NATURAL LANGUAGE PROCESSING LAB MANUAL

PROLOG:

1) Write a program in prolog to implement simple facts and Queries

Certainly! Prolog is a logic programming language. In Prolog, you define facts and rules, and you can then query the system to retrieve information based on those facts and rules. Here's a simple example:

```prolog
% Facts
father(john, jim).
father(john, ann).
father(bob, sara).
mother(jane, jim).
mother(jane, ann).
mother(amy, sara).

% Rules
parent(X, Y) :- father(X, Y).
parent(X, Y) :- mother(X, Y).

% Queries
% Is John the father of Jim?
% Query: ?- father(john, jim).
% Expected output: true.
%
% Who are the children of John?
% Query: ?- parent(john, X).
% Expected output: X = jim ; X = ann.
%
% Is Jane the mother of Jim?
% Query: ?- mother(jane, jim).
% Expected output: true.
%
% Who are the parents of Sara?
% Query: ?- parent(X, sara).
% Expected output: X = bob ; X = amy.
```

~~~
```
~~~

In this example, we have facts about fathers and mothers, and we define a rule `parent/2` that defines someone as a parent if they are a father or a mother. The queries at the end demonstrate how to ask Prolog for information based on the defined facts and rules.

2) Write a program in prolog to implement simple arithmetic

Certainly! Prolog can be used for simple arithmetic operations. Here's an example program with some basic arithmetic facts and queries:

~~~
```prolog
% Facts
addition(X, Y, Result) :- Result is X + Y.
subtraction(X, Y, Result) :- Result is X - Y.
multiplication(X, Y, Result) :- Result is X * Y.
division(X, Y, Result) :- Y =\= 0, Result is X / Y.

% Queries
% Addition: What is the result of adding 5 and 3?
% Query: ?- addition(5, 3, Result).
% Expected output: Result = 8.

% Subtraction: What is the result of subtracting 7 from 10?
% Query: ?- subtraction(10, 7, Result).
% Expected output: Result = 3.

% Multiplication: What is the result of multiplying 4 and 6?
% Query: ?- multiplication(4, 6, Result).
% Expected output: Result = 24.

% Division: What is the result of dividing 9 by 3?
% Query: ?- division(9, 3, Result).
% Expected output: Result = 3.

% Division by zero (should fail):
~~~

```
% Query: ?- division(5, 0, Result).

% Expected output: false.
```

In this example, we have facts for basic arithmetic operations like addition, subtraction, multiplication, and division. The queries demonstrate how to use these facts to perform arithmetic operations and get the results. Note that the division operation checks for division by zero to avoid errors.

3) Write a program in prolog to solve Monkey banana problem

The Monkey and Banana problem is a classic puzzle where a monkey is in a room with a bunch of bananas hanging from the ceiling. The monkey wants to reach the bananas, and there is a chair in the room that the monkey can use to climb and reach the bananas. However, the monkey can only carry one banana at a time.

Here's a simple Prolog program to solve the Monkey and Banana problem using state representation and actions:

```prolog
% Initial state: monkey at door, on floor, bananas on ceiling, chair at middle of the room.

initial_state(state(at_door, on_floor, at_ceiling, at_middle)).

% Final state: monkey has the bananas.

final_state(state(_, _, at_monkey, _)).

% Actions

% The monkey can walk to the middle of the room.

walk(state(at_door, on_floor, B, X), state(at_middle, on_floor, B, X)).

% The monkey can climb the chair.

climb(state(at_middle, on_floor, B, X), state(at_middle, on_chair, B, X)).

% The monkey can push the chair to the bananas.

push(state(P, on_floor, at_ceiling, P), state(P, on_floor, at_monkey, P)).

% Define the goal state and solve the problem.

goal_state(State) :- final_state(State).

% Plan to reach the goal state.
```

```
plan(State, []) :- goal_state(State).
plan(State, [Action | Actions]) :-
    perform(Action, State, NextState),
    plan(NextState, Actions).

% Perform actions
perform(Action, State, NextState) :-
    call(Action, State, NextState).

% Example query to solve the problem:
% ?- initial_state(InitialState), plan(InitialState, Plan).
```

This Prolog program defines the initial and final states, actions (walk, climb, push), and a plan to solve the problem. The `plan/2` predicate is used to find a sequence of actions that transform the initial state into the final state. The example query demonstrates how to use the program to find a solution.

4) Write a program in prolog to solve Tower of Hanoi

The Tower of Hanoi is a classic problem in computer science and mathematics. The problem consists of three pegs and a number of disks of different sizes. The task is to move the entire stack of disks from one peg to another, obeying the following simple rules:

1. Only one disk can be moved at a time.
2. Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack or on an empty peg.
3. No disk may be placed on top of a smaller disk.

Here is a Prolog program to solve the Tower of Hanoi problem:

```prolog
% Define the Tower of Hanoi predicate
hanoi(N) :- move(N, left, center, right).

% Define the base case for moving 0 disks (no action required)
move(0, _, _, _) :- !.

% Define the recursive case for moving N disks
```

```
move(N, Source, Target, Auxiliary) :-
    M is N - 1,
    move(M, Source, Auxiliary, Target),
    format('Move disk ~w from ~w to ~w~n', [N, Source, Target]),
    move(M, Auxiliary, Target, Source).

% Example query:
% ?- hanoi(3).
```

In this program, `hanoi/1` is the main predicate that initiates the Tower of Hanoi solving process. The `move/4` predicate defines the base case (moving 0 disks requires no action) and the recursive case for moving N disks. It prints the steps for moving disks between the source, target, and auxiliary pegs.

To use the program, you can query it with the desired number of disks, for example:

```prolog
?- hanoi(3).
```

This will output the sequence of moves needed to solve the Tower of Hanoi problem with 3 disks. You can replace `3` with any other positive integer to solve the problem with a different number of disks.

