Experiment No: 01 Lexical Analyzer

- **Aim:**
  To design & implement a lexical analyzer in python that processes a sentence & generates tokens (words, Punctuation, identifiers).

- **Theory:**
  A lexical analyzer (also called a lexer or scanner) is the first phase of a Compiler or NLP pipeline.
  - It reads the input sentence character by character.
  - Groups characters into meaningful sequences called tokens (e.g. words, punctuation).
  - Removes unnecessary whitespace.
  - Tokens are the smallest meaningful units for further processing (like syntax analysis).

- **Example:**
  Input: "Sachin Tendulkar" is called the God of Cricket"
  Tokens: ['Sachin', 'Tendulkar', 'is', 'called', 'the', 'God', 'of', 'Cricket']

- **objective:**
  - To understand how Text is broken into tokens
  - To Prepare data for Syntax & Semantic analysis.
  - To implement lexical analysis in python.

- Conclusion :
  - We successfully developed a lexical analyzer in Python.
  - It splits the gives sentences into tokens that can be used for further stages (syntax, semantics, etc)
  - This step is crucial in NLP as it prepares raw text for structured processing.

Experiment No: 02: Syntax Analyzer

- **Aim:**
  To design & implement a syntax analyzer in python that checks the grammatical structure of tokens generated by the lexical analyzer.

- **Theory:**
  A Syntax analyzer (parser) is the second phase of NLP / compilers.
  - It takes the sequence of tokens from the lexical analyzer.
  - verifies whether the sequence follows correct grammer rules.
  - If correct, It generates a parse tree & confirms Validity
  - otherwise, It Reports syntax errors.

- **Example:**
  Tokens: ['Sachin', 'Tendulkar', 'is', 'called', 'the', 'God', 'OF', 'Cricket']

- **Objective:**
  - To validate the syntactic structure of given tokens.
  - To ensure the input follows basic grammar rules.
  - To build a fundation for semantic analysis

- **Conclusion:**

- Implemented a syntax analyer using a simple grammer.
- verified the takes arrangement according to grammer rules.
- Demonstared how parsing helps in understanding sentence structrue.

Experiment No 03. Rule Base Model

- **Aim:**
  To design a simple rule-Based NLP Model in Python that responds to user input based on predefined patterns.

- **Theory:**
  A rule-based model relies on manually created rules:
  - Rules are written as if-else conditions of pattern matching.
  - No machine learning is required.
  - Used in early Chatbots (like ELIZA)
  - Example: if input Contains "Hello", reply "Hi there!".

- **Objective:**
  - To implement a basic rule-based model.
  - To understand how early NLP systems responded to user queries.

- **Conclusion:**
  - Rule-based models are simple but limited.
  - They can only respond to predefined patterns.
  - Foundation for advanced dialogue systems.

Experiment No. 04: Statistical Model.

- **Aim:**
  To implement a simple statistical NLP model using word co-occurrence from a small training corpus.

- **Theory:**
  A statistical model uses probabilities from data instead of fixed rules.
  - Trained on a corpus of sentences.
  - learns word frequencies and probabilities.
  - Example: If "Python" appears often with "love", the model predicts relation.

- **Objective:**
  - To understand frequency-based NLP Models.
  - To train a simple word-statistics model.

- **Conclusion:**
  - Statistical Models are more ~~fel~~ flexible than rule-based.
  - They adapt to data instead of fixed rules.

Experiment No. 05 (i) Shift - Reduce Parser

- **Aim:**
  To implement a shift - Reduce parser in python for checking sentence structure

- **Theory:**
  - A shift - Reduce parser is a bottom - up - parser
  - It usses a stack & an input buffer to stack
  - operations
    - shift → move a token from input buffer to stack
    - Reduce → Replace a sequence of tokens in stack with a non - terminal
  - Parsing Continues Until stack Reduces to the start Symbol.

- **Objective:**
  - To demonstrate how parsing can be done using stack operations.
  - To check if the sentence follows the grammer.

