

# Chapter 2

## Word level Analysis

# Contents

- Morphology analysis
- Inflectional Morphology
- Derivational Morphology
- Lemmatization
- Regular Expression
- Finite automata
- Finite state transducers(FST)
- Morphological Parsing with FST
- Lexicon free FST Porter Stemmer
- N-Grams and N-gram language model
- N-gram for spelling correction

# What is Morphology?

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

washing → wash + -ing

- How they are **formed**?

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

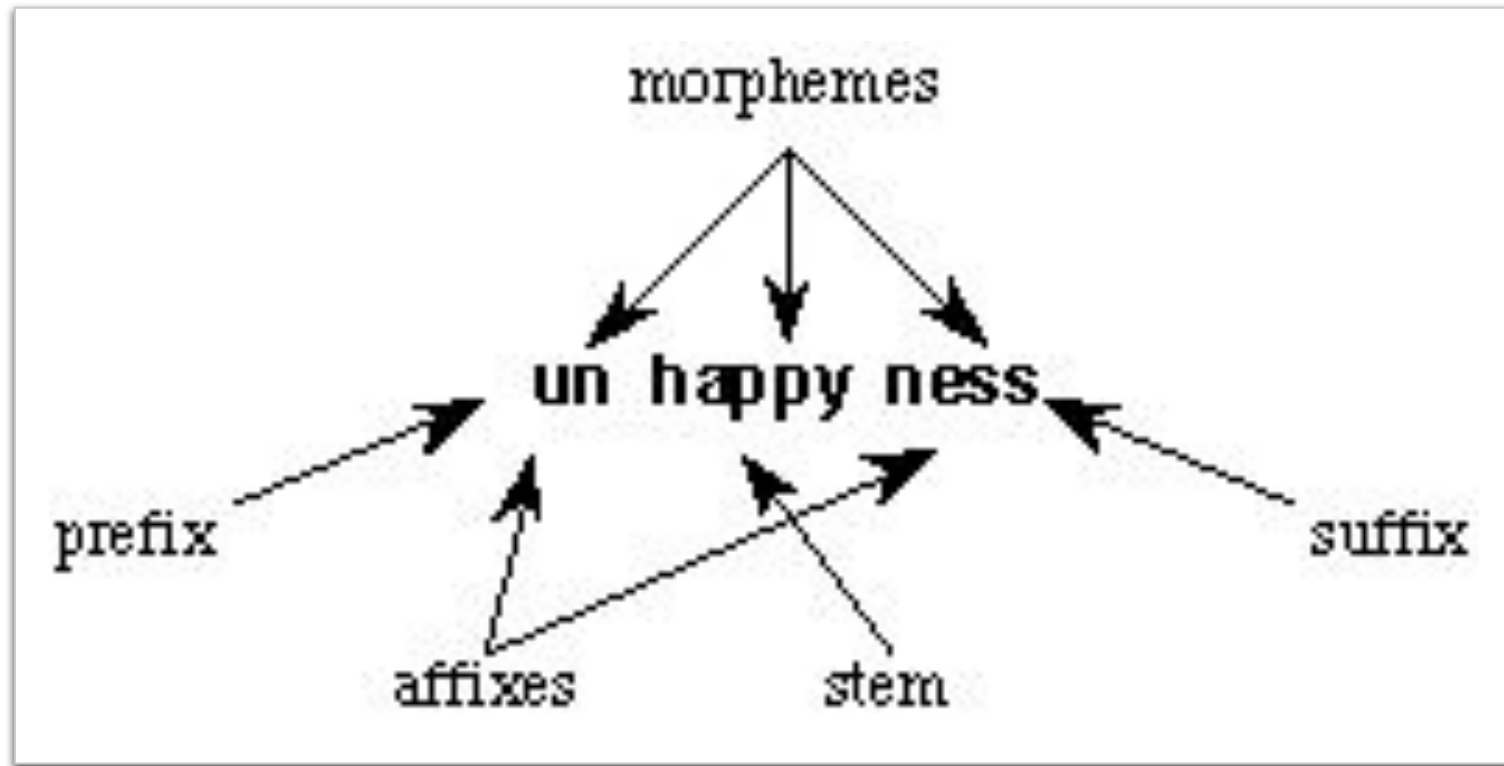
- Morphology tries to formulate rules

# Morphology

- **Morphemes:** The smallest units of language that carry meaning or function.
- **Free morphemes:** Words that can stand alone and still make sense.
- **Bound morphemes:** Morphemes that cannot stand alone, they need to be attached to a free morpheme in order to be a proper, meaningful word.
- **Affixes:** A morpheme attached to something else.
- **Root:** The core of the word. What's left when affixes are taken away.
- **Syntax:** The study of sentence structure.

# Morphology

- Morphology is the study of the structure and formation of words
- Its most important unit is the *morpheme*, which is defined as the "minimal unit of meaning"
- Papers= paper + s
- Washing=wash + ing



# Morphology

Consider a word like: "unhappiness". This has three parts:

# Why morphology is used

- Information retrieval system
  - Type **phones**... it should search for phone and phones
- Machine translation
  - **Monkeys** ate the bananas

# Morphology

- There are two morphemes,
- each carrying a certain amount of meaning.
- *un* means "not"
- *ness* means "being in a state or condition".
- ***Happy* is a *free morpheme* because it can appear on its own.**
- *Bound morphemes* have to be attached to a free morpheme, and so cannot be words in their own right.



# Morphology

- Inflectional
- Derivational

# Inflectional Morphology

- Inflection is the process of changing the form of a word so that it expresses information such as number, person, case, gender, tense.
  - but the syntactic category of the word remains unchanged. (parts of speech)
  - **Tall and taller both are adjectives**
  - As an example, the plural form of the noun in English is usually formed from the singular form by adding an s.
- 
- **car / cars**
  - **table / tables**
  - **dog / dogs**

# Inflection

- Indicates some grammatical function like

Number	लड़का (Sg)	लड़के (Pl)
Person	जाऊँगा (1st)	जाओगे (2nd)
Gender	जाऊँगा(Masc)	जाऊँगी (Fem)
Tense	गया(Past)	जाऊँगी (Future)

- Number – singular/plural
- Person- first person singular **I** or second person singular **You**

# Inflectional Morphology

- Inflection is the process of changing the form of a word so that it expresses information such as number, person, **case**, gender, tense
- **Case: determining the position of noun in the sentence**
  - **Nominative case**
    - **Objective case (or Accusative case)**
    - Dative case
    - Possessive case (or Genitive case)
    - Vocative case

A noun is said to be in the Nominative case if it is the subject of a verb.

- Mr. Ram is an intelligent boy.

Mr. Ram is a proper noun in **Nominative case**.

- Nouns or pronouns are said to be in Objective cases if they are the direct objects of verbs or if they are the objects of preposition.

- I met your sister.

"Your sister" is in **objective case**.

# Inflectional Morphology

- It doesn't take long to find examples where the simple rule given above doesn't fit.
- So there are smaller groups of nouns that form plurals in different ways:
  - **wolf / wolves**
  - **knife / knives**
  - **switch / switches.**
- A little more thought and we can think of apparently completely irregular plural forms, such as:
  - **foot / feet**
  - **child / children.**

# More on Inflection

Noun  
inflectional  
suffixes

- Plural marker –s
- Possessive marker ‘s

- Cat. Cats
- John. John’s father

Verb inflectional  
suffixes

- Third person present singular marker -s
- Past tense marker -ed
- Progressive marker -ing
- Past participle markers -en or –ed

- he **runs** fast
- She **climbs** the tree
- He **passed** the exam
- I am **going** to market
- He **walked** with his stick

Adjective  
inflectional  
suffixes

- Comparative marker -er
- Superlative marker -est

- This bat is **short**
- This bat is **shorter**
- This bat is **shortest**

# More on Inflection

- Participle
  - Word formed from a verb
  - Going, gone
- Past participle
  - PP is a verb form. For regular verbs just add the ending –ed
  - Walk ==> walked



# Spelling Rules

- Generally words are pluralized by adding –s to the end
- Words ending in –s, -z, -sh and sometimes –x require –es
  - **buses, quizzes, dishes, boxes**
- Nouns ending in –y preceded by a consonant change the –y to -ies
  - **babies, floppies**

# Derivational Morphology

- Derivation means creation of a new word from existing words by changing the grammatical category
- **Read is a verb. Adding er becomes reader which is a noun**

# Use of Derivational Morphology

- To reduce the number of forms of words to be stored. So, if there is already an entry for the base form of the verb *sing*, then it should be possible to add rules to map the nouns *singer* and *singers* onto the same entry.

