarly.
 - For example in web search, someone may type the string **woodchucks** but a useful system might want to also return pages that mention **woodchuck** with no **s**.

Lemmatization

- Lemmatization is the task of determining that two words have the same root, despite their surface differences.
- Lemmatization reduces words to their base form by considering the context in which they are used, such as “running” becoming “run”.
- The words **am**, **are**, and **is** have the shared lemma **be**; the words **dinner** and **dinners** both have the lemma **dinner**
- The most sophisticated methods for lemmatization involve complete morphological parsing of the word
 - Morphology is the study of the way words are built up from smaller meaning-bearing units called **morphemes**.
 - Two broad classes of morphemes can be distinguished: **stems**—the central morpheme of the word, supplying the main meaning—and **affixes**—adding “additional” meanings of various kinds.

Example text normalization

- Input: “running,runner,ran”
- Output: “run,runner,run”

Advantages

- It reduces the dimensionality of the data, which can improve the performance of NLP algorithms.
- It makes it easier to identify the core meaning of a word while preserving context.

Disadvantages

- It can be more computationally expensive.
- It may not be able to handle all words or forms.

Stemming

- Stemming is reduces the words to their root form by removing suffixes and prefixes, such as “running” becoming “run”. This method is helpful when working with text data that has many different versions of the same word, which can make the text harder to process.

Example text normalization

- Input: “running,runner,ran”
- Output: “run,run,run”

Advantages

- It reduces the dimensionality of the data, which can improve the performance of NLP algorithms.
- It makes it easier to identify the core meaning of a word.

Disadvantages

- It can lead to loss of information, as the root form of a word may not always be the correct form.
- It may produce non-existent words

Stemming	Lemmatization
<p>Stemming is a process that stems or removes few characters from a word, often leading to incorrect meanings and spelling.</p>	<p>Lemmatization considers the context and converts the word to its meaningful base form, which is called Lemma.</p>
<p>For instance, stemming the word 'Caring' would return 'Car'.</p>	<p>For instance, lemmatizing the word 'Caring' would return 'Care'.</p>
<p>Stemming is used in case of large dataset where performance is an issue.</p>	<p>Lemmatization is computationally expensive since it involves look-up tables and what not.</p>

The Porter Stemmer

- The naive version of morphological analysis is called stemming
- widely used stemming algorithms is the Porter (1980).

Basic Terminologies

- A consonant in a word is a letter other than A, E, I, O or U, and other than Y preceded by a consonant. (The fact that the term **consonant** is defined to some extent in terms of itself does not make it ambiguous.)
- Example: TOY the consonants are T and Y, and in SYZYGY they are S, Z and G. If a letter is not a consonant it is a vowel.
- The consonants are represented by ‘c’ and vowels are represented by ‘v’.

- A list ccc... of length greater than 0 will be denoted by C, and a list vvv... of length greater than 0 will be denoted by V.
- Any word, or part of a word, therefore has one of the four forms:
 - **CVCV ... C** → collection, management
 - **CVCV ... V** → conclude, revise
 - **VCVC ... C** → entertainment, illumination
 - **VCVC ... V** → illustrate, abundance
- These may all be represented by the single form

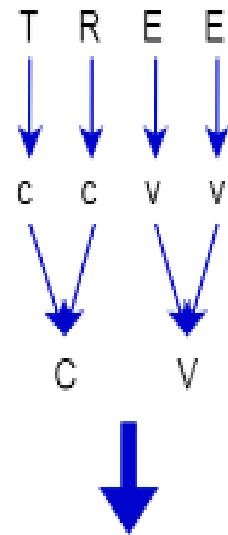
[C]VCVC ... [V]

where the square brackets denote arbitrary presence of their contents.

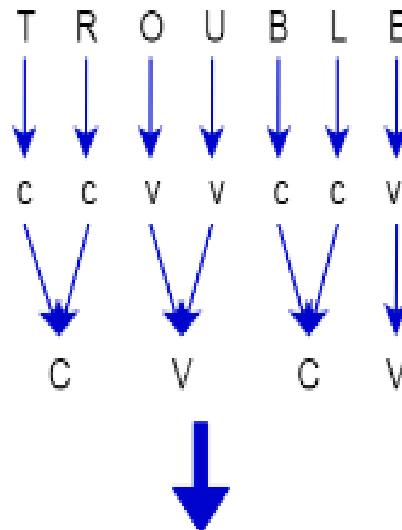
- Using (\$VC^m\$) to denote VC repeated m times, this may again be written as

[C](\$ VC^m \$)[V]

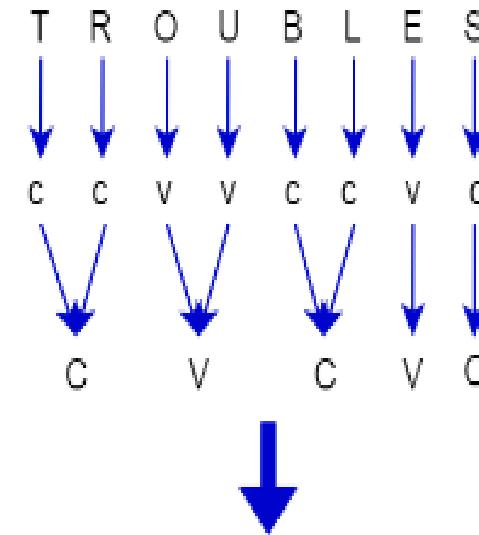
- m will be called the measure of any word or word part when represented in this form. The case m = 0 covers the null word.



$[C](VC)^m[V] \rightarrow m=0$



$[C](VC)^m[V] \rightarrow m=1$



$[C](VC)^m[V] \rightarrow m=2$

- Examples:

- $m=0$ TR, EE, TREE, Y, BY.
- $m=1$ TROUBLE, OATS, TREES, IVY.
- $m=2$ TROUBLES, PRIVATE, OATEN, ORRERY.

- The rules for removing a suffix will be given in the form

(condition) S1 -> S2

- This means that if a word ends with the suffix **S1**, and the stem before **S1** satisfies the given condition, **S1** is replaced by **S2**. The condition is usually given in terms of **m**,

Example: **(m > 1) EMENT ->**

Here **S1** is '**EMENT**' and **S2** is **null**. This would map **REPLACEMENT** to **REPLAC**, since **REPLAC** is a word part for which **m = 2**.

- The conditions may contain the following:
 - *S – the stem ends with S (and similarly for the other letters)
 - *v* – the stem contains a vowel
 - *d – the stem ends with a double consonant (e.g. -TT, -SS)
 - *o – the stem ends cvc, where the second c is not W, X or Y (e.g. -WIL, -HOP)
- And the condition part may also contain expressions with and, or and not, so that:

`(m>1 and (*S or *T))` : tests for a stem with m>1 ending in S or T, while

`(*d and not (*L or *S or *Z))` : tests for a stem ending with a double consonant other than L, S or Z.

- In a set of rules written beneath each other, only one is obeyed, and this will be the one with the longest matching **S1** for the given word.

- For example, with the following rules,

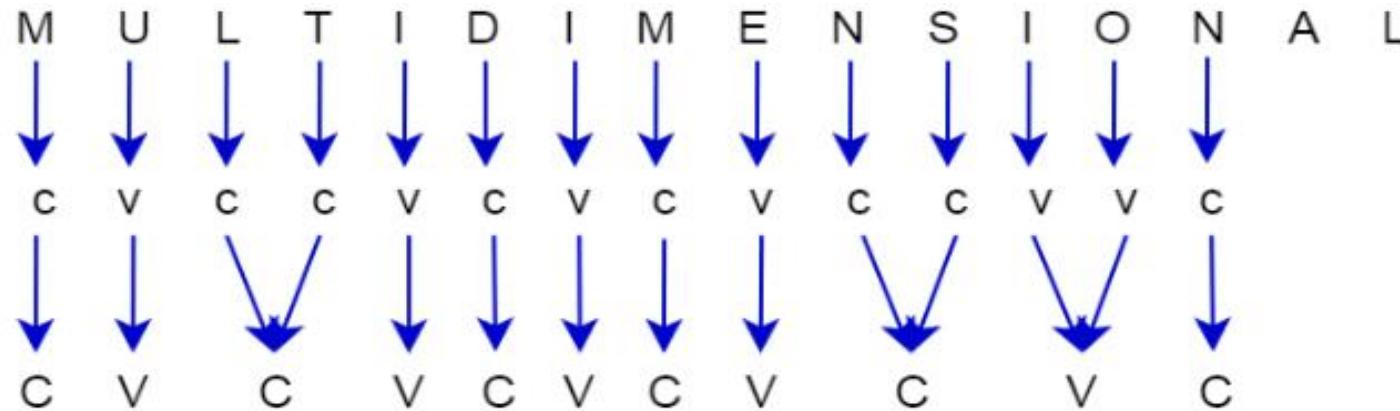
- i) SSES → SS
- ii) IES → I
- iii) SS → SS
- iv) S →

- the conditions are all null.
- CARESSES maps to CARESS since SSES is the longest match for S1.
- CARESS maps to CARESS (since S1="SS") and
- CARES to CARE (since S1="S").

- (m>0) EED -> EE (Example : feed -> feed ; agreed -> agree)
 - (v) ED -> (Example : plastered -> plaster ; bled -> bled)
 - (v) ING -> (Example : motoring -> motor ; sing -> sing)
 - S -> (Example : cats -> cat)
-
- T -> ATE (Example : conflat(ed) -> conflate)
 - BL -> BLE (Example : troubl(ed) -> trouble)
 - IZ -> IZE (Example : siz(ed) -> size)
-
- (m>0) ATIONAL -> ATE (Example : relational -> relate)
 - (m>0) TIONAL -> TION (Example : conditional -> condition ; rational -> rational)
 - (m>0) ENCI -> ENCE (Example : valenci -> valence)
 - Etc,....

Example 1:

Input word : **MULTIDIMENSIONAL**



C (VC) (VC) (VC) (VC) (VC)

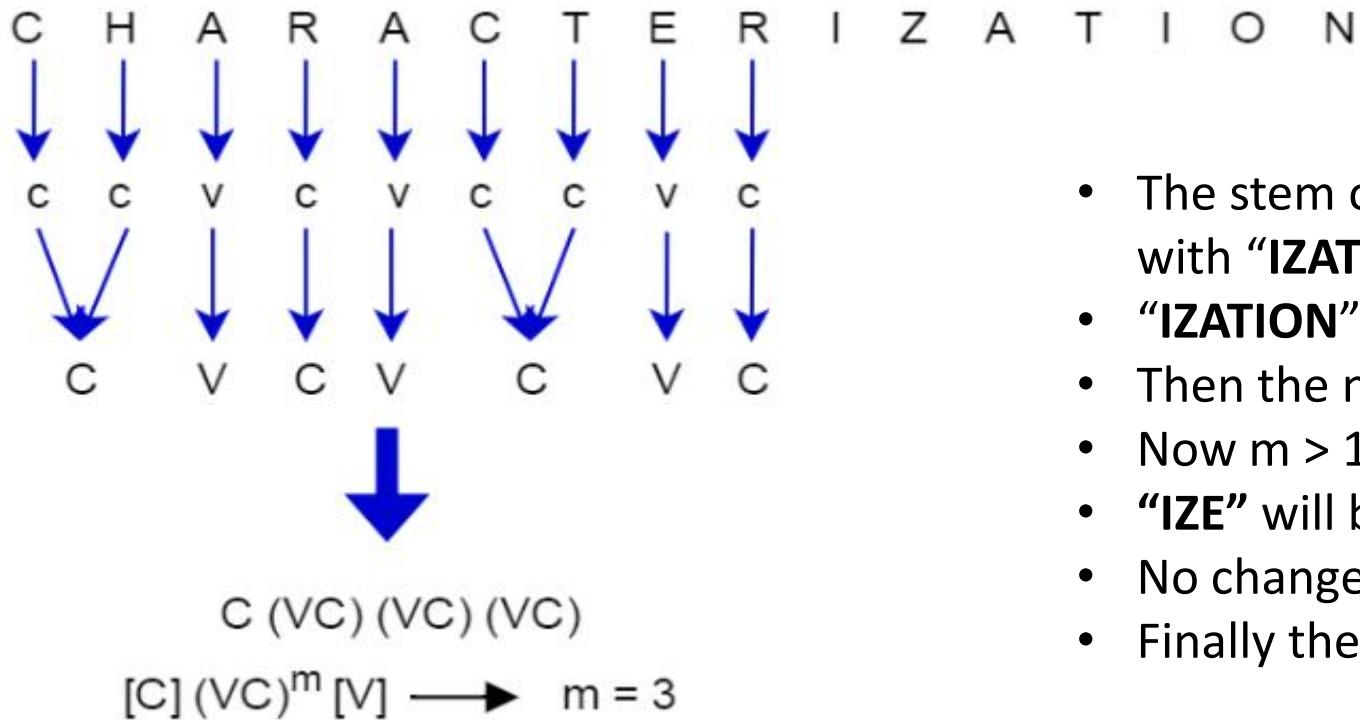
[C] (VC)^m [V] → m = 5

- The stem of the word has $m > 1$ (since $m = 5$) and ends with “AL”.
- “AL” is deleted (replaced with null).
- The word will not change the stem further.
- Finally the output will be **MULTIDIMENSION**.

(m>1) MULTIDIMENSIONAL → MULTIDIMENSION

Example 2

Input word : **CHARACTERIZATION**



- The stem of the word has $m > 0$ (since $m = 3$) and ends with “**IZATION**”.
- “**IZATION**” will be replaced with “**IZE**”.
- Then the new stem will be **CHARACTERIZE**.
- Now $m > 1$ (since $m = 3$) and the stem ends with “**IZE**”.
- “**IZE**” will be deleted (replaced with null).
- No change will happen to the stem in other steps.
- Finally the output will be **CHARACTER**.

CHARACTERIZATION → CHARACTERIZE → **CHARACTER**

Unit - 5

Probabilistic Models of Pronunciation and Spelling

Dealing with Spelling Error

- Typing is a manual process and not everyone does it correctly. Spelling mistakes occur frequently
- The detection and correction of spelling errors is an integral part of modern word-processors
- Even in applications like **Optical Character Recognition(OCR)** and **on-line handwriting recognition** the individual letters are not accurately identified
- **Optical Character Recognition(OCR)** – Automatic recognition of machine or hand-printed characters.
 - Optical scanner converts a machine or hand-printed page into a bitmap which is then passed to OCR
- **On-line handwriting recognition** – is the recognition of human printed or cursive handwriting as the user writing
 - Takes advantage of dynamic information of the input such as number and order of strokes and the speed and direction of each stroke

- Optical Character Recognition (OCR) is the process that converts an image of text into a machine-readable text format.
- OCR have higher error rates than human typists, although they tend to make different errors than typists.
 - Example: OCR systems often misread “D” as “O” or “ri” as “n” producing mis-spelled words like ***dension*** for ***derision*** or ***POQ Bach*** for ***PDQ Bach***
- There are two spelling tasks in such scenarios
 - Error Detection
 - Error Correction
- Spelling Error Detection: It is a process of detecting and sometimes providing suggestions for incorrectly spelled words in a text.
 - **Example:** Spell Checker is an application program that flags words in a document that may not be spelled correctly.

Spelling error correction

Spelling correction can be broken down into three broader problems (Kukich (1992)):

- **Non-word error detection:** Detecting spelling errors that result in non-words
Example: opportunity for opportunity , graffe for giraffe
- **Isolated-word error correction:** Check each word on its own for misspelling and correcting spelling errors that result in non-words
Example: correcting **graffe** to **giraffe**, **from** to **form**
- **Context-dependent error detection and correction (real-word error):** Using the context to help detect and correct spelling errors
- Some of these may accidentally result in an actual word
 - Typographical errors (insertion, deletion, transposition) which accidentally produce a real word **e.g. there for three**
 - Homophone or near-homophone **e.g. dessert for desert, or piece for peace**

Spelling Error Patterns

- The number and nature of spelling errors in human typed text differs from those caused by pattern-recognition devices like OCR and handwriting recognizers
- In an early study (Damerau, 1964), it is found that 80% of all misspelled words(non-word errors) were caused by **single-error misspellings**: a single one of the following errors:
 - **Insertion:** mistyping ***the*** as ***ther***
 - **Deletion:** mistyping ***the*** as ***th***
 - **Substitution:** mistyping ***the*** as ***thw***
 - **Transposition:** mistyping ***the*** as ***hte***
- Most of the research was focused on the correction of single-error misspellings

- Kukich in 1992, classified Human typing errors into two classes:
 - **Typographic Error** - Eg: Misspelling ***spell*** as ***speel***, generally related to the keyboard
 - **Cognitive Error** - Eg: Misspelling ***separate*** as ***seperate***, are caused by writers who don't know how to spell the word
- Typing errors are characterized as :
 - ***Substitutions, insertions ,deletions , transpositions***
- OCR errors are usually grouped into five classes:
 - ***Substitutions, multisubstitutions, space deletions or insertions , failures***

Detecting Non word errors

- Use a dictionary
- Use the models of morphology
- For other types of spelling correction, need a model of spelling variation.

Minimum Edit Distance

- Much of natural language processing is concerned with measuring how similar two strings are.
- For example in spelling correction, the user typed some erroneous string—let's say *graffe*—and we want to know what the user meant.
- Which is closest?
 - *graf*
 - *graft*
 - *grail*
 - *giraffe*

- Edit distance gives us a way to quantify both of these intuitions about string similarity.
- More formally, the ***minimum edit distance*** between two strings is defined as the minimum number of editing operations (operations like insertion, deletion, substitution) needed to transform one string into another.
 - ✓ **Substitution** – Change a single character from pattern s to a different character in text t, such as changing “shot” to “spot”.
 - ✓ **Insertion** – Insert a single character into pattern s to help it match text t, such as changing “ago” to “agog”.
 - ✓ **Deletion** – Delete a single character from pattern s to help it match text t, such as changing “hour” to “our”.
- For example the gap between ***intention*** and ***execution*** is five operations, which can be represented in three ways: as ***a trace***, ***an alignment*** or ***an operation list*** as shown below:
- Many important algorithms for finding string distance rely on some version of the minimum edit distance algorithm, named by Wagner and Fischer (1974).

Trace

i n t e n t i o n
e x e c u t i o n

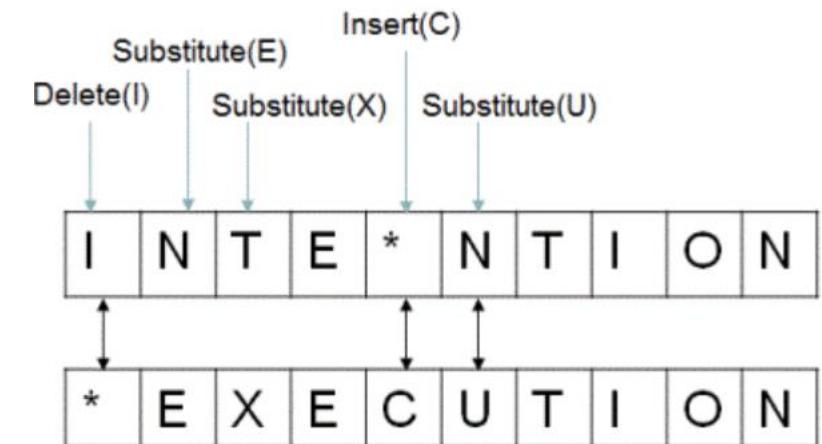
Alignment

i n t e n g t i o n
e x e c u t i o n

Operation List

delete i →	i n t e n t i o n
substitute n by e →	n t e n t i o n
substitute t by x →	e t e n t i o n
insert u →	e x e n t i o n
substitute n by c →	e x e n u t i o n
	e x e c u t i o n

- Representing the minimum edit distance between two strings as an alignment.
- The final row gives the operation list for converting the top string into the bottom string; **d** for deletion, **s** for substitution, **i** for insertion



- The gap between intention and execution, for example, is 5 (**delete an i, substitute e for n, substitute x for t, insert c, substitute u for n**)
- Given two sequences, an alignment is a correspondence between the substrings of the two sequences
- Thus, we say **I** aligns with the *empty string*, **N** with **E**, **T** with **X**, and so on
- Beneath the aligned strings is another representation; a series of symbols expressing **an operation list** for converting the top string into the bottom string: **d** for deletion, **s** for substitution, **i** for insertion

Cost Calculation-Minimum Edit Distance

- Cost or weight can be assigned to each of these operations
- The ***Levenshtein distance*** between two sequences is the simplest weighting factor in which each of the three operations has a cost of **1**
- Assume that the substitution of a letter for itself, for example, ***t*** for ***t***, has ***zero cost***
- ***The Levenshtein distance between intention and execution is 5***

- Levenshtein also proposed an alternative version of his metric in which each insertion or deletion has a cost of **1** and substitutions are not allowed.
- This is equivalent to allowing substitution, but giving each substitution a cost of **2** since any substitution can be represented by one insertion and one deletion)
- Using this version, ***the Levenshtein distance between intention and execution is 8.***
- Operations can be weighted by more complex functions like using confusion matrices (which assign a probability to each operation). Hence it becomes a ***maximum probability assignment*** instead of ***minimum edit distance***.

Defining Minimum Edit Distance

- Given two strings,
 - the source string X of length n , and
 - the target string Y of length m ,
- $D[i ; j]$ is defined as the edit distance between $X[1::i]$ and $Y[1::j]$,
 - i.e., the first i characters of X and the first j characters of Y .
- The edit distance between X and Y is thus $D[n;m]$.

Problem: Transform string $X[1\dots n]$ into $Y[1\dots m]$ by performing edit operations on string X .

Subproblem: Transform substring $X[1\dots i]$ into $Y[1\dots j]$ by performing edit operations on substring X .

Case 1: We have reached the end of either substring.

- If substring **X** is empty, insert all remaining characters of substring **Y** into **X**. The cost of this operation is equal to the number of characters left in substring **Y**.
- (' ', 'ABC') ——> ('ABC', 'ABC') (**cost = 3**)
- If substring **Y** is empty, delete all remaining characters of substring **X**. The cost of this operation is equal to the number of characters left in substring **X**.
- ('ABC', ' ') ——> (' ', ' ') (**cost = 3**)

Case 2: The last characters of substring X and Y are the same.

- If the last characters of substring X and substring Y match, nothing needs to be done – simply recur for the remaining substring **X[0...i-1], Y[0...j-1]**. As no edit operation is involved, the cost will be **0**.
- ('ACC', 'ABC') ——> ('AC', 'AB') (**cost = 0**)

Case 3: The last characters of substring X and Y are different.

- If the last characters of substring X and Y are different, return the minimum of the following operations:
- Insert the last character of Y into X. The size of Y reduces by 1, and X remains the same.
- This accounts for **X[1...i], Y[1...j-1]** as we move in on the target substring, but not in the source substring.
- ('ABA', 'ABC') ——> ('ABAC', 'ABC') == ('ABA', 'AB') (using case 2)

- **Delete** the last character of **X**. The size of **X** reduces by **1**, and **Y** remains the same.
 - This accounts for **X[1...i-1], Y[1...j]** as we move in on the source string, but not in the target string.
 - ('ABA', 'ABC') ——> ('AB', 'ABC')
-
- **Substitute (Replace)** the current character of **X** by the current character of **Y**. The size of both **X** and **Y** reduces by **1**. This accounts for **X[1...i-1], Y[1...j-1]** as we move in both the source and target string.
 - ('ABA', 'ABC') —> ('ABC', 'ABC') == ('AB', 'AB') (using case 2)
 - It is basically the same as case 2, where the last two characters match, and we move in both the source and target string, except it costs an edit operation.

