Jaypee Institute of Information Technology, Sector - 62, Noida

B.Tech CSE III Semester

Theory of Computation PBL Report

Natural Language Processing using Automata

Submitted to

Mr. Akshit Raj Patel

Submitted by

Harsh Sharma	2401030232	B5
Karvy Singh	2401030234	B5
Rudra Kumar Singh	2401030237	B5

Letter of Transmittal

Mr. Akshit Raj Patel

Department of Computer Science & IT

Subject: Submission of Project "Natural Language Processing using Automata"

Respected Sir,

We are pleased to submit our project titled *"Natural Language Processing using Automata"* as part of our coursework for the Theory of Computation course. This report documents the design and implementation of a small natural language processing pipeline that uses concepts from deterministic finite automata (DFA) and context-free grammars (CFG) for tokenization and parsing of English sentences.

We have endeavored to connect theoretical ideas from automata theory and formal languages to a practical Python implementation. In particular, we designed a DFA-based tokenizer, interpreted dependency parses in terms of CFG-style rules, and implemented a simple heuristic to decide whether a given string is likely to be a natural sentence. The project aims to demonstrate how foundational models of computation can still be used as building blocks inside modern NLP systems.

Thank you for your guidance and the opportunity to work on this project.

Sincerely,

Harsh Sharma (2401030232)
Karvy Singh (2401030234)
Rudra Kumar Singh (2401030237)

Date: November 28, 2025

Contents

1 Introduction **3**

2 Objectives **3**

3 Theoretical Background **4**
3.1 Formal Languages and Automata 4
3.2 Deterministic Finite Automata (DFA) 4
3.3 Context-Free Grammars (CFG) 5

4 DFA-Based Tokenization **5**
4.1 Design of the Tokenizer DFA 5
4.2 DFA State Diagram 6
4.3 Integration into the Program 7

5 Parsing and CFG Interpretation **7**
5.1 Dependency Trees as Parse Trees 7
5.2 Relation to Context-Free Grammars 8
5.3 Natural Sentence Heuristic 8

6 System Design and Implementation **8**
6.1 Overall Architecture 8
6.2 Key Data Structures 9
6.3 Python Implementation 9
6.4 Source Code Repository 14
6.5 Example Sentences 14

7 Program Output **15**

8 Results and Discussion **17**
8.1 Tokenization Behavior 17
8.2 Natural Sentence Classification 17
8.3 Connection to Automata and Grammars 17

9 Conclusion and Future Work **18**

1 Introduction

Natural Language Processing (NLP) is concerned with enabling computers to understand and generate human language. Modern NLP systems often rely on large machine learning models, but the theoretical foundations still come from formal languages and automata theory. Concepts such as alphabets, tokens, grammars, and parse trees play a central role in tasks like tokenization and syntactic parsing.

In this project, we focus on two classical models of computation introduced in the Theory of Computation course:

- Deterministic Finite Automata (DFA), used here to design a simple tokenizer that splits an input string into words, numbers, and symbols.
- Context-Free Grammars (CFG), used at a conceptual level to understand sentence structure and to interpret a dependency parse tree.

We implemented a Python program that:

1. Tokenizes an input sentence using a hand-designed DFA.
2. Parses the sentence using a dependency tree built from spaCy's analysis, which we interpret in terms of CFG-style rules.
3. Applies a simple grammar-based heuristic to decide whether the sentence is likely to be a "natural" English sentence.

The aim is not to build a full-scale NLP system, but to show how DFA and CFG ideas can be embedded in an end-to-end pipeline. This report describes the underlying theory, the design of our automaton and grammar, and the behavior of the implemented system on a few example sentences.

2 Objectives

The main objectives of the project are:

- To apply theoretical concepts from automata theory (DFA) to the practical task of tokenization in NLP.
- To relate context-free grammars to syntactic parsing, using dependency trees as a convenient representation.
- To design and implement a simple Python program that integrates DFA-based tokenization with CFG-style parsing.

- To use basic grammatical constraints (similar to a CFG) to heuristically judge whether a string is likely to be a valid natural language sentence.
- To gain hands-on experience bridging the gap between the Theory of Computation and real-world language processing.

3 Theoretical Background

3.1 Formal Languages and Automata

A formal language is a set of strings over an alphabet Σ. In automata theory, different models of computation recognize different classes of languages:

- Regular languages, recognized by finite automata.
- Context-free languages, generated by context-free grammars and recognized by pushdown automata.