# Stemming and Lemmatization

- Stemming and Lemmatization are **Text Normalization** (or sometimes called **Word Normalization**) techniques in the field of **Natural Language Processing** that are used to prepare text, words, and documents for further processing.
- The boy's car are different
- The boy car be
-

# Stemming and Lemmatization

The boy's car are different

- The boy car are differ. ===== stemming
- The boy car be differ. ===== lemmatization

- Boy, boys, boy's ? boy
- Am, are, is -? is
- Car, car's, cars, cars'. --? car
-

# Stemming and Lemmatization

- Stemming and Lemmatization helps us to achieve the root forms of derived words.
- Stemming and Lemmatization are widely used in **Web search results**
- For example, searching for *fish* on Google will also result in *fishes, fishing as fish* is the stem of both words.

# Difference between Stemming and Lemmatization

- **Stemming** algorithms work by cutting off the end or the beginning of the word, taking into account a list of common prefixes and suffixes that can be found in an inflected word.
- Studies----- **studi**

# Difference between Stemming and Lemmatization

- **Lemmatization**, on the other hand, takes into consideration the **morphological analysis of the words**. To do so, it is necessary to have detailed dictionaries which the algorithm can look through to link the form back to its lemma.
- **Studies == studi === study**



# Lexicon free Porter stemmer Algorithm

- Common algorithm for stemming English Language
- Algorithm consists of 5 steps
- Definitions
- CONSONANT : a letter other than AEIOU
- VOWEL: AEIOU

# Porter stemmer Algorithm

## Definitions

- CONSONANT : a letter other than AEIOU
- VOWEL: AEIOU
- With this definition, all words are of the form
- $(C) (VC)^m (V)$
- C- string of one or more consonants
- V- String of one or more vowels
- M- measure of word or word part, when represented in this form  $(VC)$

# Porter stemmer algorithm

- **Rules** are of the form

**(condition) S1 → S2**

where S1 and S2 are suffixes.

**This means that if a word ends with a suffix S1, and the stem before S1 satisfies the given condition, then S1 is replaced by S2**

# Porter stemmer algorithm

- **Conditions:**

1.  $m$  --- measure of the stem  $m = k$  or  $m > k$ , where  $k$  is an integer
2.  $*S$  --- the stem ends with a given letter  $S$
3.  $*v$  --- the stem contains a vowel
4.  $*d$  --- the stem ends in double consonant (TT, SS)
5.  $*o$  --- the stem ends with a consonant-vowel-consonant sequence, where the final consonant is not  $w, x$  or  $y$ , (e.g., *wil*, *hop*)

- Condition part may also contain expressions with **and/or/not**

# Porter stemmer Algorithm

- $m$  is the *measure* of the word  $(VC)^m$ .
  - $m = 0$ : TR, EE, TREE, Y, BY
  - $m = 1$ : TRO**UBLE**, O**ATS**, TRE**ES**, **IVY**
  - $m = 2$ : TRO**UBLE****S**, PR**IV****ATE**, O**AT****EN**, ORRERY

# Porter stemmer algorithm

- Step 1a. (basic step)
- SSES-> SS
  - **Caresse -> caress**
- IES-> I
  - **Ponies-> poni**
  - **Ties-> ti**
- SS->SS
  - **Caress-> caress**
- S-> NULL
  - **Cats-> cat**

# Porter stemmer algorithm

- Step 1b)
- (m>1) EED -> EE
  - Condition verified : **agre**ed**** -> agree
  - Condition not verified : **fe**ed**** -> feed
- (\*V\*) ED -> NULL (if any vowel is present)
  - Condition verified : **pl**ast**er**ed**** -> plaster (suffix ed is removed)
  - Condition not verified : bled -> bled. **(ed is removed)**
- (\*V\*) ING -> NULL
  - Condition verified : motoring-> motor. **(ing is removed)**
  - Condition not verified : sing -> sing

# Porter stemmer algorithm

- Cleanup step
- These rules are run if second and third rule in 1b apply
- **AT->ATE** ( if vowels are present and after removing the suffix, if word ends with AT, replace it with ATE)
  - Conflat(ed) ---> conflate. ( two vowels are present )
- **BL -> BLE**
  - Troubl( ing). → trouble



# Porter stemmer algorithm

- (\*d & ! ( \*L or \*S or \*Z )). ->>> single letter
  - If stem ends in double consonant and **that** consonant is not L/S/Z.
  - Condition verified : **hopp(ing)** -> hop
  - Condition not verified : fall(ing) -> fall. (Consonant ends with L)
- (m=1 & \*O) [?] E
  - the stem ends with a consonant-vowel-consonant sequence, where the final consonant is not w, x or y,
  - Condition verified : **fil(ing)** -> file
  - Condition not verified : fail -> fail. (ail is Vowel vowel consonant)

# Porter stemmer algorithm

- Step 1c
- Y elimination (\*V\*) Y → I
  - Condition verified: happy → happi
  - Condition not verified: sky – sky
- Step 2. DERIVATIONAL MORPHOLOGY, I
  - (m>0) ATIONAL → ATE. (IF M > 0 AND word is ending with ATIONAL)
    - **RELATIONAL → RELATE**
  - (m>0) IZATION → IZE
    - **GENERALIZATION → GENERALIZE**
  - (m>0) BILITI → BLE
    - **SENSIBILITI → SENSIBLE**

# Porter stemmer algorithm

- Step 3: Derivational Morphology, II
  - (m>0) ICATE -> IC
    - **TRIPPLICATE ? TRIPLIC**
  - (m>0) FUL -> NULL
    - **HOPEFUL ? HOPE**
  - (m>0) NESS -> NULL
    - **GOODNESS ? GOOD**
- Step 4: Derivational Morphology, III
  - (m>0) ANCE -> NULL
    - **ALLOWANCE ? ALLOW**
  - (m>0) ENT -> NULL
    - **DEPENDENT ? DEPEND**
  - (m>0) IVE -> NULL
    - **EFFECTIVE ? EFFECT**

# Porter stemmer algorithm

- STEP 5 (clean up)
  - Step 5a
    - (m>1) E -> NULL
      - **PROBATE? PROBAT**
    - (m=1 & !\*O) NESS -> NULL
      - **GOODNESS? GOOD**
  - Step 5b
    - (m>1 & \*d & \*L) ? single letter
      - Condition verified: **control** ? control
      - Condition not verified: roll ? roll

# Regular expression in NLP

- SEARCH for **cat** in browser
- Search for **cats** in browser
- RE is a language for specifying a text search string
- Used in MS word for searching a word
- RE requires a pattern to be searched for
- And a corpus of text to be searched through

# corpus

- In non-functional [linguistics](#), a sentence is a textual unit consisting of one or more words that are grammatically linked. In functional linguistics, a sentence is a unit of written texts delimited by graphological features such as upper case letters and markers such as periods, question marks, and exclamation marks. This notion contrasts with a curve, which is delimited by phonologic features such as pitch and loudness and markers such as pauses; and with a clause, which is a sequence of words that represents some process going on throughout time.<sup>[1]</sup> This entry is mainly about *sentence* in its non-functional sense, though much work in functional **linguistics** is indirectly cited or considered such as the categories of Speech Act Theory.
- A sentence can include words grouped meaningfully to express a statement, **lingusitics**, exclamation, request. A sentence is a set of words that in principle tells a complete thought (although it may make little sense taken in isolation out of context). It may be a simple phrase, but it conveys enough meaning to imply a clause, even if it is not explicit; for example, "Two" as a sentence (in answer to the question "How many were there?") implies the clause "There were two." Typically a sentence contains a subject and predicate. **lingusitics**
- In the teaching of writing skills (composition skills), students are generally required to express (rather than imply) the elements of a sentence, leading to the schoolbook definition of a sentence as one that must [explicitly] include a subject and a verb. For example, in second-language acquisition, teachers often reject one-word answers that only imply a clause, commanding the student to "give me a complete sentence," by which they mean an explicit one.
- As with all language expressions, sentences might contain function and content words and contain properties such as characteristic intonation and timing patterns.

# Regular expressions in NLP

- Imp topic in NLP
- As we need to find out patterns in NL text
- RE helps in finding out patterns in text
  
- If we are using a data set/ corpus/ text file
- And u want to search for some pattern like email address/ phone number/ similar type of text..

# Regular expressions in NLP

- RE methods
  - findall()
  - search()
  - split()



# Regular expressions in NLP

- If we are using a data set/ corpus/ text file
- And u want to search for some pattern like email address/ phone number/ similar type of text..
- Phone number has a fixed pattern
- Email address has a fixed pattern
- If u know the pattern, use the behavior of the pattern to find out the exact pattern.