5) Write a program in prolog to solve 8 Puzzle problems

Solving the 8-Puzzle problem in Prolog involves representing the puzzle states, defining the possible moves, and implementing a search algorithm. Here's a simple program that uses a depth-first search to solve the 8-Puzzle:

```prolog
% Solving the 8-Puzzle problem

% Initial state representation: [1, 2, 3, 4, 5, 6, 7, 8, 0]
% 0 represents the empty tile.
initial_state([1, 2, 3, 4, 5, 6, 7, 8, 0]).

% Final state representation
final_state([1, 2, 3, 4, 5, 6, 7, 8, 0]).
```

```
% Move actions
move(Action, State, NewState) :-
    select(0, State, Empty, State1),
    select(N, State1, Action, State2),
    select(Empty, NewState, N, Action).

% Depth-first search
dfs(State, _, Actions) :- final_state(State), Actions = [].
dfs(State, Visited, [Action | Actions]) :-
    move(Action, State, NewState),
    \+ member(NewState, Visited),
    dfs(NewState, [NewState | Visited], Actions).

% Solve the puzzle
solve(Actions) :-
    initial_state(InitialState),
    dfs(InitialState, [InitialState], Actions).

% Example query:
% ?- solve(Actions).
```

This program uses a simple depth-first search to find a sequence of moves (`Actions`) to reach the goal state. The `move/3` predicate defines the possible moves, and the `dfs/3` predicate implements the depth-first search algorithm. The `solve/1` predicate initiates the solving process.

You can run the example query `solve(Actions)` to find a solution. The result will be a list of actions that transform the initial state into the final state. The initial and final states are represented as lists where 0 represents the empty tile. You can modify the `initial_state` and `final_state` predicates to represent different initial and final states if needed.

6) Write a program in prolog to solve 4-Queens problem

The 4-Queens problem is a classic problem in chess where you need to place four queens on a 4x4 chessboard in such a way that no two queens threaten each other. In Prolog, you can solve this problem using a constraint logic programming approach. Here's a simple program to solve the 4-Queens problem:

```prolog
```

```
% 4-Queens problem

% The main predicate to solve the problem
four_queens(Board) :-
    Board = [Q1, Q2, Q3, Q4],
    % Each queen is in a different row
    all_distinct(Board),
    % Each queen is in a different column
    permutation(Board, [1, 2, 3, 4]),
    % Check for diagonal threats
    is_diagonal(Q1, 1, Q2, 2),
    is_diagonal(Q1, 1, Q3, 3),
    is_diagonal(Q1, 1, Q4, 4),
    is_diagonal(Q2, 2, Q3, 3),
    is_diagonal(Q2, 2, Q4, 4),
    is_diagonal(Q3, 3, Q4, 4).

% Check if two queens are in a diagonal threat
is_diagonal(I, R1, J, R2) :-
    RD is abs(R1 - R2),
    RD =:= abs(I - J).

% Example query:
% ?- four_queens(Board).
```

In this program, `four_queens/1` is the main predicate that represents the solution to the 4-Queens problem. It uses the `all_distinct/1` constraint to ensure that each queen is in a different row and the `permutation/2` constraint to ensure that each queen is in a different column. The `is_diagonal/4` predicate checks for diagonal threats between queens.

You can query `four_queens(Board)` to get a solution. The `Board` variable will be instantiated with a list representing the column positions of the queens on the chessboard. If there are multiple solutions, Prolog will backtrack and provide alternative solutions upon request.

7) Write a program in prolog to solve Traveling salesman problem

The Traveling Salesman Problem (TSP) is a classic optimization problem where the goal is to find the shortest possible route that visits a given set of cities and returns to the starting city. Solving TSP optimally for a large number of cities is computationally expensive, so heuristic methods are often used. Here's a simple program in Prolog using a brute-force approach to find the shortest Hamiltonian circuit (the circuit that visits each city exactly once):

```prolog
% Traveling Salesman Problem

% Define cities and their distances
distance(city1, city2, 10).
distance(city1, city3, 15).
distance(city1, city4, 20).
distance(city2, city3, 35).
distance(city2, city4, 25).
distance(city3, city4, 30).

% Find the shortest Hamiltonian circuit
tsp(Path, Cost) :-
    findall([Cost, Path], hamiltonian_circuit(Path, Cost), Solutions),
    sort(Solutions, [MinSolution | _]),
    member([Cost, Path], [MinSolution | _]).

% Find Hamiltonian circuits
hamiltonian_circuit(Path, Cost) :-
    cities(Cities),
    permutation(Cities, Path),
    path_cost(Path, Cost),
    closed_circuit(Path).

% Define the list of cities
cities([city1, city2, city3, city4]).

% Calculate the cost of a path
path_cost([_], 0).
path_cost([City1, City2 | Rest], Cost) :-
    distance(City1, City2, Distance),
    path_cost([City2 | Rest], RemainingCost),
    Cost is Distance + RemainingCost.

% Check if a path is a closed circuit
closed_circuit([First | Rest]) :-
    append(Rest, [First], Closed),
    all_distinct(Closed).

% Example query:
% ?- tsp(Path, Cost).
```

In this program, `tsp/2` is the main predicate that finds the shortest Hamiltonian circuit. It uses `hamiltonian_circuit/2` to find all Hamiltonian circuits and then selects the one with the minimum cost. The `distance/3` predicate defines the distances between cities, and `path_cost/2` calculates the cost of a given path.

Please note that this approach uses a brute-force method, which may not be efficient for large numbers of cities. More advanced algorithms, such as dynamic programming or heuristics like the nearest neighbor algorithm or genetic algorithms, are often used for practical applications of the Traveling Salesman Problem.

8) Write a program in prolog for Water jug problem

The Water Jug problem is a classic puzzle where you are given two jugs of capacities `Jug1Capacity` and `Jug2Capacity`, and your goal is to measure a specific amount of water using these jugs. You can perform the following actions:

1. Fill a jug from the tap.
2. Empty a jug on the ground.
3. Pour water from one jug into another until either the jug from which water is poured is empty or the jug into which water is poured is full.