NLP tasks can be viewed in terms of these formalisms:

- Tokenization: mapping a raw character sequence to a sequence of tokens. This is often regular in nature and can be handled by DFA or regular expressions.
- Parsing: checking whether a token sequence obeys the syntactic rules of a language and building a parse tree. This is naturally modeled by CFGs.

3.2 Deterministic Finite Automata (DFA)

A deterministic finite automaton is a 5-tuple

$$M = (Q, \Sigma, \delta, q_0, F)$$

where

- Q is a finite set of states.
- Σ is a finite input alphabet.
- $\delta : Q \times \Sigma \to Q$ is the transition function.
- $q_0 \in Q$ is the start state.
- $F \subseteq Q$ is the set of accepting states.

In our tokenizer, the alphabet consists of characters (letters, digits, whitespace, punctuation). The DFA state encodes which kind of token we are currently reading (word, number, or symbol), or whether we are in the start state between tokens.

3.3 Context-Free Grammars (CFG)

A context-free grammar is a 4-tuple

$$G = (V, \Sigma, R, S)$$

where

- V is a finite set of variables (non-terminals).
- Σ is a finite set of terminals (tokens).
- R is a finite set of production rules of the form $A \to \alpha$, with $A \in V$ and $\alpha \in (V \cup \Sigma)^*$.
- $S \in V$ is the start symbol.

A simple CFG fragment for English sentences might look like:

$$\begin{aligned}
S &\to NP\ VP \\
NP &\to Det\ N \mid Pronoun \\
VP &\to V\ NP \mid V \\
Det &\to \text{“the”} \mid \text{“a”} \\
N &\to \text{“man”} \mid \text{“dog”} \mid \text{“parser”} \\
V &\to \text{“saw”} \mid \text{“build”} \mid \text{“is”} \\
Pronoun &\to \text{“I”}
\end{aligned}$$

A parser based on this grammar would build a parse tree whose root is S. While our implementation does not explicitly implement a CFG parser, it uses dependency trees and simple grammatical heuristics that are conceptually similar to enforcing such rules.

4 DFA-Based Tokenization

4.1 Design of the Tokenizer DFA

Our custom tokenizer class `DFATokenizer` classifies each character into one of the following categories:

- Alphabetic characters: part of a word token (`WORD`).
- Digits: part of a number token (`NUMBER`).
- Other characters (punctuation, operators, etc.): symbol tokens (`SYMBOL`).
- Whitespace: token separator, not included in any token.

The DFA used for tokenization has the following states:

- **START**: not currently reading a token.
- **WORD**: currently reading a sequence of letters (and possibly apostrophes) forming a word.
- **NUMBER**: currently reading a sequence of digits forming a number.

When the DFA moves from **WORD** or **NUMBER** back to **START**, the accumulated characters are emitted as a token of the appropriate type. Symbol characters are emitted immediately as `SYMBOL` tokens.

4.2 DFA State Diagram

We now present the DFA diagram describing the core behavior of the tokenizer. Here, "letter" denotes an alphabetic character, "digit" a numeric character, "ws" whitespace, and "sym" any other symbol.

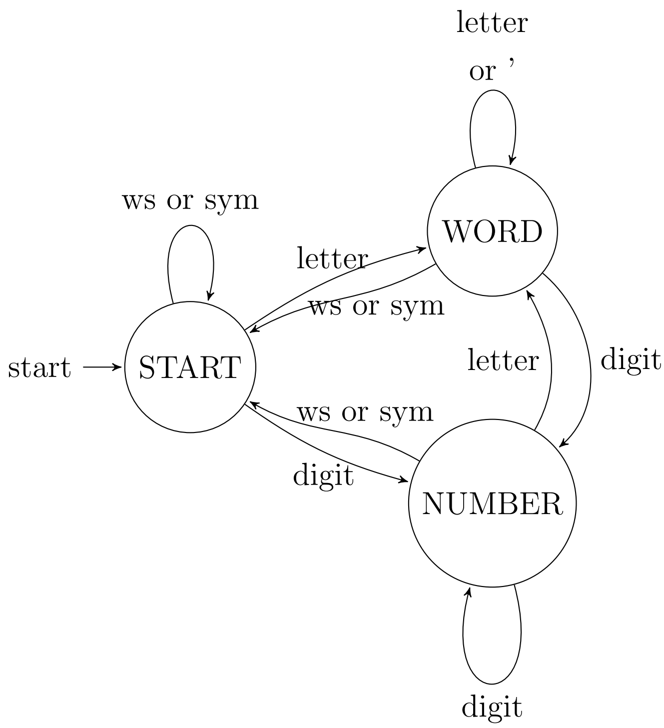

Figure 1: DFA for DFA-based tokenizer (`DFATokenizer`).