# Regular expressions in NLP

- Text: My phone number is 2221113333
- Search [phone in text]
- **re module** is available in python to perform indirect search
- `re.search(pattern, text)`

# Regular expressions in NLP

```
import re
```

```
Pattern='phone'
```

```
Text: My phone number is 2221113333
```

```
re.search(pattern, text)
```

- **Match object; span=(3,8), match='phone'**

# Regular expressions in NLP

```
import re
```

```
Pattern='phone'
```

```
Text: My phone number is 2221113333
```

```
xx=re.search(pattern, text)
```

```
xx.span() ➤ (3,8)
```

```
xx.start() ➤ 3
```

```
xx.end() ➤ 8
```

- **Match object; span=(3,8), match='phone'**

# Regular expressions in NLP

```
import re
Pattern='phone'
Text: My phone is a brand new phone
```

```
xx=re.search(pattern, text)
xx.span()
```

- Match object; **span=(3,8)**, match='phone'
- Not showing second match of phone

# Regular expressions in NLP

```
import re
```

```
Pattern='phone'
```

```
Text: My phone is a brand new phone
```

```
xx=re.findall("phone", text)
```

```
for xx in re.findall('phone', text):  
    print(xx)
```

- **Phone**
- **phone**

# Regular expressions in NLP

- Text: My phone number is 222-111-3333
- If the phone number is following a pattern like ISD/STD/NUMBER
- Pattern= r'\d\d\d-\d\d\d-\d\d\d\d'
- Ph=re.search(pattern, text)
- Ph.group()       // It returns entire match
- **222-111-3333**

# Regular expressions in NLP

- `re.findall(r "\d$", " this ends with number 3")`
- [3]. **Searches for a digit at the end**
- `re.findall(r "^ \d", " 3 is my favorite number")`
- [3]. **Searches for a digit at the beginning**
- Text= there are 3 numbers inside 4 boxes of 2 tiffins"
- `re.findall(r "[^ \d]+", text]`
- **All the integers in the text will be ignored**

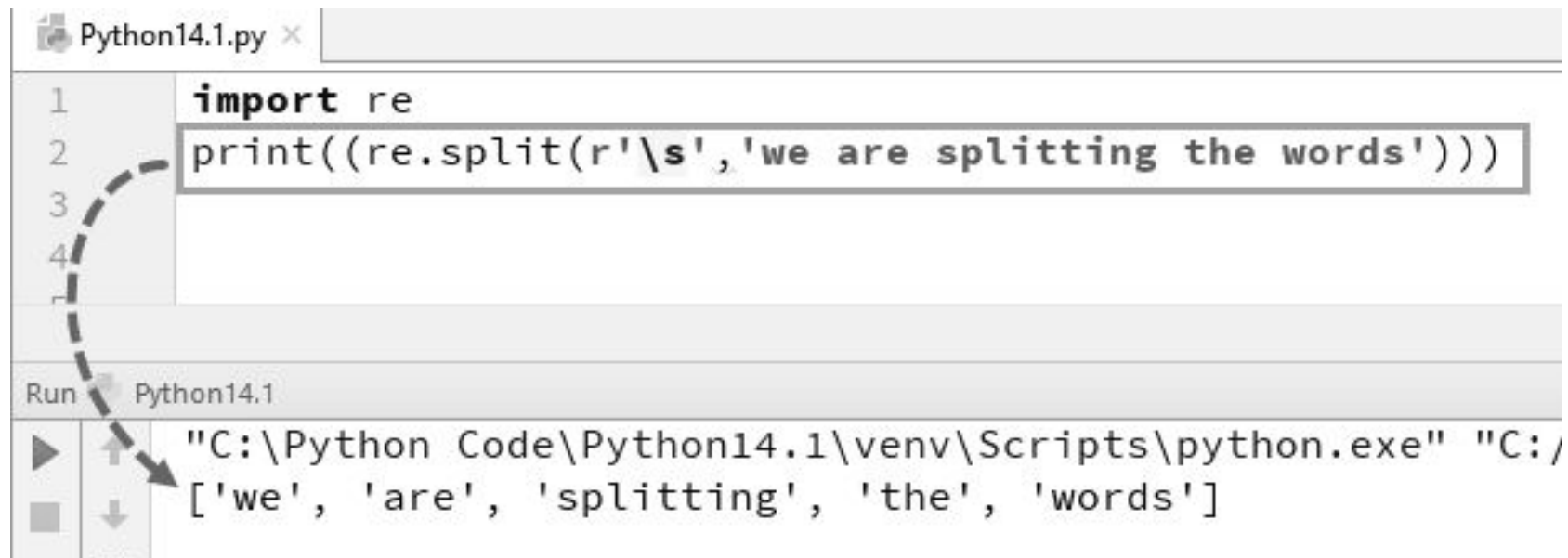


# Regular expressions in NLP

- Text=" This is a string ! But it has punctuation. How to remove it?"
- `re.findall(r "[^!.? ]+", text)`

# Regular expressions in NLP

- **Example of `\s` expression in `re.split` function**
- `s"`: This expression is used for creating a space in the string



The screenshot shows a Python IDE window titled "Python14.1.py". The code editor contains the following Python code:

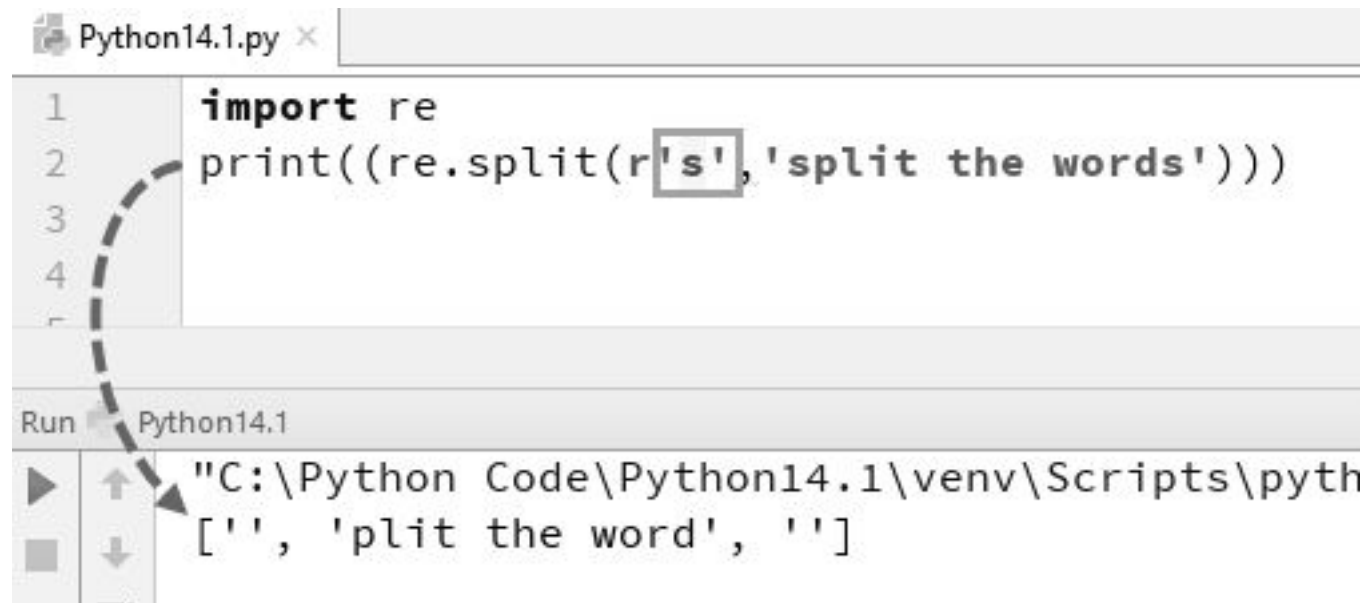
```
1 import re
2 print((re.split(r'\s', 'we are splitting the words')))
```

A dashed arrow points from the `\s` in the code to the output below. The output is displayed in a console window titled "Python14.1" and shows the command prompt path and the resulting list of words:

```
"C:\Python Code\Python14.1\venv\Scripts\python.exe" "C:/
['we', 'are', 'splitting', 'the', 'words']
```

# Regular expressions in NLP

- what happens if you remove "\" from s. There is no 's' alphabet in the output, this is because we have removed '\' from the string, and it evaluates "s" as a regular character and thus split the words wherever it finds "s" in the string.



```
Python14.1.py x
1 import re
2 print((re.split(r's', 'split the words')))
3
4
Run Python14.1
"C:\Python Code\Python14.1\venv\Scripts\pyth
['', 'plit the word', '']
```

# RE use in NLP

- Validate data fields (dates, email address)
- Filter text ( disallow spam mails)
- Identify particular strings in text

# Finite state automata in NLP

- Many NLP applications require the use of a dictionary (often called a *lexicon*) of some kind
- some applications require the use of several dictionaries.
- Dictionaries are used to hold several kinds of information, most frequently syntactic and semantic information.
- One method of storing the information would be to simply list each word as part of the program
- word(diva). word(divagate). word(divalent). word(divan).

