Eliote Schuler

B00976713

CSCI 2115 Theory of Computer Science

Final project part 2

Language Description

This Parser parses the tokenized .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

In order of appearance above, this is the form the are in after being tokenized by the Scanner, and is what the Parser takes in as an input:

- <lbrace, {>
- <rbrace, }>
- <lbrack, [>
- <rbrack,]>
- <col, :>
- <com, ,>
- <str, “value”>
- <int, 1234>
- <float, 1.234>
- <bool, true>, <bool, false>
- <null: null>

Below is the Grammar for .json

```
Value : dict
      | list
      | STRING
      | NUMBER
      | ”true” | ”false” | ”null”
list : ”[” value (”, ” value)* ”]”
dict : ”{” pair (”, ” pair)* ”}”
pair : STRING ” : ” value
```

The parse tree is constructed using indentation to visually represent the hierarchical structure of the JSON syntax, with non-terminals forming the parent nodes and terminals forming the leaves. It begins by checking what the current Value, as shown above in the grammar, the current token is. If a left bracket is seen for example, that indicates the beginning of a list, so the Parser outputs:

```
list
        [
```

As can be seen above, upon seeing a left bracket, a terminal value, the Parser outputs which non-terminal it belongs to, in this case a list, and then makes a new line, indents, and adds the left bracket. This is now the minimum indentation level, until the list is complete, which happens when a right bracket (<rbrack, [>) is seen. The parser then looks at the next token, which can be any value, since according to the Grammar, a list contains values. Let's say the next token is a left brace (<lbrace, {>), again, this is a terminal, so, just like the list that was seen before, the parser eats the left brace, and outputs this:

```
list
        [
        dict
                {
```

As the example above shows above, the parser outputs dict, and then indents, and adds the left brace, which is now the minimum indentation level. You can see that before the left brace, the minimum indentation level was at the left brace, as dict is printed at the same indentation level. Now let's say the Parser sees a String value (<str, "example">). Because the Parser is now within a dictionary, which only contains pairs, and commas <com, ,> between each pair, this means that this string is the key value in a pair, therefore, the parser outputs pair, indents, and displays the string value, followed by a colon, which is eaten to ensure that that is indeed the token following the string value, since that is the expected syntactic structure of a pair:

```
list
        [
        dict
                {
                pair
                        STRING: example
                        :
```

A pair is then completed with a value token, in this example we'll just use a number. To demonstrate how commas are shown, we will also have a second pair value in this dictionary as well, which has a float as its value:

```
list
        [
        dict
```

```
        {
        pair
                STRING: example
                :
                value
                        NUMBER: 123
        ,
        pair
                STRING: example
                :
                value
                        NUMBER: 123.456
        }
    ]
```

Because the value in this case is a number, which is a terminal, value is printed first, and then the number value is indented in the next line, in order to have an accurate parse tree representation of the syntax of the language. The way a syntax error such as a missing bracket or comma is handled will be explained in the error handling section later.

In short, when the parser encounters a terminal, such as a string or a number, it outputs the corresponding non-terminal it belongs to first, followed by a new line and an indented representation of the terminal itself. This indentation reflects the syntactic structure of the JSON language, as defined by the grammar. The parser keeps track of how many tokens have "affected" the current indentation level, which ensures that each branch of the tree is indented accurately until it reaches a terminal value. Once a terminal is reached, the parser reduces the indentation back to the previous level, indicating that the current non-terminal has been fully processed and the branch is complete. If there are nested non-terminals, the indentation level is reduced accordingly, step by step, until the entire structure is parsed. This is how, at a high level, the Parser outputs a visual representation of the Parse tree.

There was no modification needed to make the language non-ambiguous, as the JSON grammar isn't ambiguous by nature:

- Dictionaries are represented by { and }, with pair values within it, separated by commas.
- Lists are only represented by [and], and each item within it is separated by commas
- Pairs always follow the same Structure, "String : Value"

This means that any valid JSON input will produce a valid parse tree, since none of these non-terminals are ambiguous.