Defining Minimum Edit Distance(Levenshtein)

- Initialization

$$D(i, 0) = i$$

$$D(0, j) = j$$

- Recurrence Relation:

For each $i = 1 \dots M$

For each $j = 1 \dots N$

$$D(i, j) = \min \begin{cases} D(i-1, j) + 1 \\ D(i, j-1) + 1 \\ D(i-1, j-1) + 2; & \text{if } X(i) \neq Y(j) \\ 0; & \text{if } X(i) = Y(j) \end{cases}$$

- Termination:

$D(N, M)$ is distance

Dynamic Programming for Minimum Edit Distance

- We will compute $D(n,m)$ **bottom up**, combining solutions to subproblems.
- Compute **base cases** first:
 - $D(i,0) = i$
 - a source substring of length i and an empty target string requires i deletes.
 - $D(0,j) = j$
 - a target substring of length j and an empty source string requires j inserts.
- Having computed $D(i,j)$ for small i, j we then compute larger $D(i,j)$ based on previously computed smaller values.
- The value of $D(i, j)$ is computed by taking the minimum of the three possible paths through the matrix which arrive there:

$$D[i, j] = \min \begin{cases} D[i - 1, j] + \text{del-cost}(\text{source}[i]) \\ D[i, j - 1] + \text{ins-cost}(\text{target}[j]) \\ D[i - 1, j - 1] + \text{sub-cost}(\text{source}[i], \text{target}[j]) \end{cases}$$

The Edit distance table

N	9									
O	8									
I	7									
T	6									
N	5									
E	4									
T	3									
N	2									
I	1									
#	0	1	2	3	4	5	6	7	8	9
	#	E	X	E	C	U	T	I	O	N

The Edit Distance Table

N	9										
O	8										
I	7										
T	6										
N	5										
E	4										
T	3										
N	2										
I	1										
#	0	1	2	3	4	5	6	7	8	9	
	#	E	X	E	C	U	T	I	O	N	

Edit Distance Table

N	9									
O	8									
I	7									
T	6									
N	5									
E	4									
T	3									
N	2									
I	1									
#	0	1	2	3	4	5	6	7	8	9
	#	E	X	E	C	U	T	I	O	N

$$D(i,j) = \min \begin{cases} D(i-1,j) + 1 \\ D(i,j-1) + 1 \\ D(i-1,j-1) + \begin{cases} 2; & \text{if } S_1(i) \neq S_2(j) \\ 0; & \text{if } S_1(i) = S_2(j) \end{cases} \end{cases}$$

The Final Edit Distance Table

$$D(i,j) = \min \begin{cases} D(i-1,j) + 1 \\ D(i,j-1) + 1 \\ D(i-1,j-1) + \begin{cases} 2; & \text{if } S_1(i) \neq S_2(j) \\ 0; & \text{if } S_1(i) = S_2(j) \end{cases} \end{cases}$$

N	9	8	9	10	11	12	11	10	9	8
O	8	7	8	9	10	11	10	9	8	9
I	7	6	7	8	9	10	9	8	9	10
T	6	5	6	7	8	9	8	9	10	11
N	5	4	5	6	7	8	9	10	11	10
E	4	3	4	5	6	7	8	9	10	9
T	3	4	5	6	7	8	7	8	9	8
N	2	3	4	5	6	7	8	7	8	7
I	1	2	3	4	5	6	7	6	7	8
#	0	1	2	3	4	5	6	7	8	9
	#	E	X	E	C	U	T	I	O	N

Backtrace for computing alignments

- Edit distance isn't sufficient
 - We often need to **align** each character of the two strings to each other
- We do this by keeping a “backtrace”
- Every time we enter a cell, remember where we came from
- When we reach the end,
 - Trace back the path from the upper right corner to read off the alignment

Backtrace for computing alignments

n	9	↓ 8	↙ ↖ 9	↙ ↖ 10	↙ ↖ 11	↙ ↖ 12	↓ 11	↓ 10	↓ 9	↙ 8	
o	8	↓ 7	↙ ↖ 8	↙ ↖ 9	↙ ↖ 10	↙ ↖ 11	↓ 10	↓ 9	↙ 8	← 9	
i	7	↓ 6	↙ ↖ 7	↙ ↖ 8	↙ ↖ 9	↙ ↖ 10	↓ 9	↙ 8	← 9	← 10	
t	6	↓ 5	↙ ↖ 6	↙ ↖ 7	↙ ↖ 8	↙ ↖ 9	↙ 8	← 9	← 10	← 11	
n	5	↓ 4	↙ ↖ 5	↙ ↖ 6	↙ ↖ 7	↙ ↖ 8	↙ ↖ 9	↙ 10	↙ ↖ 11	↙ 10	
e	4	↙ 3	← 4	↙ ↖ 5	← 6	← 7	← 8	↙ ↖ 9	↙ ↖ 10	↓ 9	
t	3	↙ ↖ 4	↙ ↖ 5	↙ ↖ 6	↙ ↖ 7	↙ ↖ 8	↙ 7	← 8	↙ ↖ 9	↓ 8	
n	2	↙ ↖ 3	↙ ↖ 4	↙ ↖ 5	↙ ↖ 6	↙ ↖ 7	↙ ↖ 8	↓ 7	↙ ↖ 8	↙ 7	
i	1	↙ ↖ 2	↙ ↖ 3	↙ ↖ 4	↙ ↖ 5	↙ ↖ 6	↙ ↖ 7	↙ 6	← 7	← 8	
#	0	1	2	3	4	5	6	7	8	9	
	#	e	x	e	c	u	t	i	o	n	

Backtrace for computing alignments

- Base conditions:

$$D(i, 0) = i \quad D(0, j) = j$$

Termination:

$D(N, M)$ is distance

- Recurrence Relation:

For each $i = 1 \dots M$

For each $j = 1 \dots N$

$$D(i, j) = \min \begin{cases} D(i-1, j) + 1 & \text{deletion} \\ D(i, j-1) + 1 & \text{insertion} \\ D(i-1, j-1) + 2; & \begin{cases} \text{if } X(i) \neq Y(j) & \text{substitution} \\ 0; & \text{if } X(i) = Y(j) \end{cases} \end{cases}$$

$$\text{ptr}(i, j) = \begin{cases} \text{LEFT} & \text{insertion} \\ \text{DOWN} & \text{deletion} \\ \text{DIAG} & \text{substitution} \end{cases}$$

Str1=abcdef
str2=azced

f	6	5	6	5	4	5
e	5	4	5	4	3	4
d	4	3	4	3	4	4
c	3	2	3	2	4	5
b	2	1	2	3	4	5
a	1	0	1	2	3	4
#	0	1	2	3	4	5
	#	a	z	c	e	d

```

function MIN-EDIT-DISTANCE(source, target) returns min-distance

    n  $\leftarrow$  LENGTH(source)
    m  $\leftarrow$  LENGTH(target)
    Create a distance matrix distance[n+1, m+1]

    # Initialization: the zeroth row and column is the distance from the empty string
    D[0,0] = 0
    for each row i from 1 to n do
        D[i,0]  $\leftarrow$  D[i-1,0] + del-cost(source[i])
    for each column j from 1 to m do
        D[0,j]  $\leftarrow$  D[0,j-1] + ins-cost(target[j])

    # Recurrence relation:
    for each row i from 1 to n do
        for each column j from 1 to m do
            D[i,j]  $\leftarrow$  MIN( D[i-1,j] + del-cost(source[i]),
                                D[i-1,j-1] + sub-cost(source[i], target[j]),
                                D[i,j-1] + ins-cost(target[j]) )

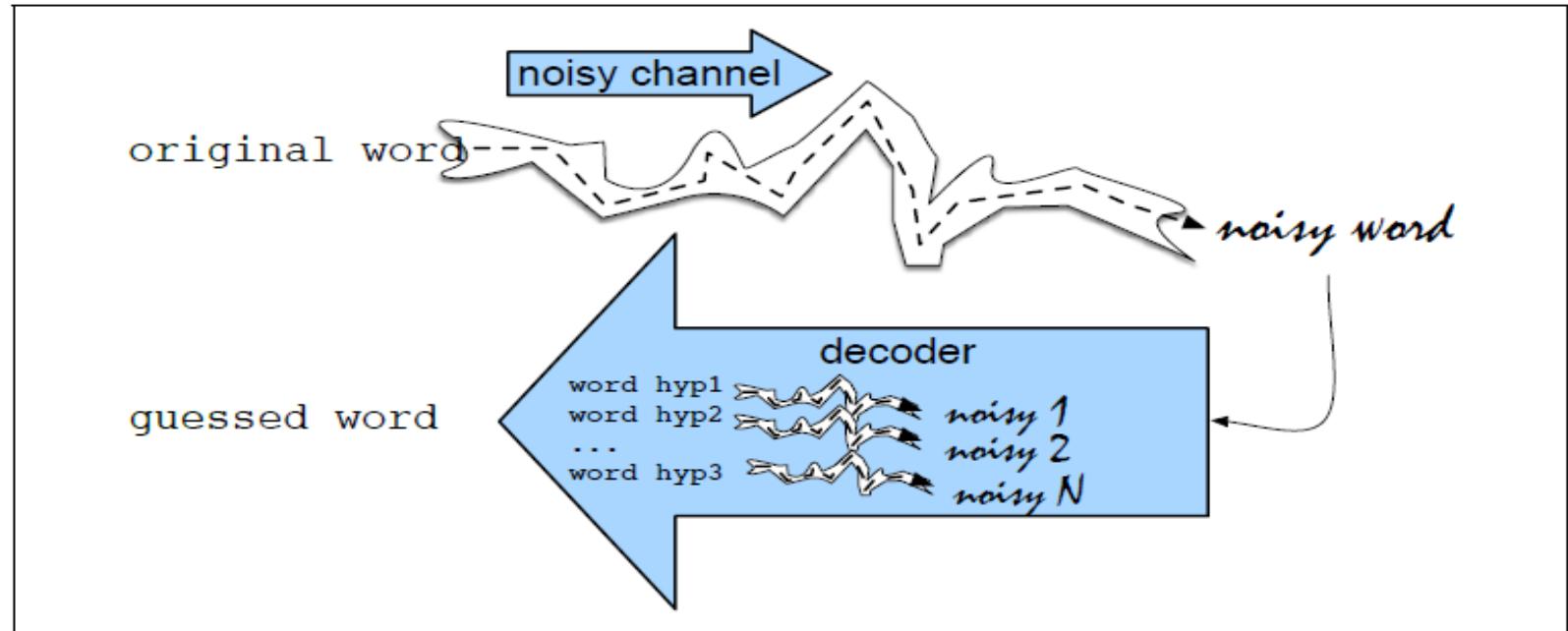
    # Termination
    return D[n,m]

```

Noisy Channel Model

- The **noisy channel model** is a framework that computers use to check spelling, question answering, recognize speech, and perform machine translation.
- It aims to determine the correct word if you type its misspelled version or mispronounce it.
- The noisy channel model can correct several typing mistakes, including missing letters (changing “leter” to “letter”), accidental letter additions (replacing “misstake” with “mistake”), swapped letters (changing “recieve” to “receive”), and replacing incorrect letters (replacing “fimite” with “finite”).
- The noisy channel model was applied to the spelling correction task by researchers at AT&T Bell Laboratories (Kernighan et al. 1990, Church and Gale 1991) and IBM Watson Research (Mays et al., 1991)

Figure : Noisy Channel Model



- In the noisy channel model, we imagine that the surface form we see is actually a “distorted” form of an original word passed through a noisy channel.
- Language is generated and passed through a noisy channel.
- Resulting noisy data are received
- Goal is to recover the original data from the noisy data.
- The **decoder passes each hypothesis** through a model of this channel and picks the word that best matches the surface noisy word.

- The intuition of the noisy channel model is to treat the misspelled word as, if a correctly spelled word had been “distorted” by being passed through a noisy communication channel.
- This channel introduces “noise” in the form of substitutions or other changes to the letters, making it hard to recognize the “true” word
- **Goal:** To build a model of the channel. Given this model, we then find the true word by passing every word of the language through the model of the noisy channel and seeing which one comes the closest to the misspelled word.

Application	Input	Output	$p(y)$	$p(x y)$
Machine Translation	L1 word sequences	L2 word sequences	$p(L1)$ in a language model	translation model
Optical Character Recognition (OCR)	actual text	text with OCR errors	prob of language text	model of OCR errors
Part of Speech (POS) tagging	POS tag sequences	English word sequence	prob of POS sequences	probability of word given tag
Speech Recognition	word sequences	acoustic speech signal	prob of word sequences	acoustic model
Document classification	class label	word sequence in document	class prior probability	$p(L1)$ from each class

Using Bayesian Model in Noisy Channel:

- This noisy channel model is a kind of **Bayesian inference**.
- We see an observation **x (a misspelled word)** and our job is to find the word **w** that generated this misspelled word
- Out of all possible words in the **vocabulary V** we want to find the word **w** such that **P(w|x)** is highest.
- We use the hat notation \hat{w} to mean “our estimate of the correct word”.

$$\hat{w} = \operatorname{argmax}_{w \in V} P(w | x) \quad 5.1$$

- The function **argmax_x f (x)** means “the **x** such that **f (x)** is maximized”.
- Equation 5.1 thus means, that out of all words in the vocabulary, we want the particular word that maximizes the right-hand side **P(w|x)**.

- The channel has a probabilistic interpretation, whereby you assume that the original pristine version has some ***probability (prior)***, and the addition of specific noise has some ***probability (conditional)***.
- These probabilities can be used to find the most likely original version given the actually observed signal.

Bayesian Classification

- The intuition of Bayesian classification is to use Bayes' rule to transform Eq. 5.1 into a set of other probabilities
- Bayes' rule is presented in Eq. 5.2; it gives us a way- to break down any conditional probability $P(a|b)$ into three other probabilities:

$$P(a|b) = \frac{P(b|a)P(a)}{P(b)} \quad (5.2)$$

- Substitute Eq. 5.2 into Eq. 5.1 to get Eq. 5.3:

$$\hat{w} = \operatorname{argmax}_{w \in V} \frac{P(x|w)P(w)}{P(x)} \quad (5.3)$$

- We can conveniently simplify Eq. 5.3 by dropping the denominator $P(x)$
- Since we are choosing a potential correction word out of all words, we need to compute for each word.

$$\frac{P(x|w)P(w)}{P(x)}$$

- But $P(x)$ doesn't change for each word; we always ask about the most likely word for the same observed error x , which must have the same probability $P(x)$.
- Thus, we can choose the word that maximizes this simpler formula

$$\hat{w} = \operatorname{argmax}_{w \in V} P(x|w)P(w) \quad (5.4)$$

- To summarize, the noisy channel model says that we have some true underlying word w , and we have a noisy channel that modifies the word into some possible likelihood of misspelled observed surface form.
- The likelihood (or channel model) of the noisy channel model producing any particular observation sequence x is modeled by $P(x|w)$
- The prior probability of a hidden word is modeled by $P(w)$
- We can compute the most probable word given that we've seen some observed misspelling x by multiplying the prior probability $P(w)$ and the likelihood $P(x|w)$ and choosing the word for which this product is greatest.

- We apply the noisy channel approach to correct non-word spelling errors by taking any word not in our spell dictionary, generating a list of candidate words, ranking them according to Eq. 5.4, and picking the highest-ranked one.
- We can modify Eq. 5.4 to refer to this list of candidate words instead of the full vocabulary V as follows:

$$\hat{w} = \operatorname{argmax}_{w \in C} \underbrace{P(x|w)}_{\text{channel model}} \underbrace{P(w)}_{\text{prior}} \quad (5.5)$$

Non--word spelling error example

- Two stages of Algorithm are:
 - Propose candidate corrections by finding words that have a similar spelling to the input word.
 - Score the candidates
- The **first stage of the algorithm proposes candidate corrections** by finding words that have a similar spelling to the input word.
- Analysis of spelling error data has shown that the majority of spelling errors consist of a ***single-letter change***. Hence, make the simplifying assumption that these candidates have an edit distance of 1 from the error word
- To find this list of candidates minimum edit distance algorithm is used. It is extended so that in addition to insertions, deletions, and substitutions, we'll add a fourth type of edit, **transpositions**, in which two letters are swapped
- The version of edit distance with transposition is called **Damerau-Levenshtein edit distance**.

- Applying all such single transformations to *acress* yields the list of candidate words:

Error	Correction	Correct Letter	Error Letter	Transformation Position (Letter #)	Type
acress	actress	t	—	2	deletion
acress	cress	—	a	0	insertion
acress	caress	ca	ac	0	transposition
acress	access	c	r	2	substitution
acress	across	o	e	3	substitution
acress	acres	—	2	5	insertion
acress	acres	—	2	4	insertion

Figure 5.24 Candidate corrections for the misspelling *acress*, together with the transformations that would have produced the error (after Kernighan et al. (1990)). “—” represents a null letter.

- The second stage of the algorithm is to **Score the candidates.**
- Once we have a set of candidates, score each one using Eq. 5.5 requires that we compute the **prior and the channel model**
- The prior probability of each correction $P(w)$ is the language model probability of the word w in context, which can be computed using any language model.
- So the probability of a particular correction word w is computed by dividing the count of w by the number N of words in the corpus

$$P(w) = \text{Count}(w)/N$$
- For this example let's start in the following table by assuming a unigram language model
- Let's use the corpus of Kernighan et al. (1990), which is the 1988 AP newswire corpus of 44 million words. Since in this corpus the word actress occurs 1343 times out of 44 million, the word acres 2879 times, and so on, the resulting unigram prior probabilities are:

c	freq(c)	p(c)
actress	1343	.0000315
cress	0	.000000014
caress	4	.0000001
access	2280	.000058
across	8436	.00019
acres	2879	.000065

Estimate the likelihood $P(x|w)$

- A simple model might estimate, for example, $p(\text{acress}|\text{across})$ just using the number of times that the letter **e** was substituted for the letter **o** in some large corpus of errors.
- To compute the probability for each edit in this way we'll need a confusion matrix that contains counts of errors.
- In general, a confusion matrix lists the matrix number of times one thing was confused with another.
- Thus for example a confusion matrix will be a square matrix of size **26x26** (or more generally $|A| \times |A|$, for an alphabet **A**) that represents the number of times one letter was incorrectly used instead of another.
- The cell labeled **[t, s]** in an insertion confusion matrix would give the count of times that **t** was inserted after **s**.
- Four confusion matrices will be used:

`del[x,y]: count(xy typed as x)`

`ins[x,y]: count(x typed as xy)`

`sub[x,y]: count(x typed as y)`

`trans[x,y]: count(xy typed as yx)`

- Four confusion matrix will be used:

$\text{del}[x,y]$: $\text{count}(xy \text{ typed as } x)$

$\text{ins}[x,y]$: $\text{count}(x \text{ typed as } xy)$

$\text{sub}[x,y]$: $\text{count}(x \text{ typed as } y)$

$\text{trans}[x,y]$: $\text{count}(xy \text{ typed as } yx)$

- $\text{del}[x,y]$ contains the number of times in the training set that the characters xy in the correct word were typed as x .
- $\text{ins}[x,y]$ contains the number of times in the training set that the character x in the correct word was typed as xy .
- $\text{sub}[x,y]$ the number of times that x was typed as y .
- $\text{trans}[x,y]$ the number of times that xy was typed as yx .

- Once we have the confusion matrices, we can estimate $P(x|w)$ as follows where (w_i is the i th character of the correct word w) and x_i is the i th character of the typo x :

$$P(x|w) = \begin{cases} \frac{\text{del}[x_{i-1}, w_i]}{\text{count}[x_{i-1}w_i]}, & \text{if deletion} \\ \frac{\text{ins}[x_{i-1}, w_i]}{\text{count}[w_{i-1}]}, & \text{if insertion} \\ \frac{\text{sub}[x_i, w_i]}{\text{count}[w_i]}, & \text{if substitution} \\ \frac{\text{trans}[w_i, w_{i+1}]}{\text{count}[w_iw_{i+1}]}, & \text{if transposition} \end{cases} \quad (5.6)$$

c	freq(c)	p(c)	p(t c)	p(t c)p(c)	%
actress	1343	.0000315	.000117	3.69×10^{-9}	37%
cress	0	.000000014	.00000144	2.02×10^{-14}	0%
caress	4	.0000001	.00000164	1.64×10^{-13}	0%
access	2280	.000058	.000000209	1.21×10^{-11}	0%
across	8436	.00019	.0000093	1.77×10^{-9}	18%
acres	2879	.000065	.0000321	2.09×10^{-9}	21%
acres	2879	.000065	.0000342	2.22×10^{-9}	23%

Figure 5.25 Computation of the ranking for each candidate correction. Note that the highest ranked word is not *actress* but *acres* (the two lines at the bottom of the table), since *acres* can be generated in two ways. The *del[]*, *ins[]*, *sub[]*, and *trans[]* confusion matrices are given in full in Kernighan et al. (1990).

- This implementation of the Bayesian algorithm predicts *acres* as the correct word (at a total normalized percentage of 44%), and **actress** as the second most likely word.
- Unfortunately, the algorithm was wrong in a text string here: The writer's intention becomes clear from the context: *...was called a “stellar and versatile across whose combination of sass and glamour has defined her...”*. The surrounding words make it clear that *actress* and not *acres* was the intended word.
- This is the reason that in practice we use trigram (or larger) language models in the noisy channel model, rather than unigrams.

END

Unit 6

NGRAMS

N-Grams and Language models

- The simplest model that assigns probabilities to sentences and sequences of words, is known as **N-gram model**. It predicts the next word from the previous **N-1** words. (word prediction)
- Models that assign probabilities to sequences of words are called **language Models or LMs**.
- Computing the probability of the next word will turn out to be closely related to computing the probability of a sequence of words.
- The following sequence, for example, has a **non-zero probability** of appearing in a text:
'... all of a sudden I notice three guys standing on the sidewalk...'
while this same set of words in a different order has a very low probability:
'on guys all I of notice sidewalk three a sudden standing the'

- N-grams that assign a conditional probability to possible next words can be used to assign a joint probability to an entire sentence.
 - N-gram model is one of the most important tools in speech and language processing.
-
- N-grams are essential in any task in which we have to identify words in noisy, ambiguous input.
 - In speech recognition, for example, the input speech sounds are very confusable and many words sound extremely similar.

Probabilistic Language Model-Applications

- Context Sensitive Spelling Correction

Example: ***The office is about 15 minuets from my house.***

Minuets means ***slow ballroom dance***

$P(\text{minutes from my house}) > P(\text{minuets from my house})$

- Natural language Generation

Whenever you have to generate sentences again, look into which particular generation has the higher probability?

- Speech Recognition

$P(\text{I saw a man}) >> P(\text{eyes awe of an})$

Application of N-Grams

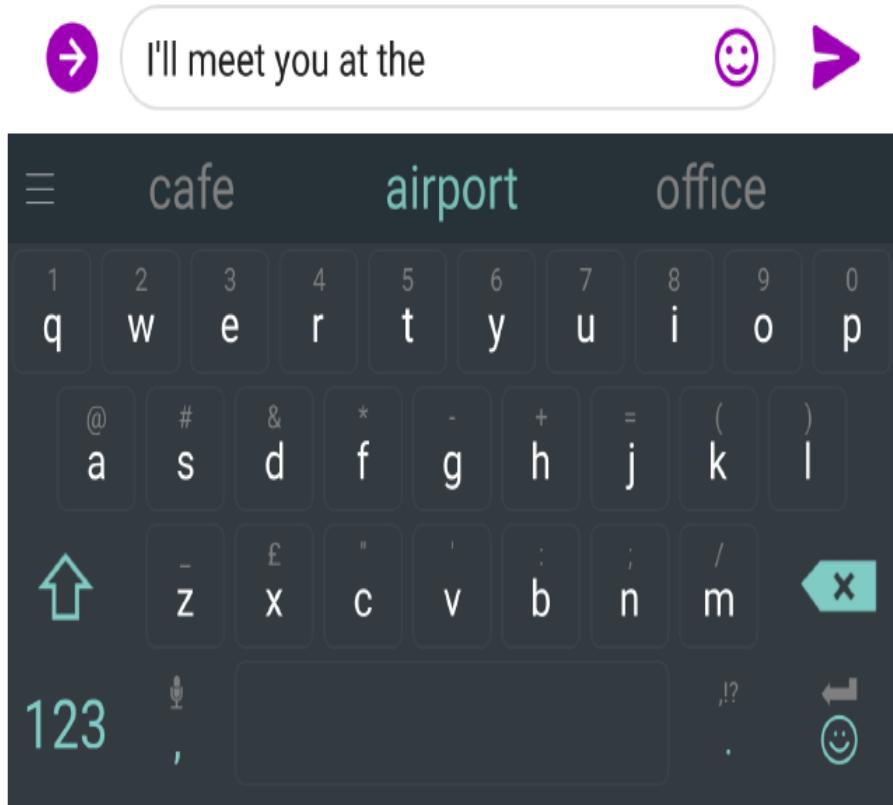
- **Augmentative communication-** (Newell et al., Communication 1998) systems that help the disabled.
- People who are unable to use speech or sign language to communicate, like the physicist Steven Hawking, can communicate by using simple body movements to select words from a menu that are spoken by the system.
- Word prediction can be used to suggest ***likely words for the menu.***
- Besides these sample areas, *N*-grams are also crucial in NLP tasks like **part-of-speech tagging, natural language generation, and word similarity**, as well as in applications from **authorship identification and sentiment extraction** to **predictive text input** systems for cell phones.

Probabilistic Language Model-Applications

- **Machine Translation:** The process of using artificial intelligence to automatically translate text from one language to another without human involvement.
- Modern machine translation goes beyond simple word-to-word translation to communicate the full meaning of the original language text in the target language.
- It analyzes all text elements and recognizes how the words influence one another.
 - -problem of collocations
Example: $P(\text{high winds}) > P(\text{large winds})$
 $P(\text{I went to the movie}) > P(\text{I flew to the movie})$

- Completion Prediction
 - Language model also predicts the completion of a sentence
 - Please turn off your cell
 - Your program does not.....
 - Predictive text input systems can guess what you are typing and give choices on how to complete it

Language Models Everyday



what is the |

what is the **weather**
what is the **meaning of life**
what is the **dark web**
what is the **xfl**
what is the **doomsday clock**
what is the **weather today**
what is the **keto diet**
what is the **american dream**
what is the **speed of light**
what is the **bill of rights**

Google Search I'm Feeling Lucky

N-GRAMS

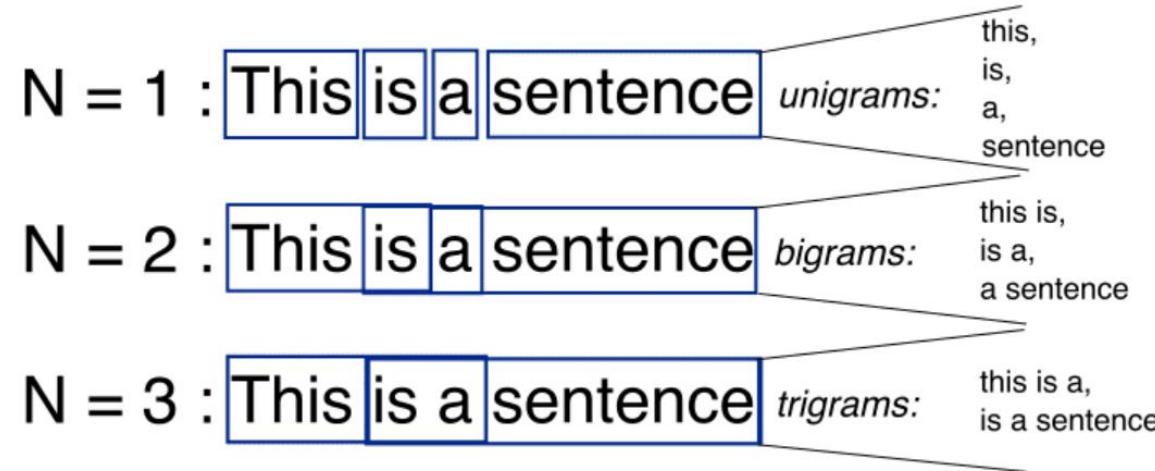


Figure 2 Uni-gram, Bi-gram, and Tri-gram Model

- An N-gram is a sequence of N words: **For eg: *Please turn your homework***
- A 2-gram (or bigram) is a two-word sequence of words like “please turn”, “turn your”, or “your homework”
- A 3-gram (or trigram) is a three-word sequence of words like “please turn your”, or “turn your homework”.

