NLP UNIT-1: Introduction to NLP, Regular Expressions, DFA

Natural Language Processing (NLP) is a subfield of **Artificial Intelligence (AI)** and **Linguistics** that enables machines to understand, interpret, and generate human languages. It bridges the gap between **human communication** and **computer understanding**.

Example:

• Voice assistants like Alexa, Siri, and Google Assistant.

• Machine translation systems (Google Translate).

• Chatbots and question-answering systems.

2. Importance of NLP

• Enables **human-computer interaction** in natural languages.

• Automates text processing tasks (translation, summarization, sentiment analysis).

• Useful in healthcare, finance, education, law, and customer service.

• Drives **data-driven decision-making** by extracting knowledge from large text datasets.

3. Core Components of NLP

1. **Syntax (Structure of Language):**

o Parsing, POS tagging, grammar analysis.

o Example: "The cat sat on the mat" → identifies subject, verb, object.

2. **Semantics (Meaning of Language):**

o Word sense disambiguation, semantic roles, logical forms.

o Example: “Bank” (riverbank vs. financial bank).

3. **Pragmatics (Context of Language):**

o Considers context, intent, and world knowledge.

o Example: “Can you open the window?” is a request, not a question about ability.

4. Common NLP Tasks

• **Tokenization:** Splitting text into words/sentences.

• **Stopword Removal:** Filtering common words like “is, the, a.”

• **Stemming & Lemmatization:** Reducing words to base form (running → run).

• **Named Entity Recognition (NER):** Identifying entities (e.g., names, places).

- **Sentiment Analysis:** Detecting opinions and emotions.
- **Machine Translation:** Translating text from one language to another.
- **Speech Processing:** Converting speech to text and vice versa.

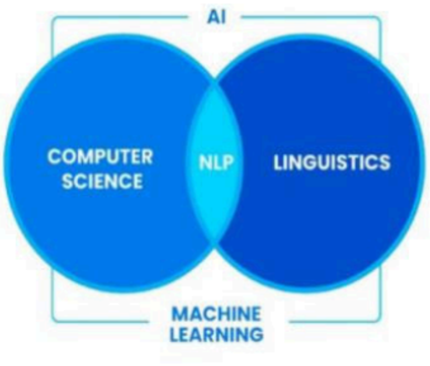
5. Challenges in NLP

- **Ambiguity:** Words and sentences may have multiple meanings.
- **Resource Scarcity:** Low-resource languages lack sufficient corpora.
- **Context Understanding:** Computers struggle with sarcasm, irony, and metaphors.
- **Multilinguality:** Handling multiple languages and dialects.

6. Applications of NLP

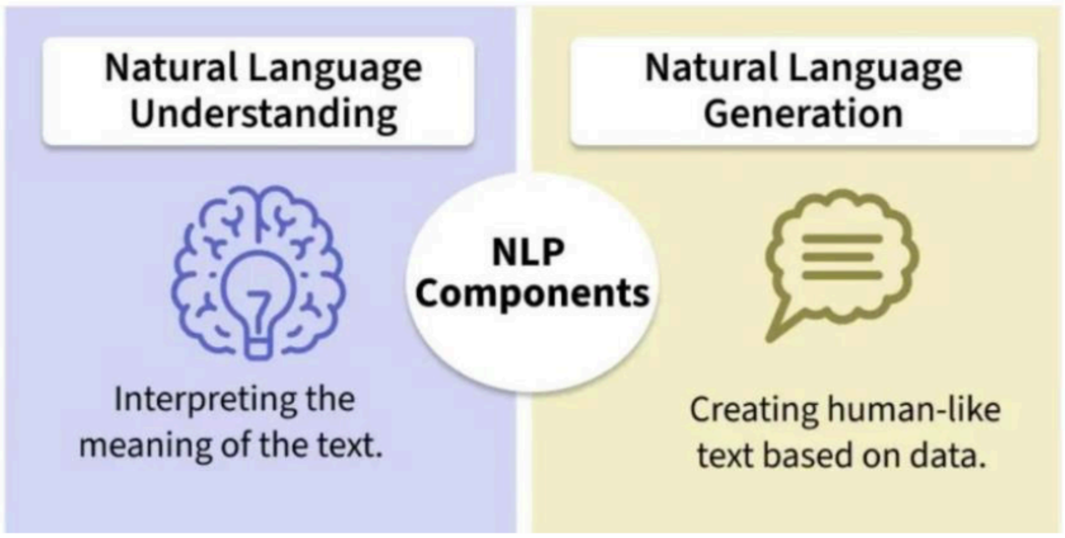
- **Search Engines** (Google, Bing).
- **Virtual Assistants** (Siri, Alexa).
- **Healthcare** (clinical text analysis, medical chatbots).
- **Social Media Analytics** (sentiment detection, trend analysis).
- **Education** (automated essay grading, language learning apps).

