The Parser (Syntax Analysis)

I. Language Description

1. VALUE

A **value** in JSON cal be one of the following:

- **dict** : A JSON object, which is a collection of key-value pairs
- **list** : A JSON array, which is an ordered list of values
- **STRING** : Text data enclosed in double quotes
- **NUMBER** : A numerical value, which could be an integer or a floating-point number
- **“true”, “false” and “null”** : Boolean values(true or false) or a null value (null)

2. LIST

- A **list**(or JSON array) starts with an opening square bracket ‘[‘ and ends with a closing square bracket ‘]’
- Inside the brackets, there can be multiple **value**s, separated by commas ‘,’
- The **(, value)*** means that after the first value, there can be zero or more additional **value**s each preceded by a comma
- Example of a **list:** [1, “hello”, true, {“key”: “value”}, [2,3]]

3. DICT

- A **dict**(or JSON object) starts with an opening curly brace ‘{‘ and ends with a closing curly brace ‘}’
- Inside the braces, there are **pair**s(key-value pairs) separated by commas
- The **(, pair)*** means that after the first **pair**, there can be zero or more additional **pair**s each preceded by a comma.
- Example of a **dict:** { "name": "Alice", "age": 30, "isStudent": false }

4. PAIR

- A **pair** is a key-value pair, where the key is a **STRING** followed by a colon ‘:’ and then a **value**
- The key(which is always a **STRING** in JSON) represents an attribute or a property, and the **value** represents the data associated with that attribute
- Example of a **pair:** “name”: “Alice”

Putting things together

- JSON data can either be a **dict**(object) or a list(array) at the top level
- Within a **dict**, you have **pair**s, which map keys(strings) to values
- Within a **list**, you have a sequence of values(which can be strings, numbers, objects, arrays, booleans or **null**)
-

Overview

This parser is designed for a JSON-like structure, using a **DFA** for lexical analysis to tokenize the input. The goal of this project is to parse JSON data and build a **parse tree** that captures its hierarchical structure.

JSON Grammar

The parser relies on the following formal grammar, simplified for JSON:

- **Object**: {} with key-value pairs
- **Array**: [] with comma-separated values
- **Values**: Strings, numbers, booleans (true/false), and null (null)

Alphabet and Tokenization

The alphabet used for tokenizing includes:

- Symbols: **{**, **}**, **[**, **]**, **:**, **,**
- Strings: Alphanumeric sequences within quotes(**“”**)
- Numbers: Digits
- Keywords: **true**, **false**, and **null**

The DFA lexer is used to identify token types, such as **LBRACE** (for **{**), **STRING**, **NUMBER**, and others. These tokens are then passed to the parser, which validates syntax and builds the parse tree.

II. Code Explanation

The parser uses a **recursive descent** approach, where each grammatical construct has an associated function. This modular structure allows the parser to handle nested objects and arrays by recursively calling functions.

Class Structure

1. **Node Class**: Represents nodes in the parse tree.
    - **Attributes**:
        - **label** for node identification,
        - **is_leaf** to denote if the node is a leaf,
        - **parent** to track the parent node.
    - **Functions**:
        - **add_child** for adding child nodes.
        - **pre_order_traversal_print** and **pre_order_traversal_output** for printing the tree structure.
2. **ParseTree Class**: Represents the entire parse tree and is responsible for constructing it based on the input tokens.
    - **Attributes**:
        - **lexer**: The DFA lexer instance.
        - **current_token**: The current token being processed.
        - **current_token_index**: Tracks the position in the token stream.
    - **Functions**:
        - **get_next_token**: Advances to the next token.
        - **parse_list**: Handles array structures.
        - **parse_dict**: Handles object structures.
        - **parse**: The main function that initiates parsing based on the starting token.