Here's a Prolog program to solve the Water Jug problem:

```prolog
% Water Jug Problem

% Initial state representation: [Jug1, Jug2]
% The initial state represents both jugs being empty.
```

```
initial_state([0, 0]).

% Final state representation: [TargetAmount, _]
% The goal is to have a specific amount of water in Jug1.
final_state([TargetAmount, _]).

% Actions
fill_jug1([_, Jug2], [Jug1Capacity, Jug2]) :-
    Jug1Capacity > 0,
    Jug1 is Jug1Capacity,
    write('Fill Jug1 '), write(Jug1), nl.

fill_jug2([Jug1, _], [Jug1, Jug2Capacity]) :-
    Jug2Capacity > 0,
    Jug2 is Jug2Capacity,
    write('Fill Jug2 '), write(Jug2), nl.

empty_jug1([_, Jug2], [0, Jug2]) :-
    write('Empty Jug1'), nl.

empty_jug2([Jug1, _], [Jug1, 0]) :-
    write('Empty Jug2'), nl.

pour_jug1_to_jug2([Jug1, Jug2], [NewJug1, NewJug2]) :-
    TotalCapacity is Jug1 + Jug2,
    NewJug1 is max(0, TotalCapacity - Jug2),
    NewJug2 is min(TotalCapacity, Jug2 + Jug1),
    write('Pour Jug1 to Jug2 '), write(NewJug1), write(' '), write(NewJug2), nl.

pour_jug2_to_jug1([Jug1, Jug2], [NewJug1, NewJug2]) :-
    TotalCapacity is Jug1 + Jug2,
    NewJug2 is max(0, TotalCapacity - Jug1),
    NewJug1 is min(TotalCapacity, Jug1 + Jug2),
    write('Pour Jug2 to Jug1 '), write(NewJug1), write(' '), write(NewJug2), nl.
```

```
% Solve the problem using depth-first search
solve(State, Actions) :-
    depth_first_search(State, [State], Actions).

% Depth-first search
depth_first_search(State, _, []) :-
    final_state(State).

depth_first_search(State, Visited, [Action | Actions]) :-
    perform(Action, State, NewState),
    \+ member(NewState, Visited),
    depth_first_search(NewState, [NewState | Visited], Actions).

% Perform actions
perform(fill_jug1, State, NewState) :- fill_jug1(State, NewState).
perform(fill_jug2, State, NewState) :- fill_jug2(State, NewState).
perform(empty_jug1, State, NewState) :- empty_jug1(State, NewState).
perform(empty_jug2, State, NewState) :- empty_jug2(State, NewState).
perform(pour_jug1_to_jug2, State, NewState) :- pour_jug1_to_jug2(State, NewState).
perform(pour_jug2_to_jug1, State, NewState) :- pour_jug2_to_jug1(State, NewState).

% Example query:
% ?- initial_state(InitialState), solve(InitialState, Actions).
```

This program defines actions to fill, empty, and pour water between the jugs. The `solve/2` predicate initiates the depth-first search to find a sequence of actions to reach the goal state. You can modify the `TargetAmount`, `Jug1Capacity`, and `Jug2Capacity` as needed for different variations of the problem.

NLP

1)word analysis

Creating a complete NLP program for word analysis involves utilizing various libraries and tools. Python is commonly used for NLP tasks, and libraries such as NLTK (Natural Language Toolkit) and spaCy can be very helpful. Below is a simple example using NLTK for tokenization, stemming, and part-of-speech tagging:

```python
```

```
import nltk
from nltk.tokenize import word_tokenize
from nltk.stem import PorterStemmer
from nltk import pos_tag

nltk.download('punkt')

def word_analysis(text):
    # Tokenization
    tokens = word_tokenize(text)
    print("Tokens:", tokens)

    # Stemming
    stemmer = PorterStemmer()
    stemmed_words = [stemmer.stem(token) for token in tokens]
    print("Stemmed Words:", stemmed_words)

    # Part-of-Speech Tagging
    pos_tags = pos_tag(tokens)
    print("POS Tags:", pos_tags)

# Example text
text_to_analyze = "Natural Language Processing is fascinating!"

# Perform word analysis
word_analysis(text_to_analyze)
```

In this example:
- `word_tokenize` is used for tokenization to break the text into words.
- `PorterStemmer` is used for stemming to reduce words to their root form.
- `pos_tag` is used for part-of-speech tagging.

Make sure to install NLTK before running the code:

```bash
pip install nltk
```

This is a basic example, and depending on your specific requirements, you might need to use more advanced techniques, such as lemmatization, named entity recognition, or word embeddings. Additionally, spaCy is another popular library that provides a more efficient and convenient way to perform various NLP tasks.

2) Word Generation

Word generation in NLP typically involves using language models to generate new words that follow the patterns and structures observed in the training data. In this example, I'll demonstrate how to use a simple character-level language model to generate new words. For more sophisticated word generation tasks, you might want to explore more advanced models like recurrent neural networks (RNNs) or transformers.

Here's a basic Python program using a character-level Markov chain model for word generation:

```python
import random
from collections import defaultdict

class WordGenerator:
    def __init__(self, n=2):
        self.n = n  # Order of the Markov chain
        self.model = defaultdict(list)

    def train(self, corpus):
        for word in corpus:
            word = word.lower()
            for i in range(len(word) - self.n):
                prefix = word[i:i + self.n]
                next_char = word[i + self.n]
                self.model[prefix].append(next_char)

    def generate_word(self, length=5, seed=None):
        if seed is None:
            seed = random.choice(list(self.model.keys()))
        else:
            seed = seed.lower()

        current_prefix = seed[-self.n:]

        generated_word = seed

        for _ in range(length - self.n):
            next_char = random.choice(self.model.get(current_prefix, ['']))
            generated_word += next_char
            current_prefix = generated_word[-self.n:]

        return generated_word

# Example usage
corpus = ["apple", "banana", "cherry", "grape", "orange", "pear"]

word_generator = WordGenerator(n=2)
word_generator.train(corpus)

generated_word = word_generator.generate_word(length=8)
print("Generated Word:", generated_word)
```

In this example:

- The `WordGenerator` class initializes with a specified order `n` for the Markov chain.
- The `train` method takes a corpus of words and builds the Markov chain model based on character sequences of length `n`.
- The `generate_word` method generates a new word by starting with a seed and randomly selecting subsequent characters based on the Markov chain model.