NLP UNIT-1: Introduction to NLP, Regular Expressions, DFA



What is Natural Language Processing?

- NLP helps machines to process human language (text or audio).
- Used in speech recognition, language translation and text summarization.


Natural Language Understanding
Interpreting the meaning of the text.
NLP Components
Natural Language Generation
Creating human-like text based on data.

Origins (Brief History of NLP)

Historically, speech and language processing has been treated very differently in computer science, electrical engineering, linguistics, and psychology/cognitive science. Because of this diversity, speech and language processing encompasses a number of different but overlapping fields in these different departments: **computational linguistics** in linguistics, **natural language processing** in computer science, **speech recognition** in electrical engineering, **computational psycholinguistics** in psychology.

Foundational Insights: 1940s and 1950s

The earliest roots of the field date to the intellectually fertile period just after World War II that gave rise to the computer itself. This period from the 1940s through the end of the 1950s saw intense work on two foundational paradigms: the **automaton** and **probabilistic** or **information-theoreticmodels**. The automaton arose in the 1950s out of Turing’s (1936) model of algorithmic computation, considered by many to be the foundation of modern computer science.

Turing's work led first to the **McCulloch-Pitts neuron** (McCulloch and Pitts, 1943), a simplified model of the neuron as a kind of computing element that could be described in terms of propositional logic, and then to the work of Kleene (1951) and (1956) on finite automata and regular expressions. Shannon (1948) applied probabilistic models of discrete Markov processes to automata for language. Drawing the idea of a finitestate Markov process from Shannon's work, Chomsky (1956) first considered finitestate machines as a way to characterize a grammar, and defined a finite-state language as a language generated by a finite-state grammar. These early models led to the field of **formal language theory**, which used algebra and set theory to define formal languages as sequences of symbols. This includes the context-free grammar, first defined by Chomsky (1956) for natural languages but independently discovered by Backus (1959) and Naur et al. (1960) in their descriptions of the ALGOL programming language. The second foundational insight of this period was the development of probabilistic algorithms for speech and language processing, which dates to Shannon's other contribution: the metaphor of the **noisy channel** and **decoding** for the transmission of language through media like communication channels and speech acoustics. Shannon also borrowed the concept of **entropy** from thermodynamics as a way of measuring the information capacity of a channel, or the information content of a language, and performed the first measure of the entropy of English using probabilistic techniques.

It was also during this early period that the sound spectrograph was developed (Koenig et al., 1946), and foundational research was done in instrumental phonetics that laid the groundwork for later work in speech recognition. This led to the first machine speech recognizers in the early 1950s. In 1952, researchers at Bell Labs built a statistical system that could recognize any of the 10 digits from a single speaker (Davis et al., 1952). The system had 10 speaker-dependent stored patterns roughly representing the first two vowel formants in the digits. They achieved 97–99%accuracy by choosing the pattern which had the highest relative correlation coefficient with the input.

The Two Camps: 1957–1970

By the end of the 1950s and the early 1960s, speech and language processing had split very cleanly into two paradigms: symbolic and stochastic. The symbolic paradigm took off from two lines of research. The first was the work of Chomsky and others on formal language theory and generative syntax throughout the late 1950s and early to mid 1960s, and the work of many linguistics and computer scientists on parsing algorithms, initially top-down and bottom-up and then via dynamic programming. One of the earliest complete parsing systems was Zelig Harris's Transformations and Discourse Analysis Project (TDAP), which was implemented between June 1958 and July 1959 at the University of Pennsylvania (Harris, 1962).2 The second line of research was the new field of artificial intelligence. In the summer of 1956 John McCarthy, Marvin Minsky, Claude Shannon, and Nathaniel Rochester brought together a group of researchers for a two-month workshop on what they decided to call artificial intelligence (AI). Although AI always included a minority of researchers focusing on stochastic and statistical algorithms (include probabilistic models and neural nets), the major focus of the new field was the work on reasoning and logic typified by Newell and Simon’s work on the Logic Theorist and the General Problem Solver. At this point early natural language understanding systems were built, These were simple systems that worked in single domains mainly by a combination of pattern matching and keyword search with simple heuristics for reasoning and question-answering. By the late 1960s more formal logical systems were developed. The stochastic paradigm took hold mainly in departments of statistics and of electrical engineering. By the late 1950s the Bayesian method was beginning to be applied to the problem of optical character recognition. Bledsoe and Browning (1959) built a Bayesian system for text-recognition that used a large dictionary and computed the likelihood of each observed letter sequence given each word in the dictionary by multiplying the likelihoods for each letter. Mosteller andWallace (1964) applied Bayesian methods to the problem of authorship attribution on *The Federalist* papers. The 1960s also saw the rise of the first serious testable psychological models of human language processing based on transformational grammar, as well as the first on-line corpora: the Brown corpus of American English, a 1 million word collection of samples from 500 written texts from different genres (newspaper, novels, non-fiction, academic, etc.), which was assembled at Brown University in 1963–64 (Kuˇcera and Francis, 1967; Francis, 1979; Francis and Kuˇcera, 1982), and William S. Y. Wang’s 1967 DOC (Dictionary on Computer), an on-line Chinese dialect dictionary.