# Finite state automata in NLP

- This may well be a good way of storing a small dictionary for a language like English that has very few ways of changing the end of words (eg "divan", "divans ")
- If we take a section of a moderate-sized English dictionary more or less at random:
- **diva**, **div**agate, **div**alent, **div**an, **div**aricate, **dive**, **div**er, **div**erge, **div**ergent, **div**ers, **div**erse, **div**ersify
- We can see that many letters at the beginning of words are repeated and obviously we might be able to economise on storage space and processing by some kind of data structure that allows sharing of common letters.

# Finite state automata in NLP

- **Components of networks**

- There are some standard symbols used in drawing networks. These are:

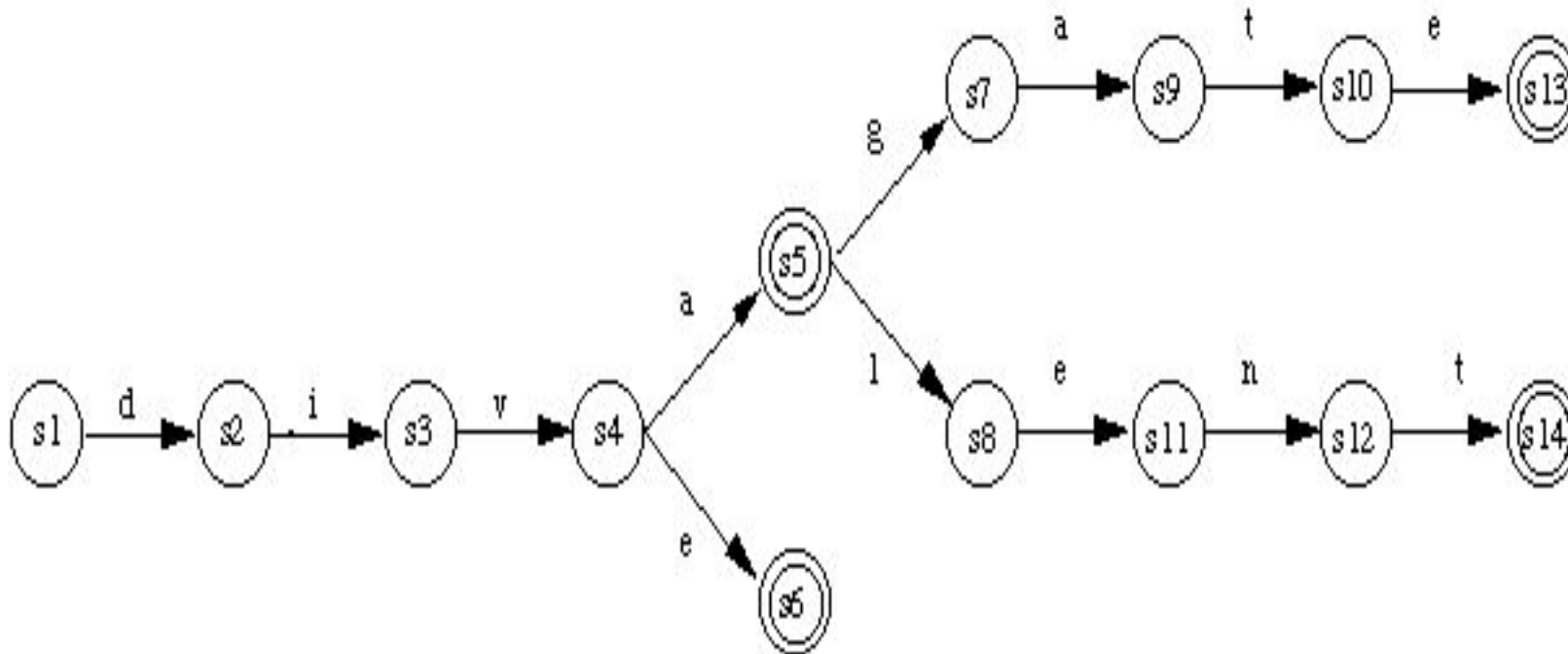
- *State:* 

- *Final state:* 

- *Arc :* 

# Finite state automata in NLP

- We'll work from the extract of a dictionary given above. We'll draw the network that represents the words:
- "diva", "dive", "divagate" and "divalent".





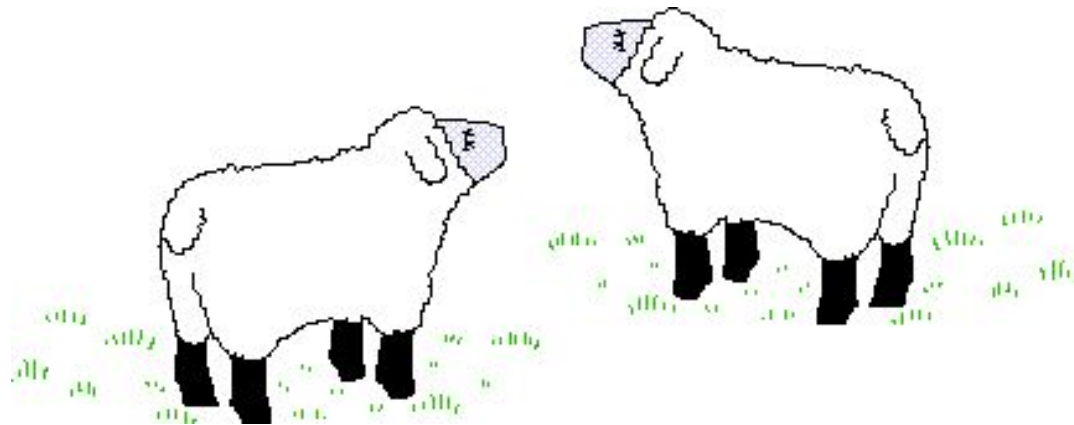
# Finite state automata in NLP

- If we look at this network, we can see (starting from the state on the far left) that we can trace the individual words of each word by simply jumping rightwards from state to state.
- There are two ways of using this network:
- one is to imagine that we have some input letters and to decide whether they are a word in the language (we call this *recognition*) or to use the network to enumerate all the words

# Natural Language

- Language created naturally by humans
- Rules are derived by studying the pattern the natural language follows
- Hence the grammatical rules of natural languages keep on changing as languages keep on changing with time

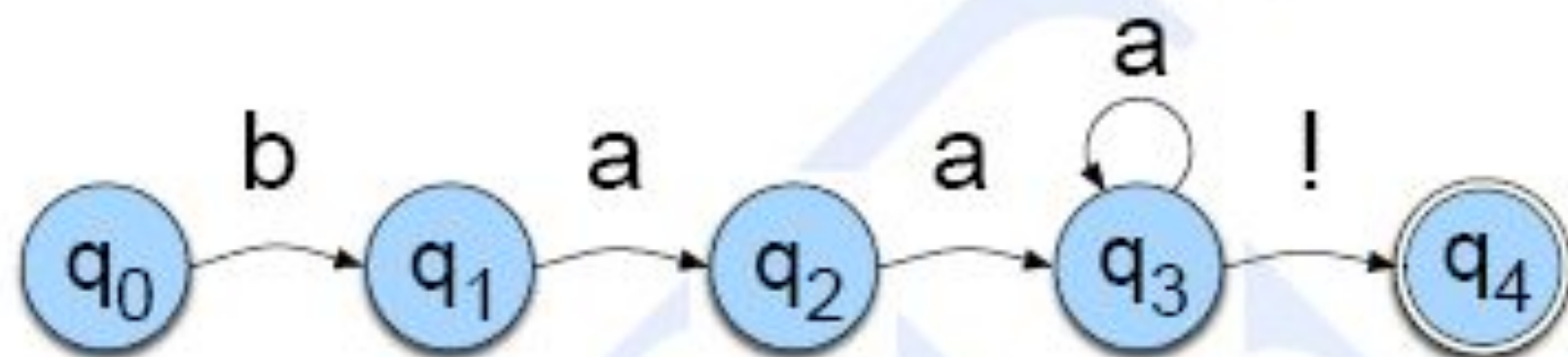
# Finite state automata



- Using an FSA to recognize sheeptalk
- Baa!
- Baaa!
- Baaaa!
- Baaaaa!
- Baaaaaa!
- Try to make an FSA for it

# Finite state automata

- Assume it to be corpus
  - Find pattern from the corpus
  - Analyze the corpus
- 
- Starts with B and ends with !
  - Common word is **aa** repeated everywhere
  - Minimum two occurrences of a



10 A finite-state automaton for talking sheep.

---

# Finite state transducers

- Finite State Transducers (FSTs) as “a finite state machine with two tapes: an input tape and an output tape.”