Keep in mind that this is a simplistic example, and more advanced models like neural networks might be better suited for more complex and context-aware word generation tasks.

3)Morphology

Morphology in NLP involves studying the internal structure of words and their formation from morphemes. Here's a simple Python program using the Natural Language Toolkit (NLTK) to perform basic morphological analysis, including tokenization, stemming, and lemmatization:

```python
import nltk
from nltk.tokenize import word_tokenize
from nltk.stem import PorterStemmer, WordNetLemmatizer

nltk.download('punkt')
nltk.download('wordnet')

def morphology_analysis(text):
    # Tokenization
    tokens = word_tokenize(text)
    print("Tokens:", tokens)

    # Stemming
    stemmer = PorterStemmer()
    stemmed_words = [stemmer.stem(token) for token in tokens]
    print("Stemmed Words:", stemmed_words)

    # Lemmatization
    lemmatizer = WordNetLemmatizer()
    lemmatized_words = [lemmatizer.lemmatize(token) for token in tokens]
    print("Lemmatized Words:", lemmatized_words)

# Example text
text_to_analyze = "Natural Language Processing involves the study of linguistic structures."

# Perform morphology analysis
morphology_analysis(text_to_analyze)
```

In this example:

- `word_tokenize` is used for tokenization to break the text into words.
- `PorterStemmer` is used for stemming to reduce words to their root form.
- `WordNetLemmatizer` is used for lemmatization to reduce words to their base or dictionary form.

Make sure to install NLTK before running the code:

```bash
pip install nltk
```

This program demonstrates the basics of morphology analysis. Depending on your specific needs, you might want to explore more advanced techniques, handle multiple languages, or consider using other NLP libraries like spaCy for a more comprehensive analysis.

4) N-Grams

N-grams are contiguous sequences of n items (characters, words, or tokens) from a given sample of text or speech. Here's a simple Python program using NLTK to generate n-grams from a given text:

```python
import nltk
from nltk import ngrams
from nltk.tokenize import word_tokenize

nltk.download('punkt')

def generate_ngrams(text, n):
    # Tokenize the text into words
    words = word_tokenize(text)

    # Generate n-grams
    n_grams = list(ngrams(words, n))

    return n_grams

# Example text
text_to_analyze = "Natural Language Processing is fascinating and powerful."

# Specify the value of 'n' for n-grams
n_value = 3
```

```
# Generate n-grams
result_ngrams = generate_ngrams(text_to_analyze, n_value)

# Print the result
print(f"{n_value}-grams:", result_ngrams)
```

In this example:

- `word_tokenize` is used to tokenize the input text into words.
- `ngrams` function from NLTK is used to generate n-grams of a specified order.

Make sure to install NLTK before running the code:

```bash
pip install nltk
```

You can change the value of `n_value` to generate n-grams of different orders. The resulting output will be a list of tuples, each containing a sequence of n words. Note that in practice, it's common to use n-grams for various NLP tasks, such as language modeling, feature extraction, and text analysis.

5) N-Grams Smoothing

Smoothing is a technique used in NLP to handle the issue of unseen n-grams, where some n-grams might have zero probabilities in the training data. One common method is add-one (Laplace) smoothing. Below is a simple Python program that generates n-grams with add-one smoothing using NLTK:

```python
import nltk
from nltk import ngrams
from nltk.tokenize import word_tokenize
from nltk.probability import FreqDist

nltk.download('punkt')

def add_one_smoothing_ngrams(text, n):
    # Tokenize the text into words
    words = word_tokenize(text)

    # Generate n-grams
    n_grams = list(ngrams(words, n))

    # Calculate frequencies of n-grams
    freq_dist = FreqDist(n_grams)

    # Vocabulary size (unique n-grams)
    vocab_size = len(set(n_grams))

    # Add-one smoothing
    smoothed_n_grams = [(gram, (freq_dist[gram] + 1) / (len(n_grams) + vocab_size)) for gram in n_grams]

    return smoothed_n_grams

# Example text
text_to_analyze = "Natural Language Processing is fascinating and powerful."

# Specify the value of 'n' for n-grams
n_value = 2

# Generate n-grams with add-one smoothing
result_ngrams = add_one_smoothing_ngrams(text_to_analyze, n_value)

# Print the result
print(f"{n_value}-grams with Add-One Smoothing:", result_ngrams)
```

In this example:

- `word_tokenize` is used to tokenize the input text into words.
- `ngrams` function from NLTK is used to generate n-grams of a specified order.
- `FreqDist` is used to calculate the frequencies of n-grams.
- Add-one smoothing is applied to adjust the probabilities of each n-gram.

Make sure to install NLTK before running the code:

```bash
pip install nltk
```

You can modify the `n_value` variable to generate n-grams of different orders. The resulting output will be a list of tuples, each containing an n-gram and its smoothed probability.

6) WORD TOKENIZER

In Natural Language Processing (NLP), word tokenization is the process of breaking a text into individual words or tokens. Here's a simple Python program using the Natural Language Toolkit (NLTK) to perform word tokenization:

```python
import nltk
from nltk.tokenize import word_tokenize

nltk.download('punkt')

def word_tokenizer(text):
    # Tokenize the text into words
    tokens = word_tokenize(text)

    return tokens

# Example text
text_to_tokenize = "Word tokenization is an essential step in natural language processing."

# Tokenize the text
tokenized_words = word_tokenizer(text_to_tokenize)

# Print the result
print("Tokenized Words:", tokenized_words)
```

In this example:

- `word_tokenize` from NLTK is used to tokenize the input text into words.