Four Paradigms: 1970–1983

The next period saw an explosion in research in speech and language processing and the development of a number of research paradigms that still dominate the field. The **stochastic** paradigm played a huge role in the development of speech recognition algorithms in this period, particularly the use of the Hidden Markov Model and the metaphors of the noisy channel and decoding, developed independently by Jelinek, Bahl, Mercer, and colleagues at IBM’s Thomas J. Watson Research Center, and by Baker at Carnegie Mellon University, who was influenced by the work of Baum and colleagues at the Institute for Defense Analyses in Princeton. AT&T’s Bell Laboratories was also a center for work on speech recognition and synthesis; see Rabiner and Juang (1993) for descriptions of the wide range of this work. The **logic-based** paradigm was begun by the work of Colmerauer and his colleagues on Q-systems and metamorphosis grammars (Colmerauer, 1970, 1975), the forerunners of Prolog, and Definite Clause Grammars (Pereira and Warren, 1980). Independently, Kay’s (1979) work on functional grammar, and shortly later, Bresnan and Kaplan’s (1982) work on LFG, established the importance of feature structure unification. The **natural language understanding** field took off during this period, beginning with TerryWinograd’s SHRDLU system, which simulated a robot embedded in a world of toy blocks (Winograd, 1972). The program was able to accept natural language text commands *(Move the red block on top of the smaller green one)* of a hitherto unseen complexity and sophistication. His system was also the first to attempt to build an extensive (for the time) grammar of English, based on Halliday's systemic grammar.

Winograd's model made it clear that the problem of parsing was well-enough understood to begin to focus on semantics and discourse models. Roger Schank and his colleagues and students (in what was often referred to as the *Yale School*) built a series of language understanding programs that focused on human conceptual knowledge such as scripts, plans and goals, and human memory organization (Schank and Albelson, 1977; Schank and Riesbeck, 1981; Cullingford, 1981; Wilensky, 1983; Lehnert, 1977). This work often used network-based semantics (Quillian, 1968; Norman and Rumelhart, 1975; Schank, 1972; Wilks, 1975b, 1975a; Kintsch, 1974) and began to incorporate Fillmore's notion of case roles (Fillmore, 1968) into their representations (Simmons, 1973). The logic-based and natural-language understanding paradigms were unified on systems that used predicate logic as a semantic representation, such as the LUNAR question-answering system (Woods, 1967, 1973).

The **discourse modeling** paradigm focused on four key areas in discourse. Grosz and her colleagues introduced the study of substructure in discourse, and of discourse focus (Grosz, 1977; Sidner, 1983), a number of researchers began to work on automatic reference resolution (Hobbs, 1978), and the **BDI** (Belief-Desire-Intention) framework for logic-based work on speech acts was developed (Perrault and Allen, 1980; Cohen and Perrault, 1979).

Empiricism and Finite State Models Redux: 1983–1993

This next decade saw the return of two classes of models which had lost popularity in the late 1950s and early 1960s, partially due to theoretical arguments against them such as Chomsky's influential review of Skinner's *Verbal Behavior* (Chomsky, 1959). The first class was finite-state models, which began to receive attention again after work on finite-state phonology and morphology by Kaplan and Kay (1981) and finite-state models of syntax by Church (1980). A large body of work on finite-state models will be described throughout the book. The second trend in this period was what has been called the "return of empiricism"; most notably here was the rise of probabilistic models throughout speech and language processing, influenced strongly by the work at the IBM Thomas J. Watson Research Center on probabilistic models of speech recognition. These probabilistic methods and other such data-driven approaches spread from speech into part-of-speech tagging, parsing and attachment ambiguities, and semantics. This empirical direction was also accompanied by a new focus on model evaluation, based on using held-out data, developing quantitative metrics for evaluation, and emphasizing the comparison of performance on these metrics with previous published research. This period also saw considerable work on natural language generation.