# Finite State Transducers

- Finite State Transducers (FSTs) is directed graphs, with a few special properties:
- Edges (“Transitions”) have input/output labels
  - Sometimes, labels are “empty” (indicated here as lowercase epsilon  $\epsilon$ )
- Traversing through to the end of an FST implies the generation (or indication of) a set of string relations

# Applications of FST

- FSTs are used for a variety of different applications:
- Word Inflections. For example, pluralizing words (cat -> cats, dog -> dogs, goose -> geese, etc.).
- Morphological Parsing; i.e., extracting the “properties” of a word (e.g., computers -> computer + [Noun] + [Plural])



# FST

- FSAutomata
  - An FSA represents a set of strings. e.g.  
 $\{walk, walks, walked, love\ loves, loved\}$ 
    - *Regular language.*
  - A recognizer function.  
`recognize(str) -> true or false`
- FSTransducers
  - An FST represents a set of *pairs of strings* (think of as input,output pairs)  
 $\{ (walk, walk+V+PL), (walk, walk+N+SG), (walked, walk+V+PAST) \dots \}$ 
    - *Regular relation.* (Not a function!)
  - A *transducer* function: maps input to zero or more outputs.  
`transduce(walk) --> {walk+V+PL, walk+N+SG}`  
Can return multiple answers if ambiguity: e.g. if you don't have POS-tagged input, "walk" could be the verb "They walk to the store" versus the noun "I took a walk".
  - Generic *inversion* and *composition* operations.

# Morphological parsing with FST

---

English	
Input	Morphologically Parsed Output
cats	cat +N +PL
cat	cat +N +SG
cities	city +N +Pl
geese	goose +N +Pl
goose	goose +N +Sg
goose	goose +V
gooses	goose +V +1P +Sg
merging	merge +V +PresPart
caught	catch +V +PastPart
caught	catch +V +Past

# Morphological Parsing

- With FSA
  - Cats  $\rightarrow$  cat
- With FST
  - Cats  $\rightarrow$  cat+N+Pl

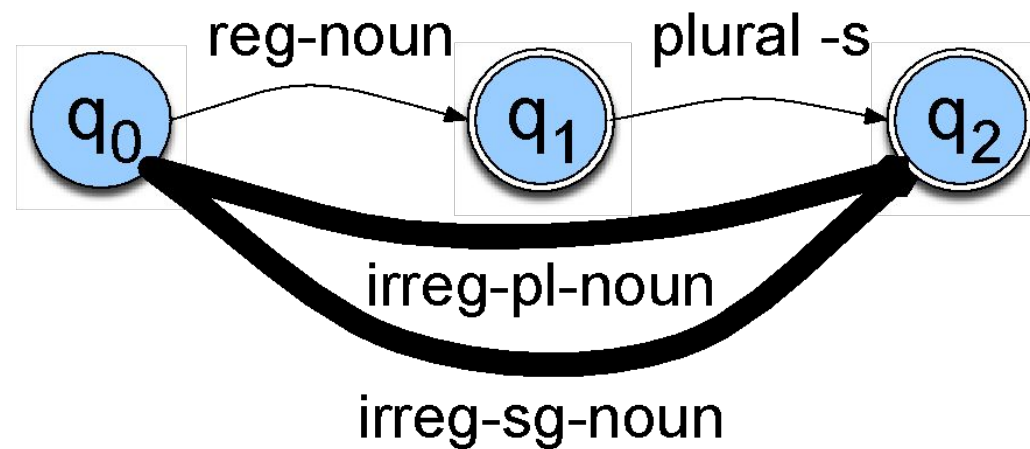
# To build a morphological parser

---

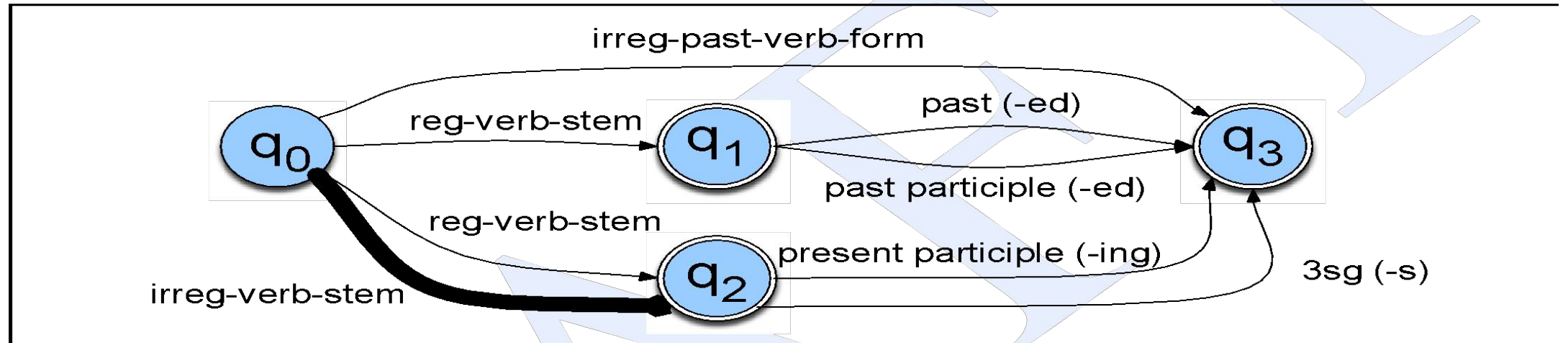
1. **lexicon:** the list of stems and affixes, together with basic information about them (whether a stem is a Noun stem or a Verb stem, etc.).
2. **morphotactics:** the model of morpheme ordering that explains which classes of morphemes can follow other classes of morphemes inside a word. For example, the fact that the English plural morpheme follows the noun rather than preceding it is a morphotactic fact.
3. **orthographic rules:** these **spelling rules** are used to model the changes that occur in a word, usually when two morphemes combine (e.g., the  $y \rightarrow ie$  spelling rule discussed above that changes *city* + *-s* to *cities* rather than *citys*).

# FSA

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



# FSA



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

# WHY FST

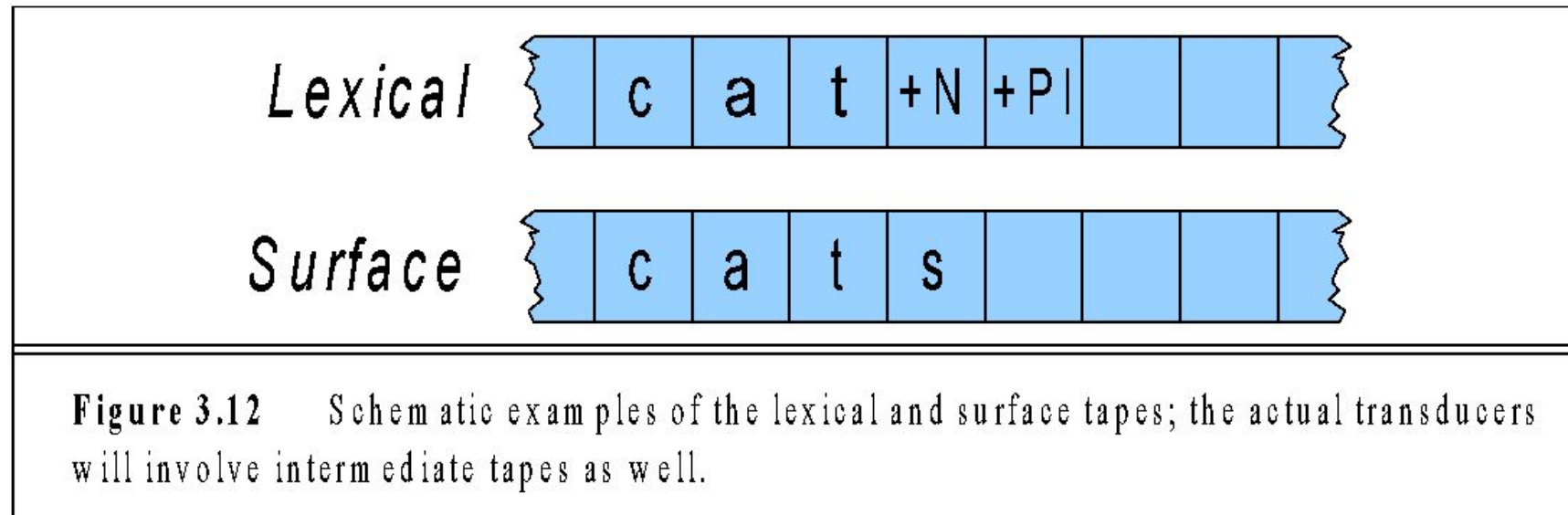
- CATS → CAT+N+PL. (WHY???????)
- This information can be used in MT
- Cats are running. (translate into Marathi)

