The Parser (Semantic 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**

Building the Parse Tree

- **Tokenization:**
    - The input string is first tokenized into meaningful components by the DFA lexer
- **Grammar Rules:**
    - I use a context-free grammar that supports LL(1) parsing. Each rule corresponds to a specific structure of the language:
        - **Value :** A scalar, list or dictionary
        - **Dictionary :** {key : value, …}\
        - **List :** [value, value, …]
        - **String :** Enclosed in double quotes
        - **Number :** A valid integer or decimal

- **Parse tree construction:**
    - Parsing proceeds top-down using recursive descent. For each non-terminal symbol, a corresponding function processes the tokens and builds nodes in the tree.
    - Each node in the parse tree represents a syntactic construct, such as a **Dictionary**, **List**, **String**, or **Number**.
    - Leaf nodes hold terminal symbols (literals), while intermediate nodes represent composite structures.

Ensuring Non-Ambiguity of Modified Grammar

- **Modified Grammar:**
    - The grammar has been modified to ensure it adheres to LL(1) properties:
        - No left recursion
        - Lookahead of one token is sufficient to decide the next rule
    - For example, ambiguous constructs like **value** being both a **list** and **dict** are resolved by the distinct opening characters (**[** vs. **{**)

- **Why it's Not Ambiguous:**
    - **Deterministic Decisions:** Each construct starts with a unique terminal (**{, [** or a literal)
    - **No Overlapping Derivations:** No input string can generate more than one valid parse tree under the grammar.
    - **Clear Separation of Rules:** Each terminal symbol corresponds to a single, unambiguous syntactic category. (**STRING** starts with ("), **KEYWORD** start with the letters (t,f,n), **NUMBER** is considered anything else)

The Abstract Syntax Tree

The parser generates an Abstract Syntax Tree(AST) to represent the hierarchical structure of the input JSON data. The construction of the AST is done during the parsing phase, after the input is tokenized by the DFA(scanner).

Building the AST

Tokenization : The DFA breaks the input JSON string into a sequence of tokens, such as **STRING, NUMBER, COLON** etc…
Parsing : The parser processes the token stream to build the tree structure where each node corresponds to a JSON element. The parser follows the standard JSON grammar to construct a tree with the following general structure:

- **dict / list :** Root node depending on the first token input into the parser (**LBRACE**, **LBRACKET**)
- **dict :** Each key-value pair in the JSON object is represented as a node in the tree. The key is always a **STRING** leaf.
- **list :** Arrays are represented by a list of child nodes that correspond to the elements in the array.
- **leaf :** The individual values (**STRING, NUMBER, BOOL**) are represented as leaf nodes in the tree.

Attribute Grammar

```
Dict → { Key : Value (, Key : Value)* }
List → [ Value (, Value)* ]
Value → STRING | NUMBER | KEYWORD | List | Dict
Key → STRING
```

- **Dict**: A dictionary is represented as a set of key-value pairs, enclosed in curly braces {}. Each key is associated with a value, and multiple pairs are separated by commas.
- **List**: A list is represented as a sequence of values, enclosed in square brackets [], with values separated by commas.
- **Value**: A value can be a string, a number, a keyword, another list, or a dictionary, allowing for nested structures.
- **Key**: A key within a dictionary must be a string.

List of Semantic Errors

Type I : Invalid Decimal Numbers
Type II : Empty Key
Type III : Invalid Numbers
Type IV : Reserved Words as Dictionary Key
Type V : No duplicate Keys in Dictionary
Type VI : Consistent Types for List Elements
Types VII : Reserved Words as Strings

The parser uses token streams directly from the DFA, and there is no intermediary (All the tokens used by the parser were generated by the DFA only).