N-GRAMS: Counting Words in Corpus

- Probabilities are based on counting things.
- Counting of things in natural language is based on a corpus (plural corpora), an online collection of text CORPUS or speech.
- Two popular corpora:
 - **Brown**
 - **Switchboard**
- The **Brown corpus** is a 1-million-word collection of samples from 500 written texts from different genres (newspaper, novels, non-fiction, academic, etc.), assembled at Brown University in 1963-64.

How many words are in the following Brown sentence?

Example: He stepped out into the hall, was delighted to encounter a water brother.

- Has 13 words if we don't count punctuation marks as words, 15 if we count punctuation. Whether we treat period ("."), comma (","), and so on as words depends on the task

N-GRAMS-Counting Words in Corpus

- The **Switchboard corpus** of telephone conversations between strangers was collected about 3 million words in the early 1990s.
- Such corpora of spoken language don't have punctuation, but do introduce other complications with regard to defining words called **utterance** i.e., the spoken correlate of a sentence:

Example: I do uh main- mainly business data processing

- This **utterance** has two kinds of **disfluencies**.
 - The broken-off word main- is called a **fragment**.
 - Words like uh and um are called **fillers** or **filled pauses**.

SIMPLE (UNSMOOTHED) N-GRAMS

- **Goal:** To compute the probability of a **word w** given some **history h**, or $P(w|h)$.
- Suppose the history h is “*its water is so transparent that*” and we want to know the probability that the next word is *the*: $P(\text{the}|\text{its water is so transparent that})$.
- One way is to estimate it from relative frequency counts.

How to estimate these probabilities

- One way is to estimate it from **relative frequency counts**.
- For example, we could take a very large corpus, count the number of times we see '**its water is so transparent that**', and count the number of times this is followed by **the**.
- This would be answering the question "Out of the times we saw the history h , how many times was it followed by the word w ",

$$P(\text{the} | \text{its water is so transparent that}). = \frac{C(\text{its water is so transparent that the})}{C(\text{its water is so transparent that})}$$

Chain Rule of Probability

Disadvantage of Relative frequency count:

- This method of estimating probabilities depends upon **word counts** that work fine in many cases, but it won't be enough to give good estimates in most cases.
 - This is because language is creative; new sentences are created all the time, and we won't always be able to count entire sentences.
 - Even simple extensions of the example sentence may have counts of zero on the web (such as "***Walden Pond's water is so transparent that the***").
- The **joint probability** of an entire sequence of words needs lots of estimation.
- **Solution: Chain Rule of Probability**

Chain Rule of Probability

Let a sequence of **N** words either as $w_1 \dots w_n$ or w_1^n .

For the joint probability of each word in a sequence having a particular value $P(X = w_1; Y = w_2; Z = w_3; \dots; W = w_n)$ we'll use $P(w_1; w_2; \dots; w_n)$.

$$\begin{aligned} P(X_1 \dots X_n) &= P(X_1)P(X_2|X_1)P(X_3|X_1^2) \dots P(X_n|X_1^{n-1}) \\ &= \prod_{k=1}^n P(X_k|X_1^{k-1}) \end{aligned}$$

Applying the chain rule to words, we get

$$\begin{aligned} P(w_1^n) &= P(w_1)P(w_2|w_1)P(w_3|w_1^2) \dots P(w_n|w_1^{n-1}) \\ &= \prod_{k=1}^n P(w_k|w_1^{k-1}) \end{aligned}$$

The Chain Rule

The Chain Rule applied to compute joint probability of words in sentence

$$P(w_1 w_2 \dots w_n) = \prod_i P(w_i | w_1 w_2 \dots w_{i-1})$$

$$\begin{aligned} P(\text{"its water is so transparent"}) &= \\ P(\text{its}) \times P(\text{water} | \text{its}) \times P(\text{is} | \text{its water}) \\ \times P(\text{so} | \text{its water is}) \times P(\text{transparent} | \text{its water is so}) \end{aligned}$$

- The chain rule shows the link between computing the joint probability of a sequence and computing the conditional probability of a word given previous words.,
- But using the chain rule doesn't really seem to help us! We don't know any way to compute the exact probability of a word given a long sequence of preceding words,
- The intuition of the N-gram model is that instead of computing the probability of a word given its entire history, we will approximate the history by just the last few words.

The **bigram** model, for example, approximates the probability of a word given all the previous words $P(w_n|w_1^{n-1})$ by using only the conditional probability of the preceding word $P(w_n|w_{n-1})$. In other words, instead of computing the probability

$$P(\text{the}|\text{Walden Pond's water is so transparent that})$$

we approximate it with the probability

$$P(\text{the}| \text{that})$$

When we use a bigram model to predict the conditional probability of the next word we are thus making the following approximation:

$$P(w_n|w_1^{n-1}) \approx P(w_n|w_{n-1})$$

Markov Assumption

- Markov Models are the class of probabilistic models that assume that we can predict the probability of some future unit without looking too far into the past

Markov Assumption

- Simplifying assumption:



Andrei Markov (1856~1922)

$$P(\text{the} \mid \text{its water is so transparent that}) \approx P(\text{the} \mid \text{that})$$

- Or maybe

$$P(\text{the} \mid \text{its water is so transparent that}) \approx P(\text{the} \mid \text{transparent that})$$

Markov Assumption

$$P(w_1 w_2 \dots w_n) \approx \prod_i P(w_i | w_{i-k} \dots w_{i-1})$$

- In other words, we approximate each component in the product

$$P(w_i | w_1 w_2 \dots w_{i-1}) \approx P(w_i | w_{i-k} \dots w_{i-1})$$

Only previous k words are considered

Thus the general equation for this N -gram approximation to the conditional probability of the next word in a sequence is:

$$P(w_n | w_1^{n-1}) \approx P(w_n | w_{n-N+1}^{n-1})$$

Given the bigram assumption for the probability of an individual word, we can compute the probability of a complete word sequence by

$$P(w_1^n) \approx \prod_{k=1}^n P(w_k | w_{k-1})$$

Markov Assumption

- $P(\text{the} \mid \text{its water is so transparent that})$
- An N-gram model uses only N-1 words of prior context
 - Unigram: $P(\text{the})$
 - Bigram: $P(\text{the} \mid \text{that})$
 - Trigram: $P(\text{the} \mid \text{transparent that})$

N-Gram Model

- We can extend to trigrams, 4-grams, 5-grams
- In general this is an insufficient model of language
 - because language has **long-distance dependencies**:

“The computer(s) which I had just put into the machine room on the fifth floor is (are) crashing.”

- But we can often get away with N-gram models

Estimating Bigram Probabilities

- An intuitive way to estimate probabilities is called **maximum likelihood estimation or MLE**
- We get maximum likelihood estimation the MLE estimate for the parameters of an **N-gram model by getting counts from a corpus, and normalizing the counts so that they lie between 0 and 1**
- For example, to compute a particular bigram probability of a word y given a previous word x , we'll compute the count of the bigram $C(xy)$ and normalize by the sum of all the bigrams that share the same first word x

Estimating Bigram Probabilities

- The Maximum Likelihood Estimate (MLE)
 - relative frequency based on the empirical counts on a training set

$$P(w_i | w_{i-1}) = \frac{\text{count}(w_{i-1}, w_i)}{\text{count}(w_{i-1})}$$

$$P(w_i | w_{i-1}) = \frac{c(w_{i-1}, w_i)}{c(w_{i-1})}$$

c – count

Estimating Bigram Probabilities

An example

$$P(w_i | w_{i-1}) \stackrel{\text{MLE}}{=} \frac{c(w_{i-1}, w_i)}{c(w_{i-1})}$$

< s > I am Sam < /s >
< s > Sam I am < /s >
< s > I do not like green eggs and ham < /s >

$$P(I | < s >) = \frac{2}{3} = .67$$

$$P(< /s > | Sam) = \frac{1}{2} = 0.5$$

$$P(Sam | < s >) = \frac{1}{3} = .33$$

$$P(Sam | am) = \frac{1}{2} = .5$$

$$P(am | I) = \frac{2}{3} = .67$$

$$P(do | I) = \frac{1}{3} = .33$$

Estimating Bigram Probabilities

	<s>	I	AM	SAM	</s>
<s>	0	0.67	0	0	0
I	0	0	0.67	0	0
AM	0	0	0	0.5	0
SAM	0	0.5	0	0	0.5
</s>	0	0	0	0	0

$$\begin{aligned} P(<\text{s}> | \text{AM SAM } <\text{s}>) &= P(\text{I} | <\text{s}>) * P(\text{AM} | \text{I}) * P(\text{SAM} | \text{AM}) * P(<\text{/s}> | \text{SAM}) \\ &= 0.67 * 0.67 * 0.5 * 0.5 = 0.1122 \end{aligned}$$

Example:

- Unigram Model

$$P(w_1, w_2, \dots, w_n) = P(w_1) * P(w_2) * P(w_3) * \dots * P(w_{n-1})$$

Example:

$$\begin{aligned} & P(\text{the}, \text{prime}, \text{minister}, \text{of}, \text{our}, \text{country}) \\ &= P(\text{the}) * P(\text{prime}) * P(\text{minister}) * P(\text{of}) * P(\text{our}) \\ &\quad * P(\text{country}) \end{aligned}$$

- Bigram Model:

$$P(w_1, w_2, \dots, w_n) = P(w_1) * P(w_2 | w_1) * P(w_3 | w_2) * \dots * P(w_n | w_{n-1})$$

Example:

$$P(\text{the}, \text{prime}, \text{minister}, \text{of}, \text{our}, \text{country})$$

$$= P(\text{the}) * P(\text{prime} | \text{the}) * P(\text{minister} | \text{prime})$$

$$* P(\text{of} | \text{minister}) * P(\text{our} | \text{of}) * P(\text{country} | \text{our})$$

- Trigram Model

$$P(w_1, w_2, \dots, w_n)$$

$$\begin{aligned} &= P(w_1) * P(w_2 | w_1) * P(w_3 | w_1, w_2) * P(w_4 | w_2, w_3) * \dots \\ &\quad * P(w_n | w_{n-2}, w_{n-1}) \end{aligned}$$

Example:

$$P(\text{the}, \text{prime}, \text{minister}, \text{of}, \text{our}, \text{country})$$

$$\begin{aligned} &= P(\text{the}) * P(\text{prime} | \text{the}) * P(\text{minister} | \text{the}, \text{prime}) \\ &\quad * P(\text{of} | \text{prime}, \text{minister}) * P(\text{our} | \text{minister}, \text{of}) \\ &\quad * P(\text{country} | \text{of}, \text{our}) \end{aligned}$$

A bigger example: Berkeley Restaurant Project sentences

- Berkeley Restaurant Project, a dialogue system from the last century that answered questions about a database of restaurants in Berkeley, California (Jurafsky et al., 1994). X
- Here are some sample user queries, lowercased and with no punctuation (a representative corpus of 9332 sentences is on the website):
- can you tell me about any good cantonese restaurants close by
- mid priced thai food is what i'm looking for
- tell me about chez panisse
- can you give me a listing of the kinds of food that are available
- i'm looking for a good place to eat breakfast

A bigger example: Berkeley Restaurant Project sentences

	i	want	to	eat	chinese	food	lunch	spend
i	5	827	0	9	0	0	0	2
want	2	0	608	1	6	6	5	1
to	2	0	4	686	2	0	6	211
eat	0	0	2	0	16	2	42	0
chinese	1	0	0	0	0	82	1	0
food	15	0	15	0	1	4	0	0
lunch	2	0	0	0	0	1	0	0
spend	1	0	1	0	0	0	0	0

Figure 4.1 Bigram counts for eight of the words (out of $V = 1446$) in the Berkeley Restaurant Project corpus of 9332 sentences.

A bigger example: Berkeley Restaurant Project sentences

Raw bigram probabilities $P(w_i | w_{i-1}) = \frac{c(w_{i-1}, w_i)}{c(w_{i-1})}$

- Normalize by unigrams:

	i	want	to	eat	chinese	food	lunch	spend
	2533	927	2417	746	158	1093	341	278

- Result:

	i	want	to	eat	chinese	food	lunch	spend
i	0.002	0.33	0	0.0036	0	0	0	0.00079
want	0.0022	0	0.66	0.0011	0.0065	0.0065	0.0054	0.0011
to	0.00083	0	0.0017	0.28	0.00083	0	0.0025	0.087
eat	0	0	0.0027	0	0.021	0.0027	0.056	0
chinese	0.0063	0	0	0	0	0.52	0.0063	0
food	0.014	0	0.014	0	0.00092	0.0037	0	0
lunch	0.0059	0	0	0	0	0.0029	0	0
spend	0.0036	0	0.0036	0	0	0	0	0

A bigger example: Berkeley Restaurant Project sentences

Here are a few other useful probabilities:

$$P(i | \langle s \rangle) = 0.25$$

$$P(\text{english} | \text{want}) = 0.0011$$

$$P(\text{food} | \text{english}) = 0.5$$

$$P(\langle /s \rangle | \text{food}) = 0.68$$

Now we can compute the probability of sentences like *I want English food* or *I want Chinese food* by simply multiplying the appropriate bigram probabilities together, as follows:

$$\begin{aligned} P(\langle s \rangle \ i \ \text{want} \ \text{english} \ \text{food} \ \langle /s \rangle) \\ &= P(i | \langle s \rangle)P(\text{want} | I)P(\text{english} | \text{want}) \\ &\quad P(\text{food} | \text{english})P(\langle /s \rangle | \text{food}) \\ &= .25 \times .33 \times .0011 \times 0.5 \times 0.68 \\ &= .000031 \end{aligned}$$

A bigger example: Berkeley Restaurant Project sentences

- Compute the probability of **I want chinese food**
- $P(< s > \text{ I want chinese food } / s >)$
- $= P(I | < s >) P(\text{want} | I) P(\text{chinese} | \text{want}) P(\text{food} | \text{chinese}) P(< / s > | \text{food})$
- $= 0.25 \times 0.33 \times 0.0065 \times 0.52 \times 0.68$
- $= 0.00018$
- These probabilities get super tiny when we have longer inputs more infrequent words

- **Training corpus:**
 - *< s > I am from Manipal </ s >*
 - *< s > I am a teacher </ s >*
 - *< s > students are good and are from various cities </ s >*
 - *< s > students from Manipal do engineering </ s >*
- **Test data:**
 - *< s > students are from Manipal </ s >*

- As per the Bigram model, the test sentence can be expanded as follows to estimate the bigram probability;

$P(< s > \text{ students are from Manipal } < /s >)$

$$= P(\text{students} | < s >) * P(\text{are} | \text{students}) * P(\text{from} | \text{are}) \\ * P(\text{Manipal} | \text{from}) * P(< /s > | \text{Manipal})$$

$$P(w_n | w_{n-1}) = \frac{\text{count}(w_{n-1}, w_n)}{\text{count}(w_{n-1})}$$

$$P(\text{are} | \text{students}) = \frac{\text{count}(\text{students are})}{\text{count}(\text{students})} = \frac{1}{2}$$

$$P(\text{students} | < s >) = \frac{\text{count}(< s > \text{ students})}{\text{count}(< s >)} = \frac{2}{4} = \frac{1}{2}$$

$$P(\text{from} | \text{are}) = \frac{\text{count}(\text{are from})}{\text{count}(\text{are})} = \frac{1}{2}$$

$$P(\text{Manipal} | \text{from}) = \frac{\text{count}(\text{from Manipal})}{\text{count}(\text{from})} = \frac{2}{3}$$

$$P(< /s > | \text{Manipal}) = \frac{\text{count}(\text{Manipal } < /s >)}{\text{count}(\text{Manipal})} = \frac{1}{2}$$

$$P(< s > \text{ students are from Manipal } < /s >) = 1/2 * 1/2 * 1/2 * 2/3 * 1/2 = 0.0416$$

Method-2

- Unigram count matrix

<s>	students	are	from	Manipal	</s>
4	2	2	3	2	4

- Bigram count matrix

		w_n				
		students	are	from	Manipal	</s>
w_{n-1}	<s>	2	0	0	0	0
	students	0	1	1	0	0
	are	0	0	1	0	0
	from	0	0	0	2	0
	Manipal	0	0	0	0	1

- Bigram probability matrix (normalized by unigram counts)

		w_n				
		students	are	from	Manipal	</s>
w_{n-1}	<s>	2/4	0/4	0/4	0/4	0/4
	students	0/2	1/2	1/2	0/2	0/2
	are	0/2	0/2	1/2	0/2	0/2
	from	0/3	0/3	0/3	2/3	0/3
	Manipal	0/2	0/2	0/2	0/2	1/2

$$P(<\mathbf{s}> \text{ students} \text{ are} \text{ from} \\ \text{ Manipal} </\mathbf{s}>) = 1/2 * 1/2 * 1/2 * 2/3 * 1/2 = 0.0416$$

- The training-and-testing paradigm can be used to evaluate different N-gram architectures.
- To compare different language models , take a corpus and divide it into a **training set and a test set**.
- Then we train the two different N-gram models on the training set and see which one better models the test set.
- There is a useful metric for how well a given statistical model matches a test corpus, called **perplexity**.
- **Perplexity** is based on computing the probability of each sentence in the test set; intuitively, whichever model assigns a higher probability to the test set (hence more accurately predicts the test set) is a better model.
- Since the evaluation metric is based on test set probability, it's important not to let the test sentences into the training set.
- Suppose we are trying to compute the probability of a particular “test” sentence. If the test sentence is part of the training corpus, we will mistakenly assign it an artificially high probability when it occurs in the test set. This situation is called **training on the test set**.
- Training on the test set introduces a bias that makes the probabilities all look too high and causes huge inaccuracies in perplexity.

- In addition to training and test sets, other divisions of data are often useful.
- Sometimes we need an extra source of data to augment the training set. Such extra data is called a **held-out** set, because we hold it out from our training set when we train our N-gram counts. The held-out corpus is then used to set some other parameters.
- Finally, sometimes we need to have multiple test sets.
- This happens because we might use a particular test set so often that we implicitly tune to its characteristics. Then we would definitely need a fresh test set which is truly unseen. In such cases, we call the initial test set the **development test set or, devset**.

EVALUATING N-GRAMS: PERPLEXITY

- The best way to evaluate the performance of a language model is to embed it in an application and measure the total performance of the application.
- Such end-to-end evaluation is called **extrinsic** evaluation, and also sometimes called **in vivo evaluation** (Sparck Jones and Galliers, 1996).
- Extrinsic evaluation is the only way to know if a particular improvement in a component is really going to help the task at hand.
- Thus for speech recognition, we can compare the performance of two language models by running the speech recognizer twice, once with each language model, and seeing which gives the more accurate transcription.
- Unfortunately, end-to-end evaluation is often very expensive; evaluating a large speech recognition test set, for example, takes hours or even days.
- Thus we would like a metric that can be used to quickly evaluate potential improvements in a language model.
- **An intrinsic evaluation** metric is one which measures the quality of a model independent of any application.
- **Perplexity is the most common intrinsic evaluation metric** for N-gram language models. While an (intrinsic) improvement in perplexity does not guarantee an (extrinsic) improvement in speech recognition performance (or any other end-to-end metric), it often correlates with such improvements.
- Thus it is commonly used as a quick check on an algorithm and an improvement in perplexity can then be confirmed by an end-to-end evaluation.

- The intuition of perplexity is that given two probabilistic models, the better model is the one that has a tighter fit to the test data, or predicts the details of the test data better.
- We can measure better prediction by looking at the probability the model assigns to the test data; the better model will assign a higher probability to the test data.
- More formally, the perplexity (sometimes called PP for short) of a language model on a test set is a function of the probability that the language model assigns to that test set.
- For a test set $W = w_1 w_2 \dots w_N$, the perplexity is the probability of the test set, normalized by the number of words:

$$\begin{aligned} \text{PP}(W) &= P(w_1 w_2 \dots w_N)^{-\frac{1}{N}} \\ &= \sqrt[N]{\frac{1}{P(w_1 w_2 \dots w_N)}} \end{aligned}$$

We can use the chain rule to expand the probability of W :

$$\text{PP}(W) = \sqrt[N]{\prod_{i=1}^N \frac{1}{P(w_i|w_1 \dots w_{i-1})}}$$

Thus if we are computing the perplexity of W with a bigram language model, we get:

$$\text{PP}(W) = \sqrt[N]{\prod_{i=1}^N \frac{1}{P(w_i|w_{i-1})}}$$

- Because of the inverse in equation, the higher the conditional probability of the word sequence, the lower the perplexity.
- Thus minimizing perplexity is equivalent to maximizing the test set probability according to the language model.

<i>N</i> -gram Order	Unigram	Bigram	Trigram
Perplexity	962	170	109

- What we generally use for word sequence is the entire sequence of words in some test set.
- Since this sequence will cross many sentence boundaries, **we need to include the begin- and end-sentence markers <s> and </s> in the probability computation.**
- **We also need to include the end-of-sentence marker </s> (but not the beginning-of-sentence marker <s>)** in the total count of word tokens N.

Practical Issues

- There is a major problem with the maximum likelihood estimation process we have seen for training the parameters of an N-gram model.
- This is the problem of sparse data caused by the fact that our maximum likelihood estimate was based on a particular set of training data.
- For any N-gram that occurred a sufficient number of times, we might have a good estimate of its probability.
- But because any corpus is limited, some perfectly acceptable English word sequences are bound to be missing from it.
- This missing data means that the N-gram matrix for any given training corpus is bound to have a very large number of cases of putative “zero probability N-grams” that should really have some non-zero probability.
- Furthermore, the MLE method also produces poor estimates when the counts are non-zero but still small.
- We need a method which can help get better estimates for these zero or low frequency counts.
- Zero counts turn out to cause another huge problem. The perplexity metric defined above requires that we compute the probability of each test sentence.
- But if a test sentence has an N-gram that never appeared in the training set, the Maximum Likelihood estimate of the probability for this N-gram, and hence for the whole test sentence, will be zero!
- This means that in order to evaluate our language models, we need to modify the MLE method to assign some non-zero probability to any N-gram, even one that was never observed in training.
- For these reasons, we'll want to modify the maximum likelihood estimates for computing N-gram probabilities, focusing on the N-gram events that we incorrectly assumed had zero probability.
- We use the term smoothing for such modifications that address the poor estimates that are due to variability in small data sets.

Practical Issues

- We do everything in log space
 - Avoid underflow
 - (also adding is faster than multiplying)

$$\log(p_1 \times p_2 \times p_3 \times p_4) = \log p_1 + \log p_2 + \log p_3 + \log p_4$$

How do you handle unseen n-grams?

- **Smoothing**

Use some of the probability mass to cover unseen events

- **Backoff**

Use counts from a smaller context

- **Interpolation**

Combine multiple sources of information appropriately weighted

Smoothing

- Since there are a combinatorial number of possible word sequences, many rare (but not impossible) combinations never occur in training, so MLE incorrectly assigns zero to many parameters (***sparse data***).
- If a new combination occurs during testing, it is given a probability of zero and the entire sequence gets a probability of zero (i.e. infinite perplexity).
- Modify the maximum likelihood estimates for computing N-gram probabilities, focusing on the N-gram events that we incorrectly assumed had zero probability.
- We use the term **smoothing** for such modifications that address the poor estimates that are due to variability in small data sets
- The name comes from the fact that(looking ahead a bit) we **will be saving a little bit of probability mass from the higher counts, and piling it instead on the zero counts**, making the distribution a little less jagged.

Laplace Smoothing

- One simple way to do smoothing might be just to take our matrix of bigram counts, before we normalize them into probabilities, and add one to all the counts. This algorithm is called [Laplace smoothing, or Laplace's Law](#)
- Laplace smoothing does not perform well enough to be used in modern N-gram models, but it introduces many of the concepts that is used in other smoothing algorithms and also gives us a useful baseline
- **Laplace smoothing to unigram probabilities:**
 - Recall that the unsmoothed maximum likelihood estimate of the unigram probability of the word w_i is its count c_i normalized by the total number of word tokens N :

$$P(w_i) = \frac{c_i}{N}$$

Laplace Smoothing

- Laplace smoothing merely adds one to each count (hence its alternate name **add one smoothing**)
- Since there are V words in the vocabulary, and each one got incremented, we also need to adjust the denominator to take into account the extra V observations

$$P_{\text{Laplace}}(w_i) = \frac{c_i + 1}{N + V}$$

Laplace Smoothing

- Instead of changing both the numerator and denominator it is convenient to describe how a smoothing algorithm affects the numerator, by defining an **adjusted count c^***
- This adjusted count is easier to compare directly with the MLE counts, and can be turned into a probability like an MLE count by normalizing by N
- To define this count, since we are only changing the numerator, in addition to adding one we'll also need to multiply by a normalization factor $\frac{N}{N+V}$.

$$c_i^* = (c_i + 1) \frac{N}{N+V}$$

We can now turn c_i^* into a probability P_i^* by normalizing by N .

Laplace Smoothing

- A related way to view smoothing is as **discounting (lowering)** some non-zero counts in order to get the probability mass that will be assigned to the zero counts.
- Thus, instead of referring to the discounted counts c^* , we might describe a smoothing algorithm in terms of a relative discount d_c , the ratio of the discounted counts to the original counts:

$$d_c = \frac{c^*}{c}$$

Unigram Smoothing Example

- Tiny Corpus, V=4; N=20

$$P_{\nu}(w_i) = \frac{c_i + 1}{N + V}$$

Word	True Ct	Unigram Prob	New Ct	Adjusted Prob
eat	10	.5	11	.46
British	4	.2	5	.21
food	6	.3	7	.29
happily	0	.0	1	.04
	20	1.0	~20	1.0

11/(20+4)

5/(20+4)

Example: Berkeley Restaurant

	i	want	to	eat	chinese	food	lunch	spend
i	5	827	0	9	0	0	0	2
want	2	0	608	1	6	6	5	1
to	2	0	4	686	2	0	6	211
eat	0	0	2	0	16	2	42	0
chinese	1	0	0	0	0	82	1	0
food	15	0	15	0	1	4	0	0
lunch	2	0	0	0	0	1	0	0
spend	1	0	1	0	0	0	0	0

Fig 1:Unsmoothed Bigram Counts

	i	want	to	eat	chinese	food	lunch	spend
i	6	828	1	10	1	1	1	3
want	3	1	609	2	7	7	6	2
to	3	1	5	687	3	1	7	212
eat	1	1	3	1	17	3	43	1
chinese	2	1	1	1	1	83	2	1
food	16	1	16	1	2	5	1	1
lunch	3	1	1	1	1	2	1	1
spend	2	1	2	1	1	1	1	1

Fig 2:-Add-one smoothed bigram counts for eight of the words (out of V = 1446) in the Berkeley Restaurant Project corpus of 9332 sentences.

