The Scanner (Lexical 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**)

II. The DFA

Table that depicts the DFA in table form

STATE	INPUT	NEXT STATE	OUTPUT
S0	{	S0	LBRACE
S0	}	S0	RBRACE
S0	[	S0	LBRACKET
S0	]	S0	RBRACKET
S0	:	S0	COLON
S0	,	S0	COMMA
S0	“	STRING	-
S0	Digit or -	NUMBER	-
S0	t, f, n	KEYWORD	-
S0	Whitespace	S0	(ignore)
S0	Any other character	ERROR	ERROR
STRING	Any character except “	STRING	Append to string token
STRING	\” (escape char)	STRING	Append escaped character
STRING	“	S0	STRING
NUMBER	Digit, . e E + -	NUMBER	Append to

			number token
NUMBER	**Any non-number char**	**S0**	**Validate and return NUMBER**
KEYWORD	**Characters to complete true**	**S0**	**BOOL: TRUE**
KEYWORD	**Characters to complete false**	**S0**	**BOOL: FALSE**
KEYWORD	**Characters to complete null**	**S0**	**NULL**
KEYWORD	**Any other sequence**	**ERROR**	**ERROR**

III. Code Explanation

1. **State Management:**
   - Each state of the DFA corresponds to a function or a logical block within the scanner.
   - The primary states include **S0**(start), **STRING, NUMBER,** and **KEYWORD**, as well as states for recognizing JSON syntax like the braces (**{, }**), brackets (**[,]**), **colons** (**:**) and commas (**,**)
2. **Token Definitions:**
   - Tokens are defined for different JSOn elements:
     - **LBRACE, RBRACE, LBRACKET, RBRACKET, COLON, COMMA**
     - **STRING, NUMBER**
     - **BOOL (true** and **false)**
     - **NULL**
3. **Transition Logic:**
   - The **S0** function acts as the main dispatcher, determining the type of the input character and moving to the appropriate state.
   - For example:
     - If a **“** is encountered, the function transitions to **STRING** state.
     - If a digit or **-** is encountered, it transitions to **NUMBER** state.
     - If **t, f,** or **n** is encountered, it transitions to **KEYWORD** state for recognizing **true, false** or **null**
     - Recognized symbols like **{, }, [,], :,** and **,** are directly tokenized in **S0.**

4. **String Handling:**
    - In the **STRING** state, the scanner reads until it finds a closing **“:**
      - It correctly handles escape sequences (eg **\, **)
      - Once the closing **“** is found, the string is validated and added as a **STRING** token
      - If the string is improperly terminated, and error is generated
5. **Number Handling:**
    - The **NUMBER** state reads digits and handles potential decimal points
      - It reads characters until a non-number character is found
      - The gathered sequence is then validated as a valid number using Python’s **float()** function.
      - If valid, the **NUMBER** token is added to the list.
      - If invalid, it generates an error indicating the malformed number
6. **Keyword Handling:**
    - In the **KEYWORD** state, the scanner reads characters to match potential keywords:
      - If the keyword matches **true, false,** or **null**, the respective token (**BOOL** or **NULL)** is added to the token list.
      - If the characters do not match a recognized keyword, an error is raised.
7. **Error Handling:**
    - The DFA is designed to handle errors gracefully:
      - If an unrecognized character is encountered in **S0**, it triggers an error state
      - For improperly formed strings or numbers, descriptive error messages are generated.
      - This helps users identify the issue in their JSON input.

IV. How the DFA scans and tokenizes input

1. **Input Stream Processing**:
    - The input JSON is provided as a string, and the scanner reads through it character by character
    - Each character is analyzed based on its context and the DFA state transitions accordingly
    - Tokens are built incrementally, with each state appending characters to the token value until the complete token is recognized.
2. **Token Collection;**
    - Each time a valid token is identified, it is added to the token list.
    - This list represents the sequence of tokens that the parser will use for further syntax analysis.

V. Reflection

Working on this JSOn scanner project provided a deep dive into the intricacies of lexical analysis and DFA-based tokenization.

What Worked Well

1. **DFA Design and Structure:**
    - The structured approach of using a DFA allowed for a clear definition of states and transitions. This made it easier to break down the problem into smaller, manageable parts.
    - Having a state for each major component of JSON provided a clear roadmap for implementation, ensuring that the JSOn specification was met.
2. **Error Handling:**
    - Implementing error handling was a significant success. The scanner could catch common mistakes like malformed strings, unexpected characters and invalid number formats.
    - Providing detailed error messages helped to pinpoint the exact issue in the input JSON, which is crucial for debugging and refining the input.
3. **Test cases and validation:**
    - Extensive testing with different JSON samples, including edge cases, ensured that the scanner could handle a variety of inputs. It performed well with nested structures, arrays and mixed data types.
    - The tokenization process proved to be efficient and accurate, providing the correct sequence of tokens that could be used for further parsing.

What I would do differently

If I were to redo the project, I would spend more time upfront creating a More detailed DFA diagram. This would have made the implementation easier.

VI. Conclusion

My understanding of formal languages and DFA was crucial in designing and implementing this project. Defining states, transitions and accepting conditions directly shaped how I approached the tokenization problem. JSOn is a widely used data format in API's, web services and configuration files. A scanner ensures that data is well-formed before it is processed, helping prevent errors and vulnerabilities. In the real

world, lexical analysis is critical in compilers and interpreters. This project highlights how the theoretical foundations of formal languages have practical applications in ensuring data integrity and system reliability.