This DFA can be formally described by the 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where:

- $Q = \{\text{START}, \text{WORD}, \text{NUMBER}\}$.
- Σ is the set of all ASCII characters.

- $q_0 = \text{START}$.
- F can be considered $\{\text{WORD}, \text{NUMBER}, \text{START}\}$, since after finishing the input we may be in any of these states; tokens are emitted based on the state before termination.

4.3 Integration into the Program

The `DFATokenizer` is implemented as a Python class with:

- A member `state` storing the current DFA state.
- A buffer `current` storing the characters of the token being built.
- A list `tokens` storing the final tokens of type `WORD`, `NUMBER`, or `SYMBOL`.

After tokenization, word tokens are normalized to lowercase for convenience when comparing or printing. The resulting sequence of tokens represents a regular-language-level processing of the input string.

5 Parsing and CFG Interpretation

5.1 Dependency Trees as Parse Trees

For parsing, we use spaCy to obtain part-of-speech (POS) tags and dependency relations between tokens. For each input sentence, spaCy returns a set of tokens with:

- `text`: the original token string.
- `pos_`: the coarse POS tag (e.g., VERB, NOUN, AUX).
- `dep_`: the dependency relation (e.g., ROOT, nsubj, dobj).
- `head`: the head of the dependency arc.

We construct our own parse tree structure using the `DepNode` data class:

```
@dataclass
class DepNode:
    text: str
    lemma: str
    pos: str
    dep: str
    children: List["DepNode"] = field(default_factory=list)
```

The function `build_dep_tree(doc)` creates one `DepNode` per token and links them according to the dependency arcs. The token whose `head` is itself becomes the root of the tree (corresponding to the syntactic ROOT of the sentence).

5.2 Relation to Context-Free Grammars

A dependency tree can be related to a CFG-style phrase structure tree as follows:

- The ROOT verb corresponds roughly to the head of the VP in a rule like $S \rightarrow NP\,VP$.
- A token with dependency label `nsubj` (nominal subject) corresponds to the NP on the left side of the sentence.
- Objects or complements correspond to NPs or PPs attached to the verb.

Instead of explicitly writing a CFG and running a parser, we use the dependency tree as a compact representation of a derivation in some underlying CFG. spaCy's model has implicitly learned such a grammar from data. Our code then reuses the resulting tree to check simple CFG-like constraints.

5.3 Natural Sentence Heuristic

We implement the following heuristic in the function `is_likely_natural_sentence(doc)`:

1. There is exactly one ROOT token in the dependency parse.
2. The ROOT token has POS tag VERB or AUX (verb or auxiliary).
3. There exists at least one token with dependency label `nsubj`, `nsubjpass`, or `csubj` (some form of subject).

Intuitively, this corresponds to requiring the existence of a subject and a finite verb, as in the CFG rule

$$S \rightarrow NP\,VP$$

where VP contains a verbal head. If these constraints are violated, the string is unlikely to be a well-formed declarative sentence in English.

6 System Design and Implementation

6.1 Overall Architecture

The complete pipeline in `main()` performs the following steps for each input sentence:

1. Print the raw sentence.
2. Run DFA-based tokenization using `DFATokenizer`.
3. Run spaCy's pipeline to obtain tokens with POS and dependency labels.
4. Build a custom dependency tree using `build_dep_tree`.
5. Print a pretty-printed version of the dependency tree.
6. Apply `is_likely_natural_sentence` to decide if the sentence is likely natural.

6.2 Key Data Structures

- `Tok`: a data class representing a token with fields `type` and `value`. Types are `"WORD"`, `"NUMBER"`, and `"SYMBOL"`.
- `DFATokenizer`: encapsulates the DFA states, transition logic, and token emission.
- `DepNode`: represents a node of the parse tree with text, lemma, POS tag, dependency label, and children.