Enforcing Semantic Rules

- **Invalid Decimal Numbers :** To check if decimals follow the correct format, there are statements inside parse_dict() and parse_list() that, when reading a NUMBER token, check if the number ends or begins with a decimal.
- **Empty Dictionary Keys :** In the parse_dict() function, we assume that we will read a string first(key), now if this string was empty (we can check this using a boolean operator on the string variable) or contained only spaced (we can check this using the str.isspace() method) we throw an exception.
- **Reserved Words as Keys :** In the parse_dict() function, we assume that we will read a string first(key), now if this string was a keyword (true, false, null), we throw an exception.
- **Duplicate Keys in Dictionaries :** Inside the parse_dict() function, we assume that we will read a string first, we can store these keys inside a list. If we read in a string that is inside the list, this indicates a duplicate key and we will throw an error
- **Inconsistent Types in Lists :** When we're reading in the inputs in the parse_list() function, we keep track of the datatype of the input. If the datatype of the last input we read is different from the input we're currently reading, we throw an Exception.
- **Invalid Number Format :** To check if numbers follow the correct format, there are statements inside parse_dict() and parse_list() that, when reading a NUMBER token, check if the number begins with a 0 or a + sign.
- **Reserved Words as Strings :** To check if there are reserved words inside strings, there are statements inside parse_dict() and parse_list() that, when reading a STRING token, checks if the string is a keyword(true, false, null)

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

- **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.
- **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.

Constructing the AST

For semantically correct JSON inputs(Check **Enforcing Semantic Rules** for more details about how we check if JSON input is semantically correct), the parser builds an AST. The AST is a simplified, cleaner version of the parse tree, focusing on essential structural and semantic components while omitting syntactic details that are irrelevant to JSON interpretation. The AST is built alongside the parse tree, but nodes are only created for certain elements.

- **Node Simplification :** The AST contains nodes for JSON objects, arrays, and values but omits syntactic details like commas and braces.

- **Array Nodes :** Each array in the AST is represented as a list of elements, with each element's type stored for consistency checking. This differs from the parse tree, which includes intermediate nodes for commas and individual values in an array sequence.
- **Elimination of Non-Essential Tokens :** Since the AST's purpose is to represent the semantic structure rather than syntactic form, tokens such as brackets, colons and commas are removed.
- **Elimination of String Nodes :** Since the input string may initially represent a larger structure (such as an array or dictionary) without clear indication, it becomes necessary to revisit and remove the child node from the AST when this determination is made.
- **Key-value Pair Structure** : Within dictionaries, to maintain semantic clarity, a single node should encapsulate both the key and its corresponding value. Therefore, after reading the key (whether a string, keyword, or number), the associated value can be appended to the previously constructed key node.

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

The parser (Syntax analysis)

- **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.
- **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.
- **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."