# Morphological parsing

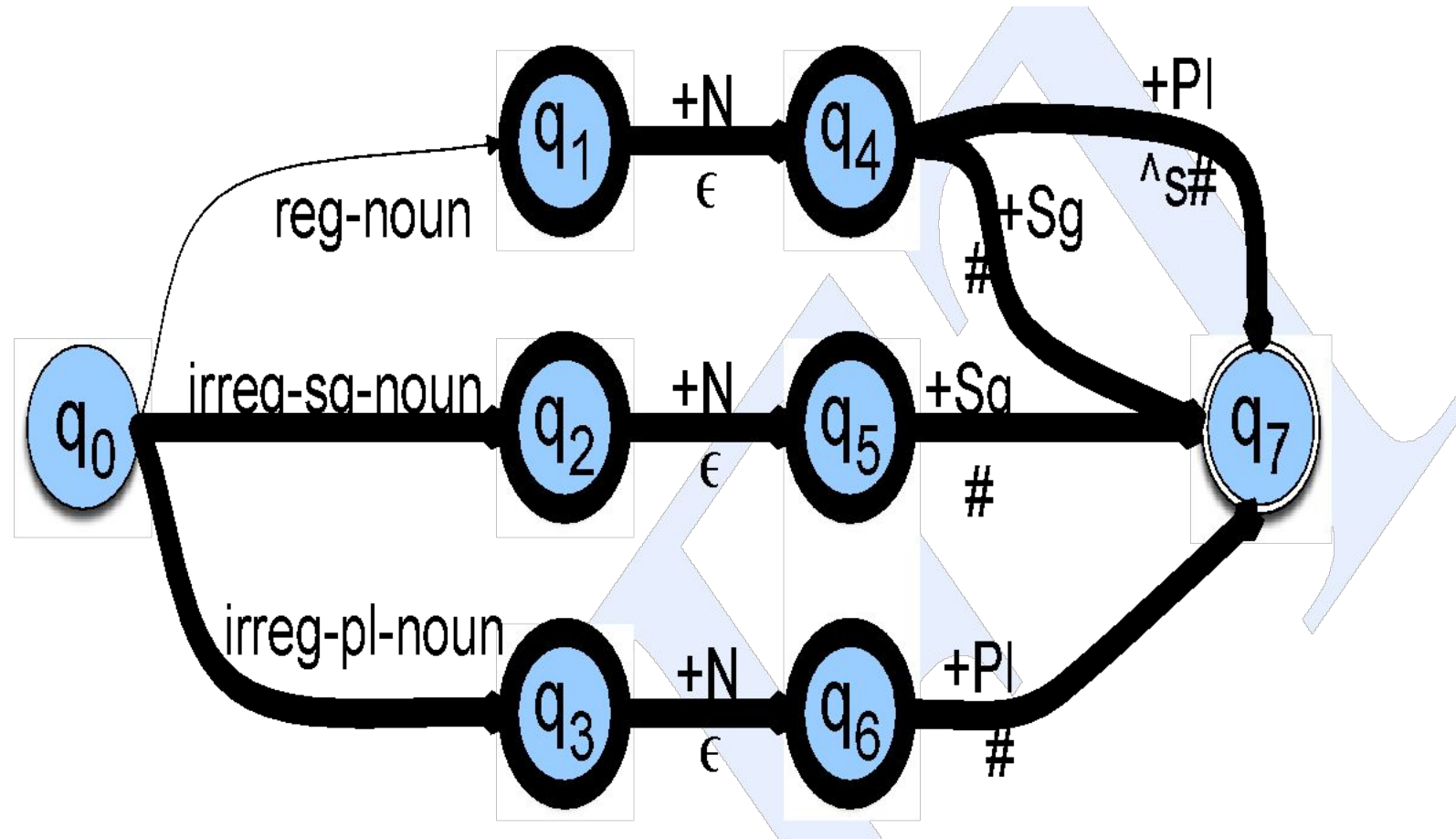
- The problem of recognizing that a word like foxes breaks down into component morphemes (fox and es)
- And building a structured representation of this fact is called Morphological parsing
- Parsing means taking the input and building some sort of linguistics structure for it
- <https://www.cs.vassar.edu/~cs395/docs/3.pdf>



# FSTs for morph parsing

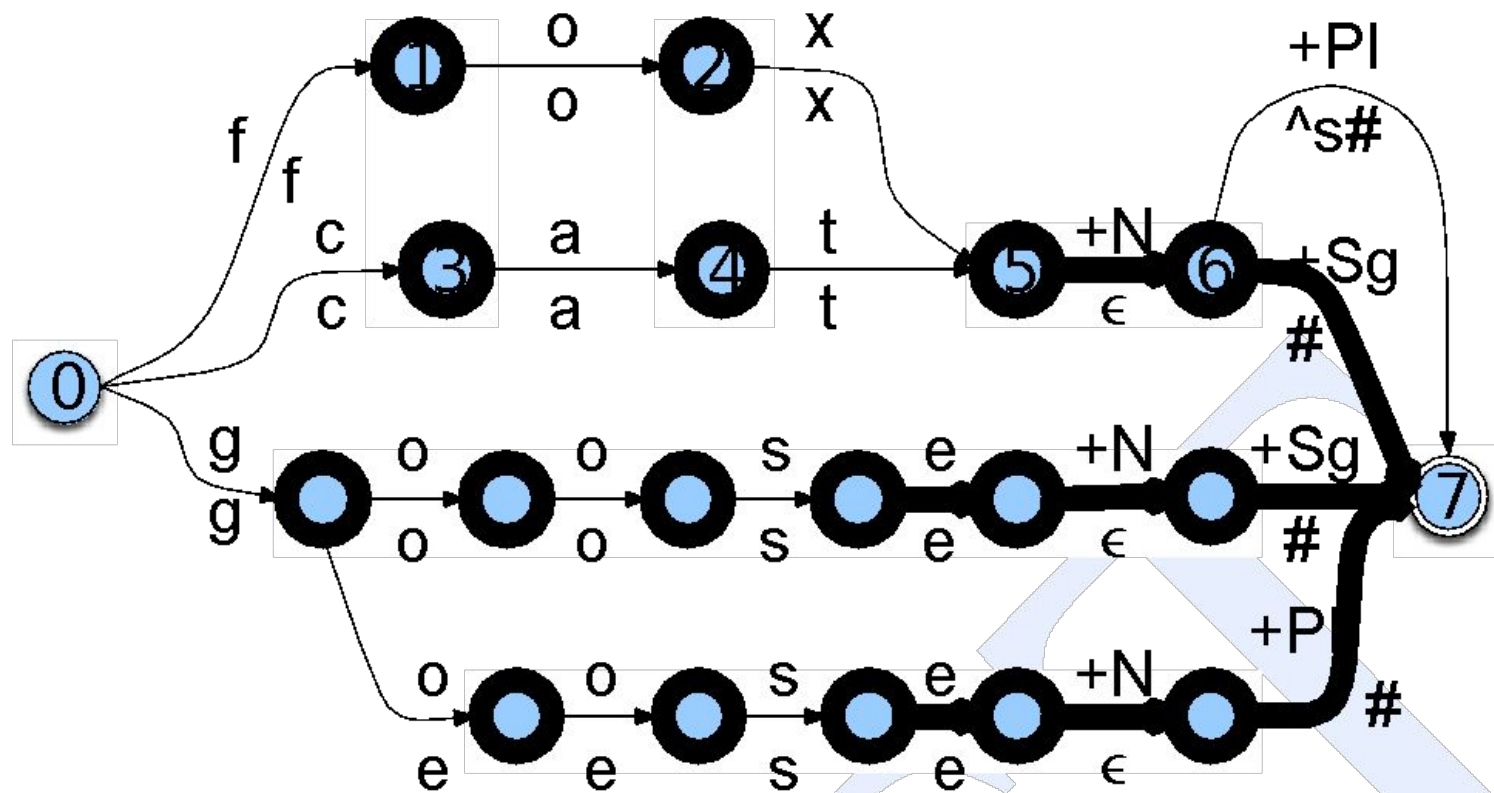


# FST FOR MORPHOLOGICAL PARSING



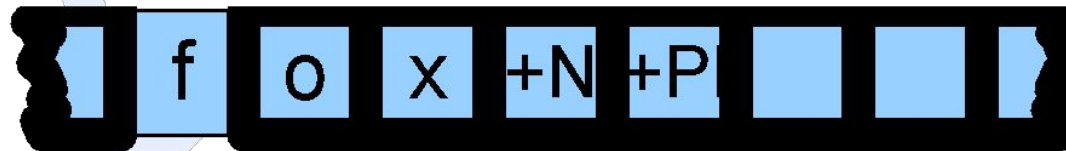
# FST with Morphological parsing

- We build a FST morphological parser
  - using morphological features ( Sg/Pl)
  - POS
  - Morpheme boundary.    ^    Cats [?].    Cat ^ s  
   unhappy [?] un^happy
  - Word boundary. #.    Cats#  
   blackbirds [?] black. # bird # s
- **ε** - Morphological features map to empty string if there is no segment corresponding to them on the output tape
- **N : ε** Noun on the upper tape will correspond to nothing on the lower tape