Example: Berkeley Restaurant

i	want	to	eat	chinese	food	lunch	spend
2533	927	2417	746	158	1093	341	278

	i	want	to	eat	chinese	food	lunch	spend
i	6	828	1	10	1	1	1	3
want	3	1	609	2	7	7	6	2
to	3	1	5	687	3	1	7	212
eat	1	1	3	1	17	3	43	1
chinese	2	1	1	1	1	83	2	1
food	16	1	16	1	2	5	1	1
lunch	3	1	1	1	1	2	1	1
spend	2	1	2	1	1	1	1	1

$$P^*(w_n | w_{n-1}) = \frac{C(w_{n-1} w_n) + 1}{C(w_{n-1}) + V}$$

For add-one smoothed bigram counts we need to augment the unigram count by the number of total word types in the vocabulary V :

Example: Berkeley Restaurant

Laplace-smoothed bigrams

$$P^*(w_n | w_{n-1}) = \frac{C(w_{n-1} w_n) + 1}{C(w_{n-1}) + V}$$

	i	want	to	eat	chinese	food	lunch	spend
i	0.0015	0.21	0.00025	0.0025	0.00025	0.00025	0.00025	0.00075
want	0.0013	0.00042	0.26	0.00084	0.0029	0.0029	0.0025	0.00084
to	0.00078	0.00026	0.0013	0.18	0.00078	0.00026	0.0018	0.055
eat	0.00046	0.00046	0.0014	0.00046	0.0078	0.0014	0.02	0.00046
chinese	0.0012	0.00062	0.00062	0.00062	0.00062	0.052	0.0012	0.00062
food	0.0063	0.00039	0.0063	0.00039	0.00079	0.002	0.00039	0.00039
lunch	0.0017	0.00056	0.00056	0.00056	0.00056	0.0011	0.00056	0.00056
spend	0.0012	0.00058	0.0012	0.00058	0.00058	0.00058	0.00058	0.00058

Add-one smoothed bigram probabilities for eight of the words (out of $V = 1446$) in the BeRP corpus of 9332 sentences

Reconstituted counts

- It is often convenient to reconstruct the count matrix, to see how much a smoothing algorithm changed the original counts.
- The adjusted counts are computed as :

$$c^*(w_{n-1}w_n) = \frac{[C(w_{n-1}w_n) + 1] \times C(w_{n-1})}{C(w_{n-1}) + V}$$

	i	want	to	eat	chinese	food	lunch	spend
i	3.8	527	0.64	6.4	0.64	0.64	0.64	1.9
want	1.2	0.39	238	0.78	2.7	2.7	2.3	0.78
to	1.9	0.63	3.1	430	1.9	0.63	4.4	133
eat	0.34	0.34	1	0.34	5.8	1	15	0.34
chinese	0.2	0.098	0.098	0.098	0.098	8.2	0.2	0.098
food	6.9	0.43	6.9	0.43	0.86	2.2	0.43	0.43
lunch	0.57	0.19	0.19	0.19	0.19	0.38	0.19	0.19
spend	0.32	0.16	0.32	0.16	0.16	0.16	0.16	0.16

Add-one reconstituted counts for eight words (of $V = 1446$) in the BeRP corpus of 9332 sentences.

$1 \times 2533 / (2533 + 1446)$

$609 \times 927 / (927 + 1446)$

Example: Berkeley Restaurant

(Big Change to the Counts!)

- Note that add-one smoothing has made a very big change to the counts **C(want to) changed from 608 to 238!**
- We can see this in probability space as well: $P(\text{to} \mid \text{want})$ decreases from .66 in the unsmoothed case to .26 in the smoothed case
- Looking at the **discount d** (the ratio between new and old counts) shows us how strikingly the counts for each prefix word have been reduced; the discount for the bigram **want to is .39**, while the discount for Chinese food is **.10**, a factor of 10!
- The sharp change in counts and probabilities occurs because too much probability mass is moved to all the zeros
- *We could move a bit less mass by adding a fractional count rather than 1 (add- δ smoothing; (Lidstone, 1920; Johnson, 1932; Jeffreys, 1948)), but this method requires a method for choosing δ dynamically, results in an inappropriate discount for many counts, and turns out to give counts with poor variance.*

Example: Berkeley Restaurant

Compare with raw bigram counts

	i	want	to	eat	chinese	food	lunch	spend
i	5	827	0	9	0	0	0	2
want	2	0	608	1	6	6	5	1
to	2	0	4	686	2	0	6	211
eat	0	0	2	0	16	2	42	0
chinese	1	0	0	0	0	82	1	0
food	15	0	15	0	1	4	0	0
lunch	2	0	0	0	0	1	0	0
spend	1	0	1	0	0	0	0	0

	i	want	to	eat	chinese	food	lunch	spend
i	3.8	527	0.64	6.4	0.64	0.64	0.64	1.9
want	1.2	0.39	238	0.78	2.7	2.7	2.3	0.78
to	1.9	0.63	3.1	430	1.9	0.63	4.4	133
eat	0.34	0.34	1	0.34	5.8	1	15	0.34
chinese	0.2	0.098	0.098	0.098	0.098	8.2	0.2	0.098
food	6.9	0.43	6.9	0.43	0.86	2.2	0.43	0.43
lunch	0.57	0.19	0.19	0.19	0.19	0.38	0.19	0.19
spend	0.32	0.16	0.32	0.16	0.16	0.16	0.16	0.16

Problem with Laplace Smoothing

- Problem: gives too much probability mass to unseen n-grams.
- For sparse sets of data over large vocabularies, such as n-grams, Laplace's law actually gives far too much of the probability space to unseen events.

Good-Turing Smoothing

- Also called Good-Turing discounting, Good-Turing estimation
- Intuition is to use the count of things we've seen once to help estimate the count of things we've never seen.
- The Good-Turing algorithm was first described by Good (1953), who credits Turing with the original idea.
- The basic insight of Good-Turing smoothing is to re-estimate the amount of probability mass to assign to N-grams with zero counts by **looking at the number of N-grams that occurred one time**.
- A word or N-gram (or any event) that occurs once is called a **singleton, or a hapax legomenon**.
- The Good-Turing intuition is to use the frequency of singletons as a re-estimate of the frequency of zero-count bigrams

Good-Turing Smoothing

- The Good-Turing algorithm is based on computing N_c , the number of N-grams that occur c times.
 - We refer to the number of N-grams that occur c times as the **frequency of frequency c** .
 - So applying the idea to smoothing the joint probability of bigrams, N_0 is the number of bigrams with count 0, N_1 the number of bigrams with count 1 (singletons), and so on.
 - We can think of each of the **N_c as a bin which stores the number of different N-grams that occur in the training set with that frequency c** .
 - Formally,
- $$N_c = \sum_{x:count(x)=c} 1$$
- The MLE count for N_c is c . The Good-Turing estimate replaces this with a smoothed count c^* , as a function of N_c+1 :

$$c^* = (c + 1) \frac{N_{c+1}}{N_c}$$

Good-Turing Smoothing

- The above equation can be used to replace the MLE counts for all the bins N1, N2, and so on.
- Instead of using this equation directly to re-estimate the smoothed count c^* for N_0 , we use the following equation for the probability P^*_{GT} for things that had zero count N_0 , or what we might call the **missing mass**.

$$P^*_{GT}(\text{things with frequency zero in training}) = \frac{N_1}{N}$$

- Here N_1 is the count of items in bin 1, i.e. that were seen once in training, and N is the total number of items we have seen in training.

Good-Turing Smoothing

- The Good-Turing method was first proposed for estimating the populations of animal species.
- Consider an illustrative example from this domain created by Joshua Goodman and Stanley Chen.
S
- Imagine you are fishing
 - There are 8 species in the lake: carp, perch, whitefish, trout, salmon, eel, catfish, bass
- You catch:
 - 10 carp, 3 perch, 2 whitefish, 1 trout, 1 salmon, 1 eel = 18 fish

What is the probability next fish caught is from a new species (one not seen in our previous catch, ie. The one has 0 count in training set)?

And how likely is it that the next species is another trout?

	unseen (bass or catfish)	trout
c	0	1
MLE p	$p = \frac{0}{18} = 0$	$\frac{1}{18}$
c^*		$c^*(\text{trout}) = 2 \times \frac{N_2}{N_1} = 2 \times \frac{1}{3} = .67$
GT p_{GT}^*	$p_{\text{GT}}^*(\text{unseen}) = \frac{N_1}{N} = \frac{3}{18} = .17$	$p_{\text{GT}}^*(\text{trout}) = \frac{.67}{18} = \frac{1}{27} = .037$

Good-Turing Smoothing

- The revised count c^* for trout was discounted from $c = 1.0$ to $c^* = .67$, (thus leaving some probability mass $p^* \text{ GT(unseen)} = 3 / 18 = .17$ for the catfish and bass).
- And since we know there were 2 unknown species, the probability of the next fish being specifically a catfish is $p^* \text{ GT(catfish)} = 1 / 2 \times 3 / 18 = .085$.

AP Newswire			Berkeley Restaurant		
c (MLE)	N_c	c^* (GT)	c (MLE)	N_c	c^* (GT)
0	74,671,100,000	0.0000270	0	2,081,496	0.002553
1	2,018,046	0.446	1	5315	0.533960
2	449,721	1.26	2	1419	1.357294
3	188,933	2.24	3	642	2.373832
4	105,668	3.24	4	381	4.081365
5	68,379	4.22	5	311	3.781350
6	48,190	5.19	6	196	4.500000

Bigram “frequencies of frequencies” and Good-Turing re-estimations for the 22 million AP bigrams from Church and Gale (1991) and from the Berkeley Restaurant corpus of 9332 sentences.

Backoff And Interpolation

- The discounting we have been discussing so far can help solve the problem of zero frequency N-grams
- But there is an additional source of knowledge we can draw on
- If we are trying to compute $P(w_n | w_{n-2} w_{n-1})$ but we have no examples of a particular trigram $w_{n-2} w_{n-1} w_n$, we can instead estimate its probability by using the bigram probability $P(w_n | w_{n-1})$.
- Similarly, if we don't have counts to compute $P(w_n | w_{n-1})$, we can look to the unigram $P(w_n)$.

Backoff And Interpolation

- In other words, sometimes using less context is a good thing, helping to generalize more for contexts that the model hasn't learned much about
- There are two ways backoff to use this N-gram “hierarchy”
- In backoff, we use the trigram if the evidence is sufficient, otherwise we use the bigram, otherwise the unigram
- In other words, we only “back off” to a lower-order N-gram if we have zero evidence for a higher-order interpolation N-gram
- By contrast, in interpolation, we always mix the probability estimates from all the N-gram estimators, weighing and combining the trigram, bigram, and unigram counts.

Backoff And Interpolation

- In simple linear interpolation, we combine different order N-grams by linearly interpolating all the models
- Thus, we estimate the trigram probability $P(w_n | w_{n-2} w_{n-1})$ by mixing together the unigram, bigram, and trigram probabilities, each weighted b:

UNIT 7

Word Classes and Part-of-Speech
Tagging

History of Parts of Speech

- Dionysius Thrax of Alexandria (c. 100 B.C.), wrote a grammatical sketch of Greek (a “techn-e”) which summarized the linguistic knowledge of his day.
- That work is the direct source of an astonishing proportion of our modern linguistic vocabulary, including among many other words, syntax, diphthong, clitic, and analogy.
- Also included are a description of eight parts-of-speech: noun, verb, pronoun, preposition, adverb, conjunction, participle, and article.
- Although earlier scholars had their own lists of parts-of speech, it was Thrax’s set of eight which became the basis for practically all subsequent part-of-speech descriptions of Greek, Latin, and most European languages for the next 2000 years.

History of Parts of Speech

- More recent lists of parts-of-speech (or tagsets) have many more word classes;
- 45 for the Penn Treebank (Marcus et al., 1993),
- 87 for the Brown corpus (Francis, 1979; Francis and Kučera, 1982), and
- 146 for the C7 tagset (Garside et al., 1997)

Parts of Speech Tagging Task

- Input: a sequence of word tokens w
- Output: a sequence of part-of-speech tags t , one per word

Parts of Speech-Advantages in NLP

- The significance of parts-of-speech (also known as **POS, word classes, tagsets, morphological classes, or lexical tags**) are:
 1. Large amount of information they give about a word and its neighbors for language processing.
 - For example these tagsets distinguish between **possessive pronouns** (my, your, his, her, its) and **personal pronouns** (I, you, he, me).
 - Knowing whether a word is a possessive pronoun or a personal pronoun can tell us what words are likely to occur in its vicinity (possessive pronouns are likely to be followed by a noun, personal pronouns by a verb).
 - This can be useful in a language model for speech recognition.

- A word's part-of-speech can tell us something about how the word is pronounced.
 - Example - the word content, can be a noun or an adjective.
- They are pronounced differently (the noun is pronounced **CO**n**tent** and the adjective **co**n**TENT**).
- Thus knowing the part-of-speech can produce more natural pronunciations in a speech synthesis system and more accuracy in a speech recognition system.
- Parts-of-speech can also be used in stemming for informational retrieval (IR), since knowing a word's part-of-speech can help tell us which morphological affixes it can take

Parts-of-speech Tagging

- **Part-of-speech tagging** is a computational method for assigning parts-of-speech to words.
- Many algorithms have been applied to this problem including
 - Hand-written rules (**rule-based tagging**),
 - Probabilistic methods (**HMM tagging** and **maximum entropy tagging**),
Transformation based tagging
 - **Memory-based tagging**

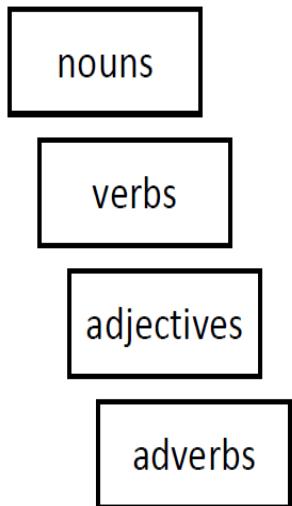
ENGLISH WORD CLASSES

- Words that somehow ‘behave’ alike:
 - Appear in similar contexts
 - Perform similar functions in sentences
 - Undergo similar transformations
- More complete definition of POS and other classes:
 - Traditionally based on **syntactic** and **morphological** function, grouping words that have similar neighboring words (their distributional properties) or take similar affixes (their morphological properties).
 - While word classes do have **semantic tendencies**—adjectives, for example, often describe properties and nouns

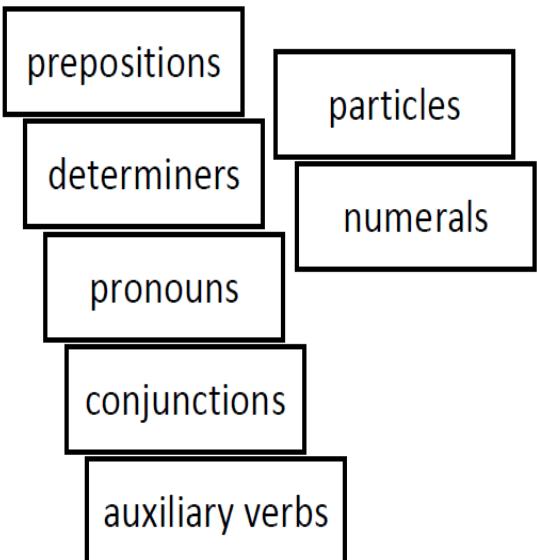
English Word Class

- Parts-of-speech can be divided into two broad supercategories:
 - **CLOSED CLASS**
 - Closed classes are those that have relatively fixed membership.
 - Example: Prepositions are a closed class because there is a fixed set of them in English;
 - **OPEN CLASS**
 - Open classes continually coined or borrowed from other languages
 - Example: nouns and verbs are open classes because new nouns and verbs can be taken from different language.

open classes



closed classes



- Closed class words are also generally **function words** like of, it, and, or you, which tend to be very short, occur frequently, and often have structuring uses in grammar.
- There are **four major open classes** that occur in the languages of the world; ***nouns, verbs, adjectives, and adverbs***

Open Word Classes

Noun

- Name given to the syntactic class in which the words for most people, places, or things occur
- Since syntactic classes like **noun** are defined syntactically and morphologically rather than semantically
 - some words for people, places, and things may not become nouns
 - Some nouns may not be words for people, places or things
- Thus, nouns include
 - Concrete terms like *ship* and *chair*,
 - Abstractions like *bandwidth* and *relationship*
 - Verb like terms like *pacing*
- Noun in English
 - Things to occur like determiners (*a goat, it's bandwidth, Plato's Republic*)
 - To take possessives (*IBM's annual revenue*)
 - To occur in plural form (*goat*)

- Nouns are grouped into **proper nouns** and **common nouns**.

- Proper nouns

- like *Regina*, *Colorado*, and *IBM*, are names of specific persons or entities.
- In English, they generally aren't preceded by articles (e.g., *the book is upstairs*, but *Regina is upstairs*).
- In written English, proper nouns are usually capitalized.

- common nouns are divided into **count nouns** and **mass nouns**.

- **Count nouns**

- are those that allow grammatical enumeration;
- they can occur in both the singular and plural (*goat/goats*, *relationship/relationships*)
- they can be counted (*one goat*, *two goats*).

- **Mass nouns**

- are used when something
- is conceptualized as a homogeneous group.
- So words like *snow*, *salt*, and *communism* are not counted (i.e., **two snows* or **two communisms*).
- Mass nouns can also appear without articles where singular count nouns cannot (*Snow is white* but not **Goat is white*)

Open Verb class

- Includes most of the words referring to actions VERB and processes, including main verbs like *draw*, *provide*, *differ*, and *go*.
 - A number of morphological forms (non-3rd-person-sg (*eat*), 3rd-person-sg (*eats*), progressive (*eating*), past participle (*eaten*))
 - A subclass of English verbs called **auxiliaries** will be discussed when we turn to closed-class forms.
-
- **Adjectives**
 - Terms describing properties or qualities
 - Most languages have adjective for the concept of color (white, black), age (young, old) and value(good,bad)
 - There are languages without adjectives Example: chineese

Adverbs

- Words viewed as modifying something
 - **Directional adverbs** or **locative adverbs** : specify the direction or location of some action (*home, here, downhill*)
 - **Degree adverbs** : specify the extent of some action, process, or property (*extremely, very, somewhat*)
 - **Manner adverbs** : describe the manner of some action or process (*slowly, slinkily, delicately*)
 - **Temporal adverb** : describe the time that some action or event took place (*yesterday, Monday*).

Closed Word Classes

- Some important closed classes in English
 - **Prepositions:** on, under, over, near, by, at, from, to, with
 - **Determiners:** a, an, the
 - **Pronouns:** she, who, I, others
 - **Conjunctions:** and, but, or, as, if, when
 - **Auxiliary verbs:** can, may, should, are
 - **Particles:** up, down, on, off, in, out, at, by
 - **Numerals:** one, two, three, first, second, third

- **Prepositions** occur before nouns, semantically they are relational
 - Indicating spatial or temporal relations, whether literal (*on it, before then, by the house*) or metaphorical (*on time, with gusto, beside herself*)
 - Other relations as well

of	540,085	through	14,964	worth	1,563	pace	12
in	331,235	after	13,670	toward	1,390	nigh	9
for	142,421	between	13,275	plus	750	re	4
to	125,691	under	9,525	till	686	mid	3
with	124,965	per	6,515	amongst	525	o'er	2
on	109,129	among	5,090	via	351	but	0
at	100,169	within	5,030	amid	222	ere	0
by	77,794	towards	4,700	underneath	164	less	0
from	74,843	above	3,056	versus	113	midst	0
about	38,428	near	2,026	amidst	67	o'	0
than	20,210	off	1,695	sans	20	thru	0
over	18,071	past	1,575	circa	14	vice	0

Figure 5.1 Prepositions (and particles) of English from the CELEX on-line dictionary.
Frequency counts are from the COBUILD 16 million word corpus.

- A **particle** is a word that resembles a preposition or an adverb, and that often combines with a verb to form a larger unit call a **phrasal verb**

So I *went on* for some days cutting and hewing timber ...

Moral reform is the effort to *throw off* sleep ...

aboard	aside	besides	forward(s)	opposite	through
about	astray	between	home	out	throughout
above	away	beyond	in	outside	together
across	back	by	inside	over	under
ahead	before	close	instead	overhead	underneath
alongside	behind	down	near	past	up
apart	below	east, etc.	off	round	within
around	beneath	eastward(s),etc.	on	since	without

Figure 5.2 English single-word particles from Quirk et al. (1985).

A closed class that occurs with nouns, often marking the beginning of a noun phrase, is the **determiners**.

One small subtype of determiners is the **articles**:

- English has three: *a*, *an*, and *the*
 - Articles begin a noun phrase.
 - Articles are frequent in English.
- **Conjunctions** are used to join two phrases, clauses, or sentences.
 - *and*, *or*, *or*, *but*
 - Subordinating conjunctions are used when one of the elements is of some sort of embedded status. *I thought that you might like some milk.*.. **complementizer**

- **Pronouns** act as a kind of shorthand for referring to some noun phrase or entity or event.
 - **Personal pronouns:** persons or entities (*you, she, I, it, me, etc*)
 - **Possessive pronouns:** forms of personal pronouns indicating actual possession or just an abstract relation between the person and some objects.
 - **Wh-pronouns:** used in certain question forms, or may act as complementizer.

it	199,920	how	13,137	yourself	2,437	no one	106
I	198,139	another	12,551	why	2,220	wherein	58
he	158,366	where	11,857	little	2,089	double	39
you	128,688	same	11,841	none	1,992	thine	30
his	99,820	something	11,754	nobody	1,684	summat	22
they	88,416	each	11,320	further	1,666	suchlike	18
this	84,927	both	10,930	everybody	1,474	fewest	15
that	82,603	last	10,816	ourselves	1,428	thyself	14
she	73,966	every	9,788	mine	1,426	whomever	11
her	69,004	himself	9,113	somebody	1,322	whosoever	10
we	64,846	nothing	9,026	former	1,177	whomsoever	8
all	61,767	when	8,336	past	984	wherefore	6
which	61,399	one	7,423	plenty	940	whereat	5
their	51,922	much	7,237	either	848	whatsoever	4
what	50,116	anything	6,937	yours	826	whereon	2
my	46,791	next	6,047	neither	618	whoso	2
him	45,024	themselves	5,990	fewer	536	aught	1
me	43,071	most	5,115	hers	482	howsoever	1
who	42,881	itself	5,032	ours	458	thrice	1
them	42,099	myself	4,819	whoever	391	wheresoever	1
no	33,458	everything	4,662	least	386	you-all	1
some	32,863	several	4,306	twice	382	additional	0
other	29,391	less	4,278	theirs	303	anybody	0
your	28,923	herself	4,016	wherever	289	each other	0
its	27,783	whose	4,005	oneself	239	once	0
our	23,029	someone	3,755	thou	229	one another	0
these	22,697	certain	3,345	'un	227	overmuch	0
any	22,666	anyone	3,318	ye	192	such and such	0
more	21,873	whom	3,229	thy	191	whate'er	0
many	17,343	enough	3,197	whereby	176	whenever	0
such	16,880	half	3,065	thee	166	whereof	0
those	15,819	few	2,933	yourselves	148	whereto	0
own	15,741	everyone	2,812	latter	142	whereunto	0
us	15,724	whatever	2,571	whichever	121	whichsoever	0

*Pronouns of English from the
CELEX on-line dictionary.*

- **Auxiliary verbs:** mark certain semantic feature of a main verb, including
 - whether an action takes place in the present, past or future (tense),
 - whether it is completed (aspect),
 - whether it is negated (polarity), and
 - whether an action is necessary, possible, suggested, desired, etc (mood).
 - Including **copula** verb *be*, the two verbs *do* and *have* along with their inflection forms, as well as a class of **modal verbs**.

can	70,930	might	5,580	shouldn't	858
will	69,206	couldn't	4,265	mustn't	332
may	25,802	shall	4,118	'll	175
would	18,448	wouldn't	3,548	needn't	148
should	17,760	won't	3,100	mightn't	68
must	16,520	'd	2,299	oughtn't	44
need	9,955	ought	1,845	mayn't	3
can't	6,375	will	862	dare	??
have	???				

*English modal verbs from
the CELEX on-line dictionary.*

Tagsets for English

- There are various standard tagsets to choose from; some have a lot more tags than others
- The choice of tagset is based on the application
- Accurate tagging can be done with even large tagsets
 - **Example**-Small 45-tag Penn Treebank Tagset, The Penn Treebank tagset was culled from the original 87-tag tagset for the Brown corpus.
 - This reduced set leaves out information that can be recovered from the identity of the lexical item.
 - This tagset has been used to label a wide variety of corpora, including the Brown corpus, the Wall Street Journal corpus, and the Switchboard corpus.