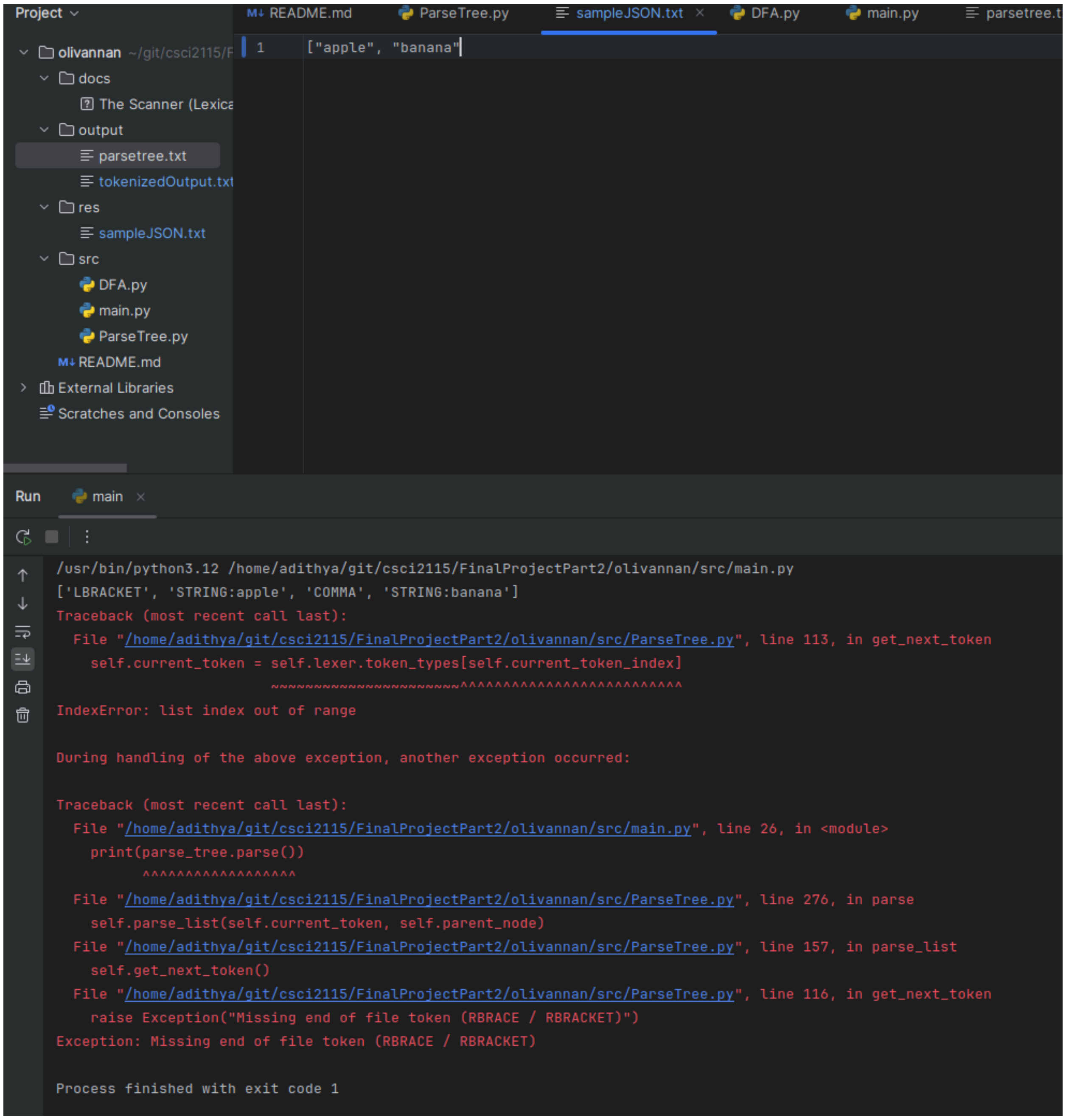
The parser (Semantic analysis)

- **Invalid Decimal Numbers (Type I) :**
    - Input : {“value”: 3.}
    - Output : Error type 1 at 3. : Invalid Decimal Number
- **Empty Key (Type II) :**
    - Input : {“”: 3.0}
    - Output : Error type 2 at . : Empty Key
- **Invalid Numbers (Type III) :**
    - Input : {“value”: 01}
    - Output : Error type 3 at 0 : InvalidNumber
- **Reserved Words as Dictionary Key (Type IV) :**
    - Input : {“true”: 3.0}
    - Output : Error type41 at true. : Reserved Words as Dictionary Key
- **No Duplicate Keys (Type V) :**
    - Input : {“value”: 3.0, “value”:4.0}
    - Output : Error type 5 at value : No Duplicate Keys
- **Consistent Types for List Elements (Type VI) :**
    - Input : [“Hello”, 3.1]
    - Output : Error type 6 at 3.1 : Inconsistent types in list
- **Reserved Words as Strings (Type VII) :**
    - Input : {“value”: “true”}

- Output : Error type 7 at true : Reserved Words as Strings

Example Scenarios

Below are some example JSON structures and the errors that would be triggered: (**Check next page**)



```
["apple", "banana"
```