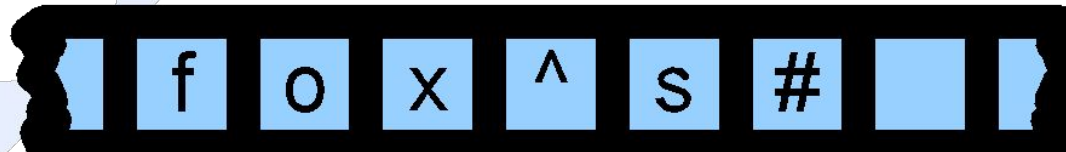


**Figure 3.14** A fleshed-out English nominal inflection FST  $T_{lex}$ , expanded from  $T_{num}$  by replacing the three arcs with individual word stems (only a few sample word stems are shown).

*Lexical*



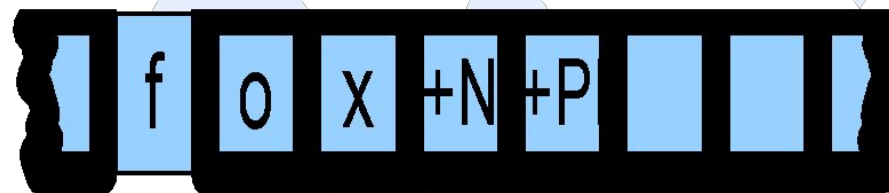
*Intermediate*



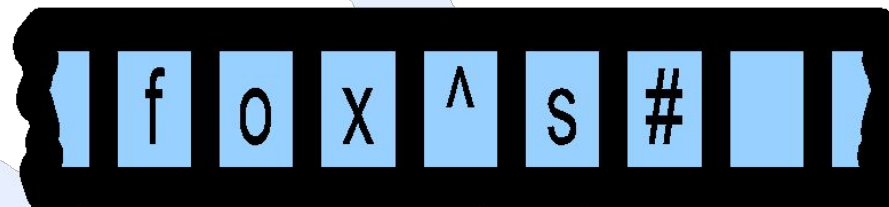
**Figure 3.15** A schematic view of the lexical and intermediate tapes.

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

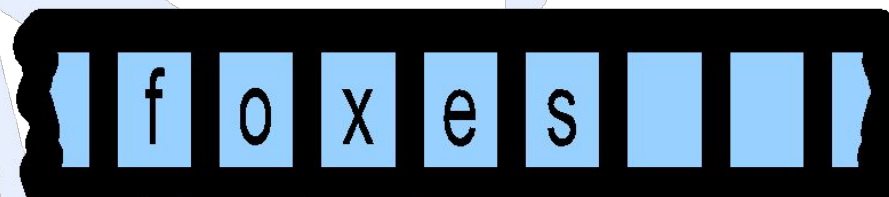
*Lexical*



*Intermediate*



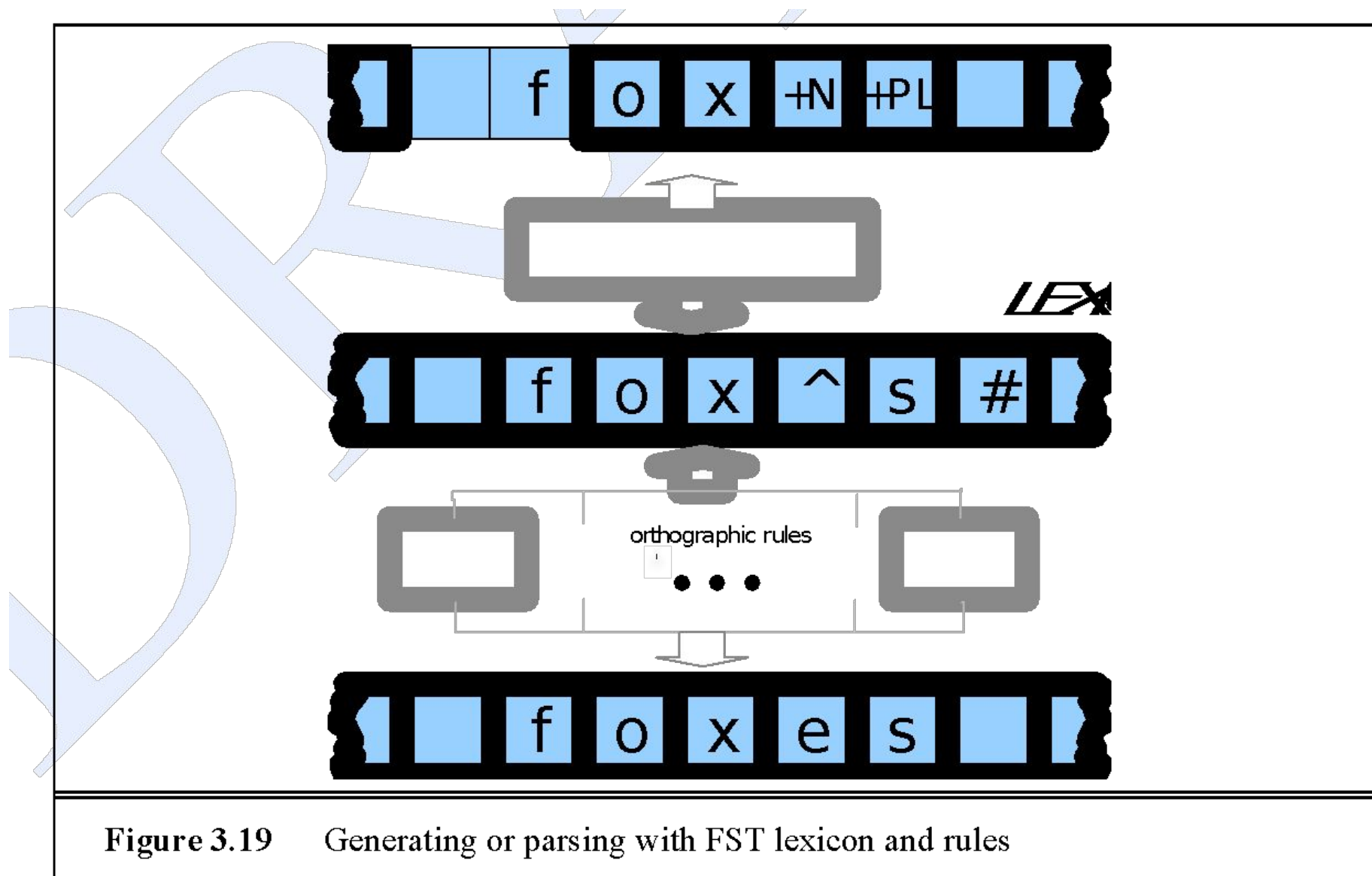
*Surface*



# Lexicon FST and rule transducers

- We now combine our lexicon FST and rule transducer for Parsing
- The lexicon transducer maps between lexical level with its stems and morphological features
- Intermediate level that represents a simple concatenation of morphemes
- Then host of transducers represent spelling rule constraint to map between intermediate level and surface level





**Figure 3.19** Generating or parsing with FST lexicon and rules

# N Grams

- Corpus
- Why it is required
- How it is processed
  - Ngrams

# N grams

- N-gram is simply a sequence of N words.
- **San Francisco (is a 2-gram)**
- **The Three Musketeers (is a 3-gram)**
- **She stood up slowly (is a 4-gram)**
- USE
- It help make next word predictions.

# N grams

- USE
- It help make next word predictions.
- Say you have the partial sentence “Please hand over your”.
- Then it is more likely that the next word is going to be “test” or “assignment” or “paper” than the next word being “school”.