The Field Comes Together: 1994–1999

By the last five years of the millennium it was clear that the field was vastly changing. First, probabilistic and data-driven models had become quite standard throughout natural language processing. Algorithms for parsing, part-of-speech tagging, reference resolution, and discourse processing all began to incorporate probabilities, and employ evaluation methodologies borrowed from speech recognition and information retrieval. Second, the increases in the speed and memory of computers had allowed commercial exploitation of a number of subareas of speech and language processing, in particular speech recognition and spelling and grammar checking. Speech and language processing algorithms began to be applied to Augmentative and Alternative Communication (AAC). Finally, the rise of the Web emphasized the need for language-based information retrieval and information extraction.

The Rise of Machine Learning: 2000–2007

The empiricist trends begun in the latter part of the 1990s accelerated at an astounding pace in the new century. This acceleration was largely driven by three synergistic trends. First, large amounts of spoken and written material became widely available through the auspices of the Linguistic Data Consortium (LDC), and other similar organizations. Importantly, included among these materials were annotated collections such as the Penn Treebank(Marcus et al., 1993), Prague Dependency Treebank(Hajiˇc, 1998), PropBank(Palmer et al., 2005), Penn Discourse Treebank(Miltsakaki et al., 2004), RSTBank(Carlson et al., 2001) and TimeBank(?), all of which layered standard text sources with various forms of syntactic, semantic and pragmatic annotations. The existence of these resources promoted the trend of casting more complex traditional problems, such as parsing and semantic analysis, as problems in supervised machine learning. These resources also promoted the establishment of additional competitive evaluations for parsing (Dejean and Tjong Kim Sang, 2001), information extraction, word sense disambiguation(Palmer et al., 2001; Kilgarriff and Palmer, 2000) and question answering(Voorhees and Tice, 1999). Second, this increased focus on learning led to a more serious interplay with the statistical machine learning community. Techniques such as support vector machines (Vapnik, 1995), multinomial logistic regression (MaxEnt) (Berger et al., 1996), and graphical Bayesian models (Pearl, 1988) became standard practice in computational linguistics. Third, the widespread availability of high-performance computing systems facilitated the training and deployment of systems that could not have been imagined a decade earlier. Finally, near the end of this period, largely unsupervised statistical approaches began to receive renewed attention. Progress on statistical approaches to machine translation(Brown et al., 1990; Och and Ney, 2003) and topic modeling demonstrated that effective applications could be constructed from systems trained on unannotated data alone. In addition, the widespread cost and difficulty of producing reliably annotated corpora became a limiting factor in the use of supervised approaches for many problems. This trend towards the use unsupervised techniques will likely increase.

NLP UNIT-1: Introduction to NLP, Regular Expressions, DFA

Challenges in NLP

• **Ambiguity**

o Words/sentences can have multiple meanings (e.g., *"bank"* → *river bank vs.*

financial bank).

• **Context Understanding**

o Handling sarcasm, irony, pragmatics, and discourse-level understanding.

• **Multilinguality**

o Different grammar, word order, morphology across languages.

• **Low-Resource Languages**

o Many Indian and African languages lack annotated corpora.

• **World Knowledge Integration**

o Machines need background knowledge to interpret meaning correctly.

• **Bias and Fairness**

o Models can amplify social, gender, or cultural biases from training data.

• **Scalability**

o Large transformer models require **huge computational resources**.

• **Ethical Issues**

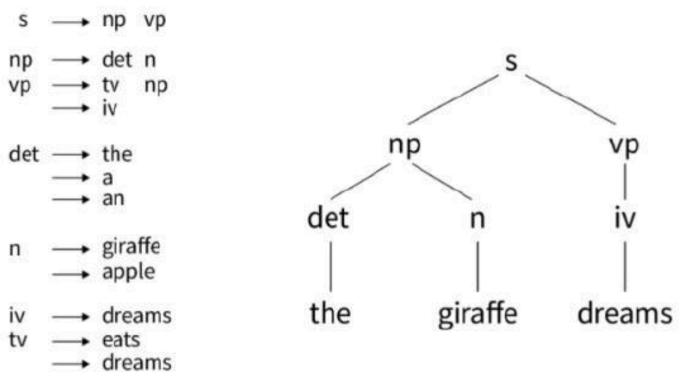
- Misinformation, deepfakes, surveillance, and misuse of NLP technology.

Language and Grammar in NLP

Language and grammar are the foundation of NLP. Grammar formalisms like CFG, PSG, and

Dependency Grammar help computationally model language, while challenges like ambiguity

and idioms make the task complex.



- Defines a set of production rules that describe all possible sentences in a language.
- Useful in parsing sentences and building syntax trees.
- **Example rule:**

$$S \rightarrow NP\ VP$$

$$NP \rightarrow Det\ Noun$$

$$VP \rightarrow Verb\ NP$$

1. Role of Language in NLP

• Language is the **medium of communication** between humans and machines.