Tagsets for English

Tag	Description	Example	Tag	Description	Example
CC	coordin. conjunction	<i>and, but, or</i>	SYM	symbol	+,%,&
CD	cardinal number	<i>one, two</i>	TO	“to”	<i>to</i>
DT	determiner	<i>a, the</i>	UH	interjection	<i>ah, oops</i>
EX	existential ‘there’	<i>there</i>	VB	verb base form	<i>eat</i>
FW	foreign word	<i>mea culpa</i>	VBD	verb past tense	<i>ate</i>
IN	preposition/sub-conj	<i>of, in, by</i>	VBG	verb gerund	<i>eating</i>
JJ	adjective	<i>yellow</i>	VBN	verb past participle	<i>eaten</i>
JJR	adj., comparative	<i>bigger</i>	VBP	verb non-3sg pres	<i>eat</i>
JJS	adj., superlative	<i>wildest</i>	VBZ	verb 3sg pres	<i>eats</i>
LS	list item marker	<i>1, 2, One</i>	WDT	wh-determiner	<i>which, that</i>
MD	modal	<i>can, should</i>	WP	wh-pronoun	<i>what, who</i>
NN	noun, sing. or mass	<i>llama</i>	WP\$	possessive wh-	<i>whose</i>
NNS	noun, plural	<i>llamas</i>	WRB	wh-adverb	<i>how, where</i>
NNP	proper noun, sing.	<i>IBM</i>	\$	dollar sign	\$
NNPS	proper noun, plural	<i>Carolinas</i>	#	pound sign	#
PDT	predeterminer	<i>all, both</i>	“	left quote	‘ or “
POS	possessive ending	<i>'s</i>	”	right quote	’ or ”
PRP	personal pronoun	<i>I, you, he</i>	(left parenthesis	[, (, {, <
PRP\$	possessive pronoun	<i>your, one's</i>)	right parenthesis	,), }, >
RB	adverb	<i>quickly, never</i>	,	comma	,
RBR	adverb, comparative	<i>faster</i>	.	sentence-final punc	. ! ?
RBS	adverb, superlative	<i>fastest</i>	:	mid-sentence punc	: ; ... --
RP	particle	<i>up, off</i>			

Figure:Penn Treebank
Part-of-Speech Tags
(including
punctuation)

Tagsets for English

- Using Penn Treebank tags, tag the following sentence from the Brown Corpus:

- The grand jury commented on a number of other topics.

The/DT grand/JJ jury/NN commented/VBD on/IN a/DT number/NN of/IN other/JJ topics/NNS ./.

- There are 70 children there

There/EX are/VBP 70/CD children/NNS there/RB

- Preliminary findings were reported in today's New England Journal of Medicine.

Preliminary/JJ findings/NNS were/VBD reported/VBN in/IN today/NN 's/POS New/NNP England/NNP Journal/NNP of/IN Medicine/NNP ./.

Tagsets for English

- How do tagsets differ?
 - Degree of granularity
 - Idiosyncratic decisions,
 - Penn Treebank doesn't distinguish **to/Prep** from **to/Inf**
 - eg. **I/PP want/VBP to/TO go/VB to/TO Zanzibar/NNP ./.**
 - Don't tag it if you can recover from a word

Part-of-speech tagging

- Part-of-speech tagging is the process of assigning a part-of-speech to each word in tagging a text.
- The input is a sequence of (tokenized) words and a tagset, and $x_1; x_2; \dots; x_n$
- The output is a sequence $y_1; y_2; \dots; y_n$ of tags, each output y_i corresponding exactly to one input x_i .
- Tagging: Process of assigning a part-of-speech or other syntactic class marker to each word in a corpus

Parts of Speech Tagging

- The input to a tagging algorithm is a string of words and a specified **tagset** of the kind described previously.

```
VB  DT  NN .
Book that flight .
```

```
VBZ DT NN VB   NN ?
Does that flight serve dinner ?
```

- Automatically assigning a tag to a word is not trivial
 - For example, *book* is **ambiguous**: it can be a verb or a noun
 - Similarly, *that* can be a determiner, or a complementizer
- The problem of POS-tagging is to **resolve** the ambiguities, choosing the proper tag for the context.

Ambiguity in the Brown Corpus

Ambiguity in the Brown corpus

- 40% of word tokens are ambiguous
- 12% of word types are ambiguous
- Breakdown of ambiguous word types:

Unambiguous (1 tag)	35,340
Ambiguous (2–7 tags)	4,100
2 tags	3,760
3 tags	264
4 tags	61
5 tags	12
6 tags	2
7 tags	1 ("still")

Why is Part of Speech Tagging harder?

Words often have more than one POS: back

- The back door: *back/JJ*
- On my back: *back/NN*
- Win the voters back: *back/RB*
- Promised to back the bill: *back/VB*

POS tagging problem

To determine the POS tag for a particular instance of a word

- Some of the most ambiguous frequent words are **that**, **back**, **down**, **put** and **set**

- Examples of the **6** different parts-of-speech for the word **back**:

- earnings growth took a **back/JJ** seat

- a small building in the **back/NN**

- a clear majority of senators **back/VBP** the bill

- Dave began to **back/VB** toward the door

- enable the country to buy **back/RP** about debt

- I was twenty-one **back/RB** then

- Most words in English are unambiguous;
 - i.e., they have only a single tag
- But many of the most common words of English are ambiguous
 - Example **can** can be an auxiliary ('to be able'), a noun ('a metal container'), or a verb ('to put something in such a metal container')
- The problem of POS-tagging is to **resolve** these ambiguities, choosing the proper tag for the context.
- Part-of-speech tagging is thus one of the many disambiguation tasks.

Approaches to POS-Tagging

- Most tagging algorithms fall into one of two classes
 - **Rule-based taggers**
 - Involve a large database of hand-written disambiguation rules to tag input sequences
 - Example, rules like : an ambiguous word is a noun rather than a verb if it follows a determiner.
 - **EngCG Tagger** is based on the Constraint Grammar architecture of Karlsson et. al. (1995)
 - **Stochastic taggers**
 - Resolve tagging ambiguities by using a training corpus to compute the probability of a given word having a given tag in a given context.
 - Example: **Hidden Markov model or HMM Tagger**
 - **Hybrid systems** (e.g. **Brill's transformation-based learning**)
 - Share features of both tagging architectures.
 - Like the rule-based tagger, it is based on rules which determine when an ambiguous word should have a given tag. Like the stochastic taggers, it has a machine-learning component: the rules are automatically induced from a previously tagged training corpus.

Rule-based Part-of-Speech Tagging

- The earliest algorithms for automatically assigning part-of-speech were based on a two stage architecture (Harris, 1962; Klein and Simmons, 1963; Greene and Rubin, 1971).
- The first stage used a dictionary to assign each word a list of potential parts-of-speech.
- The second stage used large lists of hand-written disambiguation rules to window down this list to a single part-of-speech for each word.

EngCG (English Constraint Grammer) Tagger (Voutilainen, ENGCG 1995, 1999)

- The EngCG lexicon is based on the **two-level morphology**
- It has about 56,000 entries for English word stems ,counting a word with multiple parts-of-speech (e.g., nominal and verbal senses of hit) as separate entries, and not counting inflected and many derived forms
- Each entry is annotated with a set of morphological and syntactic features
- Fig. in the next slide shows some selected words, together with a slightly simplified listing of their features; these features are used in rule writing.

EngCG ENGTWOL

Word	POS	Additional POS features
smaller	ADJ	COMPARATIVE
entire	ADJ	ABSOLUTE ATTRIBUTIVE
fast	ADV	SUPERLATIVE
that	DET	CENTRAL DEMONSTRATIVE SG
all	DET	PREDETERMINER SG/PL QUANTIFIER
dog's	N	GENITIVE SG
furniture	N	NOMINATIVE SG NOINDEFDETERMINER SG
one-third	NUM	
she	PRON	PERSONAL FEMININE NOMINATIVE SG3
show	V	PRESENT -SG3 VFIN
show	N	NOMINATIVE SG
shown	PCP2	SVOO SVO SV
occurred	PCP2	SV
occurred	V	PAST VFIN SV

Figure 5.11 Sample lexical entries from the ENGTWOL lexicon described in Voutilainen (1995) and Heikkilä (1995).

- **SG** for singular
- **SG3** for other than third-person-singular
- **ABSOLUTE** means non-comparative and non-superlative for an adjective
- **NOMINATIVE** just means non-genitive
- **PCP2** means past participle
- **PRE**, **CENTRAL**, and **POST** are ordering slots for determiners
 - Eg: (predeterminers (all) come before determiners (the): all the president's men)
- **NOINDEFDETERMINER** means that words like furniture do not appear with the indefinite determiner **a**
- **SV**, **SVO**, and **SVOO** specify the subcategorization or complementation pattern for the verb

- In the first stage of tagger,
 - each word is run through the two-level lexicon transducer and
 - the entries for all possible POS are returned.
- A set of about 1,100 constraints are then applied to the input sentences to rule out incorrect POS.

Pavlov	PALOV N NOM SG PROPER
had	HAVE V PAST VFIN SVO
	HAVE PCP2 SVO
shown	SHOW PCP2 SVOO SVO SV
that	ADV
	PRON DEM SG
	DET CENTRAL DEM SG
	CS
salivation	N NOM SG
	...

- The boldfaced entries in the table above show the desired result, in which the simple past tense tag (rather than the past participle tag) is applied to **had**, and the complementizer (CS) tag is applied to **that**
- The constraints are used in a negative way, to eliminate tags that are inconsistent with the context

- A simplified version of the constraint:

ADVERBIAL-THE RULE

Given input: “that”

if

```
(+1 A/ADV/QUANT); /* if next word is adj, adverb, or quantifier */  
(+2 SENT-LIM);      /* and following which is a sentence boundary, */  
(NOT -1 SVOC/A);   /* and the previous word is not a verb like */  
                   /* 'consider' which allows adj as object complements */
```

then eliminate non-ADV tags

else eliminate ADV tags

- The first two clauses of this rule check to see that the ***that*** directly precedes a sentence-final adjective, adverb, or quantifier
- In all other cases the adverb reading is eliminated
- The last clause eliminates cases preceded by verbs like ***consider*** or ***believe*** which can take a noun and an adjective; this is to avoid tagging the following instance of ***that*** as an adverb:

Ex: I consider that odd.
- Another rule is used to express the constraint that the complementizer sense of ***that*** is most likely to be used if the previous word is a verb that expects a complement (like ***believe***, ***think***, or ***show***), and if ***that*** is followed by the beginning of a noun phrase, and a finite verb.

Stochastic Tagger

HMM (Hidden Markov Model) is a Stochastic technique for POS tagging

To find **the most probable tag sequence** given the observation sequence of n words w_1^n , that is, find $P(t_1^n|w_1^n)$ is highest.

But $P(t_1^n|w_1^n)$ is difficult to compute and Bayesian classification rule is used:

$$P(x|y) = P(x) P(y|x) / P(y)$$

When applied to the sequence of words, **the most probable tag sequence** would be

$$P(t_1^n|w_1^n) = P(t_1^n) P(w_1^n|t_1^n)/P(w_1^n)$$

where $P(w_1^n)$ does not change and thus do not need to be calculated

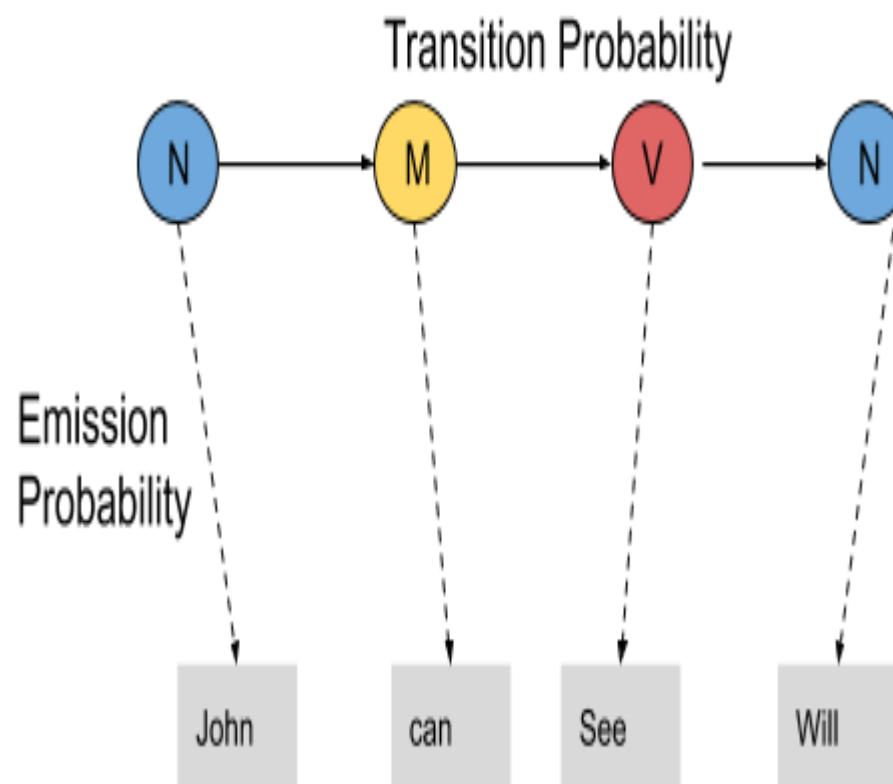
Thus, the **most probable tag sequence** is the product of two probabilities for each possible sequence:

- **Prior probability of the tag sequence. Context $P(t_1^n)$**
- **Likelihood of the sequence of words considering a sequence of (hidden) tags. $P(w_1^n|t_1^n)$**

Hidden Markov Model has the following values:

- **States:** Hidden States – POS Tags; i.e. VP, NP, JJ, DT
- **Emission Probability:** $P(w_i | t_i)$ – Probability that given a tag t_i , the word is w_i ,
e.g. $P(\text{book} | \text{NP})$ is the probability that the word **book** is a **Noun**.
- **Transition Probability Matrix:** $P(t_{i+1} | t_i)$ – Transition Probabilities from one tag t_i to another t_{i+1}
e.g. $P(\text{VP} | \text{NP})$ is the probability that current tag is **Verb** given previous tag is a **Noun**.
- **Observation:** Words
- **Initial Probability:** Probability of the initial word

Stochastic Tagger-An Example proposed by Dr. Luis Serrano



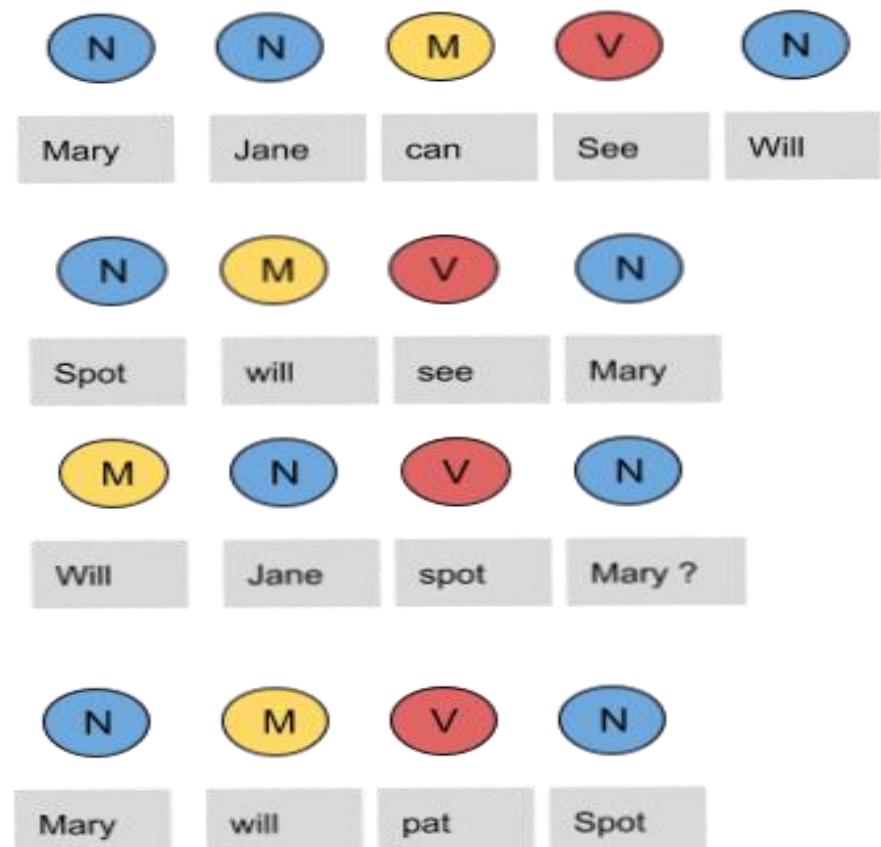
- In this example, we consider only 3 POS tags that are noun, model and verb
- Let the sentence “ **Ted will spot Will** ” be tagged as noun, model, verb and a noun
- To calculate the probability associated with this particular sequence of tags we require their **Transition probability** and **Emission probability**

Example

- The **transition probability** is the likelihood of a particular sequence for example, **how likely is that a noun is followed by a model and a model by a verb and a verb by a noun**
- It should be high for a particular sequence to be correct.
- Now, what is the probability that the word **Ted** is a noun, **will** is a model, **spot** is a verb and **Will** is a noun
- These sets of probabilities are **Emission probabilities** and should be high for our tagging to be likely

Example

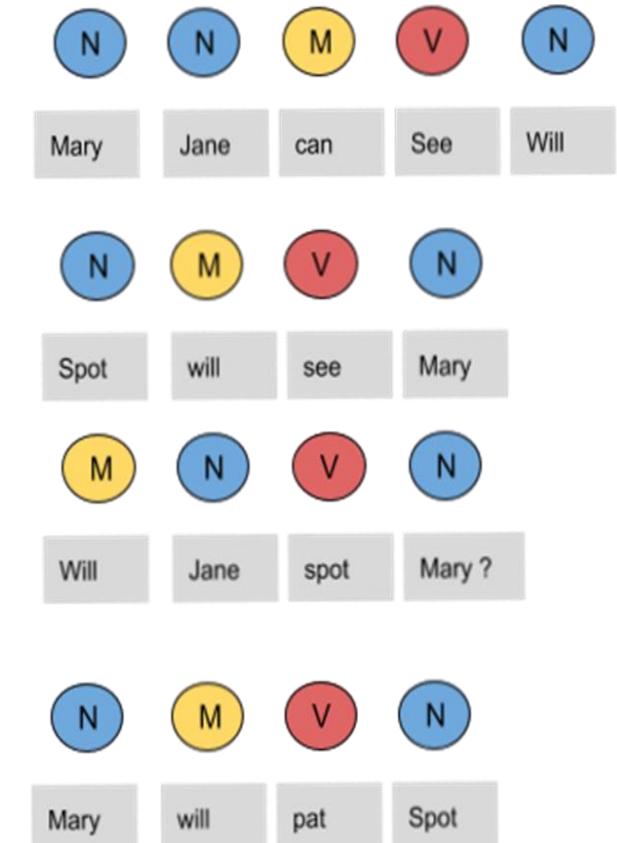
- Calculate the Emission and Transitional probabilities for the set of sentences below
- **Mary Jane can see Will**
- **Spot will see Mary**
- **Will Jane spot Mary?**
- **Mary will pat Spot**



Example

- To calculate the **emission probabilities**, create a **counting table** in a similar manner.

Words	Noun	Model	Verb
Mary	4	0	0
Jane	2	0	0
Will	1	3	0
Spot	2	0	1
Can	0	1	0
See	0	0	2
pat	0	0	1

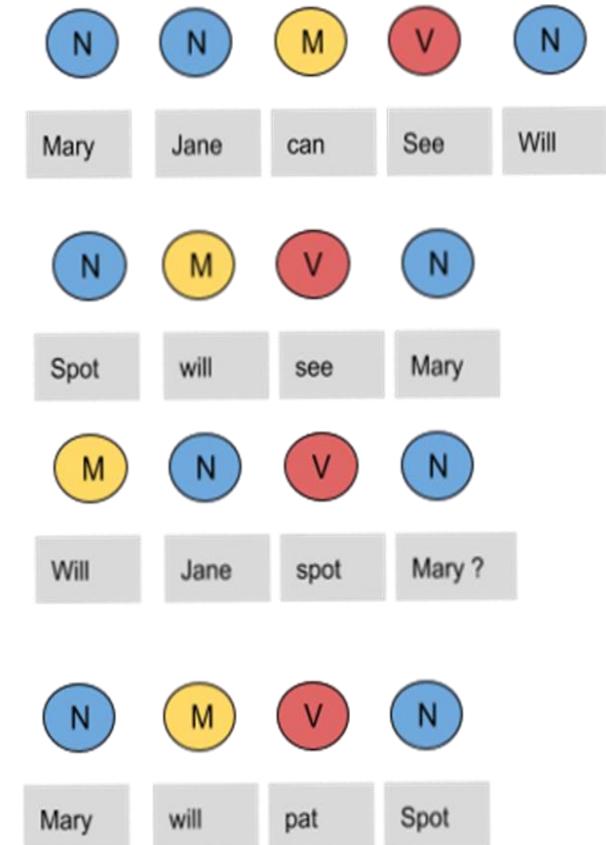


Example

Divide each column by the total number of their appearances

For example, 'noun' appears **nine times** in the above sentences so **divide** each term by **9** in the noun column. We get the following table after this operation.

Words	Noun	Model	Verb
Mary	4/9	0	0
Jane	2/9	0	0
Will	1/9	3/4	0
Spot	2/9	0	1/4
Can	0	1/4	0
See	0	0	2/4
pat	0	0	1



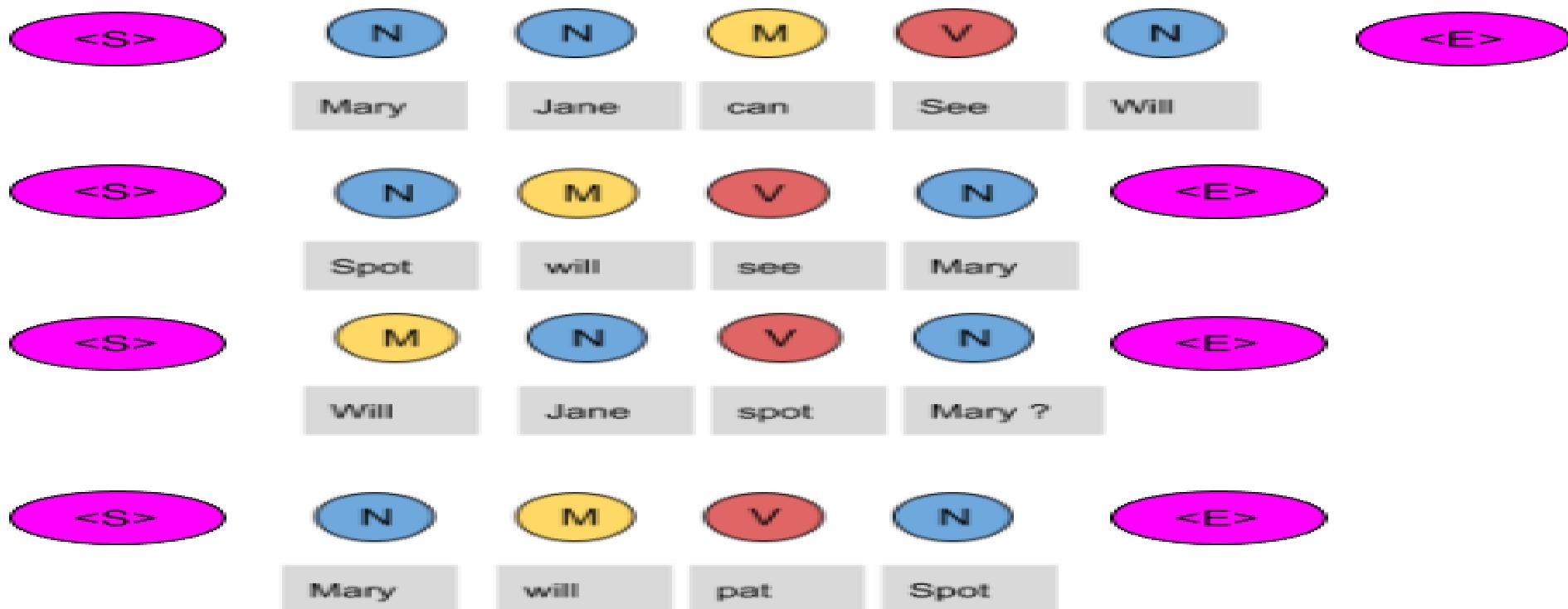
Example

From the above table, it is inferred as:

- The probability that Mary is Noun = $4/9$
- The probability that Mary is Model = 0
- The probability that Will is Noun = $1/9$
- The probability that Will is Model = $3/4$

Example

- Next, we have to calculate the transition probabilities, so define two more tags $\langle S \rangle$ and $\langle E \rangle$.



Example

- Create a table and fill it with the co-occurrence counts of the tags.

	N	M	V	<E>
<S>	3	1	0	0
N	1	3	1	4
M	1	0	3	0
V	4	0	0	0

Example

- In the above figure, we can see that the **<S>** tag is followed by the **N** tag **three** times, thus the first entry is **3**
- The **model** tag follows the **<S>** just once, thus the second entry is **1**
- In a similar manner, the rest of the table is filled.
- Next, we divide each term in a row of the table by the **total number of co-occurrences of the tag in consideration**
 - For example, The **Model tag** is followed by any other tag **four times** as shown below, thus we **divide each element in the third row by four**.