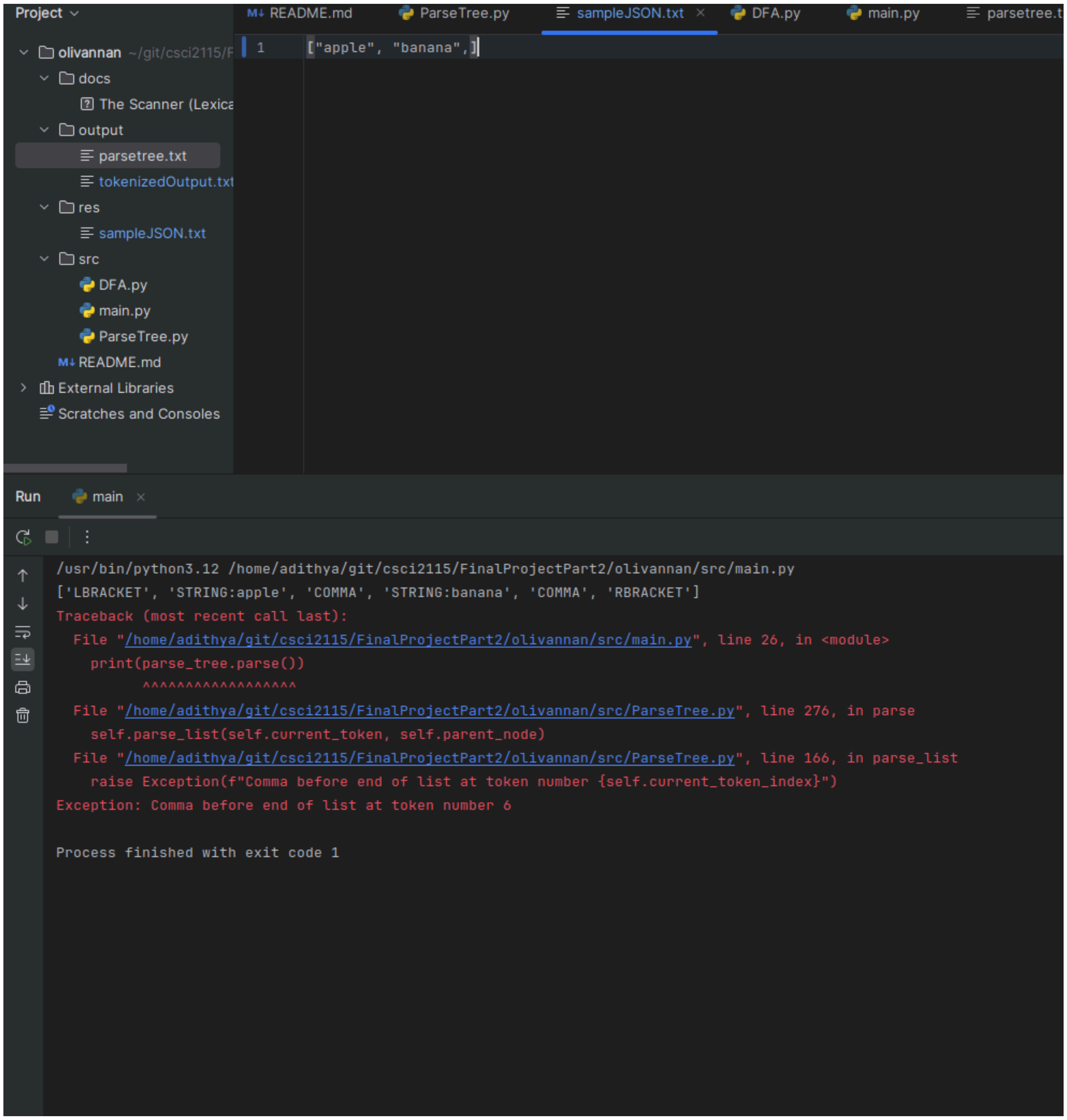
```
/usr/bin/python3.12 /home/adithya/git/csci2115/FinalProjectPart2/olivannan/src/main.py
['LBRACKET', 'STRING:apple', 'COMMA', 'STRING:banana']
Traceback (most recent call last):
  File "/home/adithya/git/csci2115/FinalProjectPart2/olivannan/src/ParseTree.py", line 113, in get_next_token
    self.current_token = self.lexer.token_types[self.current_token_index]
                         ~~~~~~~~~~~~~~~~~~~~~~^^^^^^^^^^^^^^^^^^^^^^^^^^
IndexError: list index out of range

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "/home/adithya/git/csci2115/FinalProjectPart2/olivannan/src/main.py", line 26, in <module>
    print(parse_tree.parse())
          ^^^^^^^^^^^^^^^^^^
  File "/home/adithya/git/csci2115/FinalProjectPart2/olivannan/src/ParseTree.py", line 276, in parse
    self.parse_list(self.current_token, self.parent_node)
  File "/home/adithya/git/csci2115/FinalProjectPart2/olivannan/src/ParseTree.py", line 157, in parse_list
    self.get_next_token()
  File "/home/adithya/git/csci2115/FinalProjectPart2/olivannan/src/ParseTree.py", line 116, in get_next_token
    raise Exception("Missing end of file token (RBRACE / RBRACKET)")
Exception: Missing end of file token (RBRACE / RBRACKET)

Process finished with exit code 1
```