• NLP aims to model both the **structure (syntax)** and **meaning (semantics)** of language for computational use.

• A system must handle **words, phrases, clauses, and sentences** while preserving meaning.

2. Grammar in NLP

Grammar defines the **rules and structure** of a language. In NLP, grammar is essential for:

• **Parsing**: Analyzing sentence structure.

• **Machine Translation**: Ensuring syntactic correctness.

• **Question Answering**: Understanding sentence intent.

3. Types of Grammar in NLP

(a) Morphology

• Study of **word formation** and structure.

- Example: *play → plays, playing, played.*

(b) Syntax

- Study of **sentence structure** (rules for arranging words).
- Example:

o Correct: *The cat sat on the mat.*

o Incorrect: *Cat mat sat the on.*

(c) Semantics

- Deals with **meaning of words and sentences**.
- Example: *John kicked the ball* vs. *The ball kicked John* (different meanings).

(d) Pragmatics

- Study of **contextual meaning** (depends on situation).
- Example: *"Can you pass the salt?"* → a request, not a question about ability.

(e) Discourse

- Concerned with **multiple sentences together**.
- Example: *She went to the market. She bought apples.* → pronoun *she* refers to same person.

(f) Phonology (Speech-based NLP)

- Study of **sound systems** in language.

4. Grammar Formalisms in NLP

1. **Phrase Structure Grammar (PSG)**

o Represents sentences as hierarchical structures.

o Example: Sentence (S) → Noun Phrase (NP) + Verb Phrase (VP).

2. **Dependency Grammar**

o Represents grammatical relationships as **dependencies between words**.

o Example: In *"She eats apples"*, *eats* → head verb, *she* → subject, *apples* → object.

3. **Context-Free Grammar (CFG)**

o Uses rules like **S → NP VP** to generate valid sentences.

4. **Transformational Grammar** (Chomsky)

o Describes how deep structures can be transformed into surface forms.

5. Challenges in Using Grammar for NLP

• **Ambiguity:**

o *"I saw the man with a telescope"* → who has the telescope?

• **Ellipsis:** Missing words in sentences.

• **Idioms:** Non-literal expressions (*"kick the bucket"* ≠ kick + bucket).

• **Cross-linguistic variation:** Different languages follow different grammar rules.

Regular Expressions

One of the unsung successes in standardization in computer science has been the **regular expression** (**RE**), a language for specifying text search REGULAR strings. The regular expression languages used for searching texts in UNIX (vi, Perl, Emacs, grep), Microsoft Word (version 6 and beyond), and WordPerfect are almost identical, and many RE features exist in the various Web search engines. Besides this practical use, the regular expression is an important theoretical tool throughout computer science and linguistics.

A regular expression (first developed by Kleene (1956) but see the History section for more details) is a formula in a special language that is used for specifying simple STRINGS classes of **strings**. A string is a sequence of symbols; for the purpose of most text based search techniques, a string is any sequence of alphanumeric characters (letters, numbers, spaces, tabs, and punctuation). For these purposes a space is just a character like any other, and we represent it with the symbol .

Formally, a regular expression is an algebraic notation for characterizing a set of strings. Thus they can be used to specify search strings as well as to define a language in a formal way. We will begin by talking about regular expressions as a way of specifying searches in texts, and proceed to other uses.

Definition: A regular expression is a sequence of symbols and characters that defines a

search pattern, mainly for string matching and text processing.

• **Role in NLP**:

o Tokenization (splitting sentences/words).

o Pattern matching (finding dates, phone numbers, emails).

o Text normalization (removing punctuation, replacing characters).

o Morphological analysis (finding word stems, suffixes, prefixes).

• **Examples**:

o \d+ → matches numbers (e.g., 123).

o [A-Z][a-z]+ → matches proper nouns (e.g., India, John).

Basic Matchers of RE

Format	**Description**	**Example**
abc	Literal characters	cat
.	Any single character (except newline)	b.t
\	Escapes a special character	\.
|	Alternation (OR)	gray|grey

Character Sets in RE

Format	**Description**	**Example**
[abc]	Any one character in the brackets	[bt]at
[^abc]	Any character NOT in the brackets	[^b]at
[a-z]	Range of lowercase letters	[a-c]at
[0-9]	Any single digit	ID-[0-9]

Meta Characters in RE

Format	Description	Example
\d	Any digit (same as [0-9])	\d\d
\D	Any non-digit	\D\D
\w	Word character (alphanumeric + _)	\w
\W	Non-word character	\W
\s	Whitespace (space, tab, newline)	\d\s\d

