Eliote Schuler

B00976713

CSCI 2115 Theory of Computer Science

# Final project part 3

## Language Description

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

 list : ”[” value (”,” value)* ”]”

 dict : ”{” pair (”,” pair)* ”}”

 pair : STRING ” : ” value
```

`Grammar for JSON language`

The abstract Syntax tree is built very similarly to how the Parse tree in part 2, but the output method doesn’t include things like value, list, dict or pair as well as terminal labels. This is because the abstract Syntax tree doesn’t need to display non-terminals, or terminal labels. I did this by removing the if blocks that handled non-terminals and terminal labels in my output method, as well as adding a prune method that removes said Grammar types from the list returned by detectType() to ensure there were no errors when actually building the abstract syntax tree. The outputted abstract syntax tree is a correctly indented, non-terminal and terminal-label free .txt file.

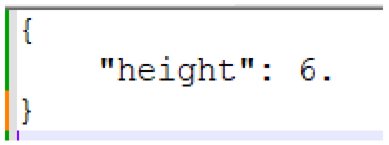
The program is able to detect 7 different error types:

- Error 1: Invalid decimal errors
    - Decimal numbers must have digits on both sides of the decimal point
- Error 2: Empty Key
    - The key of a dictionary should not be empty
- Error 3: Invalid numbers
    - Numeric values follow proper formatting and do not include leading zeros before a decimal place.
    - Numeric values have no leading + sign.
- Error 4: Reserved keywords as Dictionary key
    - true and false cannot be a key, as they are reserved for boolean values, however True and False can be, as those are not reserved keywords.
- Error 5: No duplicate keys in a single dictionary
    - Each key in a dictionary must be unique.

- Error 6: Consistent data types as list elements
  - All elements in a list must be the same datatype, i.e: a string followed by a number in a list would be invalid
- Error 7: Reserved keywords as Strings
  - Reserved keywords, true and false, can not be used as a string value.

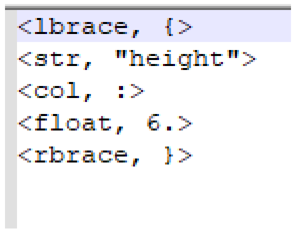
Here are all the original inputs used to test each error. On the left is the original JSON text, while on the right is the same text file in tokenized form:

Error 1 JSON format:



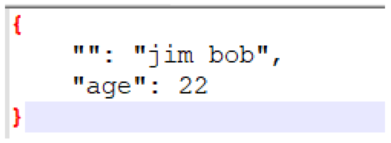
```
{
    "height": 6.
}
```

