JSON PARSER

SIDHANT KUMAR
B00961444

CSCI 2115
THEORY OF COMPUTER SCIENCE

Language Description

The given parser makes Abstract Syntax Tree (AST) for a JSON syntax like language following grammar rules for list, dict, String, Number, Boolean, and null. This part of the project contains Abstract Syntax Tree instead of parse tree, Abstract Syntax Tree which are like a compressed form of parse trees [1]. An AST is essentially a simplified version of a parse tree. It is simpler to reason about the structure of the code when the nodes of an AST are abstracted away from the specifics of the language's syntax [2]. ASTs omit unnecessary syntax and focus on semantic relationships, making them more practical for compilation, whereas Parse Trees show complete syntactic structure according to grammar rules [2]. So basically, non-terminals and terminals does not play a role in AST and AST are much easier to understand than parse trees due to its more compact form and also, as it is easier to read [3].

This part of the project is detecting all types of semantic errors given, i.e. from Error type 1 to Error type 7. The error types are as follows:

- Level C
    - Error Type 1: Invalid Decimal Numbers. If a number is encountered with value .1 or 1. then error type 1 is raised.
    - Error Type 2: Empty Key. If any key in a dict is empty ("" or " ") then error type 2 is raised.
- Level B
    - Error Type 3: Invalid Numbers. If a number is encountered starting with a 0 or '+' then error type 3 is raised.
    - Error Type 4: Reserved Words as Dictionary Key. If any key in a dict is keyword ("true", "false") then error type 4 is raised.
- Level A
    - Error Type 5: No Duplicate Keys. If a key in a dict is repeated in the same dict then error type 5 is raised.
    - Error Type 6: Consistent Types for List Elements. If every element in a list is not of the same type, then error type 6 is raised.
    - Error Type 7: Reserved Words as String. If anywhere a keyword ("true" , "false") is used as a String then error type 7 is raised.

4

Now, each of the input files (from input1.txt to input7.txt) demonstrates the above explained errors.

- input1.txt
  [3. , 123.0 , 456.78] . The token stream is :

  ```
  <LBRAC, '['>
  <NUMBER, 3.>
  <COMMA, ','>
  <NUMBER, 123.0>
  <COMMA, ','>
  ```

```
<NUMBER, 456.78>
<RBRAC, ']'>
```

The output of the parser is:
Error detected during parsing --> Error type 1 at 3.: Invalid Decimal Numbers

- input2.txt
  { “ “ : 123.0 , “ABC” : 456.0}. The token stream is :

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

The output of the parser is:
Error detected during parsing --> Error Type 2 at : Empty Key

- input3.txt
  [1.24 , 123.04 , 013]. The token stream is :

  ```
  <LBRAC, '['>
  <NUMBER, 1.24>
  <COMMA, ','>
  <NUMBER, 123.04>
  <COMMA, ','>
  <NUMBER, 013>
  <RBRAC, ']'>
  ```

The output of the parser is:
Error detected during parsing --> Error type 3 at 013: Invalid Numbers

- input4.txt
  { “True” : 123.0 , ‘true’ : 456.0 }. The token stream is :

  ```
  <LPAREN, '{'>
  <STRING, 'True'>
  <COLON, ':'>
  <NUMBER, 123.0>
  <COMMA, ','>
  ```

```
<STRING, 'true'>
<COLON, ':'>
<NUMBER, 456.0>
<RPAREN, '}'>
```

The output of the parser is:
Error detected during parsing --> Error type 4 at true : Reserved Words as Dictionary Key

- input5.txt
  { “True” : 123.0 , ‘True’ : 456.0 }. The token stream is :

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

  The output of the parser is:
  Error detected during parsing --> Error type 5 at True: No Duplicate Keys in Dictionary

- input6.txt
  [“ABC” , 123.0 , 456.13]. The token stream is :

  ```
  <LBRAC, '['>
  <STRING, 'ABC'>
  <COMMA, ','>
  <NUMBER, 123.0>
  <COMMA, ','>
  <NUMBER, 456.13>
  <RBRAC, ']'>
  ```

  The output of the parser is:
  Error detected during parsing --> Error type 6 at 123.0: Consistent Types for List Elements

- input7.txt
  [“true” , “false” , “Hello”]. The token stream is :

  ```
  <LBRAC, '['>
  ```

```
<STRING, 'true'>
<COMMA, ','>
<STRING, 'false'>
<COMMA, ','>
<STRING, 'Hello'>
<RBRAC, ']'>
```

The output of the parser is:

```
Error detected during parsing --> Error type 7 at true: Reserved Words as Strings
```