Make sure to install NLTK before running the code:

```bash
pip install nltk
```

The `word_tokenizer` function takes a text input and returns a list of tokenized words. You can use this basic word tokenizer as a starting point for more advanced tokenization tasks. Depending on your specific requirements, you might explore other tokenization techniques or use more sophisticated tokenizers provided by libraries like spaCy.

7) SENTENCE TOKENIZER

Sentence tokenization is the process of breaking a text into sentences. Here's a simple Python program using the Natural Language Toolkit (NLTK) to perform sentence tokenization:

```python
import nltk
from nltk.tokenize import sent_tokenize


nltk.download('punkt')


def sentence_tokenizer(text):
    # Tokenize the text into sentences
    sentences = sent_tokenize(text)


    return sentences


# Example text
text_to_tokenize = "Sentence tokenization is important. It helps break text into meaningful sentences. NLTK provides tools for this task."


# Tokenize the text into sentences
tokenized_sentences = sentence_tokenizer(text_to_tokenize)
```

```python
# Print the result
print("Tokenized Sentences:")
for i, sentence in enumerate(tokenized_sentences, start=1):
    print(f"{i}. {sentence}")
```

In this example:

- `sent_tokenize` from NLTK is used to tokenize the input text into sentences.

Make sure to install NLTK before running the code:

```bash
pip install nltk
```

The `sentence_tokenizer` function takes a text input and returns a list of tokenized sentences. The example text has three sentences, and the program prints each tokenized sentence along with its index.

Keep in mind that sentence tokenization can be more complex in certain cases, especially with abbreviations, numbers, or other linguistic challenges. Libraries like spaCy also provide robust sentence tokenization capabilities. Depending on your specific needs, you might choose the tokenizer that best fits your use case.

8) PARAGRAPH TOKENIZER

Paragraph tokenization involves breaking a longer text into individual paragraphs. In NLTK, paragraph tokenization is not explicitly defined as it is for word or sentence tokenization. However, you can create a simple program to achieve paragraph tokenization based on the presence of newline characters (`\n`) or other paragraph delimiters. Here's an example:

```python
def paragraph_tokenizer(text):
    # Split the text into paragraphs based on newline characters
    paragraphs = text.split('\n\n')  # Adjust as needed for your specific text format

    return paragraphs

# Example text
text_to_tokenize = """
```

```
Paragraph 1: NLTK is a powerful library for natural language processing.
It provides tools for tasks such as tokenization, stemming, and part-of-speech tagging.

Paragraph 2: The library is widely used in the field of artificial intelligence.
It is a valuable resource for researchers and developers working on NLP projects.
"""

# Tokenize the text into paragraphs
tokenized_paragraphs = paragraph_tokenizer(text_to_tokenize)

# Print the result
print("Tokenized Paragraphs:")
for i, paragraph in enumerate(tokenized_paragraphs, start=1):
    print(f"{i}. {paragraph.strip()}")
```

In this example:

- The `paragraph_tokenizer` function uses the `split` method to break the text into paragraphs based on double newline characters (`\n\n`). You may need to adjust this based on the specific format of your text.

- The example text includes two paragraphs separated by double newline characters.

- The program then prints each tokenized paragraph along with its index.

Keep in mind that the effectiveness of paragraph tokenization can depend on the structure of your input text. If your paragraphs are delimited differently, you may need to modify the `paragraph_tokenizer` function accordingly. Additionally, more advanced tools like spaCy also provide efficient sentence and paragraph tokenization features.

9) CORPORA

In Natural Language Processing (NLP), a corpus (plural: corpora) refers to a collection of texts used for linguistic analysis. Corpora play a crucial role in training and evaluating NLP models. Below is a simple Python program that demonstrates how to load and use a corpus using the Natural Language Toolkit (NLTK):

```python
import nltk
from nltk.corpus import reuters
```

```
nltk.download('reuters')

def load_reuters_corpus():
    # Load the Reuters corpus from NLTK
    corpus = reuters

    # Display some basic information about the corpus
    print("Number of Categories:", len(corpus.categories()))
    print("Categories:", corpus.categories()[:10])
    print("File IDs in 'crude' category:", corpus.fileids('crude')[:5])

    # Access and print the text of a specific document
    document_id = 'test/14826'
    document_text = corpus.raw(document_id)
    print("\nText of Document (ID:", document_id, "):\n", document_text[:500])

# Load and explore the Reuters corpus
load_reuters_corpus()
```

In this example:

- We use the `reuters` corpus provided by NLTK. The Reuters corpus is a collection of news documents.

- The `load_reuters_corpus` function loads the corpus and prints some basic information, such as the number of categories, a list of categories, and file IDs in the 'crude' category.

- The program then accesses and prints the text of a specific document with the file ID 'test/14826'.

Make sure to install NLTK before running the code:

```bash
pip install nltk
```

This is a basic example using NLTK's built-in corpora. Depending on your specific needs, you may want to explore other corpora, such as the Brown corpus, Gutenberg corpus, or even custom corpora for your specific domain. Corpora are essential for tasks like training language models, extracting features, and evaluating algorithms in NLP.