6.3 Python Implementation

Listing 1: Python implementation of DFA-based tokenizer and dependency-tree parser

```
from dataclasses import dataclass, field
from typing import List, Optional
import spacy

@dataclass
class Tok:
    type: str   # "WORD", "NUMBER", "SYMBOL"
    value: str

class DFATokenizer:
    def __init__(self):
        self.state = "START"
        self.current = ""
        self.tokens: List[Tok] = []

    def reset(self):
        self.state = "START"
                self.current = ""
```

```python
        self.tokens.clear()

    def emit(self, type_):
        if self.current:
            self.tokens.append(Tok(type_, self.current))
            self.current = ""

    def tokenize(self, text: str) -> List[Tok]:
        self.reset()
        for ch in text:
            if self.state == "START":
                if ch.isspace():
                    continue
                elif ch.isalpha():
                    self.state = "WORD"
                    self.current += ch
                elif ch.isdigit():
                    self.state = "NUMBER"
                    self.current += ch
                else:
                    self.tokens.append(Tok("SYMBOL", ch))

            elif self.state == "WORD":
                if ch.isalpha() or ch == "'":
                    self.current += ch
                else:
                    self.emit("WORD")
                    self.state = "START"
                    # reprocess ch
                    if ch.isspace():
                        continue
                    elif ch.isdigit():
                        self.state = "NUMBER"
                        self.current += ch
                    else:
                        self.tokens.append(Tok("SYMBOL", ch))

            elif self.state == "NUMBER":
                if ch.isdigit():
                    self.current += ch
                else:
```

```
                    self.emit("NUMBER")
                    self.state = "START"
                    # reprocess ch
                    if ch.isspace():
                        continue
                    elif ch.isalpha():
                        self.state = "WORD"
                        self.current += ch
                    else:
                        self.tokens.append(Tok("SYMBOL", ch))

        if self.state == "WORD":
            self.emit("WORD")
        elif self.state == "NUMBER":
            self.emit("NUMBER")

        # normalize words to lowercase for convenience
        norm = []
        for t in self.tokens:
            if t.type == "WORD":
                norm.append(Tok("WORD", t.value.lower()))
            else:
                norm.append(t)
        return norm

@dataclass
class DepNode:
    text: str
    lemma: str
    pos: str
    dep: str
    children: List["DepNode"] = field(default_factory=list)

    def pretty(self, level=0) -> str:
        indent = "  " * level
        out = f"{indent}{self.text} ({self.pos}, {self.dep})\n"
        for child in self.children:
            out += child.pretty(level + 1)
        return out


```

```
def build_dep_tree(doc) -> Optional[DepNode]:
    if len(doc) == 0:
        return None

    # Create nodes for each token
    nodes = [DepNode(t.text, t.lemma_, t.pos_, t.dep_) for t in doc]

    root = None
    for i, tok in enumerate(doc):
        if tok.head.i == tok.i:
            # This is the ROOT token
            root = nodes[i]
        else:
            head_node = nodes[tok.head.i]
            head_node.children.append(nodes[i])

    return root

def is_likely_natural_sentence(doc) -> bool:
    #exactly one ROOT, ROOT is a verb or auxiliar, at least one nominal subject
 #uusing spacy learned grammar as a reference, not defining our cfg

    roots = [t for t in doc if t.dep_ == "ROOT"]
    if len(roots) != 1:
        return False

    root = roots[0]
    if root.pos_ not in ("VERB", "AUX"):
        return False

    has_subject = any(t.dep_ in ("nsubj", "nsubjpass", "csubj") for t in doc)
    if not has_subject:
        return False

    return True
```

```python
def main():
    # load spacy model (predefined "grammar")
    # make sure you installed it first:
    #   python -m spacy download en_core_web_sm
    nlp = spacy.load("en_core_web_sm")

    sentences = [
        "The man saw a dog.",
        "cat table green quickly.",
        "I will build a parser using automata.",
        "x1 + x2 = 10"
    ]

    tokenizer = DFATokenizer()

    for s in sentences:
        print("=" * 60)
        print("Sentence:", s)

        #  low-level DFA tokenization
        my_tokens = tokenizer.tokenize(s)
        print("\nDFA tokens:")
        for t in my_tokens:
            print("  ", t)

        # spacy analysis
        doc = nlp(s)

        print("\nspaCy tokens / POS / dep (reference grammar):")
        for t in doc:
            print(f"  {t.i:2d}: {t.text:10s} POS={t.pos_:6s} DEP={t.dep_:10s} HEAD={t.head.i}")

        # Use spacy arcs to build our parse tree
        tree = build_dep_tree(doc)
        print("\nOur dependency tree (built manually):")
        if tree:
            print(tree.pretty().rstrip())
        else:
            print("  <empty>")