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

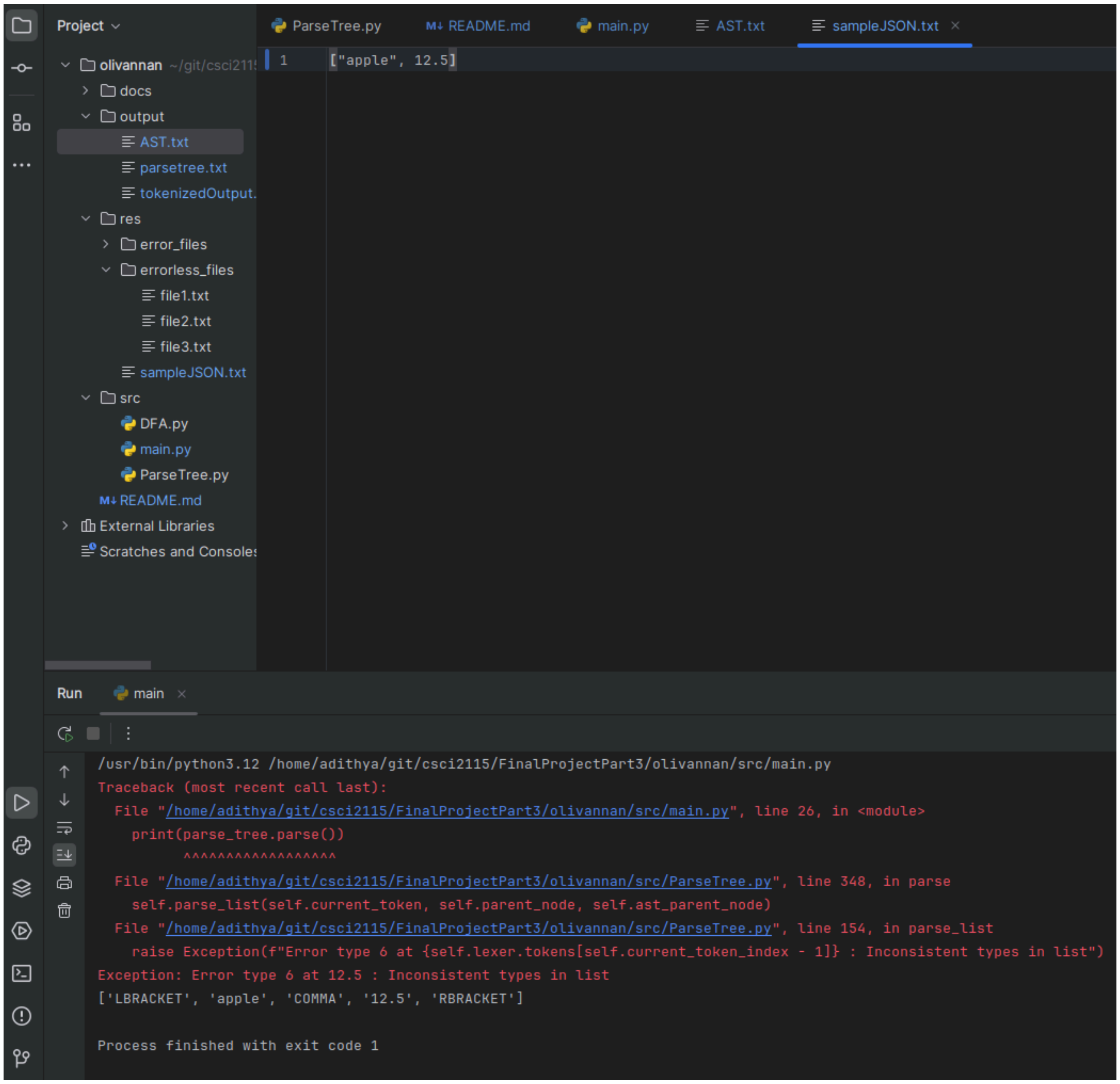


```
["apple", "banana",]
```

```
/usr/bin/python3.12 /home/adithya/git/csci2115/FinalProjectPart2/olivannan/src/main.py
['LBRACKET', 'STRING:apple', 'COMMA', 'STRING:banana', 'COMMA', 'RBRACKET']
Traceback (most recent call last):
  File "/home/adithya/git/csci2115/FinalProjectPart2/olivannan/src/main.py", line 26, in <module>
    print(parse_tree.parse())
          ^^^^^^^^^^^^^^^^^^
  File "/home/adithya/git/csci2115/FinalProjectPart2/olivannan/src/ParseTree.py", line 276, in parse
    self.parse_list(self.current_token, self.parent_node)
  File "/home/adithya/git/csci2115/FinalProjectPart2/olivannan/src/ParseTree.py", line 166, in parse_list
    raise Exception(f"Comma before end of list at token number {self.current_token_index}")
Exception: Comma before end of list at token number 6

Process finished with exit code 1
```