10) PROBABILISTIC PARSING

Probabilistic parsing involves using statistical models to assign probabilities to different parses of a given sentence. The Natural Language Toolkit (NLTK) provides a probabilistic parser called the `EarleyChartParser` that is based on Earley parsing algorithm. Below is a simple example program using probabilistic parsing in NLTK:

```python
import nltk
from nltk import CFG
from nltk.parse import EarleyChartParser

# Define a probabilistic context-free grammar (PCFG)
pcfg_grammar = CFG.fromstring("""
    S -> NP VP [1.0]
    NP -> Det N [0.5] | NP PP [0.4] | 'John' [0.1]
    Det -> 'the' [0.6] | 'a' [0.4]
    N -> 'man' [0.5] | 'dog' [0.3] | 'cat' [0.2]
    VP -> V NP [0.7] | VP PP [0.3]
    V -> 'chased' [0.4] | 'saw' [0.6]
    PP -> P NP [1.0]
    P -> 'with' [0.7] | 'in' [0.3]
""")

def probabilistic_parsing(sentence):
    # Tokenize the sentence
    tokens = nltk.word_tokenize(sentence)

    # Create a probabilistic Earley chart parser
    parser = EarleyChartParser(pcfg_grammar)

    # Parse the sentence and print the parse tree with probabilities
    for tree in parser.parse(tokens):
```

```python
    print("Parse Tree with Probability:", tree)

# Example sentence
example_sentence = "the man saw a cat with a dog"

# Perform probabilistic parsing
probabilistic_parsing(example_sentence)
```

In this example:

- We define a Probabilistic Context-Free Grammar (PCFG) using the `CFG` class in NLTK. Each production rule is associated with a probability.

- The `probabilistic_parsing` function tokenizes the input sentence and uses the `EarleyChartParser` to perform probabilistic parsing.

- The program then prints parse trees with associated probabilities.

Make sure to install NLTK before running the code:

```bash
pip install nltk
```

You can adjust the PCFG rules and probabilities in the grammar to suit your specific parsing needs. Probabilistic parsing is particularly useful in cases where multiple valid parses exist for a given sentence, and probabilities help in selecting the most likely interpretation.

11) PROBABILISTIC CONTEXT FREE GRAMMER

In Natural Language Processing (NLP), Probabilistic Context-Free Grammar (PCFG) is an extension of Context-Free Grammar (CFG) where each production rule is associated with a probability. Here's an example program using the Natural Language Toolkit (NLTK) to create and use a simple PCFG:

```python
import nltk
from nltk import PCFG
from nltk.parse import EarleyChartParser

# Define a probabilistic context-free grammar (PCFG)
pcfg_grammar = PCFG.fromstring("""
    S -> NP VP [1.0]
    NP -> Det N [0.5] | NP PP [0.4] | 'John' [0.1]
    Det -> 'the' [0.6] | 'a' [0.4]
    N -> 'man' [0.5] | 'dog' [0.3] | 'cat' [0.2]
    VP -> V NP [0.7] | VP PP [0.3]
    V -> 'chased' [0.4] | 'saw' [0.6]
    PP -> P NP [1.0]
    P -> 'with' [0.7] | 'in' [0.3]
""")

def probabilistic_parsing(sentence):
    # Tokenize the sentence
    tokens = nltk.word_tokenize(sentence)

    # Create a probabilistic Earley chart parser
    parser = EarleyChartParser(pcfg_grammar)

    # Parse the sentence and print the parse tree with probabilities
    for tree in parser.parse(tokens):
        print("Parse Tree with Probability:", tree)

# Example sentence
example_sentence = "the man saw a cat with a dog"

# Perform probabilistic parsing
probabilistic_parsing(example_sentence)
```

In this example:

- We use the `PCFG` class in NLTK to define a probabilistic context-free grammar.

- Each production rule is associated with a probability (the numbers in square brackets).

- The `probabilistic_parsing` function tokenizes the input sentence and uses the `EarleyChartParser` to perform probabilistic parsing.

- The program then prints parse trees with associated probabilities.

Make sure to install NLTK before running the code:

```bash
pip install nltk
```

You can adjust the PCFG rules and probabilities in the grammar to suit your specific parsing needs. Probabilistic parsing is particularly useful in cases where multiple valid parses exist for a given sentence, and probabilities help in selecting the most likely interpretation.

12) LEARNING GRAMMAR

Learning grammar in the context of Natural Language Processing (NLP) often involves training models to understand and generate grammatically correct sentences. One common approach is to use supervised machine learning with labeled data. Here's a simple example using NLTK to train a part-of-speech (POS) tagger, which is a fundamental aspect of grammatical analysis:

```python
import nltk
from nltk.corpus import treebank
from nltk.tag import DefaultTagger, UnigramTagger, BigramTagger, TrigramTagger
from nltk.chunk import RegexpParser

nltk.download('treebank')

def train_pos_tagger():
    # Load the Penn Treebank corpus for training
    tagged_sentences = treebank.tagged_sents()

    # Split the data into training and testing sets
    train_data = tagged_sentences[:3000]
    test_data = tagged_sentences[3000:]

    # Train a POS tagger using the backoff approach
    default_tagger = DefaultTagger('NN')
    unigram_tagger = UnigramTagger(train_data, backoff=default_tagger)
    bigram_tagger = BigramTagger(train_data, backoff=unigram_tagger)
    trigram_tagger = TrigramTagger(train_data, backoff=bigram_tagger)

    # Evaluate the tagger on the test data
    accuracy = trigram_tagger.evaluate(test_data)
    print("Trigram Tagger Accuracy:", accuracy)

    return trigram_tagger

def parse_sentence(tagged_sentence):
    # Define a simple chunking grammar using regular expressions
    chunking_grammar = r"""
        NP: {<DT>?<JJ>*<NN>}   # Chunk NP (noun phrases)
        VP: {<VB.*><NP|PP|CLAUSE>+$}    # Chunk VP (verb phrases)
        CLAUSE: {<NP><VP>}   # Chunk clauses
    """

    # Create a chunk parser with the defined grammar
    chunk_parser = RegexpParser(chunking_grammar)

    # Apply chunking to the tagged sentence
    tree = chunk_parser.parse(tagged_sentence)

    # Display the parse tree
    print("\nParse Tree:")
    tree.pretty_print()

# Train a POS tagger
pos_tagger = train_pos_tagger()
```

```
# Example sentence
example_sentence = "The quick brown fox jumps over the lazy dog."

# Tokenize and tag the example sentence
tokenized_sentence = nltk.word_tokenize(example_sentence)
tagged_sentence = pos_tagger.tag(tokenized_sentence)

# Display the tagged sentence
print("\nTagged Sentence:", tagged_sentence)

# Parse the tagged sentence
parse_sentence(tagged_sentence)
```

In this example:

- We use the Penn Treebank corpus from NLTK for training and testing data.