Code Explanation:

The Parser begins by reading in a .txt file, which has been outputted by a Scanner that has tokenized the .json file. Within the file, each token is separated by a new line. That text file is then sent to the Parser class, where the Tokens are then put into an array, or a list, where each index is a single token.

```
self.token_list = input_text.splitlines()
```

The pointer is initialized at 0, and will iterate through the list using the parse() method within the Parser class.

The parse method calls another method, called determineType(). This method is responsible for taking in the token at the current index, and recognizing which token it is, so that it can then return which type of Grammar it is in the JSON grammar. This gets returned into a list called parseList. Specifically, determineType() returns a list of Objects of Type Grammar.

```
def parse(self):
    parseList = [self.determineType()]
    outputList = self.output(parseList)
    if self.hasError:
        for i in range(len(self.eatError)):
            outputList.append(self.eatError[i])
    return outputList

    PAIR = "PAIR"
    DICT = "DICT"
    BOOL = "BOOL"
    LBRACE = 'LBRACE' #'{'
    RBRACE = 'RBRACE' #'}'
    LBRACK = 'LBRACK' #'['
    RBRACK = 'RBRACK' #']'
    COL = 'COL' #':'
    COM = 'COM' #','
```

Grammar contains a label for the Token, which is another object of type GrammarType. GrammarType essentially stores all the different possible grammar items described at the beginning of the language description section. Furthermore, the Grammar also contains the value of a Token, if it has one. The cases where this could happen is where the Token is a terminal, for example. The Grammar class also has a __repr__ method, which essentially works like the toStringMethod() in Java.

```
def determineType(self):

    #skip if it's an empty
    if self.current_token == "":
        return

    if "str" in self.current_token[:4]:
        return self.parse_String()

    if "int" in self.current_token[:4] or "float" in self.current_token[:6]:
        return self.parse_Num()

    if "null" in self.current_token[:5]:
        return self.parse_Null()

    if "lbrace" in self.current_token[:7]:
        return self.parse_Dict()

    if "bool" in self.current_token[:5]:
        return self.parse_Bool()

    if "lbrack" in self.current_token[:7]:
        return self.parse_List()

    else:
        errorValue = f"invalid token type: {self.current_token}"
        self.eatError.append(errorValue)
        self.errorLocation.append(self.position)
        self.hasError = True
```

Now that we understand what determineType() returns, let's take a closer look at how it works.

The parser follows a recursive descent algorithm, where each parsing method corresponds to a non-terminal in the grammar and may call itself or other methods to handle nested cases. determineType() is what initiates this "recursive descent" by calling said methods.

Shown to the left is how it "detects" which kind of Grammar the current Token is. Because of how different every Grammar Type is formatted, a different way to Parse each token was required, and this method is responsible for calling the correct parsing method depending on the current token it is seeing.

Each line in the input file specified the type of token it was, as well as its value. This made it easy for this section of code to determine which parse method it had to call in order to Parse that value and possibly values coming after it, if it was a non-terminal.

Below is a simple example of what an input file could look like. After the first triangle bracket, the type of token is described, and after the comma, the value of that token is listed. The value is what the Parser will end up outputting in the ParseTree at the end.

```
<lbrace, {>
<str, "name">
<col, :>
<str, "jim bob">
<com, ,>
<str, "age">
<col, :>
<int, 22>
<com, ,>
<str, "height">
<col, :>
<float, 6.2>
<com, ,>
<str, "criminal">
<col, :>
<bool, false>
<com, ,>
<str, "father">
<col, :>
<bool, true>
<com, ,>
<str, "income">
<col, :>
<null, null>
<rbrace, }>
```

If “str” was seen within a token, then the self.parse_String() method is called. This method is pretty simple, as it does not need to recursively call itself, or other functions, since a String is a terminal value. The method is shown below. In order to keep things consistent between methods, I made every single one of these parsing methods return items in lists, even if there is a single item in the list. This was the easiest way I could find to deal with recursive methods, such as parse_Dict(), since there was no telling how large the return value could possibly be.

```
def parse_String(self):
    returnValue = []
    returnValue.append(Grammar(GrammarType.STRING, self.current_token[6:].strip('">'), is_terminal: True))
    return returnValue
```