Example



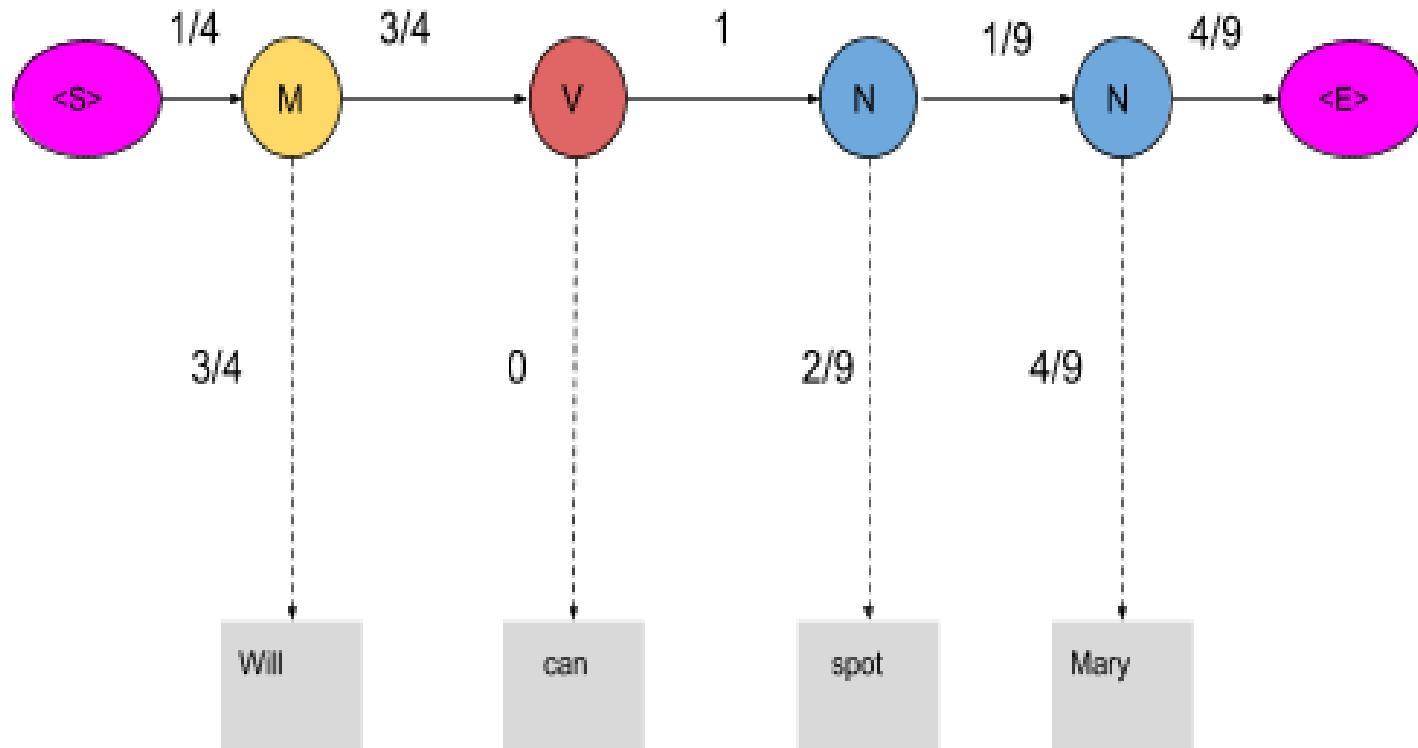
Example

	N	M	V	<E>
<S>	3/4	1/4	0	0
N	1/9	3/9	1/9	4/9
M	1/4	0	3/4	0
V	4/4	0	0	0

- These are the respective transition probabilities for the above four sentences

Example

- Now how does the HMM determine the **appropriate sequence of tags for a particular sentence from the above tables?** Let us find it out.
- Take a new sentence and tag them with wrong tags.
Let the sentence, '**Will can spot Mary**' be tagged as-
 - Will as a modal
 - Can as a verb
 - Spot as a noun
 - Mary as a noun



Words	Noun	Model	Verb
Mary	4/9	0	0
Jane	2/9	0	0
Will	1/9	3/4	0
Spot	2/9	0	1/4
Can	0	1/4	0
See	0	0	2/4
pat	0	0	1

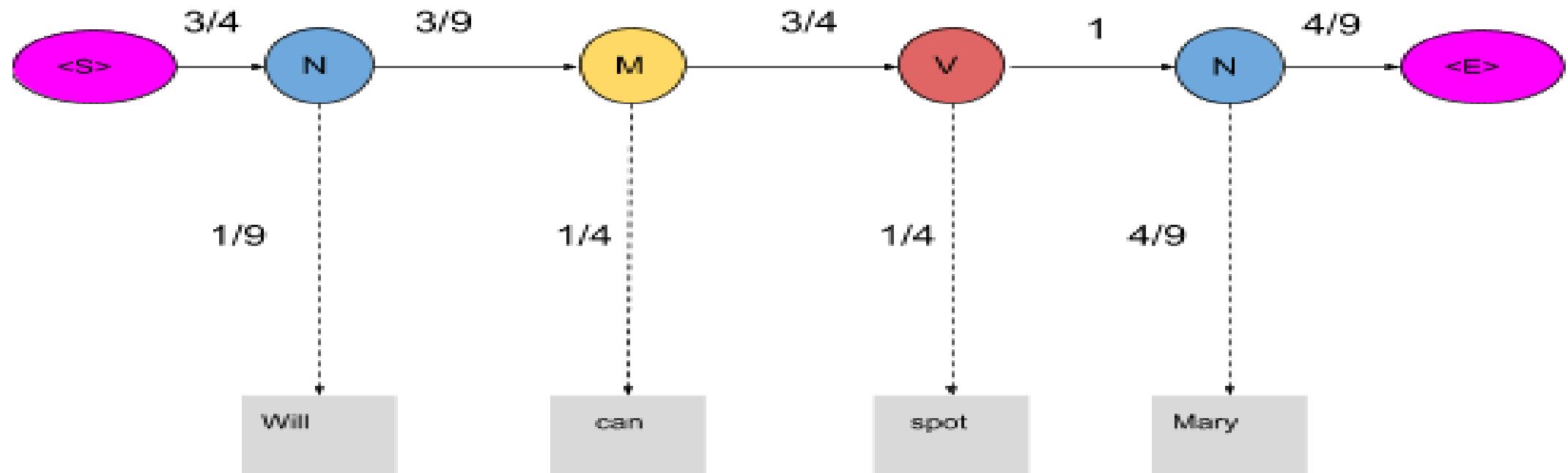
N M V <E>

- Now the product of these probabilities is the likelihood that this sequence is right
- Since the tags are not correct, the product is zero.
- $1/4 * 3/4 * 3/4 * 0 * 1 * 2/9 * 1/9 * 4/9 * 4/9 = 0$**

<S>	3/4	1/4	0	0
N	1/9	3/9	1/9	4/9
M	1/4	0	3/4	0
V	4/4	0	0	0

Example

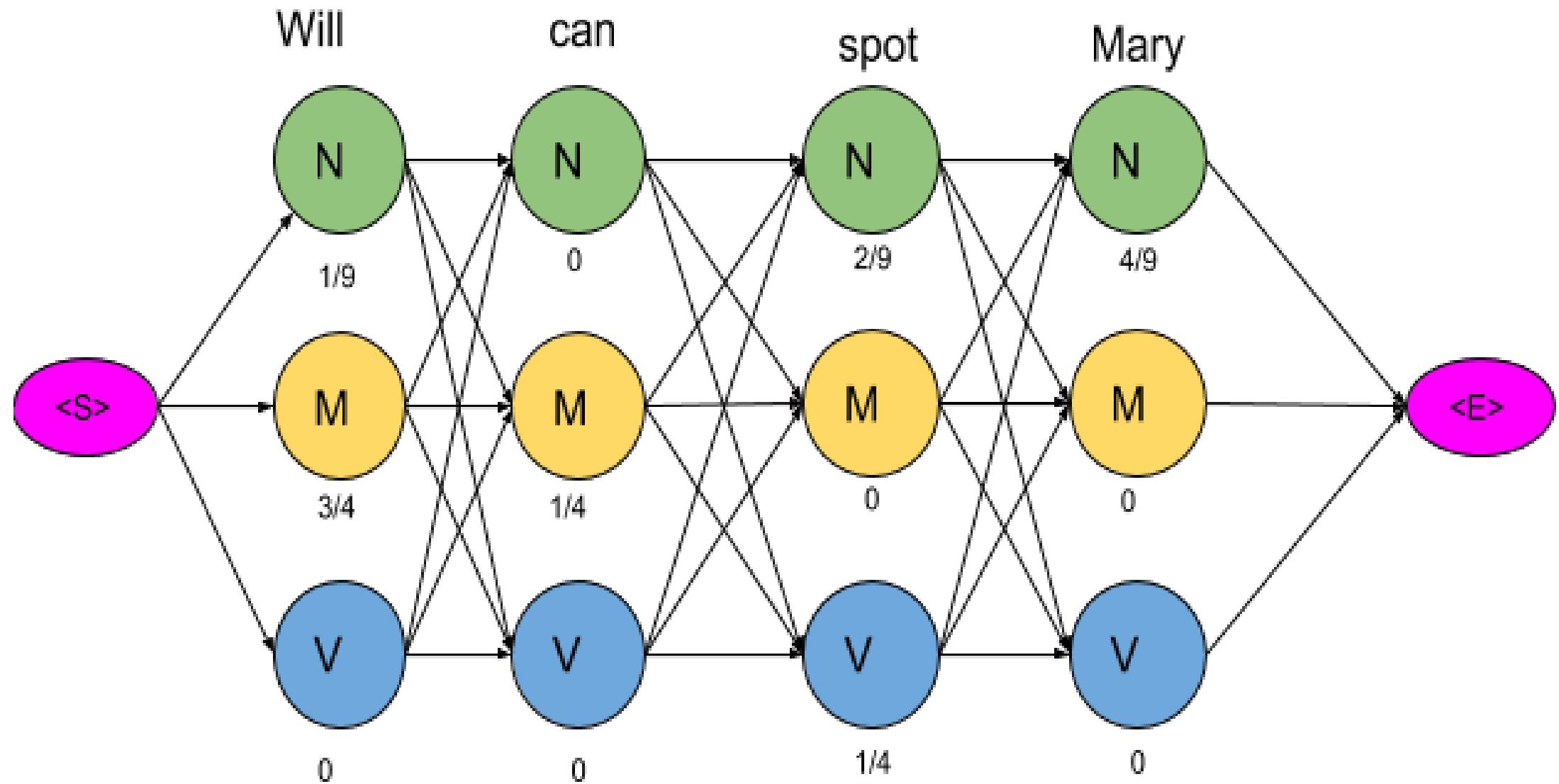
- When these words are correctly tagged, we get a probability greater than zero as shown below:



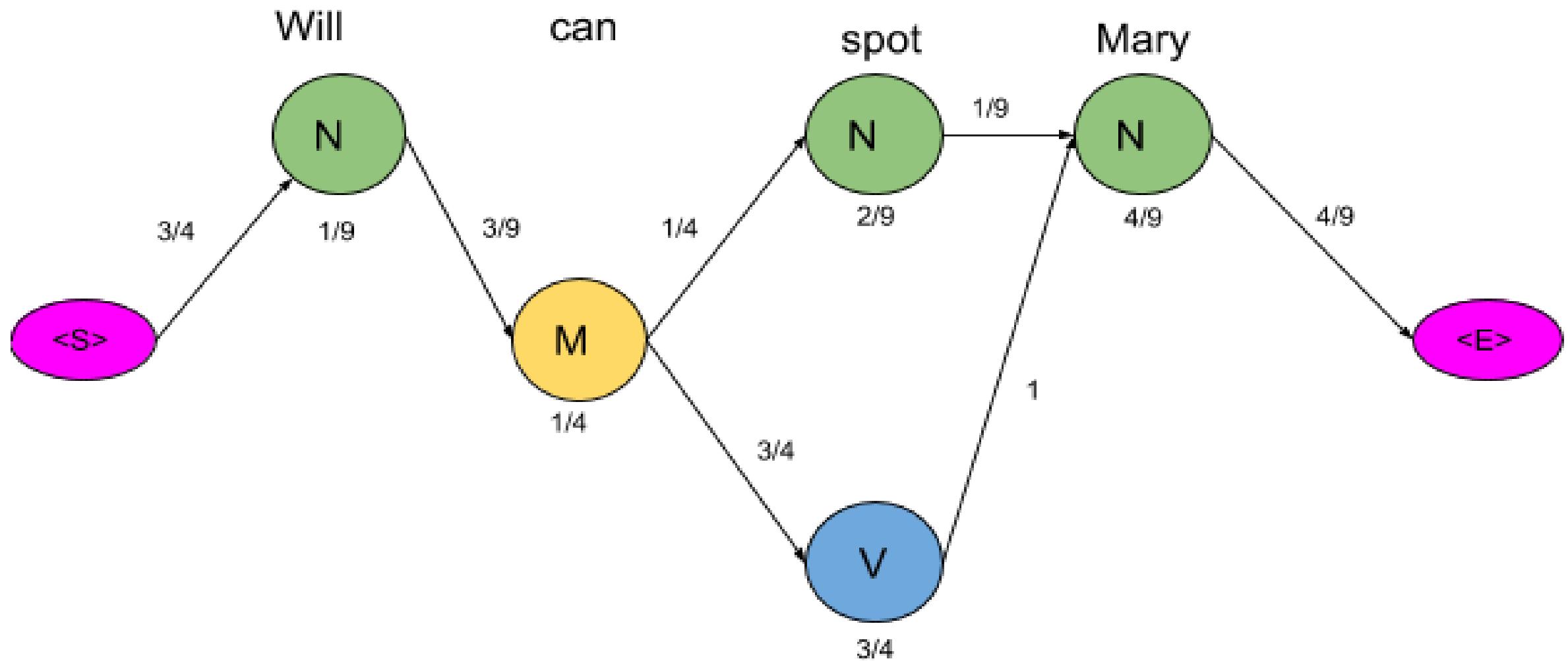
- Calculating the product of these terms we get,

$$3/4 * 1/9 * 3/9 * 1/4 * 3/4 * 1/4 * 1 * 4/9 * 4/9 = 0.00025720164$$

- For our example, keeping into consideration just three POS tags we have mentioned, 81 different combinations of tags can be formed
- In this case, calculating the probabilities of all 81 combinations seems achievable
- But when the task is to tag a larger sentence and all the POS tags in the Penn Treebank project are taken into consideration, the number of possible combinations grows exponentially, and this task seems impossible to achieve.
- let us visualize these 81 combinations as paths and using the transition and emission probability, mark each vertex and edge as shown in the figure



The next step is to delete all the vertices and edges with probability zero, also the vertices which do not lead to the endpoint are removed



Example

- Now there are only two paths that lead to the end, let us calculate the probability associated with each path.
- $\langle S \rangle \rightarrow N \rightarrow M \rightarrow N \rightarrow N \rightarrow \langle E \rangle = 3/4 * 1/9 * 3/9 * 1/4 * 1/4 * 2/9 * 1/9 * 4/9 * 4/9 = 0.00000846754$
- $\langle S \rangle \rightarrow N \rightarrow M \rightarrow N \rightarrow V \rightarrow \langle E \rangle = 3/4 * 1/9 * 3/9 * 1/4 * 3/4 * 1/4 * 1 * 4/9 * 4/9 = 0.00025720164$
- **Clearly, the probability of the second sequence is much higher and hence the HMM is going to tag each word in the sentence according to this sequence.**

- Given an HMM and an input string, the **Viterbi algorithm** is used to decode the optimal tag sequence
- For any model, such as an HMM, that contains hidden variables, the task of determining which sequence of variables is the underlying source of some sequence of observations is called the **decoding** task.
- The **Viterbi** algorithm is the most common decoding algorithm used for HMMs, whether for part-of-speech tagging or for speech recognition.
- The term **Viterbi** is common in speech and language processing, but this is a standard application of the classic **dynamic programming** algorithm, and looks a lot like the **minimum edit distance** algorithm

Formal definition of HMM

- The goal of HMM decoding is to choose the tag sequence that is most probable given the observation sequence of n words w_1^n :

$$\hat{t}_1^n = \operatorname{argmax}_{t_1^n} P(t_1^n | w_1^n) \quad (10.4)$$

by using Bayes' rule to instead compute:

$$\hat{t}_1^n = \operatorname{argmax}_{t_1^n} \frac{P(w_1^n | t_1^n)P(t_1^n)}{P(w_1^n)} \quad (10.5)$$

Furthermore, we simplify Eq. 10.5 by dropping the denominator $P(w_1^n)$:

$$\hat{t}_1^n = \operatorname{argmax}_{t_1^n} P(w_1^n | t_1^n)P(t_1^n) \quad (10.6)$$

HMM taggers make two further simplifying assumptions. The first is that the probability of a word appearing depends only on its own tag and is independent of neighboring words and tags:

$$P(w_1^n | t_1^n) \approx \prod_{i=1}^n P(w_i | t_i) \quad (10.7)$$

Formal definition of HMM

The second assumption, the **bigram** assumption, is that the probability of a tag is dependent only on the previous tag, rather than the entire tag sequence;

$$P(t_1^n) \approx \prod_{i=1}^n P(t_i | t_{i-1}) \quad (10.8)$$

Plugging the simplifying assumptions from Eq. 10.7 and Eq. 10.8 into Eq. 10.6 results in the following equation for the most probable tag sequence from a bigram tagger, which as we will soon see, correspond to the **emission probability** and **transition probability** from the HMM of Chapter 9.

$$\hat{t}_1^n = \operatorname{argmax}_{t_1^n} P(t_1^n | w_1^n) \approx \operatorname{argmax} \prod_{i=1}^n \overbrace{P(w_i | t_i)}^{\text{emission}} \overbrace{P(t_i | t_{i-1})}^{\text{transition}} \quad (10.9)$$

Transformation-Based Tagging

- Also called **Brill tagging**, is an instance of the **Transformation-Based Learning** (TBL) approach to machine learning (Brill,1995)
- It draws inspiration from both the rule-based and stochastic taggers.
- Like the rule-based taggers, TBL is based on rules that specify what tags should be assigned to what words.
- But like the stochastic taggers, TBL is a machine learning technique, in which rules are automatically induced from the data.
- Like some but not all of the HMM taggers, TBL is a supervised learning technique; it assumes a pre-tagged training corpus

TBL Concept

- The TBL algorithm has a set of tagging rules.
- A corpus is first tagged using the broadest rule, that is, the one that applies to the most cases.
- Then a slightly more specific rule is chosen, which changes some of the original tags.
- Next an even narrower rule, which changes a smaller number of tags (some of which might be previously changed tags).

How TBL Rules Are Applied?

- Before the rules apply, the tagger labels every word with its most-likely tag. We get these most-likely tags from a tagged corpus.
- For example, in the Brown corpus, ***race*** is most likely to be a noun:

$$P(\text{NN} \mid \text{race}) = .98$$

$$P(\text{VB} \mid \text{race}) = .02$$

- This means that the two examples of ***race*** that we saw above will both be coded as **NN**.
- In the first case, this is a mistake, as NN is the incorrect tag:
is/VBZ expected/VBN to/TO race/NN tomorrow/NN
- In the second case this *race* is correctly tagged as an NN:
the/DT race/NN for/IN outer/JJ space/NN

- After selecting the most-likely tag, Brill's tagger applies its transformation rules.
- As it happens, Brill's tagger learned a rule that applies exactly to this mistagging of ***race***
Change NN to VB when the previous tag is TO
- This rule would change ***race/NN*** to ***race/VB*** in exactly the following situation since it is preceded by ***to/TO***

expected/VBN to/TO race/NN → expected/VBN to/TO race/VB

How TBL rules are learned?

Brill's TBL has three major stages:

1. It first labels every word with its most likely tag.
 2. It then examines every possible transformation and selects the one that results in the most improved tagging.
 3. Finally, it then re-tags the data according to this rule.
-
- The last two stages are repeated until some stopping criterion is reached, such as insufficient improvement over the previous pass.
 - Note that stage two requires that TBL knows the correct tag of each word; that is, TBL is a supervised learning algorithm.
 - The output of the TBL process is an ordered list of transformations; these then constitute a “**tagging procedure**” that can be applied to a new corpus.

Challenges in POS Tagging:

- The design of the training set or **training corpus** needs to be carefully considered.
 - If the training corpus is too specific to the task or domain, the probabilities may be too narrow and not generalize well to tagging sentences in very different domains.
 - But if the training corpus is too general, the probabilities may not do a sufficient job of reflecting the task or domain.
- The problem with having a fixed training set, development set, and test set is that in order to save lots of data for training, the test set might not be large enough to be representative.
 - Thus a better approach would be to somehow use **all** our data both for training and test
 - The idea is to use **cross-validation**.
 - In cross-validation, we randomly choose a training and test set division of our data, train our tagger, and then compute the error rate on the test set.
 - Generally, 10-cross validation is performed

- Tag Indeterminacy and Tokenization
 - Tag indeterminacy arises when a word is ambiguous between multiple tags and it is impossible or very difficult to disambiguate
 - Common tag indeterminacies include adjective versus preterite versus past participle (JJ/VBD/VBN), and adjective versus noun as a prenominal modifier (JJ/NN).
- Unknown Words
 - All the tagging algorithms we have discussed require a dictionary that lists the possible parts-of-speech of every word.
 - But the largest dictionary will still not contain every possible word
 - Therefore, in order to build a complete tagger we cannot always use a dictionary to give us $p(w|t)$. We need some method for guessing the tag of an unknown word.

END

Unit 8

Context Free Grammar for English

- Geoff Pullum noted in a talk that “almost everything most educated Americans believe about English grammar is wrong”.
- In this chapter we make a preliminary stab at addressing some of these gaps in our knowledge of grammar and syntax, as well as introducing some of the formal mechanisms that are available for capturing this knowledge
- The word **syntax** comes from the Greek *sýntaxis*, meaning SYNTAX “setting out together or arrangement”,
 - refers to the way words are arranged together.

- Various syntactic notions are seen till now:
 - The regular languages - offered a simple way to represent the ordering of strings of words
 - Compute probabilities - for these word sequences.
 - Part-of-speech categories - could act as a kind of equivalence class for words
- The present one introduces sophisticated notions of syntax and grammar that go well beyond these simpler notions.
- Three main new ideas are introduced in this chapter:
 - **Constituency**
 - **Grammatical relations**
 - **Subcategorization and dependency**

Constituency

- The fundamental idea of constituency is that groups of words may behave as a single unit or phrase, called a **constituent**.
 - For example we will see that a group of words called a **noun phrase** often acts as a unit;
 - noun phrases include single words like *she* or *Michael* and phrases like *the house*, *Russian Hill*, and *a well-weathered, three-story structure*.

- How do words group together in English? Consider the **noun phrase**, a sequence of words surrounding at least one noun.
 - Examples of noun phrases:
 - three parties from Brooklyn
 - Harry the Horse
 - a high-class spot such as Mindy's
 - the Broadway coppers
 - they
- How do we know that these words group together (or “form constituents”)? **One piece of evidence is that they can all appear in similar syntactic environments, for example before a verb.**
 - three parties from Brooklyn *arrive*...
 - a high-class spot such as Mindy's *attracts*...
 - the Broadway coppers *love*...
 - they *sit*

- But while the whole noun phrase can occur before a verb, this is not true of each of the individual words that make up a noun phrase.
 - Thus to correctly describe facts about the ordering of these words in English, we must be able to say things like “**Noun Phrases can occur before verbs**”.
-
- Other kinds of evidence for constituency come from what are called **preposed** or **postposed** constructions.
 - For example, the prepositional phrase ***on September seventeenth*** can be placed in a number of different locations in the following examples, including **preposed at the beginning, and postposed at the end**:

On September seventeenth, I'd like to fly from Atlanta to Denver
I'd like to fly *on September seventeenth* from Atlanta to Denver
I'd like to fly from Atlanta to Denver *on September seventeenth*

- But again, while the entire phrase can be placed differently, the individual words making up the phrase cannot be:
 - * **On September**, I'd like to fly **seventeenth** from Atlanta to Denver
 - * **On** I'd like to fly **September seventeenth** from Atlanta to Denver
 - * I'd like to fly **on September** from Atlanta to Denver **seventeenth**
- This chapter will introduce the use of **context-free grammars**, a formalism that will allow us to model these constituency facts.

Grammatical relations

- Grammatical relations are a formalization of ideas from traditional grammar such as SUBJECTS and OBJECTS, and other related notions.
- In the following sentence the noun phrase ***She*** is the SUBJECT and ***a mammoth breakfast*** is the OBJECT:
 - ***She ate a mammoth breakfast***

Subcategorization and dependency relations

- **Subcategorization and dependency relations** refer to certain kinds of relations between words and phrases.
- For example the verb ***want*** can be followed by an infinitive, as in ***I want to fly to Detroit***, or a noun phrase, as in ***I want a flight to Detroit***.
- But the verb ***find*** cannot be followed by an infinitive (****I found to fly to Dallas***). These are called facts about the ***subcategorization*** of the verb

- None of the syntactic mechanisms that we've discussed up until now can easily capture such phenomena.
- They can be modeled much more naturally by grammars that are based on **context-free grammars (CFG)**
- CFG is integral to many computational applications including grammar checking, semantic interpretation, dialogue understanding, and machine translation.
- They are powerful enough to express sophisticated relations among the words in a sentence and efficient algorithms exist for parsing sentences

Context-Free Grammar

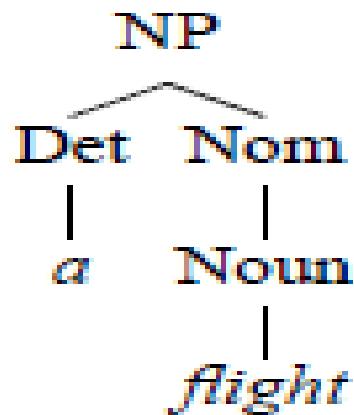
- The most commonly used mathematical system for modeling constituent structure in English and other natural languages is the **Context-Free Grammar**, or **CFG**.
- Context free grammars are also called **Phrase-Structure Grammars**, and the formalism is equivalent to what is also called **Backus-Naur 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 symbols. For example, the following productions express that a **NP (or noun phrase)**, can be composed of either a **Proper Noun or a determiner (Det)** followed by a **Nominal**; a Nominal can be one or more Nouns.
 - $NP \rightarrow Det\ Nominal$
 - $NP \rightarrow ProperNoun$
 - $Nominal \rightarrow Noun\ | Nominal\ Noun$

- Context-free rules can be **hierarchically embedded**, so we can combine the previous rules with others like the following which express facts about the lexicon:
 - $\text{Det} \rightarrow a$
 - $\text{Det} \rightarrow \text{the}$
 - $\text{Noun} \rightarrow \text{flight}$
- The symbols that are used in a CFG are divided into two classes.
 - 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
 - The symbols that express clusters or generalizations of these are called ***non-terminals***

- In each context free rule, the item to the right of the **arrow (\rightarrow)** is an ordered list of **one or more terminals and non-terminals**, while to the left of the arrow is a **single non-terminal symbol** expressing some cluster or generalization.
- Notice that in the lexicon, the non-terminal associated with each word is its lexical category, or part-of-speech

- A CFG can be thought of in two ways:
 - as a device for generating sentences, and
 - as a device for assigning a structure to a given sentence.
- As a generator, we can read the →arrow as “rewrite the symbol on the left with the string of symbols on the right
 - The string **a flight** can be derived from the non-terminal NP
 - 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** (commonly shown inverted with the root at the top)

- In the parse tree for *a flight* , we say that the node ***NP*** **immediately dominates** the node ***Det*** and the node ***Nom***.
- We say that the node ***NP*** **dominates** all the nodes in the tree (*Det*, *Nom*, *Noun*, *a*, *flight*).
- The formal language defined by a CFG is the set of strings that are derivable from the designated **start symbol**.