Attribute Grammar

An attribute grammar is an enhanced context-free grammar where each symbols has attributes, and semantic rules for copying, evaluating, and enforcing restrictions on these attributes [1]. The attribute grammar for this part of the project according to the given grammar would be:

value -> dict	{value.type = “dict”}
value -> list	{value.type = “list”}
value -> STRING	{value.type = “STRING”, value.val = “ ”(STRING.val)}
value -> NUMBER	{value.type = “NUMBER”, value.val = NUMBER.vl}
value -> “true”	{value.type = “Bool”, value.val = true}
value -> “false”	{value.type = “Bool”, value.val = false}
value -> “null”	{value.type = “NULL”}
list -> “[“ value (“,” value)*”]”	{list.type = “list}
dict -> “{“ pair (“,” pair)*”}”	{dict.type = “dict”}
pair -> STRING “:” value	{pair.key = STRING.val, pair.val = value.val}

The above made attribute grammar is adapted from [1] (and tutorial assignment 8 solution as mentioned in the feedback). The production rule implemented by this attribute grammar follows a variety of very strict rules. It follows rule such as, if the input starts with a ‘{‘ then it considers it to be dict, similarly if it starts with ‘[‘ it considers it to be list. So according to the attribute grammar production rules, Value goes to any of list, dict, string, number, Boolean, null. So, the mentioned value becomes dict or list when it starts with ‘{‘ and ‘[‘ respectively. Now in each element of a list there is a Value which could take any of the value, and a dict contains pair. And if the parser encounters anything beyond the above-mentioned rules, then it throws a number of specific errors (From Error 1-7) with a specific message as where the error is and what the error is.

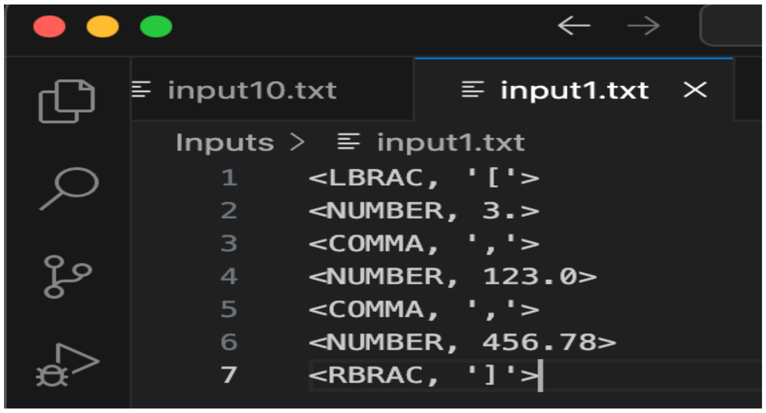
Code Explanation

The code of this part of the project is quite similar to the project part 2’s code. So, it parses the given token stream while following grammar rules very closely. It checks if during parsing it encounter any of the predetermined error types (Error Type 1 – 7) mentioned above under Language Description it raises an error and if it does not encounter anything then the input is considered to be semantically correct. Now for semantically correct input the code provides an Abstract Syntax Tree instead of parse tree (as of Part 2). The output implemented as an Abstarct Tree Syntax is quite similar of that of parse tree but removing unnecessary token values like brackets '{', '}' ,'[' , ']' and punctuations like ',' , ':' . Thus, providing a compact form of output than a parse tree.

Error Handling

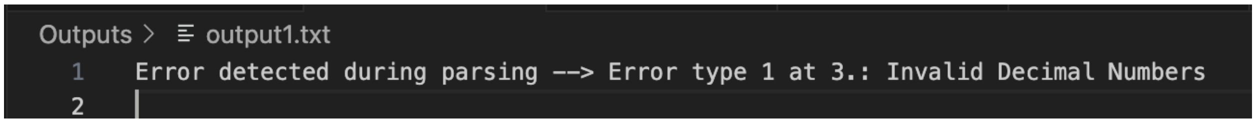
The code raises an error if any of the error types mentioned under Language Description is encountered. For example:

1.) For this input (input1.txt)