Fig. 2: Trailing comma before end of list

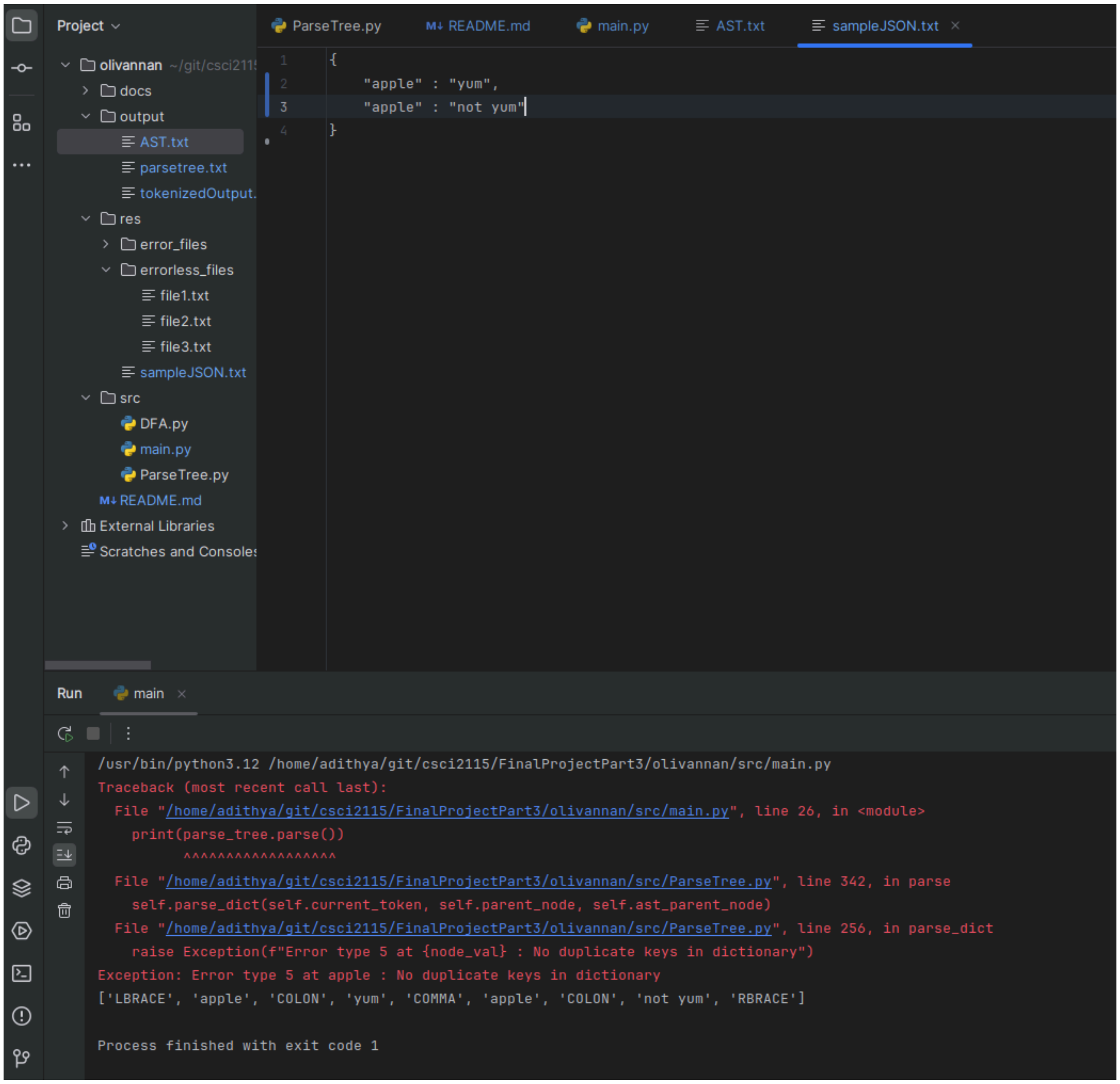


```
["apple", 12.5]
```

```
/usr/bin/python3.12 /home/adithya/git/csci2115/FinalProjectPart3/olivannan/src/main.py
Traceback (most recent call last):
  File "/home/adithya/git/csci2115/FinalProjectPart3/olivannan/src/main.py", line 26, in <module>
    print(parse_tree.parse())
          ^^^^^^^^^^^^^^^^^^
  File "/home/adithya/git/csci2115/FinalProjectPart3/olivannan/src/ParseTree.py", line 348, in parse
    self.parse_list(self.current_token, self.parent_node, self.ast_parent_node)
  File "/home/adithya/git/csci2115/FinalProjectPart3/olivannan/src/ParseTree.py", line 154, in parse_list
    raise Exception(f"Error type 6 at {self.lexer.tokens[self.current_token_index - 1]} : Inconsistent types in list")
Exception: Error type 6 at 12.5 : Inconsistent types in list
['LBRACKET', 'apple', 'COMMA', '12.5', 'RBRACKET']

Process finished with exit code 1
```

Fig. 3: Semantic error : inconsistent types in list



```
{
    "apple" : "yum",
    "apple" : "not yum"
}
```

```
/usr/bin/python3.12 /home/adithya/git/csci2115/FinalProjectPart3/olivannan/src/main.py
Traceback (most recent call last):
  File "/home/adithya/git/csci2115/FinalProjectPart3/olivannan/src/main.py", line 26, in <module>
    print(parse_tree.parse())
          ^^^^^^^^^^^^^^^^^^
  File "/home/adithya/git/csci2115/FinalProjectPart3/olivannan/src/ParseTree.py", line 342, in parse
    self.parse_dict(self.current_token, self.parent_node, self.ast_parent_node)
  File "/home/adithya/git/csci2115/FinalProjectPart3/olivannan/src/ParseTree.py", line 256, in parse_dict
    raise Exception(f"Error type 5 at {node_val} : No duplicate keys in dictionary")
Exception: Error type 5 at apple : No duplicate keys in dictionary
['LBRACE', 'apple', 'COLON', 'yum', 'COMMA', 'apple', 'COLON', 'not yum', 'RBRACE']

Process finished with exit code 1
```

Fig. 4: Semantic error : no duplicate keys in dictionary