All terminal value parsing methods worked in this manner. They are usually only called after a list, dictionary or pair parse methods are called. Let’s take a look at how each of them work in closer detail, starting with parse_list().

```
def parse_List(self):
    self.eat("[")
    returnValue = []
    returnValue.append(Grammar(GrammarType.LIST, value: "list", is_terminal: False))
    returnValue.append(Grammar(GrammarType.LBRACK, value: "[", is_terminal: True))
    token = self.determineType()
    returnValue.append(Grammar(GrammarType.VALUE, value: "value", is_terminal: False))
    returnValue.append(token)
    self.advance()

    while "rbrack" not in self.current_token[:7] and self.position < len(self.token_list)-2:
        if "com" in self.current_token[:4]:
            returnValue.append(Grammar(GrammarType.COM, value: ",", is_terminal: True))
            self.eat(",")
            if "lbrack" in self.current_token[:7]:
                token = self.parse_List()
            elif "lbrace" in self.current_token[:7]:
                token = self.determineType()
            else:
                token = self.determineType()
                self.advance()
            returnValue.append(Grammar(GrammarType.VALUE, value: "value", is_terminal: False))
            returnValue.extend(token)
        else:
            self.eat(",")
    self.eat("]")
    returnValue.append(Grammar(GrammarType.RBRACK, value: "]", is_terminal: True))
    return returnValue
```

When a left brace is detected in determineType()it calls the parse_list method. The method immediately “eats” the left bracket. The eat method has two functions. One of them is used for syntactic error checking, which will happen near the end of this method. The reason it’s being used here is to advance the pointer to the next token in the list. This is done because we are now in “list parsing mode”, and we need to see what the values within the list

are in order to be able to parse the list in full. The GrammarType list is first to be appended in the list, followed by the left bracket grammar type. These are added first since they must appear first within the Parse tree. Next, determineType is called again to detect what type of token the first value in the list is. It is then appended to the return list of the parse_List() method, and advance() is called to move the pointer to the next token. The while loop then takes care of the rest before returning the list.
The while loop is used for the Kleene-* property of the list, since there can be one or more items in any list. This while loop continues until either a right bracket is detected in the current token, or the end of the input text list is reached. Each value in a list must be separated by a comma, so the very first thing the while loop checks is if the current token is indeed a comma. If it is not, the eat() method is called expecting a comma, which then throws an error due to the syntactic error of a missing comma. Error handling will be explained in greater detail later on in this document. If a left bracket is seen, this indicates that another list is within the initial list that was detected, so parse_List() is recursively called. Conversely, if a left brace is detected, it means a dictionary has been detected, meaning the program needs to switch to the “dictionary parsing mode”, so parse_dict() is called. Since we have taken care of nested list and dictionary cases, the else block is left is to parse whatever token that is recognized by determineType(). The Grammar Type value is then appended to the return list, followed by whatever value(s) the token has. Once the while loop breaks, the eat() method is called expecting a right bracket. If the current token is not a right bracket, an error is thrown, otherwise the pointer is moved forward, the right bracket is added to the return list, and that list is then returned back to the parse() method.

```
def parse_Dict(self):
    self.eat("{")
    returnValue = []
    returnValue.append(Grammar(GrammarType.DICT, value: "dict", is_terminal: False))
    returnValue.append(Grammar(GrammarType.LBRACE, value: "{", is_terminal: True))
    toString = self.determineType()
    #getting here and rbrace is the current token, and then self.advance() is called
    #check if this throws an error for mismatched braces
    if "str" in self.current_token:
        getpairs = self.parse_Pair()
        returnValue.append(getpairs)
        if "rbrace" in self.current_token[:7]:
            self.eat("}")
            returnValue.append(Grammar(GrammarType.RBRACE, value: "}", is_terminal: True))
            return returnValue
    self.advance()
    while "com" in self.current_token[:4] and "}" not in self.current_token and self.position < len(self.token_list)-2:
        returnValue.append(Grammar(GrammarType.COM, value: ",", is_terminal: True))
        self.eat(",")
        # toString = self.determineType()
        if "str" in self.current_token:
            getpairs = self.parse_Pair()
            returnValue.append(getpairs)
            self.advance()
    self.eat("}")
    returnValue.append(Grammar(GrammarType.RBRACE, value: "}", is_terminal: True))
    return returnValue
```