- **Conclusion:-**
  - Implement a shift - Reduce parser.
  - Verified sentence grammer using stack operation

(ii) Recursive Descent Parser

- **Aim:**
  To implement a Recursive Descent parser in Python for sentence parsing.

- **Theory:**
  - A Recursive Descent parser is a top-down parser
  - Implements grammer rules as Recursive function
  - Example: S → NP VP
    NP → 'sachin' 'Tendulkar'
    VP → 'is' 'called' NP

- **Objective:**
  - To demonstrate top-down parsing using recursive function
  - To validate sentence grammer

- **Conclusion:**
  - Recursive Descent parser successfully validate sentences.
  - useful for simple grammarsbut not efficient for large ambiguous grammers.

Experiment No: 06. Semantic Analyzer

- Aim:
  To implement a Semantic Analyzer in Python to check the meaning / consistency of a sentence after syntax analysis.

- Theory:
  - The semantics deals with the meaning of a sentence, Not just grammer.
  - A semantic Analyzer ensures
    - words are compatible
    - Subjects & verbs make logical sense.
  - Common approach:
    - Rule Based checks
    - word embeldings
    - ontology- based models
  - Example:
  - Sentence: "Sachin Tendulkar plays cricket"
  → Valid
  - Sentence: "cricket plays sachin Tendulkar."
  → Invalid

- objective:
  - To check if the tokens from a meaningful Sentence
  - To move beyond grammer into logical correctess

- Conclusion:
  - Implemented a simple semantic analyzer
  - Demonstrated how semantic rules can detect
  - meaningful vs. meaningless sentences.

Experiment NO-07: N-Gram Models

Aim:

To implement N-Gram models (Unigrams, Bigrams, Trigrams) & compute their frequency in python

Theory:

- An N-Gram is a sequence of N consecutive words in a sentence of corpus.
- Types
  - Unigram (1-word) → Example: "I", "Love", "NLP".
  - Bigram (2 words) → Example: "I love", "love NLP"
  - Trigram (3-words) → Example: "I Love NLP"
- Application
  - Predicting next words
  - Text generation
  - Language modeling.

objective:

- To generate Unigrams, bigrams, trigrams from a corpus
- To compute their frequencies for probability based NLP models.

Conclusion:

- Successfully generated N-Grams
- observed how frequency distribution helps in building statistical language models.

Experiment No: 08 Language model Evaluation Using perplexity

- **Aim:**
  To evaluate a language model using preplexity (PPL) in python

- **Theory:**
  - Perplexity is a metric used to evaluate how well a language model predicts a sample.
  - Formula :-

$$PPL = 2^{-\frac{1}{N}\sum_{i=1}^{N} \cdot \log_2 P(w_i / w_{i-n+1}^{i-1})}$$

- **Where:**
  - N = total words
  - P(wi/ (context) = Probability of words given previous words.
  - Lower Perplexity → Better model

- **Objective:**
  - To implement Preplexity Calculation For a given corpus
  - To evaluate Unigram / bigram models.

- **Conclusion:**
  - Perplexity provides a numerical measure of model quality.
  - Lower PPL indicates amore accurate predictive model.

Experiment - 09: Bayesian Parameter Estimation

- **Aim:**
  To implement Bayesian parameter Estimation in Python for NLP models.

- **Theory:**
  - Bayes' Theorem:

$$P(H/D) = \frac{P(D/H) \cdot P(H)}{P(D)}$$

  - $P(H/D)$ → posterior Probability.
  - $P(H)$ → prior probability.
  - $P(D/H)$ → Likelihood of Data given hypothesis
  - $P(D)$ → Evidence
  - MLE → (maximum likelihood Estimation)
    Choose parameters that maximize likelihood
    chooses parameters that maximize likelihood.
  - MAP (maximum A Posteriori):
    - Uses priort likelihood to estimate parameter
  - Example in NLP:
    Estimating probability of a word occurring in a sentence.

- **objective:**
  - To apply Bayesian estimation to NLP tasks.
  - To show difference between MLE & MAP estimation.

- **Conclusion:**

- Bayesian Estimation Refines probability by combining prior knowledge with observed data
- Helps in better word prediction & smoothing in NLP.