- We train a part-of-speech (POS) tagger using the backoff approach with a DefaultTagger, UnigramTagger, BigramTagger, and TrigramTagger.

- The trained POS tagger is then used to tag an example sentence.

- We define a simple chunking grammar using regular expressions to extract noun phrases (NP), verb phrases (VP), and clauses from the tagged sentence.

- The example sentence is parsed, and the resulting parse tree is displayed.

Make sure to install NLTK before running the code:

```bash
pip install nltk
```

This example demonstrates a basic approach to learning grammar-related tasks, specifically part-of-speech tagging and chunking. For more advanced tasks, you might explore other machine learning models, such as sequence-to-sequence models or transformers, depending on the specific NLP task you're interested in.

13) CONDITIONAL FREQUENCY DISTRIBUTIONS

Conditional Frequency Distributions (CFD) in Natural Language Processing (NLP) allow you to examine the frequency of words in a corpus under specific conditions. The Natural Language Toolkit (NLTK) provides a `ConditionalFreqDist` class that makes it easy to work with conditional frequency distributions. Here's an example program using NLTK to create and work with a conditional frequency distribution:

```python
import nltk
from nltk.probability import ConditionalFreqDist
from nltk.corpus import inaugural

nltk.download('inaugural')

def create_conditional_freq_dist(corpus, condition_function):
    # Get the words from the specified corpus
    words = inaugural.words(corpus)

    # Apply the condition function to each word
    conditions = [condition_function(word) for word in words]

    # Create a conditional frequency distribution
    cfd = ConditionalFreqDist(zip(conditions, words))

    return cfd

# Example condition function: word length
def word_length_condition(word):
    return len(word)

# Example condition function: starting letter
def starting_letter_condition(word):
    return word[0].lower()
```

```python
# Example corpus: inaugural addresses
corpus_name = '1789-Washington.txt'

# Create conditional frequency distributions based on word length and starting letter
cfd_word_length = create_conditional_freq_dist(corpus_name, word_length_condition)
cfd_starting_letter = create_conditional_freq_dist(corpus_name, starting_letter_condition)

# Display conditional frequency distributions
print("Conditional Frequency Distribution (Word Length):")
cfd_word_length.tabulate()

print("\nConditional Frequency Distribution (Starting Letter):")
cfd_starting_letter.tabulate()
```

In this example:

- We use the `inaugural` corpus from NLTK, which contains U.S. presidential inaugural addresses.

- We define two condition functions (`word_length_condition` and `starting_letter_condition`) that categorize words based on their length and starting letter, respectively.

- The `create_conditional_freq_dist` function creates a conditional frequency distribution using the specified condition function.

- The resulting conditional frequency distributions are then displayed using the `tabulate` method.

Make sure to install NLTK before running the code:

```bash
pip install nltk
```

You can modify the example by choosing a different corpus or creating custom condition functions based on your specific analysis needs. Conditional frequency distributions are useful for exploring relationships between different linguistic features in a corpus.

14) LEXICAL ANALYSER

A lexical analyzer, also known as a lexer or tokenizer, is a crucial component in the process of parsing and analyzing natural language text. It breaks down the input text into smaller units called tokens. Below is a simple example program in Python using the Natural Language Toolkit (NLTK) to create a basic lexical analyzer for tokenizing words and extracting part-of-speech tags:

```python
import nltk
from nltk import pos_tag, word_tokenize

nltk.download('punkt')
nltk.download('averaged_perceptron_tagger')

def lexical_analyzer(text):
    # Tokenize the text into words
    tokens = word_tokenize(text)

    # Perform part-of-speech tagging
    pos_tags = pos_tag(tokens)

    # Display the tokens and their part-of-speech tags
    print("Token\t\tPOS Tag")
    print("----------------------")
    for token, pos_tag in pos_tags:
        print(f"{token}\t\t{pos_tag}")

# Example text
example_text = "Natural Language Processing is an exciting field of study."

# Apply the lexical analyzer to the example text
lexical_analyzer(example_text)
```

In this example:

- The `word_tokenize` function from NLTK is used to tokenize the input text into words.

- The `pos_tag` function is used to perform part-of-speech tagging on the tokens, assigning a part-of-speech tag to each word.
- The program then prints the tokens along with their part-of-speech tags.

Make sure to install NLTK before running the code:

```bash
pip install nltk
```

This is a basic example, and you can extend the lexical analyzer based on your specific requirements. Depending on the task, you might want to include additional information such as named entities, or you might choose more advanced tokenization libraries like spaCy for more sophisticated analyses.

15) WORDNET

WordNet is a lexical database of the English language that relates words to one another in terms of synonyms, hypernyms, hyponyms, and more. The Natural Language Toolkit (NLTK) in Python provides an interface to access WordNet. Below is a simple program that demonstrates how to use NLTK to explore WordNet:

```python
import nltk
from nltk.corpus import wordnet

nltk.download('wordnet')

def wordnet_example(word):
    # Get synsets for the given word
    synsets = wordnet.synsets(word)

    if not synsets:
        print(f"No synsets found for '{word}'.")
        return

    # Display information for each synset
    for synset in synsets:
        print(f"Synset: {synset.name()}")
        print(f"POS Tag: {synset.pos()}")
        print(f"Definition: {synset.definition()}")
        print(f"Examples: {synset.examples()}")
        print()

    # Get hypernyms (more abstract terms)
    hypernyms = synsets[0].hypernyms()
    if hypernyms:
        print("Hypernyms:")
        for hypernym in hypernyms:
            print(f"- {hypernym.name()}")
        print()

# Example usage
word_to_explore = "car"
wordnet_example(word_to_explore)
```

In this example:

- The `wordnet.synsets` function is used to get a list of synsets for a given word.
- For each synset, information such as the synset name, part-of-speech (POS) tag, definition, and examples are displayed.
- The program also demonstrates how to get hypernyms (more abstract terms) for the first synset.