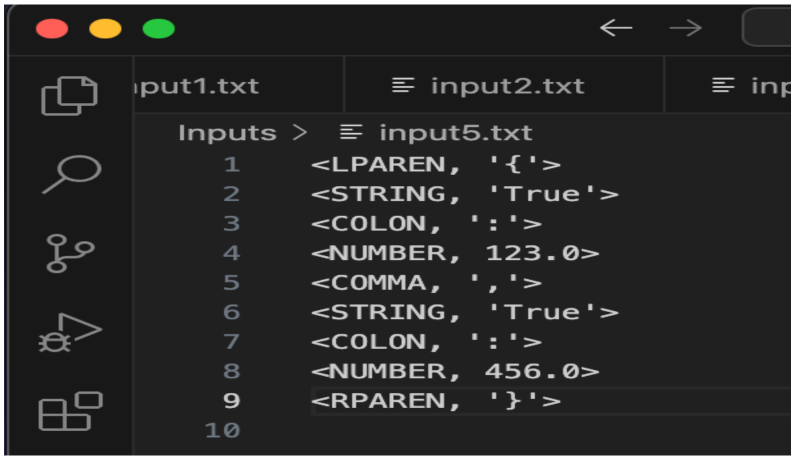
```
<LBRAC, '['>
<NUMBER, 3.>
<COMMA, ','>
<NUMBER, 123.0>
<COMMA, ','>
<NUMBER, 456.78>
<RBRAC, ']'>
```

This is the output

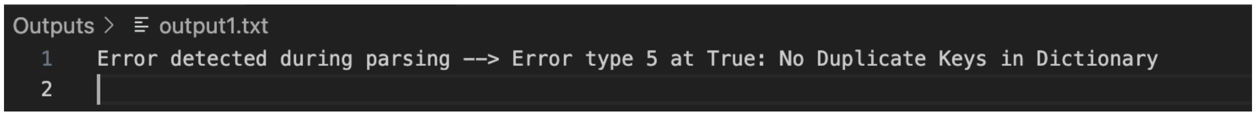


```
Error detected during parsing --> Error type 1 at 3.: Invalid Decimal Numbers
```

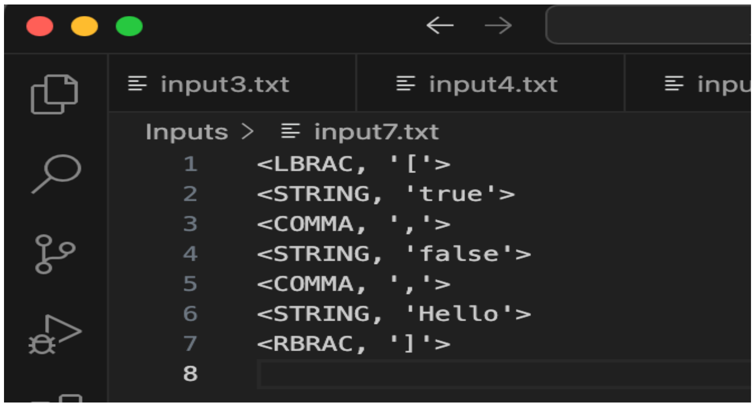
2.) For this input (input 5.txt)


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

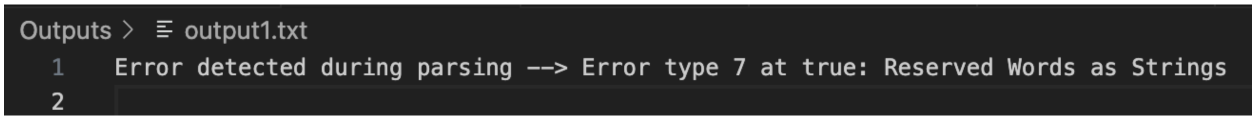
This is the output


```
Error detected during parsing --> Error type 5 at True: No Duplicate Keys in Dictionary
```

3.) For this input (input7.txt)


```
<LBRAC, '['>
<STRING, 'true'>
<COMMA, ','>
<STRING, 'false'>
<COMMA, ','>
<STRING, 'Hello'>
<RBRAC, ']'>
```

This is the output


```
Error detected during parsing --> Error type 7 at true: Reserved Words as Strings
```

Input1-7.txt are semantically incorrect following error type 1–7 respectively (it is also mentioned as comments in the code). The input8-10.txt are semantically correct which outputs abstract syntax tree.

References

[1] “Semantic Analysis - 2115-documentation,” Cs.dal.ca, 2024. https://web.cs.dal.ca/~baorui/2115-documentation/front/semantic/ (accessed Nov. 23, 2024).

[2] GeeksforGeeks, “Abstract Syntax Tree vs Parse Tree,” [Online]. Available: https://www.geeksforgeeks.org/abstract-syntax-tree-vs-parse-tree/. [Accessed: 11-Dec-2024].

[3] Stack Overflow, “What’s the difference between parse trees and abstract syntax trees (ASTs)?,” [Online]. Available: https://stackoverflow.com/questions/5026517/whats-the-difference-between-parse-trees-and-abstract-syntax-trees-asts. [Accessed: 11-Dec-2024].