- Each grammar must have one designated start symbol which is often called S .
- Since context-free grammars are often used to define sentences, S is usually interpreted as the “**sentence**” node, and the set of strings that are derivable from S is the set of sentences in some simplified version of English.

- A sentence can consist of a noun phrase followed by a **verb phrase**
- A verb phrase in English consists of a verb followed by assorted other things; for example, one kind of verb phrase consists of a verb followed by a noun phrase
- Or the verb phrase may have a verb followed by a noun phrase and a prepositional phrase
- Or the verb may be followed by a prepositional phrase alone
- A prepositional phrase generally has a preposition followed by a noun phrase

$S \rightarrow NP VP$

$VP \rightarrow Verb\ NP$

$VP \rightarrow Verb\ NP\ PP$

$VP \rightarrow Verb\ PP$

$PP \rightarrow Preposition\ NP$

I prefer a morning flight

prefer a morning flight

leave Boston in the morning

leaving on Thursday

from Los Angeles (locations, times, dates)

Context-Free Grammer Example

Noun → flights | breeze | trip | morning | ...

Verb → is | prefer | like | need | want | fly

Adjective → cheapest | non-stop | first | latest
| other | direct | ...

Pronoun → me | I | you | it | ...

Proper-Noun → Alaska | Baltimore | Los Angeles
| Chicago | United | American | ...

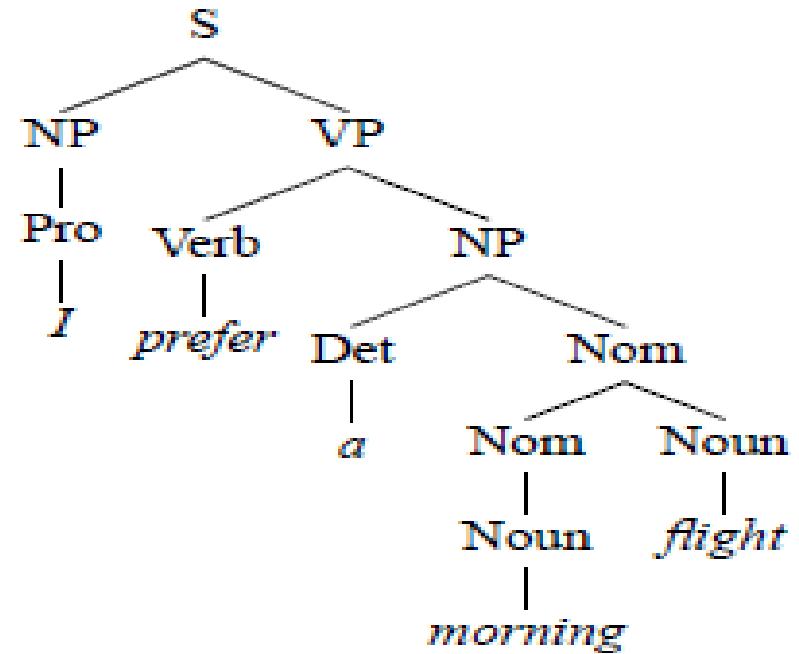
Determiner → the | a | an | this | these | that | ...

Preposition → from | to | on | near | ...

Conjunction → and | or | but | ...

Ten examples from the ATIS corpus (L_0)

$S \rightarrow NP VP$	I + want a morning flight
$NP \rightarrow$ Pronoun Proper-Noun Det Nominal $Nominal \rightarrow$ Nominal Noun Noun	I
	Los Angeles
	a + flight
	morning + flight
$VP \rightarrow$ Verb Verb NP Verb NP PP Verb PP	flights
	do
	want + a flight
	leave + Boston + in the morning
$PP \rightarrow$ Preposition NP	leaving + on Thursday
	from + Los Angeles



The grammar for L_0 , with example phrases for each rule.

The parse tree for “I prefer a morning flight” according to grammar L_0 .

- It is sometimes convenient to represent a parse tree in a more compact format called bracketed notation, essentially the same as LISP tree representations
- Example: The bracketed representation of the parse tree (previous slide)

[S [NP [Pro I]] [VP [V prefer] [NP [Det a] [Nom [N morning] [Nom [N flight]]]]]]]

- A CFG like that of L_0 defines a formal language.
- A formal language is a set of strings
- 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, are referred to as ***ungrammatical***
- In linguistics, the use of formal languages to model natural languages is called ***generative grammar***, since the language is defined by the set of possible sentences “generated” by the grammar.

Formal definition of context-free grammar

- A context-free grammar G is defined by four parameters N, Σ, P, S (technically “is a 4-tuple”):
 - N a set of **non-terminal symbols (or variables)**
 - Σ a set of **terminal symbols (disjoint from N)**
 - P a set of **rules** or productions, each of the form $A \rightarrow \beta$, where A is a nonterminal, β is a string of symbols from the infinite set of strings $(\Sigma \cup N)^*$
 - S is a designated **start symbol**

Formal definition of context-free grammar

Convention:

Capital letters like A, B, and S	Non-Terminals
S	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

- A language is defined via the concept of **derivation**
- One string derives another one if it can be rewritten as the second one via some series of **rule applications**
- If $A \rightarrow \beta$ is a production of P and α and γ are any strings in the set $(\Sigma \cup N)^*$, then we say that $\alpha A \gamma$ **directly derives** $\alpha \beta \gamma$, or $\alpha A \gamma \Rightarrow \alpha \beta \gamma$
- Derivation is then a generalization of direct derivation:
 Let $\alpha_1, \alpha_2, \dots, \alpha_m$ be strings in $(\Sigma \cup N)^*$, $m \geq 1$, such that $\alpha_1 \Rightarrow \alpha_2, \alpha_2 \Rightarrow \alpha_3, \dots, \alpha_{m-1} \Rightarrow \alpha_m$
 We say that **α_1 derives α_m , or $\alpha_1 * \Rightarrow \alpha_m$.**
- The language L_G generated by a grammar G can be defined as the set of strings composed of terminal symbols which can be derived from the designated start symbol S .

$$L_G = \{w \mid w \text{ is in } \Sigma^* \text{ and } S * \Rightarrow w\}$$
- The problem of mapping from a string of words to its parse tree is called **parsing**

SOME GRAMMAR RULES FOR ENGLISH

Sentence-Level Constructions

- There are a large number of constructions for English sentences, but four are particularly common and important:
- **Declarative structure:** Structure has a subject noun phrase followed by a verb phrase.
Eg: “*I prefer a morning flight*”.
 - *Subject NP+VP*
- **Imperative structure:** Structure often begins with a verb phrase, and have no subject. They are called imperative because they are almost always used for commands and suggestions.
Eg: “*Show the lowest fare*”.
 - $S \rightarrow VP$
- **Yes-no-question structure:** Structure are often (though not always) used to ask questions (hence the name), and begin with an auxiliary verb, followed by a subject NP, followed by a VP.
Eg: “*Do any of these flights have stops?*”.
 - $S \rightarrow Aux\ NP\ VP$

- ***Wh-question structure***: These may be broadly grouped into two classes of sentence-level structures.
- The **wh-subject-question** structure is identical to the declarative structure, except that the first noun phrase contains some wh-word.
 Eg: “*Whose flights serve breakfast?*”
 - $S \rightarrow Wh-NP VP$
- In the **wh-non-subject question** structure, the wh-phrase is not the subject of the sentence, and so the sentence includes another subject.
 - In these types of sentences the auxiliary appears before the subject NP, just as in the yes-no-question structures.
 Eg: “*What flights do you have from Burbank to Tacoma Washington?*”
 - $S \rightarrow Wh-NP Aux NP VP$

Clauses and Sentences

- ***Clause:*** A clause is the smallest grammatical unit that can express a complete proposition. In traditional grammar, Clauses are often described as forming a complete thought.
- ***Sentences:*** It may form larger structures. **S** may be in the right side of the production

The Noun Phrase

- Noun phrases consisting of a head, the central noun in the noun phrase, along with various modifiers that can occur before or after the head noun. Let's take a close look at the various parts.
- **The Determiners:**
 - Noun phrases can begin with simple lexical determiners
Examples: a stop, the flights, this flight, those flights, any flights, some flights
 - Determiners are optional in English.
 - For example, determiners may be omitted if the noun they modify is plural:
Show me *flights* from San Francisco to Denver on weekdays
 - **Mass nouns** also don't require determination
 - substance like **water** and **snow**, don't take the indefinite article “*a*”, and don't tend to pluralize.
 - Many abstract nouns are mass nouns (***music, homework***).

Noun phrase

- **The Nominal**
- The nominal construction follows the determiner and contains any pre- and post-head noun modifiers.
- As indicated in grammar L_0 , in its simplest form a nominal can consist of a single noun.
 - *Nominal → Noun*
- As we'll see, this rule also provides the basis for the bottom of various recursive rules used to capture more complex nominal constructions.

The Noun Phrase: Before Noun head

- **Predeterminers:**
 - Word classes appearing in the NP before the determiner
 - *all the flights, all flights*
- **Postdeterminers:**
 - Word classes appearing in the NP between the determiner and the head noun
 - **Cardinal numbers:** *two friends, one stop*
 - **Ordinal numbers:** *the first one, the next day, the second leg, the last flight, the other American flight, and other fares*
 - **Quantifiers:** *many fares*
 - The quantifiers, *much* and *a little* occur only with noncount nouns.

The Noun Phrase: Before Noun head

- Adjectives occur after quantifiers but before nouns.
 - *a first-class fare, a nonstop flight, the longest layover, the earliest lunch flight*
- Adjectives can be grouped into a phrase called an **adjective phrase** or **AP**.
 - AP can have an adverb before the adjective
 - *the least expensive fare*
- $NP \rightarrow (Det)(Card)(Ord)(Quant)(AP) Nominal$

The Noun Phrase: After Noun head

The Noun Phrase: After Noun head

- The three most common kinds of **non-finite** postmodifiers are the gerundive (-*ing*), -*ed*, and infinitive form.
 - A gerundive consists of a VP begins with the gerundive (-*ing*)
 - *any of those [leaving on Thursday]*
 - *any flights [arriving after eleven a.m.]*
 - *flights [arriving within thirty minutes of each other]*
 - Examples of two other common kinds
 - *the last flight to arrive in Boston*
 - *I need to have dinner served*
 - *Which is the aircraft used by this flight?*

Nominal → Nominal GerundVP

GerundVP → GerundV NP | GerundV PP | GerundV | GerundV NP PP

GerundV → being | preferring | arriving | leaving | ...

The Noun Phrase: After Noun head

- A postnominal relative clause (more correctly a **restrictive relative clause**)
 - is a clause that often begins with a **relative pronoun** (*that* and *who* are the most common).
 - The relative pronoun functions as the subject of the embedded verb,
 - *a flight that serves breakfast*
 - *flights that leave in the morning*
 - *the United flight that arrives in San Jose around ten p.m.*
 - *the one that leaves at ten thirty five*

Nominal → Nominal RelClause

RelClause → (who | that) VP

Some grammar rules for English: Coordination

- NPs and other units can be **conjoined** with **coordinations** like *and*, *or*, and *but*.
 - *Please repeat [NP [NP the flight] and [NP the coast]]*
 - *I need to know [NP [NP the aircraft] and [NP flight number]]*
 - *I would like to fly from Denver stopping in [NP [NP Pittsburgh] and [NP Atlanta]]*
 - $NP \rightarrow NP \ and \ NP$
 - $VP \rightarrow VP \ and \ VP$
 - $S \rightarrow S \ and \ S$

Some grammar rules for English: Verb Phrase and Subcategorization

- The VP consists of the verb and a number of other constituents.

$VP \rightarrow Verb$

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

- An entire embedded sentence, called **sentential complement**, can follow the verb.

You [VP [V said [S there were two flights that were the cheapest]]]

You [VP [V said [S you had a two hundred sixty six dollar fare]]]

[VP [V Tell] [NP me] [S how to get from the airport in Philadelphia to downtown]]

I [VP [V think [S I would like to take the nine thirty flight]]]

$VP \rightarrow Verb\ S$

Some grammar rules for English: Verb Phrase and Subcategorization

- Another potential constituent of the VP is another VP
 - Often the case for verbs like *want*, *would like*, *try*, *intend*, *need*

I want [_{VP} to fly from Milwaukee to Orlando]

Hi, I want [_{VP} to arrange three flights]

Hello, I'm trying [_{VP} to find a flight that goes from Pittsburgh to Denver after two p.m.]

- Recall that verbs can also be followed by *particles*, words that resemble a preposition but that combine with the verb to form a *phrasal verb*, like *take off*.
 - These particles are generally considered to be an integral part of the verb in a way that other post-verbal elements are not;
 - Phrasal verbs are treated as individual verbs composed of two words.

Some grammar rules for English: Verb Phrase and Subcategorization

- A VP can have many possible kinds of constituents, not every verb is compatible with every VP.
 - *I want a flight ...*
 - *I want to fly to ...*
 - **I found to fly to Dallas.*
- The idea that verbs are compatible with different kinds of complements
 - Traditional grammar **subcategorize** verbs into two categories (transitive and intransitive).
 - Modern grammars distinguish as many as 100 subcategories

Frame	Verb	Example
\emptyset	eat, sleep	I want to eat
NP	prefer, find leave	Find [NP the flight from Pittsburgh to Boston]
$NP\ NP$	show, give, find	Show [NP me] [NP airlines with flights from Pittsburgh]
$PP_{from}\ PP_{to}$	fly, travel	I would like to fly [pp from Boston] [pp to Philadelphia]
$NP\ PP_{with}$	help, load	Can you help [NP me] [pp with a flight]
VP_{to}	prefer, want, need	I would prefer [VP_{to} to go by United airlines]
VP_{brst}	can, would, might	I can [VP_{brst} fo from Boston]

Grammar Equivalence and Normal Form

- A formal language is defined as a (possibly infinite) set of strings of words
- Two grammars are equivalent if they generate the same set of strings
- Two kinds of grammar equivalence:
 - **Strong equivalence** - Two grammars are strongly equivalent if they generate the same set of strings and assign the same phrase structure to each sentence (allowing merely renaming of the non-terminal symbols).
 - **Weak equivalence** - Two grammars are weakly equivalent if they generate the same set of strings but do not assign the same phrase structure to each sentence.

Grammar Equivalence and Normal form

- It is sometimes useful to have a **normal form** for grammars, in which each of the productions takes a particular form.
- A context-free grammar is in **Chomsky Normal Form (CNF)** (Chomsky, 1963)
 - if it is ϵ -free and if in addition each production is either of the form $A \rightarrow BC$ or $A \rightarrow a$.
 - That is, the right-hand side of each rule either has **two non-terminal symbols or one terminal symbol**.

Chomsky normal form grammars are **binary branching**, i.e. they have binary trees.

Grammar Equivalence and Normal form

- For example, a rule of the form
 - $A \rightarrow B C D$
- can be converted into the following weakly equivalence CNF rules :
 - $A \rightarrow X D$
 - $X \rightarrow B C$
- Sometimes using binary branching can actually produce smaller grammars.

Parsing with Context Free Grammars

Parsing as Searching

- Unit 1 and unit 2 showed that finding the right path through a finite-state automaton or finding the right transduction for an input, can be viewed as a search problem.
- For finite-state automata, the search is through the space of all possible paths through a machine.
- In syntactic parsing, the parser can be viewed as searching through the space of possible parse trees to find the correct parse tree for a given sentence.
- Just as the search space of possible paths was defined by the structure of an automata, so the search space of possible parse trees is defined by grammar.

- Parse trees are directly useful in applications such as grammar checking in word-processing systems; a sentence which cannot be parsed may have grammatical errors (or at least be hard to read).
- Typically, parse trees serve as an important intermediate stage of representation for semantic analysis, and thus plays an important role in applications like **machine translation**, **question answering** and **information extraction**.
- Parsing algorithms specify how to recognize the strings of a language and assign each string to one (or more) syntactic analyses

What is Parsing?

- The process of taking a string and a grammar and returning all possible parse trees for that string
- That is, find all trees, whose root is the start symbol S , which cover exactly the words in the input
- There are two kinds of constraints that should help for parsing.
 - One set of constraints comes from the data, that is, the input sentence itself. Whatever else is true of the final parse tree, we know that there must be three leaves and they must be the words ***book***, ***that***, and ***flight***.
 - The second kind of constraint comes from the grammar. We know that whatever else is true of the final parse tree, it must have one root, which must be the **start symbol S** .

Parsing: Example

$S \rightarrow NP VP$

$S \rightarrow Aux NP VP$

$S \rightarrow VP$

$NP \rightarrow Pronoun$

$NP \rightarrow Proper-Noun$

$NP \rightarrow Det Nominal$

$Nominal \rightarrow Noun$

$Nominal \rightarrow Nominal Noun$

$Nominal \rightarrow Nominal PP$

$VP \rightarrow Verb$

$VP \rightarrow Verb NP$

$VP \rightarrow Verb NP PP$

$VP \rightarrow Verb PP$

$VP \rightarrow VP PP$

$PP \rightarrow Preposition NP$

$Det \rightarrow that | this | a$

$Noun \rightarrow book | flight | meal | money$

$Verb \rightarrow book | include | prefer$

$Pronoun \rightarrow I | she | me$

$Proper-Noun \rightarrow Houston | TWA$

$Aux \rightarrow does$

$Preposition \rightarrow from | to | on | near | through$

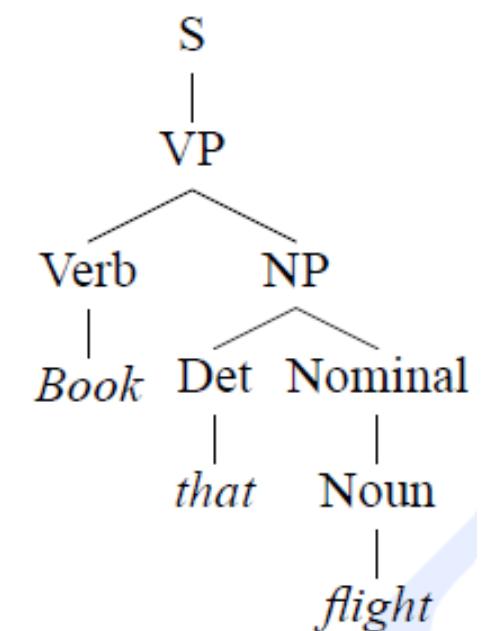


Figure : The \mathcal{L}_1 miniature English grammar and lexicon.

Types of Parsing

These two constraints give rise to two search strategies underlying most parsers:

- **Top-down parsing or goal-directed search:**
 - Builds from the root S node to the leaves
- **Bottom-up parsing or data-directed search.**
 - Parser begins with words of input and builds up trees, applying grammar rules whose RHS matches.

Top-down parsing

- Searches for a parse tree by trying to build from the **root node S** down to the **leaves**.
- Let's consider the search space that a top-down parser explores, assuming for the moment that it builds all possible trees in parallel.
- 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.
- Trees are grown downward until they eventually reach the part-of-speech categories at the bottom of the tree.
- At this point, trees whose leaves fail to match all the words in the input can be rejected

Top-down Parsing Example

$S \rightarrow NP VP$

$S \rightarrow Aux NP VP$

$S \rightarrow VP$

$NP \rightarrow Pronoun$

$NP \rightarrow Proper-Noun$

$NP \rightarrow Det Nominal$

$Nominal \rightarrow Noun$

$Nominal \rightarrow Nominal Noun$

$Nominal \rightarrow Nominal PP$

$VP \rightarrow Verb$

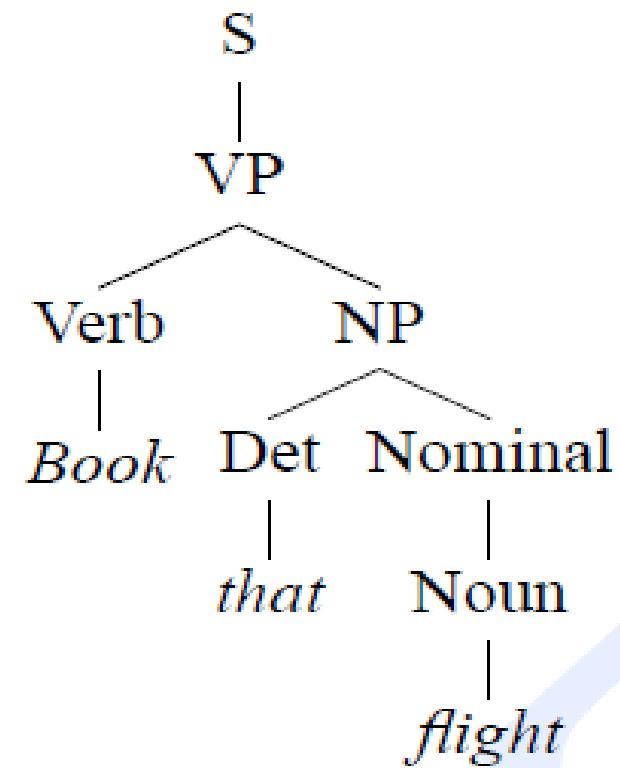
$VP \rightarrow Verb NP$

$VP \rightarrow Verb NP PP$

$VP \rightarrow Verb PP$

$VP \rightarrow VP PP$

$PP \rightarrow Preposition NP$



Bottom-up parsing

- In bottom-up parsing the parser starts with the words of the input, and tries to build trees from the words up, again by applying rules from the grammar one at a time.
- The parse is successful if the parser succeeds in building a tree rooted in the start symbol S that covers all of the input.

Bottom-up parsing for the example ‘book that flight’

$S \rightarrow NP VP$

$S \rightarrow Aux NP VP$

$S \rightarrow VP$

$NP \rightarrow Pronoun$

$NP \rightarrow Proper-Noun$

$NP \rightarrow Det Nominal$

$Nominal \rightarrow Noun$

$Nominal \rightarrow Nominal Noun$

$Nominal \rightarrow Nominal PP$

$VP \rightarrow Verb$

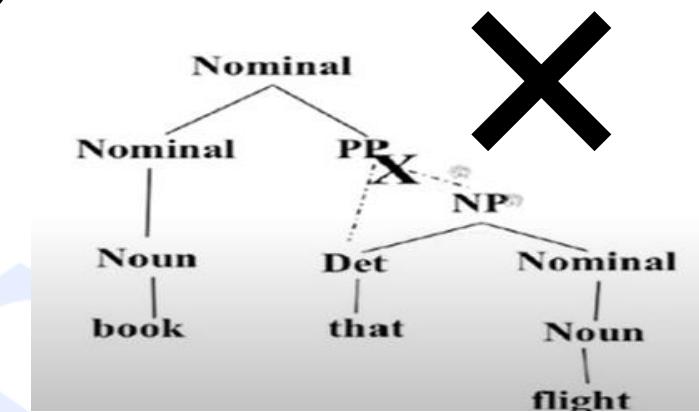
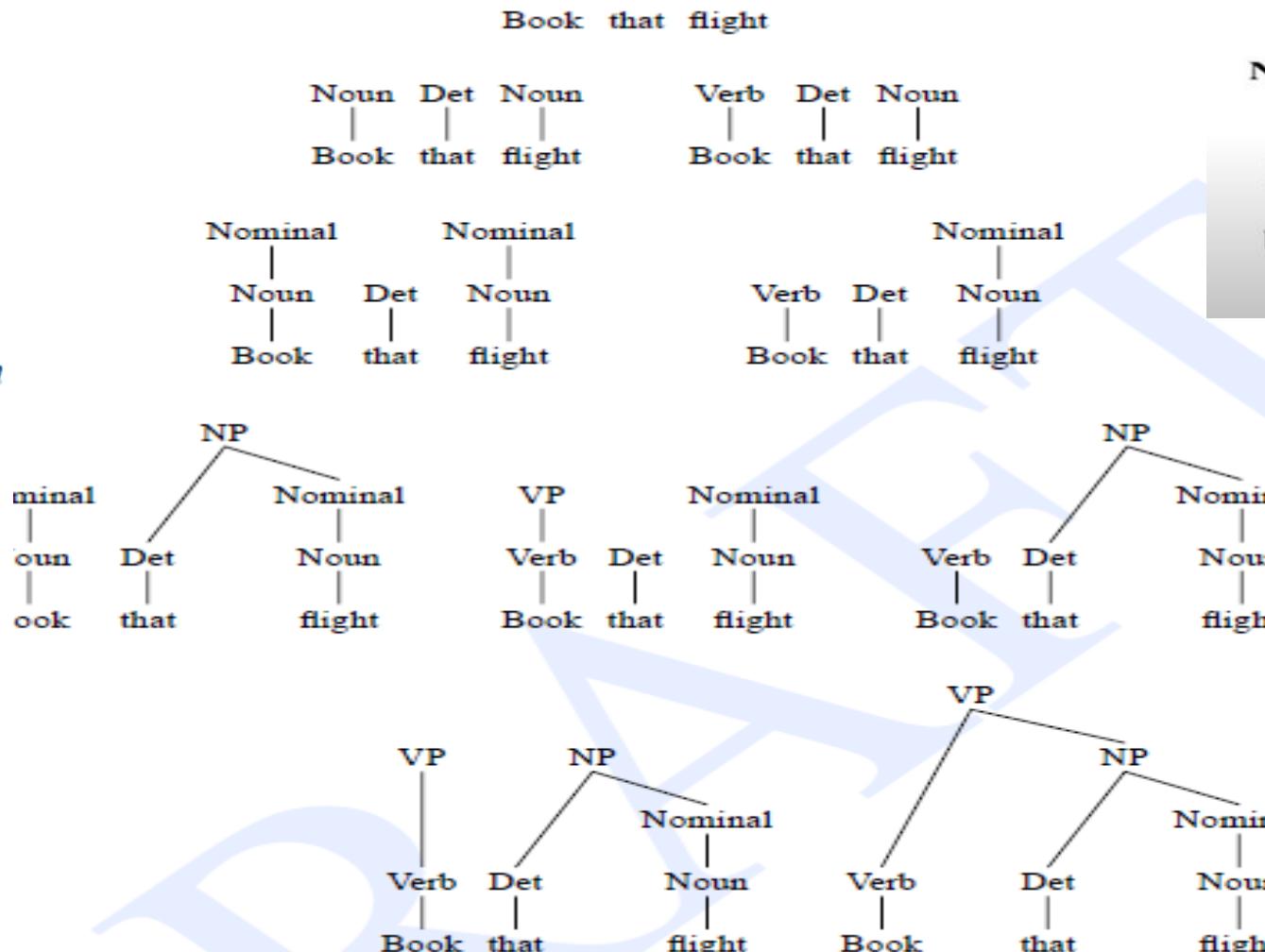
$VP \rightarrow Verb NP$