For the sake of Brevity, the parse_Dict() method works similarly to the parse_Dict() method, but is only triggered when a left brace { is detected. Since a dictionary only contains pairs, it will call parse_pair, which does most of the work for parse_Dict. As said before, parse_Dict() works similarly to parse_List(), as there is a while loop that exists for the Kleene-* property of dictionaries, as they may have more than one Pair of items. Just like parse_List(), it ensures a comma separates each value, and ensures that the dictionary is closed with a right brace.

```
def parse_Pair(self):
    returnValue = []
    returnValue.append(Grammar(GrammarType.PAIR, value: "pair", is_terminal: False))

    #make sure the first thing in the pair is a string
    if "str" in self.current_token:
        toString = self.parse_PairString()
        returnValue.append(toString)
        self.advance()
        #make sure the next token is a colon. then, advance, and find the value of that
        # <col, :> check index 6
        if "col" in self.current_token[:4]:
            returnValue.append(Grammar(GrammarType.COL, value: ":", is_terminal: True))
            self.eat(":")
```

parse_Pair() is exclusively called by parse_Dict(), and works similarly to

parse_List() and parse_Dict(). In this case, it checks syntax by ensuring that the key value is a String. It then expects a colon, followed by any value, as per the JSON grammar. This method is also capable of calling parse_List() or parse_Dict() if a left bracket or brace is detected.

Due to how all the methods can call each other, and in some cases even themselves, this parsing method follows a recursive descent algorithm; A list might detect a list, which might detect a dictionary, which detects a pair, which might contain another list as its value, and so on.

Finally, once all the tokens in the input text have been read, parse() calls output(), which is responsible for actually outputting the list created by determineType() to a text file. Due to the recursive nature of detecting all the grammar types and their values, the output method begins by calling a method called clean(). This is because since parse_dict(), parse_list() and parse_pair() may return a list of lists since they can all call each other and themselves, the list that is being sent to output() might have nested lists. In short, the clean() method takes that list, and extracts all the elements in all possible nested lists and puts them in a single “clean” list, meaning there are no more nested lists.

Output then takes this “clean” list, and begins the Parse tree output. It looks at what type of grammar the current token is, calls the __repr__ method from Grammar that was mentioned earlier, adds the appropriate amount of indents for that token, followed by its value. The result is a list of values at each index, all of which having an appropriate indentation level. This list gets sent back to parse(), which returns this list to the main method, where it is then iterated through, and each element is printed onto a new line in an output text file, creating a clean, accurate Parse tree.
In short, determineType() is responsible for detecting the token and ensuring the input file is syntactically correct. The output method then indents the values accordingly into a list to represent the JSON file in a Parse Tree. This list is then sent back to the main method, which prints the resulting list to a text file, adding each element of the list to a new line, which produces a clean Parse Tree.