Parsing Logic

- **parse_dict**: This function constructs subtrees for **JSON objects**. It:
    - Validates the presence of **{** and **}** for object boundaries.
    - Verifies pairs by matching **STRING** for keys and checking for **COLON**.
    - Recursively processes values within **pairs**, allowing for nesting of **objects** and **arrays**.
- **parse_list**: Similar to **parse_dict**, but for **JSON arrays**. This function handles:
    - Bracket matching **[**, **]** for array boundaries.
    - Recursive parsing for **nested lists** or **dictionaries** within **arrays**.
    - Proper handling of **commas** between elements.
- **parse**: The main entry point. This function checks if the input starts with an **object** (**{**) or array (**[**) and calls the appropriate function to begin parsing.

III. Error Handling

This parser implements error handling to detect and raise meaningful exceptions when parsing invalid JSON. The error handling strategy focuses on capturing errors in real-time and providing detailed messages for easy debugging. Although the parser doesn't have full recovery mechanisms to continue parsing after an error, it's designed to guide the developer toward correcting the input quickly.

Error Handling Mechanism

1. **Unexpected Tokens**: The parser expects specific tokens at each step, such as colons after keys in dictionaries, commas between items in lists, and closing braces or brackets for nested structures. Any deviation from these expectations raises an exception with details about the incorrect token and its position.
2. **Incomplete Structures**: The parser checks that all lists (**[** ... **]**) and dictionaries (**{** ... **}**) are properly closed. When an expected end token (**]** or **}**) is missing, it raises an exception, indicating that the structure was not completed.
3. **Misplaced Delimiters**:
    - **Commas in Arrays and Objects**: Commas are required to separate elements in arrays and key-value pairs in dictionaries. Extra or misplaced commas, especially before the end of an array or dictionary, raise an error to indicate a malformed structure.
    - **Missing Colons in Dictionaries**: Each key-value pair in a dictionary should have a colon (**:**) between the key and value. If a colon is missing, an exception is raised with the token position, indicating that a colon was expected.

Error Messages

Each error is accompanied by a descriptive message, including the token type, current index, and additional details to help debug the issue. Below are some of the specific messages the parser might generate:

- **Unexpected Token**: Raised if an unrecognized token appears in a specific structure, e.g., "Unexpected token in list NUMBER at token number 4."
- **Missing End Token**: Raised if a closing bracket or brace is missing, e.g., "Missing end of file token (RBRACE / RBRACKET)".
- **Misplaced Comma**: Raised if a comma appears before the end of a list or dictionary, e.g., "Comma before end of list at token number 6."
- **Missing Colon**: Raised if a colon is expected after a key but not found, e.g., "Expected COLON after key in dict, found NUMBER."

Example Scenarios

Below are some example JSON structures and the errors that would be triggered:

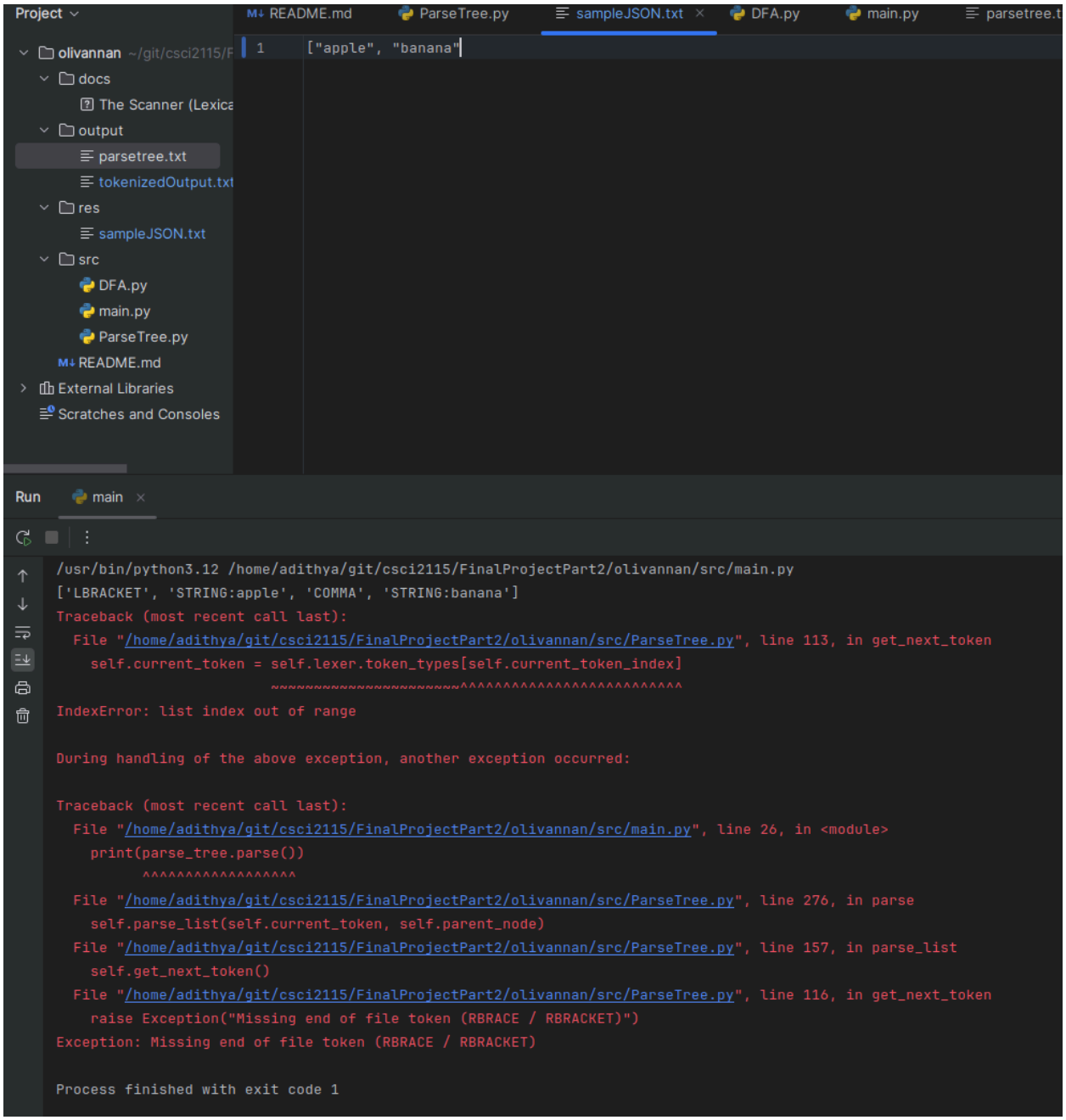

Fig.1: Unexpected Token: For instance, if an array is missing a closing bracket

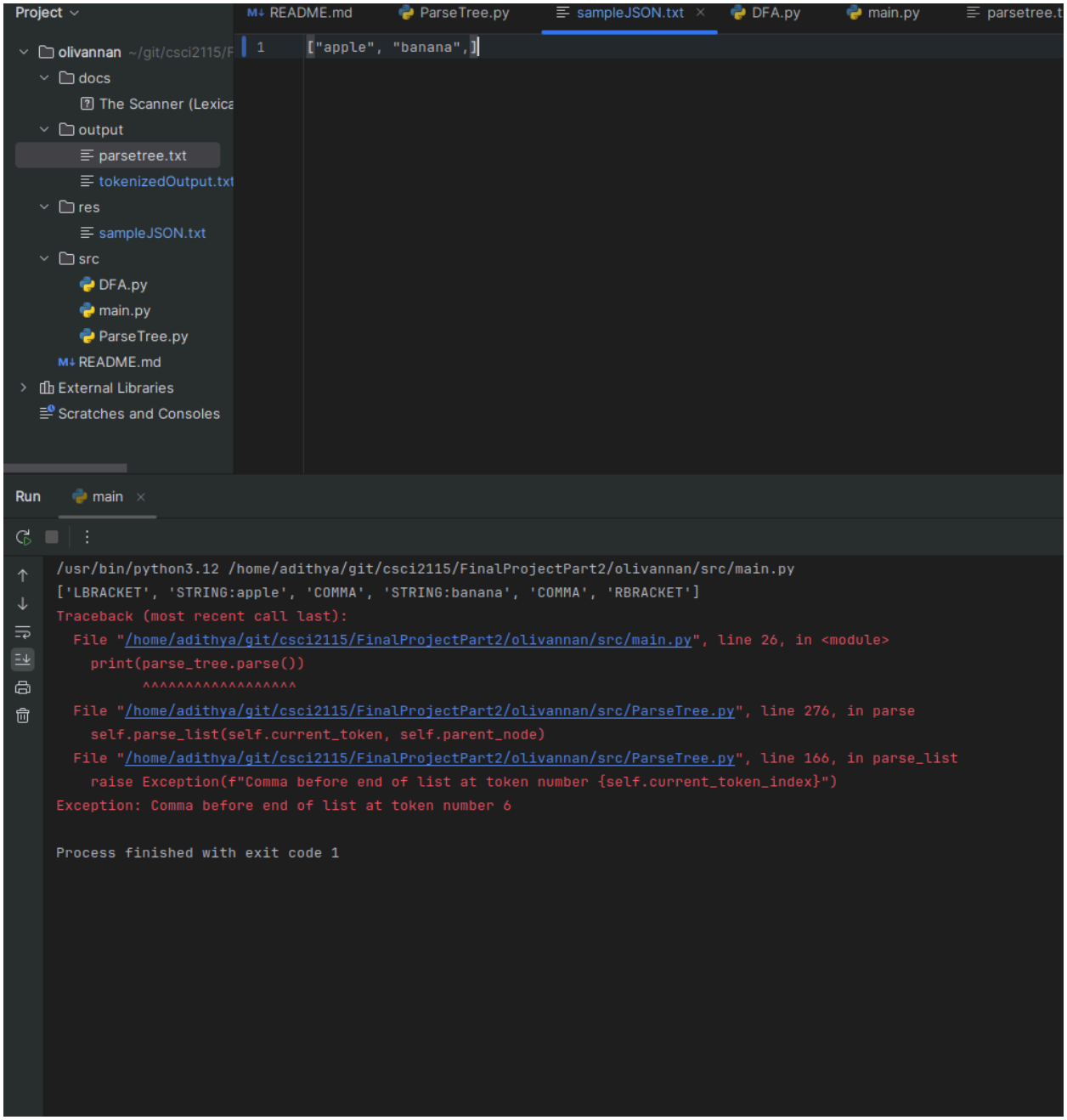

Fig. 2: Trailing comma before end of list

IV. REFLECTION

What Worked Well

- **Tree Structure Design:** The recursive descent structure allowed for clear and manageable code by dividing JSON syntax into smaller, understandable units.
- **Error Handling:** Error detection for unexpected tokens and incomplete structures added reliability, making the parser more resilient against malformed inputs.

Challenges Encountered

- **Handling Complex Nesting:** Managing deeply nested dictionaries and lists was challenging, introducing potential ambiguities and infinite loops.
- **Limited Error Reporting:** The parser currently stops on the first error, limiting feedback for debugging. Identifying multiple errors in a single run would improve its usability.

Improvements for Future Iterations

- **Enhanced Error Recovery:** Implementing an error recovery mechanism to continue parsing after minor errors would provide more comprehensive feedback.
- **Optimized Token Traversal:** Modifying traversal logic to better handle errors and improve parsing efficiency, especially for large JSON files.

Impact of Understanding Context-Free Grammars (CFG) and Recursive Descent Parsers

- **Shaping the Project:** Knowledge of CFG and LL(1) grammar influenced the project's structure
- **Connection to Real-World Applications:**
    - **Programming Languages:** Parsers like this one are required in compilers and interpreters, validating and translating syntax.
    - **Data Processing:** Parsing techniques support data validation, serialization, and deserialization in formats like JSON and XML, ensuring data integrity across applications.