$VP \rightarrow Verb NP PP$

$VP \rightarrow Verb PP$

$VP \rightarrow VP PP$

$PP \rightarrow Preposition NP$

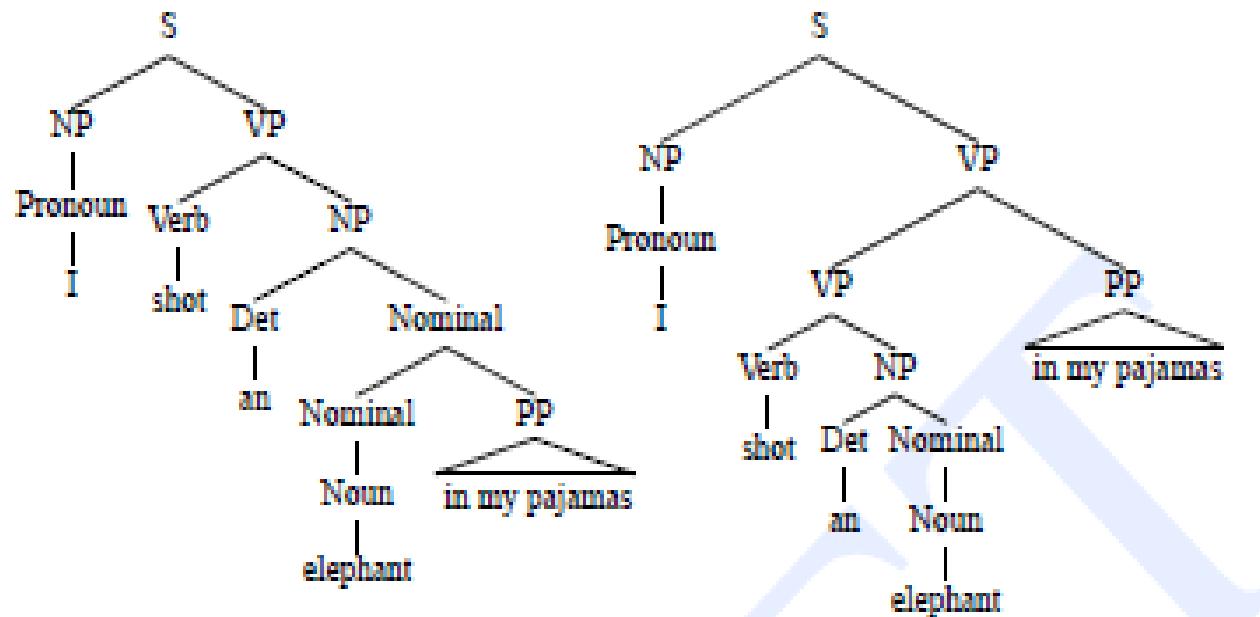


Comparing Top-Down and Bottom-Up Parsing

- The top-down strategy never wastes time exploring trees that cannot result in a full parse tree, since it begins by generating just those trees.
 - But explores many options that never connect to the actual sentence
- In the bottom-up strategy, by contrast, never explores options that do not connect to the actual sentence i.e. no hope of leading to an S
 - But can explore options that can never lead to a full parse tree

Ambiguity

- Ambiguity is a serious problem faced by parsers. (Already seen in Part-of-speech ambiguity)
- Ambiguity also arises in the syntactic structures used in parsing and is called as **structural ambiguity**.
 - Structural ambiguity occurs when the grammar assigns more than one possible parse to a sentence.
 - Groucho Marx's well-known line as Captain Spaulding is ambiguous because the phrase *in my pajamas* can be part of the *NP* headed by *elephant* or the verb-phrase headed by *shot*.



- Three common kinds of ambiguity are **attachment ambiguity**, **coordination ambiguity**, and **Local ambiguity**.
- A sentence has an **attachment ambiguity** if a particular constituent can be attached to the parse tree at more than one place.
 - The Groucho Marx sentence above is an example of PP-attachment ambiguity.
- In **coordination ambiguity** 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 as *[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.

- **Local ambiguity** occurs when some part of a sentence is ambiguous, that is, has more than one parse, even if the whole sentence is not ambiguous.
 - For example the sentence *Book that flight* is unambiguous, but when the parser sees the first word *Book*, it cannot know if it is a verb or a noun until later. Thus it must consider both possible parses

Dynamic Programming Parsing Methods

- **Ambiguity** gives rise to problems and confusions in standard bottom-up or top-down parsers
- Also to avoid extensive repeated work, caching of intermediate results is required
- **Dynamic programming** provides a framework for caching the results and also solving the ambiguity problem, like it helped with the Minimum Edit Distance
 - Recall that dynamic programming approaches systematically fill in tables of solutions to sub-problems.
 - When complete, the tables contain the solution to all the sub-problems needed to solve the problem as a whole

- The efficiency gain arises from the fact that these subtrees are discovered once, stored, and then used in all parses calling for that constituent.
 - This solves the re-parsing problem (subtrees are looked up, not re-parsed)
 - Partially solves the ambiguity problem (the dynamic programming table implicitly stores all possible parses by storing all the constituents with links that enable the parses to be reconstructed)
- Three most widely used methods are
 - The **Cocke-Kasami-Younger (CKY) algorithm**
 - bottom-up parser, requires normalizing the grammar
 - The **Earley algorithm**
 - Top-down parser, doesn't require normalizing the grammar, more complex
 - **Chart Parsing**
 - retain completed phrases in a chart and can combine top-down and bottom-up searches

CKY Parsing (Cocke-Kasami-Younger Parsing)

- Grammars used with it must be in Chomsky Normal Form (CNF)
 - Either, exactly two non-terminals on the RHS
 - Or, 1 terminal symbol on the RHS
- Bottom-up parsing stores phrases formed from all substrings in a triangular table(chart)

Conversion to CNF

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

1. Rules that mix terminals with non-terminals on the right-hand side

- Solution:** introduce a new dummy non-terminal that covers only the original terminal.
- Example:** a rule for an infinitive verb phrase such as $\text{INF-VP} \rightarrow \text{to VP}$ would be replaced by the two rules $\text{INF-VP} \rightarrow \text{TO VP}$ and $\text{TO} \rightarrow \text{to}$

2. Rules with a single non-terminal on the right are called unit productions

- Solution:** Unit productions are eliminated by rewriting the right-hand side of the original rules with the right-hand side of all the non-unit production rules that they ultimately lead to.

- **Example:** More formally, if $A * \Rightarrow B$ by a chain of one or more unit productions, and $B \rightarrow \gamma$ is a non-unit production in our grammar, then we add $A \rightarrow \gamma$ for each such rule in the grammar, and discard all the intervening unit productions.

3. Rules with right-hand sides longer than two

- **Solution:** Introduction of new non-terminals that spread the longer sequences over several new productions.
- **Example:** if we have a rule like $A \rightarrow B C \gamma$
 - we replace the leftmost pair of non-terminals with a new non-terminal and introduce a new production result in the following new rules.
 - $X_1 \rightarrow B C$
 - $A \rightarrow X_1 \gamma$

Conversion to CNF : Example

Original Grammar

S → NP VP
S → Aux NP VP
S → VP
NP → Pronoun
NP → Proper-Noun
NP → Det Nominal
Nominal → Noun
Nominal → Nominal Noun
Nominal → Nominal PP
VP → Verb
VP → Verb NP
VP → VP PP
PP → Prep NP
Pronoun → I | he | she | me
Noun → book | flight | meal | money
Verb → book | include | prefer
Proper-Noun → Houston | NWA

Chomsky Normal Form

S → NP VP
S → X1 VP
X1 → Aux NP
S → book | include | prefer
S → Verb NP
S → VP PP
NP → I | he | she | me
NP → Houston | NWA
NP → Det Nominal
Nominal → book | flight | meal | money
Nominal → Nominal Noun
Nominal → Nominal PP
VP → book | include | prefer
VP → Verb NP
VP → VP PP
PP → Prep NP
Pronoun → I | he | she | me
Noun → book | flight | meal | money
Verb → book | include | prefer
Proper-Noun → Houston | NWA

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. Binarize all rules and add to new grammar

CKY Parsing

- Involves parsing substrings of length 1, then length 2, and so on until the entire string has been parsed.
- This is useful because the shorter substrings from previous iterations can be used when applying grammar rules to parse the longer substrings.

- Let n be the number of words in the input. Think about $n + 1$ lines separating them, numbered 0 to n .
 - x_{ij} will denote the words between line i and j
 - We build a table so that x_{ij} contains all the possible non-terminal spanning for words between line i and j .
 - We build the Table bottom-up.
-
- For a sentence of length n , we will work with the upper-triangular portion of an $(n+1) \times (n+1)$ matrix.
 - Each **cell** $[i, j]$ in this matrix contains a set of non-terminals that represent all the constituents that span positions i through j of the input

CKY parsing for CFG

a 1	pilot 2	likes 3	flying 4	planes 5

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

a	pilot	likes	flying	planes
1	2	3	4	5
DT				

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

Step 1

a	pilot	likes	flying	planes
1	2	3	4	5
DT				
	NN			

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

Step 2

a	pilot	likes	flying	planes
1	2	3	4	5
DT	NP			
	NN			

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

Step 3

a	pilot	likes	flying	planes
1	2	3	4	5
DT	NP			
	NN			
		VBZ		

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

Step 4

a 1	pilot 2	likes 3	flying 4	planes 5
DT	NP			
	NN	-		
		VBZ		

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

Step 5

a 1	pilot 2	likes 3	flying 4	planes 5
DT	NP	-		
	NN	-		
		VBZ		

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

Step 6

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

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

Step 7

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

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

Step 8

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

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

Step 9

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

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

Step 11

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

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

#

Step 10

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

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

#

Step 12

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

Step13

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

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

Step 14

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

CKY Algorithm

```
function CKY-PARSE(words, grammar) returns table
    for j ← from 1 to LENGTH(words) do
        table[j − 1, j] ← {A | A → words[j] ∈ grammar}
    for i ← from j − 2 downto 0 do
        for k ← i + 1 to j − 1 do
            table[i, j] ← table[i, j] ∪
                {A | A → BC ∈ grammar,
                 B ∈ table[i, k],
                 C ∈ table[k, j] }
```

STATISTICAL PARSING

- The CKY algorithm could represent some ambiguities but were not equipped to resolve them.
- A probabilistic parser offers a solution to the problem: compute the probability of each interpretation, and choose the most-probable interpretations.
- The most commonly used probabilistic grammar is the **probabilistic context-free grammar** (PCFG), a probabilistic augmentation of context-free grammars in which each rule is associated with a probability

PROBABILISTIC CONTEXT-FREE GRAMMARS

- PCFG differs from CFG by augmenting each rule in R with a conditional probability:
$$A \rightarrow b [p]$$
- Here p expresses the probability that the given non-terminal A will be expanded to the sequence b . That is, p is the conditional probability of a given expansion b given the left-hand-side (LHS) non-terminal A .
- We can represent this probability as $P(A \rightarrow b)$ or as $P(A \rightarrow b | A)$ or as $P(\text{RHS} | \text{LHS})$
- Thus if we consider all the possible expansions of a non-terminal, the sum of their probabilities must be 1

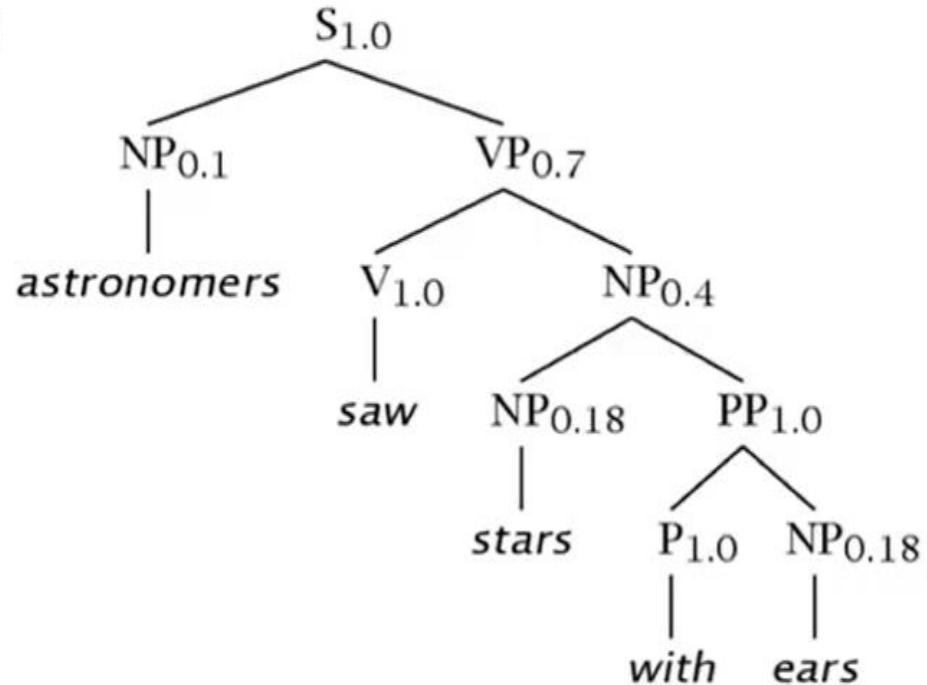
$$\sum_{\beta} P(A \rightarrow \beta) = 1$$

A Simple PFGS (in CNF)

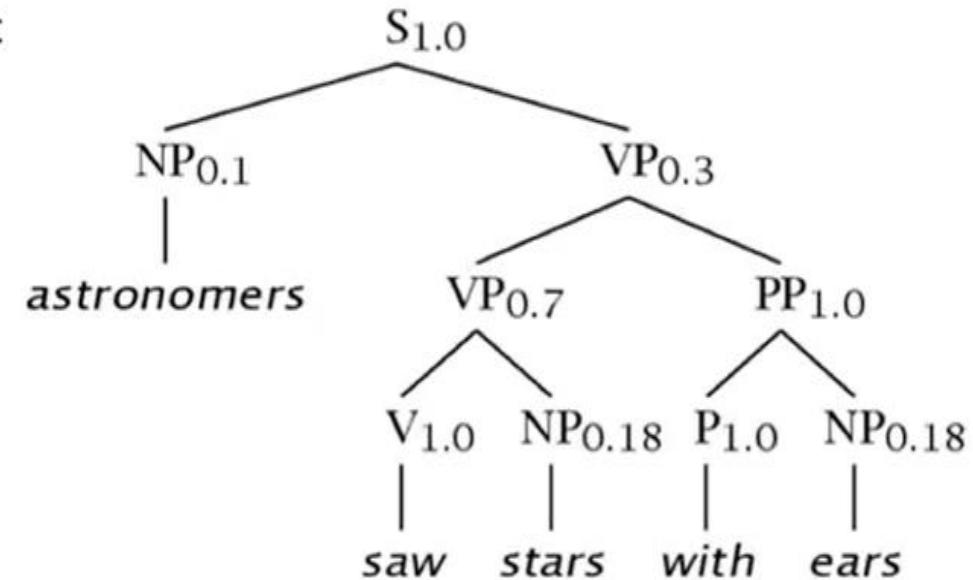
S	→	NP VP	1.0	NP	→	NP PP	0.4
VP	→	V NP	0.7	NP	→	<i>astronomers</i>	0.1
VP	→	VP PP	0.3	NP	→	<i>ears</i>	0.18
PP	→	P NP	1.0	NP	→	<i>saw</i>	0.04
P	→	<i>with</i>	1.0	NP	→	<i>stars</i>	0.18
V	→	<i>saw</i>	1.0	NP	→	<i>telescope</i>	0.1

Example trees

t_1 :



t_2 :



Probabilities of Trees and Strings

- $P(t)$: The probability of tree is the product of the probabilities of the rules used to generate it
- $P(w_{1:n})$: The probability of the string is the sum of the probabilities of the trees which have that string as their yield

Probabilities of Trees and Strings

$$\begin{aligned} w_{15} &= \text{astronomers saw stars with ears} \\ P(t_1) &= 1.0 * 0.1 * 0.7 * 1.0 * 0.4 * 0.18 \\ &\quad * 1.0 * 1.0 * 0.18 \\ &= 0.0009072 \\ P(t_2) &= 1.0 * 0.1 * 0.3 * 0.7 * 1.0 * 0.18 \\ &\quad * 1.0 * 1.0 * 0.18 \\ &= 0.0006804 \\ P(w_{15}) &= P(t_1) + P(t_2) \\ &= 0.0009072 + 0.0006804 \\ &= 0.0015876 \end{aligned}$$

Probabilities

- Parse tree 1: $.05 \times .20 \times .30 \times .20 \times .60 \times .20 \times .75 \times .10 \times .30 = 1.62 \times 10^{-6}$
- Parse tree 2: $.05 \times .05 \times .30 \times .20 \times .60 \times .75 \times .10 \times .15 \times .75 \times .30 = 2.28 \times 10^{-7}$

Features of PCFG

- As the number of possible trees for a given input grows, a PCFG gives some idea of the plausibility of a particular parse
- *But* the probability estimates are based purely on structural factors, and do not factor in lexical co-occurrence. Thus, PCFG does not give a very good idea of the plausibility of the sentence.
- Real text tends to have grammatical mistakes. PCFG avoids this problem by ruling out nothing, but by giving implausible sentences a low probability
- In practice, a PCFG is a worse language model for English than an n-gram model
- All else being equal, the probability of a smaller tree is greater than a larger tree

CKY for PCFG

<i>S</i> → <i>NP VP</i>	[1.0]
<i>VP</i> → <i>VBG NNS</i>	[0.1]
<i>VP</i> → <i>Vbz VP</i>	[0.1]
<i>VP</i> → <i>Vbz NP</i>	[0.3]
<i>NP</i> → <i>DT NN</i>	[0.3]
<i>NP</i> → <i>JJ NNS</i>	[0.4]
<i>DT</i> → <i>a</i>	[0.3]
<i>NN</i> → <i>pilot</i>	[0.1]
<i>Vbz</i> → <i>likes</i>	[0.4]
<i>Vbg</i> → <i>flying</i>	[0.5]
<i>JJ</i> → <i>flying</i>	[0.1]
<i>NNS</i> → <i>planes</i>	[.34]

CKY for PCFG

$$0.3 * 0.1 * 0.3$$

a 1	pilot 2	likes 3	flying 4	planes 5
DT [0.3]	NP [.009]	-	-	S [1.4688×10^{-5}] S [6.12×10^{-6}]
	NN [0.1]	-	-	-
		VBZ [0.4]	-	VP [.001632] VP [.00068]
			JJ [0.1] VBG [0.5]	NP [.0136] VP [.017]
				NNS [.34]

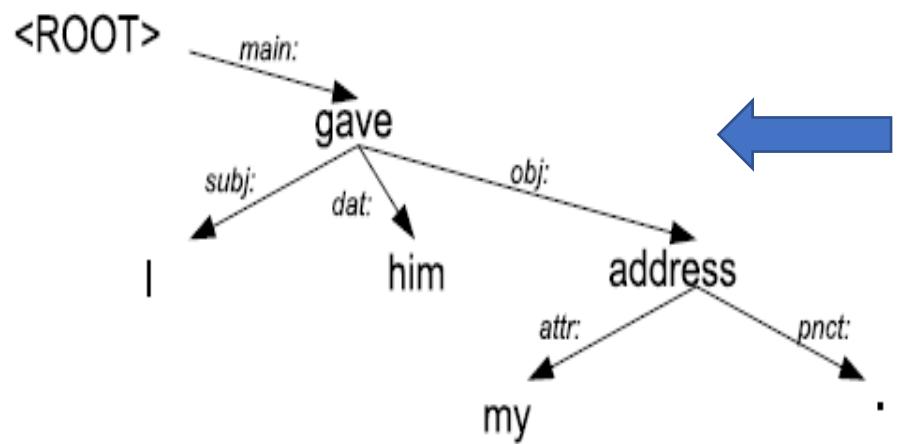
$$0.4 * 0.0136 * 0.3$$

$$0.4 * 0.017 * 0.1$$

Dependency Grammars

- **Dependency grammars** that are becoming quite important in speech and language processing, where constituents and phrase-structure rules do not play any fundamental role.
- Instead of that, the syntactic structure of a sentence is described purely in terms of words and binary semantic or syntactic relations between these words.
- It follows the notion of traditional grammar “**parsing a sentence into subject and predicate**” that is based on lexical relations rather than constituent relations.

Dependency Grammars



The links are drawn from a fixed inventory of around 35 relations, most of which roughly represent grammatical functions or very general semantic relations.

Shows an example parse of the sentence I gave him my address, using the dependency grammar formalism.

Note that there are no non-terminal or phrasal nodes; each link in the parse tree holds between two lexical nodes (augmented with the special <ROOT> node).

Dependency	Description
subj	syntactic subject
obj	direct object (incl. sentential complements)
dat	indirect object
pcomp	complement of a preposition
comp	predicate nominals (complements of copulas)
tmp	temporal adverbials
loc	location adverbials
attr	premodifying (attributive) nominals (genitives, etc.)
mod	nominal postmodifiers (prepositional phrases, etc.)

Dependency Grammars

- Advantage of dependency formalisms is the strong predictive parsing power that words have for their dependents.
- Knowing the identity of the verb is often a very useful cue for deciding which noun is likely to be the subject or the object.
- Another advantage of pure dependency grammars is their ability to handle languages with relatively **free word order**.
 - For example an *object* might occur before or after a *location adverbial* or a **comp**.

- A phrase-structure grammar would need a separate rule for each possible place in the parse tree such that an adverbial phrase could occur.
- A dependency grammar would just have one link-type representing this particular adverbial relation. Thus a dependency grammar abstracts away from word-order variation, representing only the information that is necessary for the parse.

- Convert the following CFG to CNF

- $S \rightarrow bA|aB$
- $A \rightarrow bAA|aS|a$
- $B \rightarrow aBB|bS|b$

- Step1: List the productions which are already in CNF

$A \rightarrow a$

$B \rightarrow b$

- Step2: Replace the terminals on the right by the new non terminals

- $S \rightarrow bA$
 - ✓ $S \rightarrow CA$
 - ✓ $C \rightarrow b$
- $S \rightarrow aB$
 - ✓ $S \rightarrow DB$
 - ✓ $D \rightarrow a$
- $A \rightarrow aS$
 - ✓ $A \rightarrow DS$
 - ✓ $D \rightarrow a$
- $B \rightarrow bS$
 - $B \rightarrow CS$
 - $C \rightarrow b$

- A->bAA
 - ✓ A->CAA
 - ✓ C->b
- B->aBB
 - ✓ B->DBB
 - ✓ D->a
- Step3: According to CNF RHS must contain only 2 non terminals. So, convert such productions.
 - A->CAA
 - ✓ A->EA
 - ✓ E->CA
 - B->DBB
 - ✓ B->FB
 - ✓ F->DB
- Step4: Check for productions with only one non terminal in RHS
No such productions in the given problem.

The final CFG in CNF is:

- | | |
|---------|--------------------|
| ✓ S->CA | S->CA DB |
| ✓ C->b | C->b |
| ✓ S->DB | D->a |
| ✓ D->a | A-> <u>DS EA a</u> |
| ✓ A->DS | B-> <u>CS FB b</u> |
| ✓ D->a | E->CA |
| ✓ B->CS | F->DB |
| ✓ A->EA | |
| ✓ E->CA | |
| ✓ B->FB | |
| ✓ F->DB | |

Exercise

1. Show the CYK Algorithm with the following example:

- CNF grammar G
 - $S \rightarrow AB \mid BC$
 - $A \rightarrow BA \mid a$
 - $B \rightarrow CC \mid b$
 - $C \rightarrow AB \mid a$
- w is **ababa**
- Whether **ababa** is in $L(G)$?

2. Use the CYK algorithm to find the parse tree for “ Book the flight through Houston” using the CNF form shown below.

$$S \rightarrow NP VP$$

$$S \rightarrow Aux NP VP$$

$$S \rightarrow VP$$

$$NP \rightarrow Pronoun$$

$$NP \rightarrow Proper-Noun$$

$$NP \rightarrow Det Nominal$$

$$Nominal \rightarrow Noun$$

$$Nominal \rightarrow Nominal Noun$$

$$Nominal \rightarrow Nominal PP$$

$$VP \rightarrow Verb$$

$$VP \rightarrow Verb NP$$

$$VP \rightarrow Verb NP PP$$

$$VP \rightarrow Verb PP$$

$$VP \rightarrow VP PP$$

$$PP \rightarrow Preposition NP$$

$$S \rightarrow NP VP$$

$$S \rightarrow XI VP$$

$$XI \rightarrow Aux NP$$

$$S \rightarrow book | include | prefer$$

$$S \rightarrow Verb NP$$

$$S \rightarrow X2 PP$$

$$S \rightarrow Verb PP$$

$$S \rightarrow VP PP$$

$$NP \rightarrow I | she | me$$

$$NP \rightarrow TWA | Houston$$

$$NP \rightarrow Det Nominal$$

$$Nominal \rightarrow book | flight | meal | money$$

$$Nominal \rightarrow Nominal Noun$$

$$Nominal \rightarrow Nominal PP$$

$$VP \rightarrow book | include | prefer$$

$$VP \rightarrow Verb NP$$

$$VP \rightarrow X2 PP$$

$$X2 \rightarrow Verb NP$$

$$VP \rightarrow Verb PP$$

$$VP \rightarrow VP PP$$

$$PP \rightarrow Preposition NP$$

3. Consider grammar L

$S \rightarrow AB \mid BC, A \rightarrow BA \mid a, B \rightarrow CC \mid b, C \rightarrow AB \mid a$

Let $w = baaba$. Is the word in grammar L?

END