JSON PARSER

SIDHANT KUMAR
B00961444

CSCI 2115
THEORY OF COMPUTER SCIENCE

Language Description

This part of the project covers a recursive descent parser to parse a JSON like language as well as a simplified Newick format for tree representation. The language consists of lists, dictionaries(objects), strings, numbers, booleans and null values. The alphabet of the language includes:

- Characters: Letters (either uppercase or lowercase).
- Tokens: The language defines tokens as:
    - LBRAC: '[' (Used in declaring list)
    - RBRAC: ']' (Used in ending list)
    - LPAREN: '{' (Used in declaring dict)
    - RPAREN: '}' (Used in ending dict)
    - STRING: Enclosed within double quotes, for example: "CSCI".
    - NUMBER: Any numerical value.
    - BOOL: Boolean values (true or false).
    - NULL: Represented by keyword null.
    - COMMA: ',' (Used in separating values or pairs)
    - COLON: ':' (Used in key-value pairs)

The language's grammar is intended to be clear. There is no misunderstanding regarding the proper way to interpret inputs because each rule has a distinct and unambiguous structure. For example, there is no overlap between the definitions of lists and dictionaries that can cause confusion during parsing. Each function in the recursive descent parser is closely correlated with a grammar rule, guaranteeing that the parser interprets input in a predictable fashion.

The syntactic structure of the input is captured by the parse tree that the parser produces. A grammatical construct is represented by each node in the tree, and its constituent parts are represented by child node. A dictionary for example will have a node called "dict" with child nodes for each key-value pair separated by ',' e. Similarly for a list, will have a node called "list" with the child nodes values separated by ','

Code Explanation

The parser uses a recursive descent parsing technique, which builds a parse tree for the input using a collection of recursive functions, each of which corresponds to a distinct grammatical rule. A set of tokens from an input file are used to initiate the parser. The main parts of the code are broken down as follows:

- InputReader Class: This class reads the input file from a batch of .txt files containing the token streams generated by the Scanner made in Project Part 1. It reads the input file line by line and removes the <> generated by the Scanner and then splits the value separated by ',' into token's type and value respectively. It then makes a token object with that type and value.

- Node Class: This class represents the nodes of the parse tree. Each node can have children, which allows a hierarchal structure.
- Parser Class: This is the most important class for parser as it divides the tokens according to the grammar rule, specified below:
    - value(): It determines if the current token is a list, dict , string, number, Boolean, or null.
    - list(): It parses a list, while handling commas and ensuring that the syntax is correct. As according to the grammar list starts with '[' then consists of values and then ends with ']', else raises an exception.
    - dict(): It parses a dict, while handling key-value pairs and their syntax. As according to the grammar dict starts with '{' and then consists of pairs separated by ',' and then ends with '}', else raises an exception.
    - pair(): It parses a pair while handling the syntax. Pair is often used within dict and is separated by ':', else raises an exception.
    - string(): It parses STRING.
    - number(): It parses NUMBER.
    - boolean(): It parses BOOL.
    - null(): It parses NULL.
    - eat(): It parses while eating other tokens and raising exception if the current token is different than the required one.
    - Error handling: There is exception handling in the parser to manage the exception handling due to unexpected token or syntax errors. It will display an error message if an exception is raised.

The algorithm strictly follows the grammar rules, so it makes sure that there is no syntax error. If there is then the parser raises an exception, letting the user know that there is something wrong with the code, hence allowing comprehensive validation of the input before it is accepted as valid.

Error Handling & Recovery

An essential part of the parser is error handling, which guarantees resilience and offers insightful feedback in the event of syntax problems. During the parsing process, the parser detects the errors and records them using exceptions.

For example:

- Unexpected Token: If at a given point in the grammar, a token is not recognized while parsing , then an exception will be raised to let the user know about the syntactic error.
- Comma at the end of a list: If a comma is encountered within list that is at the end with no value following it. E.g, ["abc",123,] then it raises an exception.
-

When an error is encountered then an error message is printed in the output.txt notifying the user where the error would be.

Examples of Error handling in the parser code:

1.) If we have a input file containing token stream generated by Project Part 1 (Scanner)as input:

```
<LBRAC, '['>
<STRING, 'abc'>
<COMMA, ','>
<NUMBER, 123.0>
<COMMA, ','>
<RBRAC, ']'>
```

As its suggests that there is an extra ‘,’ at the end of the list without a value following it then it raises an error and prints the exception in the output.txt

```
Outputs > output.txt
1   Error detected during parsing: Comma at the end of the list
2
```

2.) Similarly, for dict

```
<LPAREN, '{'>
<STRING, 'abc'>
<COLON, ':'>
<NUMBER, 123.0>
<COMMA, ','>
<STRING, 'def'>
<COLON, ':'>
<NUMBER, 456.0>
<COMMA, ','>
<RPAREN, '}'>
```

This is the exception raised by the parser.

```
Outputs > output.txt
1   Error detected during parsing: Expected token STRING, got RPAREN
2
```

As the error suggests that after ‘,’ there should be a String according to the grammar, but it found ‘}’. Suggesting that a pair starts with String followed by value.

The current parser just finds the errors and handles them by notifying the user about it , but it does not recover from them. A recovery mechanism could be done by skipping the unrecognized token or by taking input from the user replacing the token, which will ultimately result in a complete error free and smooth experience for users.