Quantifiers in RE

Format	Description	Example
*	0 or more times	hi*
+	1 or more times	hi+
?	0 or 1 time (optional)	colou?r
{n}	Exactly n times	\d{3}
{n,}	n or more times	\w{5,}
{n,m}	Between n and m times	\d{2,4}

Anchors in RE

Format	Description	Example
^	Start of a string/line	^Hello
$	End of a string/line	bye$
\b	Word boundary	\bcat\b

Grouping and Capturing in RE

Format	Description	Example
(...)	Capturing Group	(ha)+
(?:...)	Non-capturing Group	(?:dog|cat)
$1, $2	Back-references	Replace (\w+) (\w+)

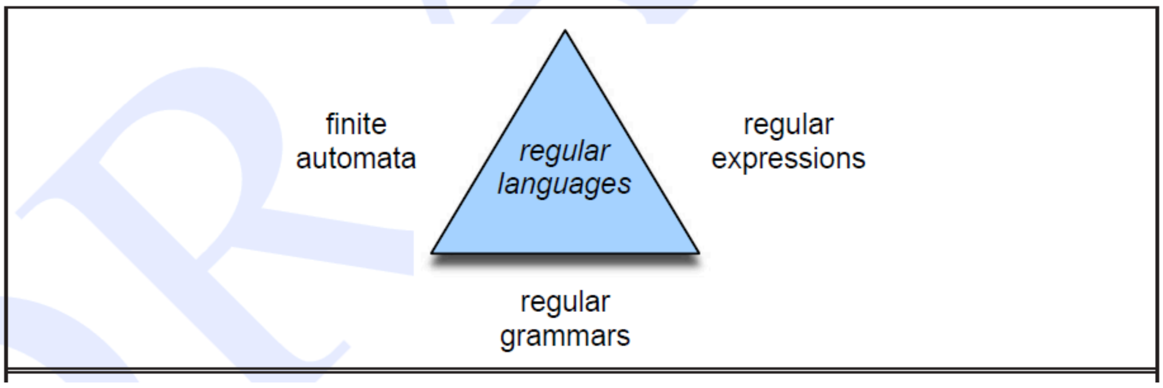
Some Basic Examples

Target Data	Pattern (Regex)
Email Address	^[\w.-]+@[\w.-]+\.[a-z]{2,}$
Phone (US)	^\d{3}-\d{3}-\d{4}$
Date (YYYY-MM-DD)	\d{4}-\d{2}-\d{2}
HTML Tags	<[^>]+>

NLP UNIT-1: Introduction to NLP, Regular Expressions, DFA

Finite State Automata

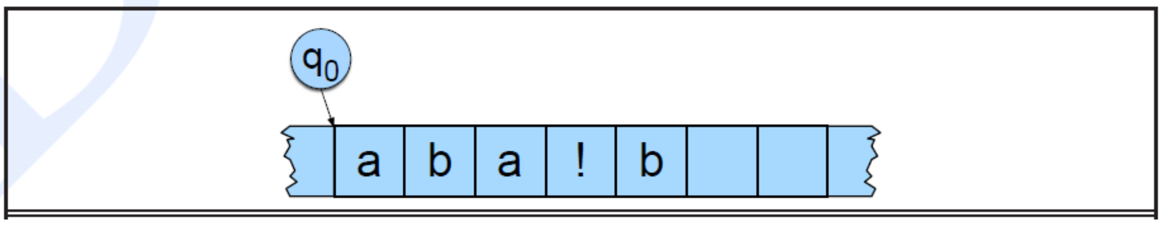
The regular expression is more than just a convenient metalanguage for text searching. First, a regular expression is one way of describing a **finite-state automaton** (**FSA**). FSA Finite-state automata are the theoretical foundation of a good deal of the computational work. Any regular expression can be implemented as a finite-state automaton. Symmetrically, any finite-state automaton can be described with a regular expression. Second, a regular expression is one way of characterizing a particular kind REGULAR LANGUAGE of formal language called a **regular language**. Both regular expressions and finite state automata can be used to describe regular languages.



Deterministic Finite State Automata (DFA)

The automaton (i.e., machine,also called **finite automaton**, **finite-state automaton**, or **FSA**) recognizes a set of strings, in this case the strings characterizing sheep talk, in the same way that a regular expression does. We represent the automaton as a directed graph: a finite set of vertices (also called nodes), together with a set of directed links between pairs of vertices called arcs. We'll represent vertices with circles and arcs with arrows. The automaton has five STATES **states**s, which are represented by nodes in the graph. State 0 is the **start state**. In our START STATE examples state 0 will generally be the start state; to mark another state as the start state we can add an incoming arrow to the start state. State 4 is the **final state** or **accepting state**, which we represent by the double circle. It also has four **transitions**, which we represent by arcs in the graph.

