**A JSON compiler frontend**

Author: Nafisah Nubah

**Language Description:**This project is a simple JSON parser that reads in **JSON (JavaScript Object Notation)** token streams (obtained from the scanner provided, or created by hand) from .txt files, carries out semantic analysis, and parses the data into an abstract syntax tree structure.

The **grammar** for JSON implemented by this parser is as follows:

value : dict  
 | list   
 | STRING   
 | NUMBER   
 | ”true” | ”false” | ”null”

list : ”[” value (”, ” value) ∗ ”]”

dict : ”{” pair (”, ” pair) ∗ ”}”

pair : STRING ” : ” value

In addition to maintaining this grammar, the parser also checks for the following types of **semantic errors**:

Type 1: Invalid Decimal Numbers

Decimal numbers must have digits on both sides of the decimal point.

Type 2: Empty Key

The key of a dictionary should not be empty.

Type 3: Invalid Numbers

Numeric values must follow proper formatting, there should not be leading zeros unless it is in decimal place, or leading + signs.

Type 4: Reserved Words as Dictionary Key

Keywords (true and false) cannot be keys of a dictionary, but True or False can, as they are not keywords defined by the grammar.

Type 5: No Duplicate Keys in Dictionary

Each key must be unique in a dictionary.

Type 6: Consistent Types for List Elements

All elements in a list must be of the same type.

The parser constructs **Abstract Syntax Trees (ASTs)** that capture a simplified representation of the syntactic structure of JSON. An abstract syntax tree is a compressed parse tree that represents the program structure rather than the parsing process, meaning that it is more concise. The nested children of Object and Array nodes are indicated using appropriate indentations. Pairs are expressed in ‘key : value’ form. If a semantic error is encountered, an error message is logged.

**Code Explanation:**The code implements a recursive descent parser with semantic analysis using Python, which uses a top-down parsing technique where each grammar rule is represented by its own function. It follows the following algorithm:

*For each non-terminal X, write a procedure parseX:*

*• Choose production X → Y1Y2...Yk based on current token with one lookahead*

*• For i =1,2,...,k:– If Yi is a terminal, match Yi with the next input token.– If Yi is a non-terminal, call parseYi.*

*• If no production rule can be matched, parsing fails (the input is syntactically incorrect).*

To start, our non-ambiguous JSON grammar is as follows:

value : dict  
 | list   
 | STRING   
 | NUMBER   
 | ”true” | ”false” | ”null”

list : ”[” value (”, ” value) ∗ ”]”

dict : ”{” pair (”, ” pair) ∗ ”}”

pair : STRING ” : ” value

This comprises of four different production rules, which are implemented using four different functions – *value(), list(), dict() and pair().* In addition, strings, numbers, Booleans and null are each recognised using individual functions to simplify the implementation of the parser.

The parser starts with a root function *parse()*, which initiates the parsing process by fetching the first token and calling *value(),* which then recognises the type of value to read. It then calls the function that handles that specific value type. Additionally, it also detects exceptions that may be thrown due to semantic errors in the input and logs meaningful error messages to the specified output file.

Functions such as *dict()* and *list()* make recursive calls to functions, corresponding to the attribute grammar. For instance, *value()* will call *dict()* if it encounters a {, which in turn will call *pair(). pair()* will call *value()* for the associated value. Each recursive call processes the tokens in a top-down fashion, consuming tokens until it reaches a leaf node in the tree, allowing for an abstract syntax tree to be constructed in this fashion.

The Node class defines and represents each individual node, with an *add\_child()* function to add children to internal nodes, creating a structure that reflects the input’s syntactic structure, and a *print\_tree()* function to format and write the output abstract syntax tree in a specified .txt file.

To determine whether an input is semantically correct or not, the parser carries out separate validations for each error type. This takes place in the following parts of the code:

**number()** – checks if a numerical value starts or ends with a decimal point, and throws a Type 1 error if true. Also checks if a numerical value that is not 0 or a decimal has leading 0’s or if it has leading + signs and throws a Type 3 error if true.

**pair()** – checks if the key is empty (or any amount of whitespaces), and throws a Type 2 error if true. Also checks if the key is a reserved keyword and throws a Type 4 error if true.

**list()** – keeps track of the data type of each element in the array by using a set (which does not allow duplicates), if the length of the set is greater than 1 it throws a Type 6 error, indicating that the elements don’t all have the same data type.

**duplicate\_key()** – this is called by dict() to check if a given key is a duplicate, and a Type 5 error is thrown if true.

For semantically correct inputs, the parser constructs an abstract syntax tree using the *print\_tree()* function. Below is an output example for a semantically correct input (check valid-input1.txt for input):

A screenshot of a computer

Description automatically generated