Make sure to install NLTK before running the code:

```bash
pip install nltk
```

This is a basic example, and WordNet provides more functionality for exploring word relationships, such as hyponyms, holonyms, meronyms, and more. You can further extend this program to explore other aspects of WordNet based on your specific needs.

16) CONTEXT FREE GRAMMAR

A Context-Free Grammar (CFG) is a formal grammar that describes the syntax of a language by specifying a set of production rules. In this example, we'll create a simple CFG and use the NLTK library to generate sentences based on that grammar.

```python
import nltk

from nltk import CFG

from nltk.parse.generate import generate


# Define a Context-Free Grammar (CFG)

cfg_rules = """

    S -> NP VP

    NP -> Det N

    VP -> V NP | V

    Det -> 'the' | 'a'

    N -> 'cat' | 'dog'

    V -> 'chased' | 'ran'

"""


cfg = CFG.fromstring(cfg_rules)


# Generate sentences based on the CFG

generated_sentences = list(generate(cfg, depth=3))


# Display the generated sentences

print("Generated Sentences:")

for sentence in generated_sentences:

    print(' '.join(sentence))
```

In this example:

- We define a simple CFG using the `CFG.fromstring` method. The production rules describe the structure of sentences, specifying the relationships between sentence (S), noun phrase (NP), verb phrase (VP), determiner (Det), noun (N), and verb (V).

- We use the `generate` function from `nltk.parse.generate` to create sentences based on the CFG. The `depth` parameter controls the maximum depth of the generated tree.

- The program then prints the generated sentences.

Make sure to install NLTK before running the code:

```bash
pip install nltk
```

This is a basic example, and you can customize the CFG rules to represent more complex grammars. CFGs are commonly used in natural language processing for syntax analysis and generation.

17) **LARGE CONTEXT FREE GRAMMAR AND PARSING**

Creating a large context-free grammar (CFG) and parsing sentences with it can be a complex task, and the specifics of the grammar would depend on the language and the structures you want to capture. In this example, I'll provide a larger CFG representing a simple English grammar and demonstrate how to parse sentences using the Natural Language Toolkit (NLTK):

```python
import nltk
from nltk import CFG

nltk.download('punkt')

# Define a larger Context-Free Grammar (CFG) for English
large_cfg_rules = """
    S -> NP VP
    NP -> Det N | Det N PP | 'I'
    VP -> V NP | VP PP
    PP -> P NP
    Det -> 'the' | 'a' | 'my'
    N -> 'cat' | 'dog' | 'house' | 'ball'
    V -> 'chased' | 'saw' | 'ate'
    P -> 'in' | 'on' | 'with'
"""
```

```
large_cfg = CFG.fromstring(large_cfg_rules)

# Example sentences
sentences_to_parse = [
    "I saw the dog with my cat.",
    "The ball in the house chased a dog.",
]

# Create a parser based on the CFG
parser = nltk.ChartParser(large_cfg)

# Parse and print the results for each sentence
for sentence in sentences_to_parse:
    tokens = nltk.word_tokenize(sentence)
    for tree in parser.parse(tokens):
        print("Sentence:", sentence)
        print("Parse Tree:")
        tree.pretty_print()
        print("\n")
```

In this example:

- The `large_cfg_rules` string defines a larger CFG for English with rules for sentences (S), noun phrases (NP), verb phrases (VP), prepositional phrases (PP), determiners (Det), nouns (N), verbs (V), and prepositions (P).

- The `sentences_to_parse` list contains example sentences to be parsed.

- The NLTK `ChartParser` is used to parse the sentences based on the defined CFG.

- The program prints the original sentence, and for each sentence, it prints one of the parse trees generated by the parser.

Make sure to install NLTK before running the code:

```bash
pip install nltk
```

Keep in mind that creating an extensive CFG for a natural language involves considering a wide range of linguistic phenomena, and this example is quite simplified. Depending on your specific use case, you might need a more comprehensive CFG. Additionally, more advanced parsing techniques, such as dependency parsing or probabilistic parsing, may be considered for more accurate analyses.

18) NAMED ENTITY RECOGNITION

Named Entity Recognition (NER) is a task in Natural Language Processing (NLP) that involves identifying and classifying named entities (such as persons, organizations, locations) in text. One common library for NER is spaCy. Below is an example program demonstrating Named Entity Recognition using spaCy in Python:

```python
import spacy

# Load the spaCy English NLP model
nlp = spacy.load("en_core_web_sm")

def named_entity_recognition(text):
    # Process the text using spaCy NLP pipeline
    doc = nlp(text)

    # Print named entities and their labels
    print("Named Entities:")
    for ent in doc.ents:
        print(f"{ent.text}: {ent.label_}")

# Example text
example_text = "Apple Inc. was founded by Steve Jobs in Cupertino, California. It is a leading technology company."

# Apply Named Entity Recognition
named_entity_recognition(example_text)
```

In this example:

- We use spaCy to load the English NLP model (`en_core_web_sm`).
- The `nlp` object is used to process the example text, resulting in a spaCy `Doc` object.
- The named entities and their corresponding labels are printed.

Make sure to install spaCy before running the code:

```bash
pip install spacy
```

You also need to download the English language model:

```bash
python -m spacy download en_core_web_sm
```

After running the program, you should see the identified named entities along with their labels.

Note: spaCy's NER model is not perfect and may not cover all domain-specific entities. For more accurate NER in specialized domains, you might need to consider training a custom NER model.

If you prefer NLTK for NER, it also provides an interface to the Stanford NER system. Keep in mind that using spaCy often results in more straightforward and efficient NER tasks.