The FSA can be used for recognizing (we also say **accepting**) strings in the following way. First, think of the input as being written on a long tape broken up into cells, with one symbol written in each cell of the tape.



The machine starts in the start state (*q*0), and iterates the following process: Check the next letter of the input. If it matches the symbol on an arc leaving the current state, then cross that arc, move to the next state, and also advance one symbol in the input. If we are in the accepting state (*q*4) when we run out of input, the machine has successfully recognized an instance of sheeptalk. If the machine never gets to the final state, either because it runs out of input, or it gets some input that doesn't match an arc (as in Fig. 2.11), or if it just happens to get stuck in some non-final state, we say the machine **rejects** or fails to accept an input.

We can also represent an automaton with a **state-transition table**. As in the graph TABLE notation, the state-transition table represents the start state, the accepting states, and what transitions leave each state with which symbols. Here's the state-transition table for the FSA.

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

We've marked state 4 with a colon to indicate that it's a final state (you can have as many final states as you want), and the /0 indicates an illegal or missing transition. We can read the first row as "if we're in state 0 and we see the input **b** we must go to state 1. If we're in state 0 and we see the input **a** or **!**, we fail".

More formally, a finite automaton is defined by the following five parameters:

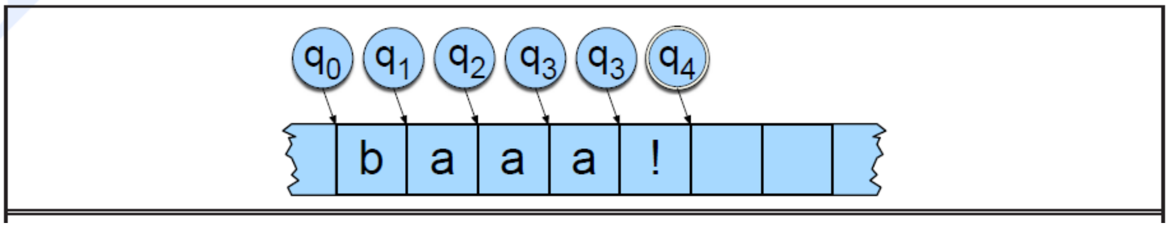
Q = *q*0*q*1*q*2 . . .*qN*−1 a finite set of *N* **states**

S a finite **input alphabet** of symbols

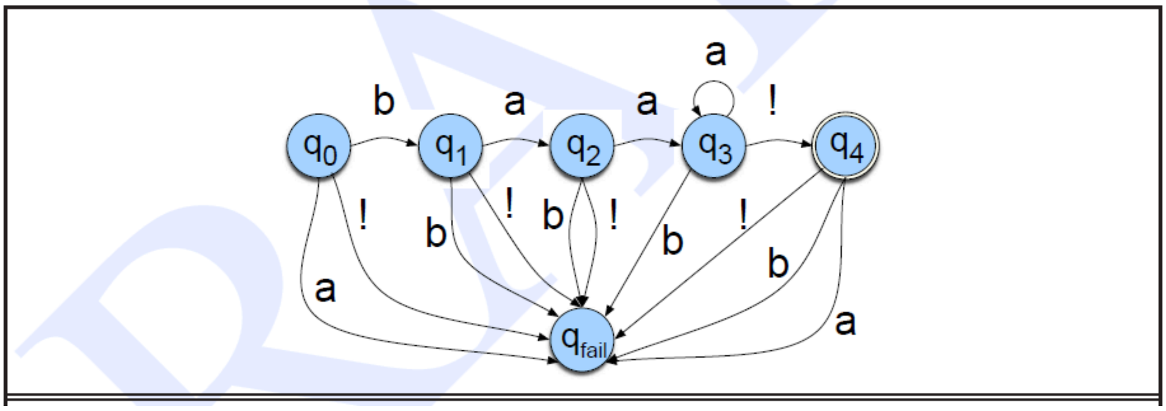
*q*0 the **start state**

F the set of **final states**, $F \subseteq Q$

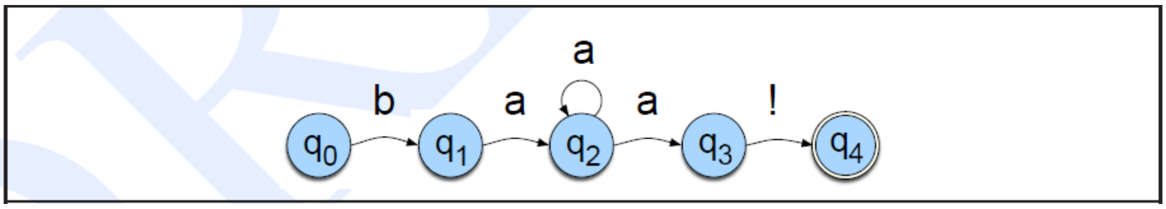
d(*q*, *i*) the **transition function** or transition matrix between states. Given a state *q* ∈ *Q* and an input symbol *i* ∈ S, d(*q*, *i*) returns a new state *q'* ∈ *Q*. d is thus a relation from *Q*×S to *Q*;