```

```python
        print("\nLikely natural language sentence?",
              "YES" if is_likely_natural_sentence(doc) else "NO")


if __name__ == "__main__":
    main()
```

6.4 Source Code Repository

The complete source code for this project is available on GitHub at: `https://github.com/Karvy-Singh/TOCSuper.git`

6.5 Example Sentences

We tested the system on four sentences:

1. `‘‘The man saw a dog.’’`
2. `‘‘cat table green quickly.’’`
3. `‘‘I will build a parser using automata.’’`
4. `‘‘x1 + x2 = 10’’`

For each sentence, the program prints:

- The DFA tokens (type and value).
- spaCy tokens with POS and dependency labels.
- The custom dependency tree.
- The result of the natural sentence heuristic.

7 Program Output

```
==========================================================
Sentence: The man saw a dog.

DFA tokens:
  Tok(type='WORD', value='the')
  Tok(type='WORD', value='man')
  Tok(type='WORD', value='saw')
  Tok(type='WORD', value='a')
  Tok(type='WORD', value='dog')
  Tok(type='SYMBOL', value='.')

spaCy tokens / POS / dep (reference grammar):
   0: The        POS=DET    DEP=det        HEAD=1
   1: man        POS=NOUN   DEP=nsubj      HEAD=2
   2: saw        POS=VERB   DEP=ROOT       HEAD=2
   3: a          POS=DET    DEP=det        HEAD=4
   4: dog        POS=NOUN   DEP=dobj       HEAD=2
   5: .          POS=PUNCT  DEP=punct      HEAD=2

Our dependency tree (built manually):
saw (VERB, ROOT)
  man (NOUN, nsubj)
    The (DET, det)
  dog (NOUN, dobj)
    a (DET, det)
  . (PUNCT, punct)

Likely natural language sentence? YES
==========================================================
```

Figure 2: Program output for the sentence “The man saw a dog.”

```
==========================================================
Sentence: cat table green quickly.

DFA tokens:
  Tok(type='WORD', value='cat')
  Tok(type='WORD', value='table')
  Tok(type='WORD', value='green')
  Tok(type='WORD', value='quickly')
  Tok(type='SYMBOL', value='.')

spaCy tokens / POS / dep (reference grammar):
   0: cat        POS=NOUN   DEP=compound   HEAD=1
   1: table      POS=NOUN   DEP=nsubj      HEAD=2
   2: green      POS=ADV    DEP=ROOT       HEAD=2
   3: quickly    POS=ADV    DEP=advmod     HEAD=2
   4: .          POS=PUNCT  DEP=punct      HEAD=2

Our dependency tree (built manually):
green (ADV, ROOT)
  table (NOUN, nsubj)
    cat (NOUN, compound)
  quickly (ADV, advmod)
  . (PUNCT, punct)

Likely natural language sentence? NO
==========================================================
```

Figure 3: Program output for the sentence “cat table green quickly.”

```
============================================================
Sentence: I will build a parser using automata.

DFA tokens:
  Tok(type='WORD', value='i')
  Tok(type='WORD', value='will')
  Tok(type='WORD', value='build')
  Tok(type='WORD', value='a')
  Tok(type='WORD', value='parser')
  Tok(type='WORD', value='using')
  Tok(type='WORD', value='automata')
  Tok(type='SYMBOL', value='.')

spaCy tokens / POS / dep (reference grammar):
   0: I          POS=PRON   DEP=nsubj      HEAD=2
   1: will       POS=AUX    DEP=aux        HEAD=2
   2: build      POS=VERB   DEP=ROOT       HEAD=2
   3: a          POS=DET    DEP=det        HEAD=4
   4: parser     POS=NOUN   DEP=dobj       HEAD=2
   5: using      POS=VERB   DEP=acl        HEAD=4
   6: automata   POS=NOUN   DEP=dobj       HEAD=5
   7: .          POS=PUNCT  DEP=punct      HEAD=2

Our dependency tree (built manually):
build (VERB, ROOT)
  I (PRON, nsubj)
  will (AUX, aux)
  parser (NOUN, dobj)
    a (DET, det)
    using (VERB, acl)
      automata (NOUN, dobj)
  . (PUNCT, punct)

Likely natural language sentence? YES
============================================================
```