# N grams

- **Coffee** is a brewed [drink](#) prepared from roasted [coffee beans](#), the seeds of [berries](#) from certain . The genus **Coffee** is native to [tropical Africa](#) (specifically having its origin in [Ethiopia](#) and [Sudan](#)) and [Madagascar](#), the [Comoros](#), [Mauritius](#), and [Réunion](#) in the Indian Ocean.<sup>[2]</sup> **Coffee** plants are now cultivated in [over 70 countries](#), primarily in the equatorial regions of the Americas, Southeast Asia, Indian subcontinent, and Africa. The two most commonly grown are [C. arabica](#) and [C. robusta](#). Once ripe, **coffee** berries are picked, processed, and dried. Dried **coffee** seeds (referred to as "beans") are [roasted](#) to varying degrees, depending on the desired flavor. Roasted beans are [ground and then brewed](#) with near-boiling water to produce the beverage known as **coffee**.
  - **Coffee** is darkly colored, bitter, slightly [acidic](#) and has a [stimulating](#) effect in humans, primarily due to its [caffeine](#) content.<sup>[3]</sup> It is one of the most popular drinks in the world,<sup>[4]</sup> and it can be prepared and presented in a variety of ways (e.g., [espresso](#), [French press](#), [caffè latte](#)). It is usually served hot, although [iced coffee](#) is a popular alternative. Clinical studies indicate that moderate **coffee** consumption is benign or mildly beneficial in healthy adults, with continuing research on whether long-term consumption lowers the risk of some diseases, although those long-term studies are of generally poor quality.
- 
- It can also help to make spelling error corrections.
  - For instance, the sentence "**cofee**" could be corrected to "**coffee**" if you knew that the word "coffee" had a high probability of occurrence after the word "drink"

# Corpus link

- <https://gagantalreja.github.io/wordPred/Word%20Corpus.txt>

# N GRAMS

- LETS BUILD A MODEL – NGRAM LANGUAGE MODEL

# NGRAM LANGUAGE MODEL

- Basically, an N-gram model predicts the occurrence of a word based on the occurrence of its  $N - 1$  previous words.
- **how far back in the history of a sequence of words should we go to predict the next word?**
- For instance, a bigram model ( $N = 2$ ) predicts the occurrence of a word given only its previous word (as  $N - 1 = 1$  in this case).
- Similarly, a trigram model ( $N = 3$ ) predicts the occurrence of a word based on its previous two words (as  $N - 1 = 2$  in this case).



# NGRAM LANGUAGE MODEL

- Let us see a way to assign a probability to a word occurring next in a sequence of words. First of all, we need a very large sample of English sentences (called **a corpus**).

- Say our corpus contains the following sentences:

He said **thank you**.

He said bye as he walked through the door.

He went to San Diego.

San Diego has nice weather.

It is raining in San Francisco.

# NGRAM LANGUAGE MODEL

- Let's assume a bigram model.
- So we are going to find the probability of a word based only on its previous word.
- In general, we can say that this probability is (the number of times the previous word 'wp' occurs before the word 'wn') / (the total number of times the previous word 'wp' occurs in the corpus) =
- $(\text{Count}(wp\ wn)) / (\text{Count}(wp))$

# NGRAM LANGUAGE MODEL

To find the probability of the word “you” following the word “thank”, we can write this as  $P(\text{you} \mid \text{thank})$  which is a conditional probability.

This becomes equal to:

- $= (\text{No. of times “Thank You” occurs}) / (\text{No. of times “Thank” occurs}) = 1/1 = 1$
- We can say with certainty that whenever “Thank” occurs, it will be followed by “You”
- (This is because we have trained on a set of only five sentences and “Thank” occurred only once in the context of “Thank You”).

# NGRAM LANGUAGE MODEL

- Let us see a way to assign a probability to a word occurring next in a sequence of words. First of all, we need a very large sample of English sentences (called **a corpus**).

- Say our corpus contains the following sentences:

He said **thank you**.

He said bye as he walked through the door.

He went to San Diego.

San Diego has nice weather.

It is raining in San Francisco.

# NGRAM LANGUAGE MODEL

- **Let's see an example of a case when the preceding word occurs in different contexts.**
- Let's calculate the probability of the word **"Diego"** coming after **"San"**. We want to find the  $P(\text{Diego} \mid \text{San})$ .
- This means that we are trying to find the probability that the next word will be "Diego" given the word "San".
- We can do this by:
  - $$= (\text{No of times "San Diego" occurs}) / (\text{No. of times "San" occurs}) = 2/3 = 0.67$$
- This is because in our corpus, one of the three preceding "San"s was followed by "Francisco".
- So, the  $P(\text{Francisco} \mid \text{San}) = 1 / 3$ .

# NGRAM LANGUAGE MODEL

- Say our corpus contains the following sentences:

He said thank you.

He said bye as he walked through the door.

He went to San Diego.

San Diego has nice weather.

It is raining in San Francisco.

- In our corpus, only “Diego” and “Francisco” occur after “San” with the probabilities  $\frac{2}{3}$  and  $\frac{1}{3}$  respectively.
- So if we want to create a next word prediction software based on our corpus, and a user types in “San”, we will give two options: “Diego” ranked most likely and “Francisco” ranked less likely. (GOOGLE.COM)

# Ngrams for spelling correction

- A word needs to be checked for spelling correctness and corrected if necessary, many a time in the *context* of the surrounding words.
- A spellchecker points to spelling errors and possibly suggests alternatives.
- An auto corrector usually goes a step further and automatically picks the most likely word. In case of the correct word already having been typed, the same is retained.

# Ngrams for spelling correction

- Given that a word  $w$  was typed, what was the intended word  $c$ ?
- Supposing  $w = \text{"the"}$ , then almost certainly,  $c = \text{"the"}$ . However, supposing,  $w = \text{"thew"}$ , then what might  $c$  be? It could be one of the following:

$\{\text{"the"}, \text{"then"}, \text{"thaw"}, \text{"thew"}, \text{"threw"}, \dots\}$

- There are many possibilities. Perhaps you actually meant to type *"thew"* and there was no spelling mistake. How do we know?
-



# Ngrams for spelling correction

- a language model learns frequencies of *n-grams* from large text corpora of a language, like English.
- Once trained, it can be used to evaluate the validity of an n-gram from that language, or to probabilistically generate new n-grams (word sequences or sentences) from that language.
-

# Ngrams for spelling correction

- For each word in sentence
  - Generate a candidate set
    - The word itself
    - All single letter edits that are English words
    - Words that are homophones
- Choose best candidate by probability maximization

# Ngrams for spelling correction

- Given an input unigram, we generate a set of candidate unigrams

$$C = \{ \text{"the"}, \text{"then"}, \text{"thaw"}, \text{"threw"}, \text{"threw"}, \dots \}$$

and then choose a  $c \in C$

- that maximizes the probability using the language model.

- 

$$\arg \max_{c \in C} P(c|w)$$

- This essentially means, pick the most frequent word among the set of candidates.

# Ngrams for spelling correction

- Perhaps this is right.
- After all, “*the*” is the most common word in English, and someone mistyping it does not seem unreasonable.
- On the other hand, perhaps the intended word was “*thaw*” and
- we assumed it was “*the*” because our language model said so.

# Building the Error Model

- given that we have observed the word  $w$ , we need to compute the intended word  $c$ .
- That is the spelling correction problem. Mathematically, it's the following:

$$\arg \max_{c \in C} P(c|w) = \arg \max_{c \in C} P(w|c)P(c)$$

- Finding the right candidate boils down to maximizing the above joint probability:
- **the unigram probability,  $P(c)$ , (given by the language model),**
- **multiplied by the probability that  $w$  is a spelling error of  $c$ , (given by the error model).**

# Error model

- Damerau-Levenshtein distance measures the distance between two strings (a word and its misspelling, for instance) in terms of the minimum number of *basic edit operations* required to transform one string to the other. These edit operations are:
- **Substitution:** Substitute a single character by another (e.g., *pwned* → *owned*)
- **Deletion:** Delete a single character (e.g., *thew* → *the*)
- **Addition:** Add a single character (e.g., *langage* → *language*)
- **Transposition:** Transpose or exchange a single pair of adjacent characters (e.g., *recieve* → *receive*)

- If you get the drift, you can see that we can generate the set of candidates for spelling correction by creating words by transforming a word using the above operations. So, the candidate set when,  $w = \text{"threw"}$ , might be:

$$C = \{\text{"thea"}, \text{"theb"}, \dots, \text{"the"}, \text{"hew"}, \dots, \text{"thewn"}, \text{"threw"}, \dots, \text{"thwe"}, \text{"tehw"}, \dots\}$$