Error Handling:

Error handling is mainly done by the eat method, which is called by parse_List(), parse_Dict(), and parse_Pair() to ensure that there are no missing commas, brackets, braces or colons. This ensures that everything is syntactically correct and there isn’t anything missing, or that there are no mismatched brackets or braces.

```
def eat(self, token_type):
    errorMessage = ""
    token = self.current_token.split()
    if token_type in token[1]:
        self.advance()
    else:
        errorMessage = f"Expected token '{token_type}' got '{token[1].strip('>')}' at line {self.position + 1} in input file"
        print(f"Expected token '{token_type}' got '{token[1].strip('>')}' at line {self.position + 1} in input file")
        self.eatError.append(errorMessage)
        self.errorLocation.append(self.position)
        self.hasError = True
        self.advance()
```

To briefly explain the code block shown above, when eat is called, the expected token type is compared to what the current token type actually is. If it is the same, the pointer is advanced, and the eat method is exited. Otherwise, an error message is created, and appended to the error messages list. The hasError boolean is set to True, which the parse() method then checks after indentation is done by the output() method. If it is true, the parse() method then appends these error messages at the end of the output file. This ensures that the program doesn't crash when there's an error. Instead, it handles it, and recovers by continuing parsing the rest of the input file.
Below are a few examples of how errors are outputted by the program.

Example 1: missing bracket

```
                ,
                value
                    BOOL: false
                ]
    ]
Expected token ']' got '}'  at line 36 in input file
```

In this example, there was no closing bracket in the list. When the while loop ended in parse_list(), eat("]") was called to ensure that the closing bracket existed. Instead, a right brace } was detected instead. Therefore, the error message in the left example was added to the output file, where the user is told what is wrong, as well as *where* in the input file the error occurred. The output file of this example can be seen in the main directory under "missingbracketoutput.txt". There, you can see that the Parser outputs a correct Parse Tree despite the error.

Example 2: missing comma

```
        ]
Expected token ',' got 'false'  at line 9 in input file
```

In this example, there was a missing comma to separate values within a list. The program successfully recovered and continued parsing the rest of the input file. The output file of this example can be seen in the main directory under "missingcommaoutput.txt". There, you can see that the Parser outputs a correct Parse Tree despite the error.

Example 3: Missing closing brace for a dictionary within a list:

```
Expected token '}' got '['  at line 24 in input file
Expected token ',' got '"test2"'  at line 25 in input file
```

In this example, a dictionary was missing a closing brace within a list, which caused a cascading error; Because there was no closing brace, the dictionary had a syntactic error, which was handled, but this meant that when this dictionary was returned to the parse_List() method, and it looked at the next token, it was expecting a comma, but it instead saw the next list value, which was test 2. Below is the input file where the error occurred, and to the right is the full output from the parser of that input file.

```
<lbrack, [>
<str, "test">
<com, ,>
<int, 64>
<com, ,>
<bool, true>
<com, ,>
<null, null>
<com, ,>
<bool, false>
<com, ,>
<lbrace, {>
<str, "name">
<col, :>
<str, "jim">
<com, ,>
<str, "species">
<col, :>
<str, "dog">
<com, ,>
<str, "age">
<col, :>
<int, 3>
<lbrack, [>
<str, "test2">
<com, ,>
<int, 68>
<com, ,>
<bool, true>
<com, ,>
<null, null>
<com, ,>
<bool, false>
<rbrack, ]>
<rbrack, ]>
```

```
list
    [
    value
        STRING: test
    ,
    value
        NUMBER: 64
    ,
    value
        BOOL: true
    ,
    value
        NULL
    ,
    value
        BOOL: false
    ,
    value
        dict
            {
            pair
                STRING: name
                :
                value
                STRING: jim
            ,
            pair
                STRING: species
                :
                value
                STRING: dog
            ,
            pair
                STRING: age
                :
                value
                    NUMBER: 3
            }
    ,
            value
                NUMBER: 68
    ,
            value
                BOOL: true
    ,
            value
                NULL
    ,
            value
                BOOL: false
    ]
Expected token '}' got '['  at line 24 in input file
Expected token ',' got '"test2"'  at line 25 in input file
```

If the text is too small, both files can be viewed in the main directory, under missingbrace.txt and missingbraceoutput.txt As can be seen from the output in the right picture, even though this missing brace caused a cascading error, the program *still* recovered and managed to output a correct Parse tree.

In short, the parser ensures that syntactic errors are detected and reported without crashing, and allows it to recover and produce a correct parse tree representation whenever possible.