Eliote Schuler

B00976713

CSCI 2115 Theory of Computer Science

Final project part 1

This scanner scans the .json language. There are 12 tokens in total:

- Left Braces: {

- Right Braces: }

- Left Brackets: [

- Right Brackets: ]

- Colon: :

- Comma: ,

- Strings: "key", "value"

- Integers: e.g., 123

- Floating-point numbers: e.g., 45.67

- Booleans: true, false

- Null: null

- EOF

A brace { indicates the beginning of an object, and is then closed by a closing brace }

A bracket [ indicates the beginning of an array, and is then closed by a closing bracket ]

A colon : separates keys and values from each other

A comma , separates key:value pairs as well as values in arrays

A period . separates a floating-point numbers between its integer part and decimal part

Literals:

A string is a combination of characters enclosed within double “ quotes “

An integer is a sequence of digits that form a whole number

A float is a number that has a decimal value

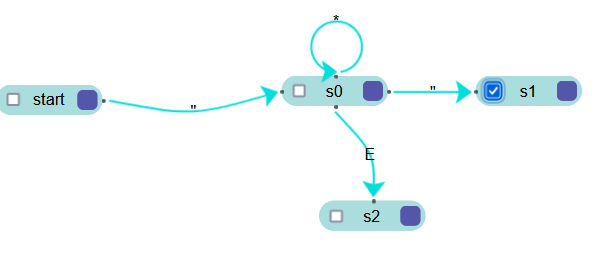
Keywords:

Boolean can be either true or false. The first character is lowercase in .json

# DFA:

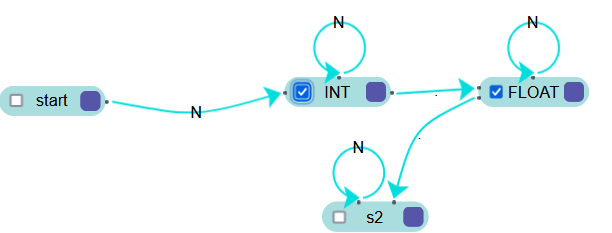
Because of how complicated this DFA is, I will break it down into sections. The start node on all these sub DFAs is the same as the start node in the main DFA, which will be shown at the end of this section.

String DFA:



If a “ is seen, the String DFA gets triggered. \* means any character whatsoever. E mean end of file/None. Essentially, if there isn’t a closing “, the DFA rejects the string

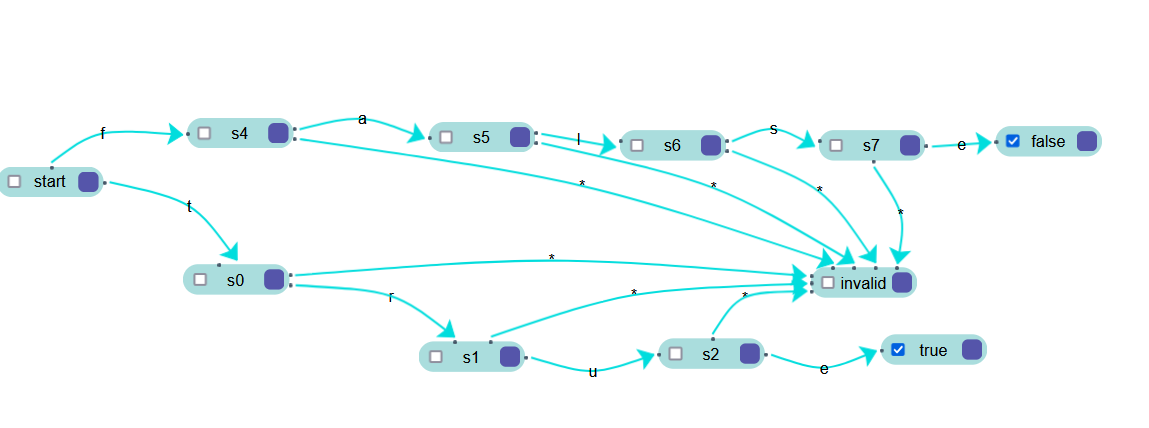
Int/Float DFA:



N means any digit. If 2 periods are seen, the number is now invalid, and is neither a float or an int.

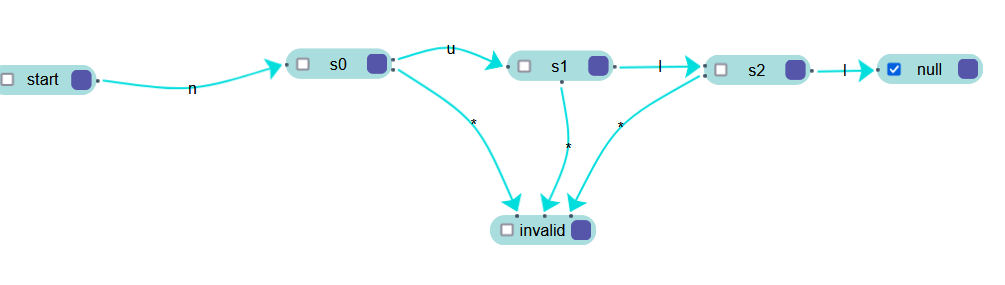
The state the string of numbers ends on determines the token type.

Boolean DFA:



This dfa is triggered when a t of f is seen. \* means any character apart from the character in the other transition node. (ex for s4, anything other than an a leads to an error)

Null DFA:



This dfa works just like the boolean dfa. It is triggered if an n is seen. \* means any character apart from the character in the other transition node.