Figure 4: Program output for the sentence “I will build a parser using automata.”

```
============================================================
Sentence: x1 + x2 = 10

DFA tokens:
  Tok(type='WORD', value='x')
  Tok(type='NUMBER', value='1')
  Tok(type='SYMBOL', value='+')
  Tok(type='WORD', value='x')
  Tok(type='NUMBER', value='2')
  Tok(type='SYMBOL', value='=')
  Tok(type='NUMBER', value='10')

spaCy tokens / POS / dep (reference grammar):
   0: x1         POS=PROPN  DEP=ROOT       HEAD=0
   1: +          POS=CCONJ  DEP=cc         HEAD=0
   2: x2         POS=PROPN  DEP=conj       HEAD=0
   3: =          POS=NOUN   DEP=conj       HEAD=0
   4: 10         POS=NUM    DEP=nummod     HEAD=0

Our dependency tree (built manually):
x1 (PROPN, ROOT)
  + (CCONJ, cc)
  x2 (PROPN, conj)
  = (NOUN, conj)
  10 (NUM, nummod)

Likely natural language sentence? NO
```

Figure 5: Program output for the expression “x1 + x2 = 10”.

8 Results and Discussion

8.1 Tokenization Behavior

For the sentence `‘‘The man saw a dog.’’`, the DFA produces tokens similar to:

Type	Value
WORD	the
WORD	man
WORD	saw
WORD	a
WORD	dog
SYMBOL	.

This shows that the tokenizer correctly distinguishes word tokens and punctuation symbols. Numbers or mathematical expressions such as `‘‘x1 + x2 = 10’’` generate a mix of `WORD`, `NUMBER`, and `SYMBOL` tokens, demonstrating how DFA-based tokenization can be used for both natural language and simple formulas.

8.2 Natural Sentence Classification

Table 1 summarizes the heuristic classification results for the four example sentences.

Table 1: Heuristic judgment of whether a sentence is likely natural.

Sentence	Likely Natural?
The man saw a dog.	YES
cat table green quickly.	NO
I will build a parser using automata.	YES
$x1 + x2 = 10$	NO

The sentences that have a clear subject and verb (`‘‘The man saw a dog.’’` and `‘‘I will build a parser using automata.’’`) satisfy the heuristic. The purely nominal and adverbial sequence `‘‘cat table green quickly.’’` and the mathematical expression lack a proper verbal ROOT with a subject, so they are correctly classified as unlikely to be natural sentences.

8.3 Connection to Automata and Grammars

The project demonstrates the following connections:

- The DFA tokenizer is a direct application of regular languages and deterministic finite automata studied in Theory of Computation.
- The dependency tree and the natural sentence heuristic are inspired by CFG ideas, where a valid sentence must expand from a start symbol S to valid constituents like NP and VP.
- By combining a DFA-based front-end with a CFG-style parsing back-end (implemented using spaCy's learned grammar), we obtain a practical yet theoretically grounded NLP pipeline.

9 Conclusion and Future Work

In this project, we implemented a small natural language processing system that combines:

- A deterministic finite automaton for tokenizing input strings into words, numbers, and symbols.
- A dependency-tree-based representation of sentence structure derived from spaCy.
- A simple context-free-grammar-inspired heuristic for detecting whether a sentence is likely to be a natural English sentence.

This work illustrates how abstract theoretical models such as DFA and CFG can be embedded into concrete applications. Even though modern NLP heavily uses statistical and neural methods, the underlying notions of tokens, parse trees, and grammatical constraints remain rooted in the theory of computation.

Possible directions for future work include:

- Designing an explicit CFG for a subset of English and implementing a top-down or bottom-up parser to compare with the dependency-based approach.
- Extending the DFA tokenizer to handle more complex token types, such as multi-word tokens or abbreviations.
- Incorporating additional grammatical features (e.g., object presence, agreement, tense) into the natural sentence heuristic.
- Evaluating the system on a larger set of sentences and analyzing false positives and false negatives.

References

1. Hopcroft, J.E., Motwani, R., Ullman, J.D. *Introduction to Automata Theory, Languages, and Computation.* Pearson.

2. Sipser, M. *Introduction to the Theory of Computation.* Cengage Learning.

3. Jurafsky, D., Martin, J.H. *Speech and Language Processing.* Pearson.

4. spaCy Documentation: `https://spacy.io/`

5. NLTK Documentation: `https://www.nltk.org/howto/parse.html`