The input *abc* will fail to be recognized since there is no legal transition out of state *q*0 on the input a. Even if the automaton had allowed an initial *a* it would have certainly failed on *c*, since *c* isn't even in the sheeptalk alphabet! We can think of these "empty" elements in the table as if they all pointed at one "empty" state, which we might call the **fail state** or **sink state**. In a sense then, we could FAIL STATE view any machine with empty transitions *as if* we had augmented it with a fail state, and drawn in all the extra arcs, so we always had somewhere to go from any state on any possible input.

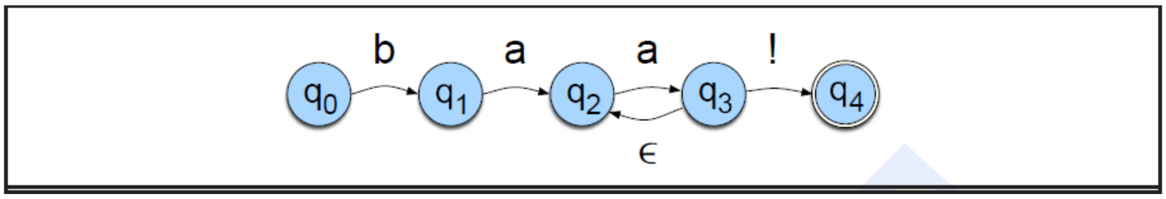


Non Deterministic FSA's



The only difference between this automaton and the previous one is that here the self-loop is on state 2 instead of state 3. Consider using this network as an automaton for recognizing sheeptalk. When we get to state 2, if we see an **a** we don't know whether to remain in state 2 or go on to state 3. Automata with decision points like this are called **non-deterministic FSAs** (or **NFSAs**). Recall by contrast NFSA that Figure 2.10 specified a **deterministic** automaton, i.e., one whose behavior during recognition is fully *determined* by the state it is in and the symbol it is looking at. A deterministic automaton can be referred to as a **DFSA**.

There is another common type of non-determinism, caused by arcs that have no symbols on them (called e-**transitions**).



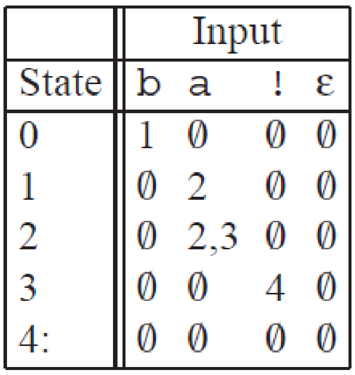
If we want to know whether a string is an instance of regular expression or not, and if we use a non-deterministic machine to recognize it, we might follow the wrong arc and reject it when we should have accepted it. That is, since there is more than one choice at some point, we might take the wrong choice. This problem of choice in non-deterministic models will come up again and again as we build computational models, particularly for parsing. There are three standard **solutions to the problem of non-determinism**:

• **Backup:** Whenever we come to a choice point, we could put a *marker* to mark where we were in the input, and what state the automaton was in. Then if it turns out that we took the wrong choice, we could back up and try another path.

• **Look-ahead:** We could look ahead in the input to help us decide which path totake.

• **Parallelism:** Whenever we come to a choice point, we could look at every alternative path in parallel.

The backup approach suggests that we should blithely make choices that might lead to deadends, knowing that we can always return to unexplored alternative choices. There are two keys to this approach: we need to remember all the alternatives for each choice point, and we need to store sufficient information about each alternative so that we can return to it when necessary. When a backup algorithm reaches a point in its processing where no progress can be made (because it runs out of input, or has no legal transitions), it returns to a previous choice point, selects one of the unexplored alternatives, and continues from there. Applying this notion to our non-deterministic recognizer, we need only remember two things for each choice point: the state, or node, of the machine that we can go to and the corresponding position on the tape. We will call the combination of the node and position the **search-state** of the recognition algorithm. To avoid confusion, we will refer to the state of the automaton (as opposed to the state of the search) as a **node** or a **machine-state.**



	Input			
State	b	a	!	ε
0	1	∅	∅	∅
1	∅	2	∅	∅
2	∅	2,3	∅	∅
3	∅	∅	4	∅
4:	∅	∅	∅	∅