All of the DFAs above are connected to this main DFA. I just didn’t have the space to add them all on one screen:

Legend:

\* means any character

N means any digit

B means the characters t or f

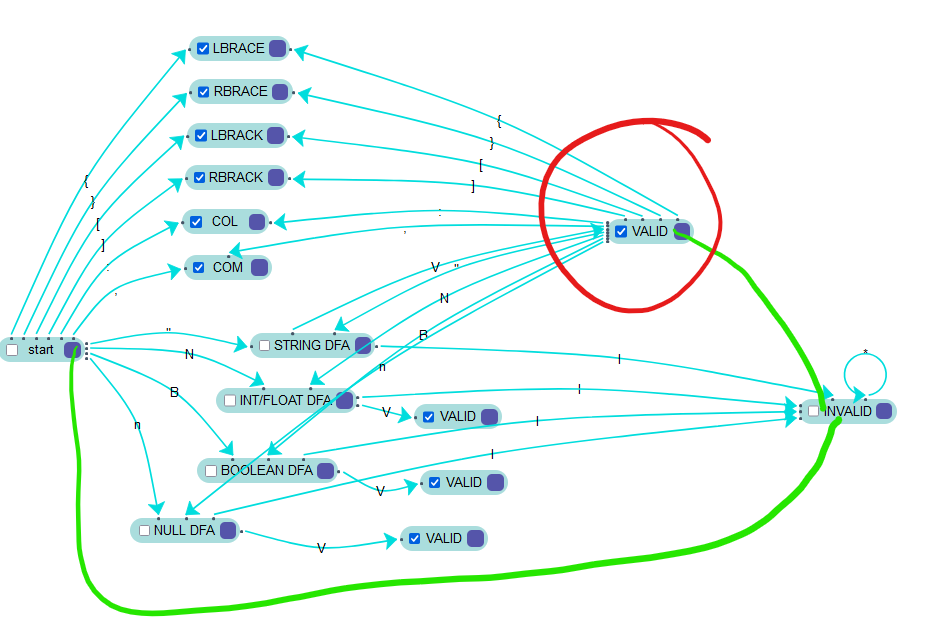
V or I means a valid or invalid state, respectively (explained more below\*)

Other characters, like n, represent that character itself.

This website allows only for 10 connections, so I had to add a green line for any character not listed above or shown in the diagram

Here is a clean image of the DFA. Every valid node does connect to every other node, just like how the VALID node in the red circle connects to everything, if it’s a valid character (I didn’t connect them all for legibility. Otherwise, if it's an invalid character, it takes the green line to the invalid state.

\*STRING DFA, INT/FLOAT DFA, BOOLEAN DFA and NULL DFA are the DFAs shown above, with the VALID state following them indicating that the string passed into the DFA was valid. If it was invalid, the INVALID node represents the node each DFA had where anything passed after reaching that node is recursive, meaning WHOLE string being parsed, in this case json files, is invalid.



# Code explanation:

The DFAs shown above are implemented in my Lexer class, specifically get\_next\_token(), which then also calls recognize\_bool(), recognize\_null(), numOrFloat() and recognize\_string(), all of which are part of the main DFA.

The tokenType class just stores the type of tokens that can be recognized. It gets called by \_\_repr\_\_() in the Token class.

The Token class is what returns what type of token a subset of the string (in this case a .json file) is. It gets called in the main method when we want to output the results to a text file.

The LexerError class is called when there is an invalid character in the String. It throws an error, tells the user where the error is and what it is.

The Lexer class is the meat of this program. It is what steps through the string and detects each token type.

\_\_init\_\_() initiates the process.

advance() is what iterates through each character in the string.

ignore\_whitespace() is self explanatory. It advances when a white space is seen.

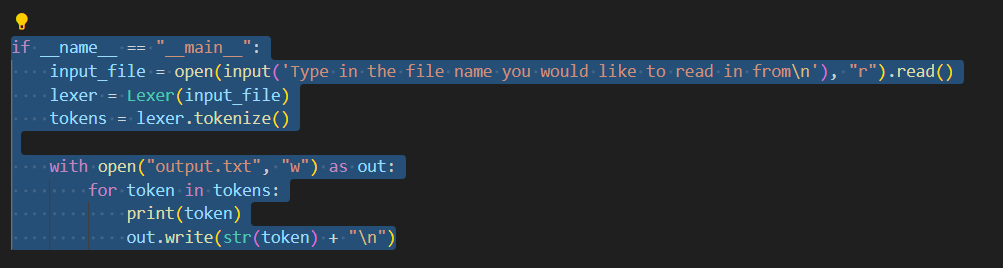
isValidChar() is used to see if a character is one of our valid characters, apart from numbers. This was made just because of a challenge I had to deal with when it came to dealing with numbers (more of that in the challenges section).

get\_next\_token() is the bulk of the class. It steps through the string using advance() and returns the type of token a character of group of characters is, or throws an error if its an unrecognized character.

tokenize() stores the results from get\_next\_token()

recognize\_null(), recognize\_bool(), numOrFloat(), and recognize\_string() recognize a null value, a boolean value, an int or a float number, and a string, respectively.

The following lines of code is what runs the whole code:



# Challenges:

One big challenge was that I had never written in python before apart from in 1109. Luckily, it’s close enough to java, so I could build upon my knowledge from csci1109 to write the code, alongside looking at the scanner example we had been provided in the tutorial. The logic is all my own, but I had to use w3school in some spots where the scanner example wasn’t relevant, or I didn’t know the specific syntax for something, such as how to read and write from a file.

Another challenge I had was being able to detect what a character directly after a number, without a whitespace was (ex: 34353.33222false). This is why I created the isValidChar() method, as it was part of my solution. I used it to detect if the number was done, and if i could break out of the while loop i was using to do that.