Error 1 Tokenized:



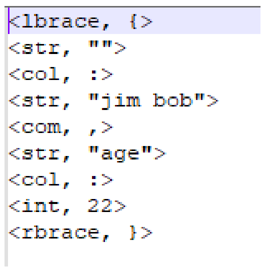
```
<lbrace, {>
<str, "height">
<col, :>
<float, 6.>
<rbrace, }>
```

Error 2 JSON format:



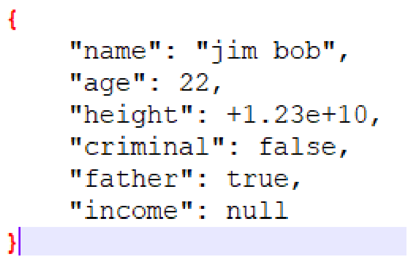
```
{
    "": "jim bob",
    "age": 22
}
```

Error 2 Tokenized:



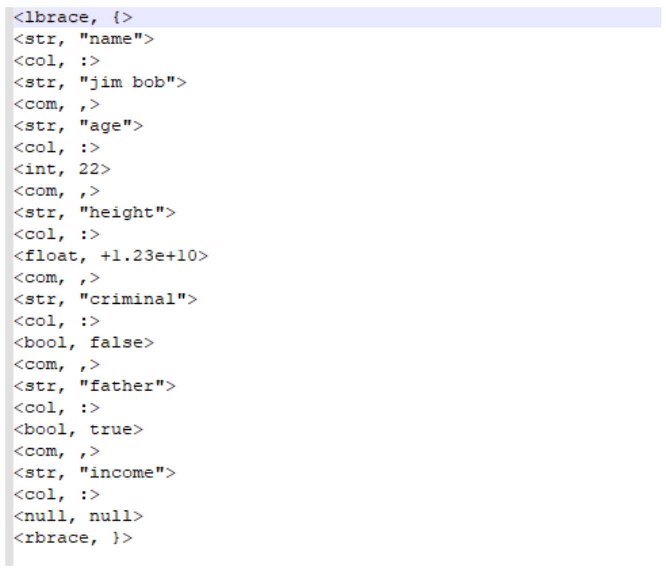
```
<lbrace, {>
<str, "">
<col, :>
<str, "jim bob">
<com, ,>
<str, "age">
<col, :>
<int, 22>
<rbrace, }>
```

Error 3 JSON format:



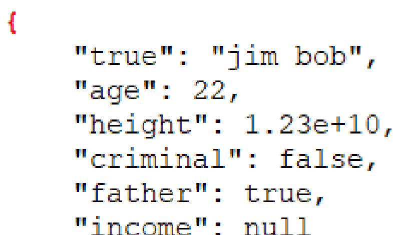
```
{
    "name": "jim bob",
    "age": 22,
    "height": +1.23e+10,
    "criminal": false,
    "father": true,
    "income": null
}
```

Error 3 Tokenized:



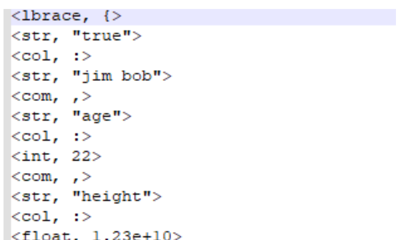
```
<lbrace, {>
<str, "name">
<col, :>
<str, "jim bob">
<com, ,>
<str, "age">
<col, :>
<int, 22>
<com, ,>
<str, "height">
<col, :>
<float, +1.23e+10>
<com, ,>
<str, "criminal">
<col, :>
<bool, false>
<com, ,>
<str, "father">
<col, :>
<bool, true>
<com, ,>
<str, "income">
<col, :>
<null, null>
<rbrace, }>
```

Error 4 JSON format:



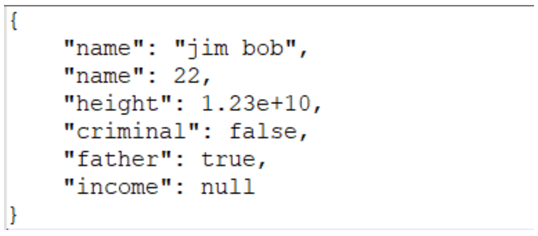
```
{
    "true": "jim bob",
    "age": 22,
    "height": 1.23e+10,
    "criminal": false,
    "father": true,
    "income": null
```

Error 4 Tokenized:



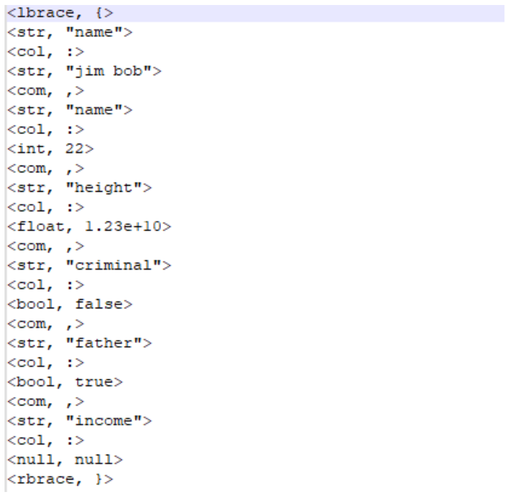
```
<lbrace, {>
<str, "true">
<col, :>
<str, "jim bob">
<com, ,>
<str, "age">
<col, :>
<int, 22>
<com, ,>
<str, "height">
<col, :>
<float, 1.23e+10>
```

Error 5 JSON format:



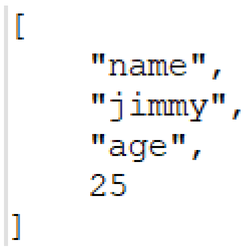
```
{
    "name": "jim bob",
    "name": 22,
    "height": 1.23e+10,
    "criminal": false,
    "father": true,
    "income": null
}
```

Error 5 Tokenized:



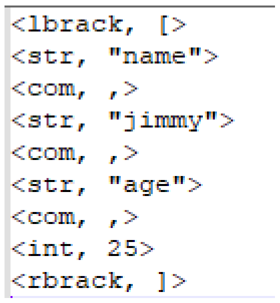
```
<lbrace, {>
<str, "name">
<col, :>
<str, "jim bob">
<com, ,>
<str, "name">
<col, :>
<int, 22>
<com, ,>
<str, "height">
<col, :>
<float, 1.23e+10>
<com, ,>
<str, "criminal">
<col, :>
<bool, false>
<com, ,>
<str, "father">
<col, :>
<bool, true>
<com, ,>
<str, "income">
<col, :>
<null, null>
<rbrace, }>
```

Error 6 JSON format:



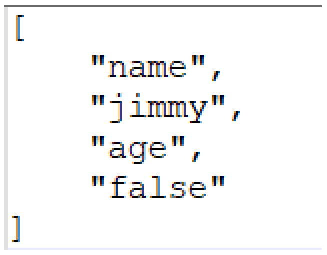
```
[
     "name",
     "jimmy",
     "age",
     25
]
```

Error 6 Tokenized:



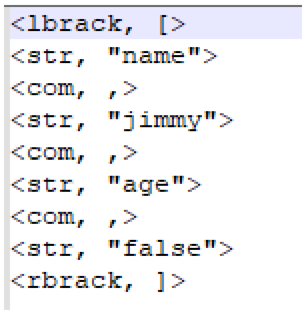
```
<lbrack, [>
<str, "name">
<com, ,>
<str, "jimmy">
<com, ,>
<str, "age">
<com, ,>
<int, 25>
<rbrack, ]>
```

Error 7 JSON format:



```
[
     "name",
     "jimmy",
     "age",
     "false"
]
```

Error 7 Tokenized:



```
<lbrack, [>
<str, "name">
<com, ,>
<str, "jimmy">
<com, ,>
<str, "age">
<com, ,>
<str, "false">
<rbrack, ]>
```

## Attribute Grammars

```
value -> dict {$0.valid if $1.valid} {$0.type = dict}
value -> list {$0.valid if $1.valid, } {$0.type = list}
value -> STRING{$0.valid if $1 not equal to “true”, “false”} {$0.type = STRING}
value -> “true”{$0.valid if $1 equal to “true”}{$0.type=”true”}
value -> “false”{$0.valid if $1 equal to “true”}{$0.type=”false”}
value -> “null”{$0.valid if $1 equal to “true”}{$0.type=”null”}
value -> NUMBER{$0.valid if $1 first_char is numeric or equal to -
                          and first_char not equal to 0 if second_char not equal to .
                          and last_char not equal to .}{$0.type = NUMBER}
pair -> STRING “:” Value {$0.valid if $1 is equal to STRING and $1 not equal to “”, “true”, “false”
        and $2.valid, $0.key equal to $1.value}{$0.type = pair}
list -> “[“ value (“,” value)* “]” {$0.valid if $1.valid and if $n.valid if exists
                               $n not valid if $n.type not equal to $1.type}{$0.type = list}
dict -> “{“ pair (“,” pair)* “}” {$0.valid if $1.valid and if $n.valid if exists
                               $n not valid if $n.key not unique}
```

$n is used to handle kleene-* values, since we can’t tell how many values there may be due to the properties of kleene-*

For $n, the semantic checks for $n set it back to not valid if they do not meet the semantic rules.

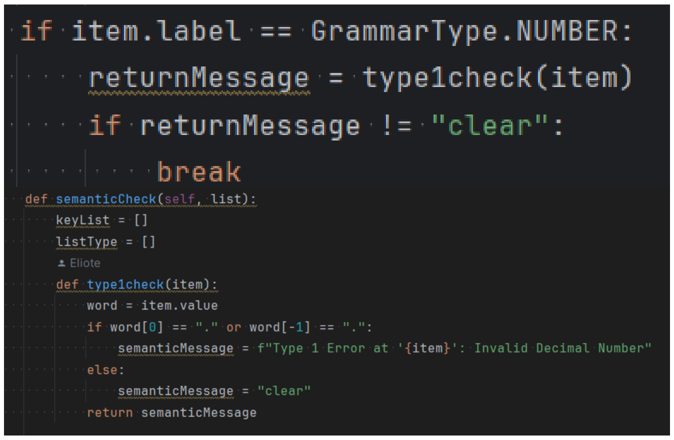
Each Attribute grammars enforce the semantic rules for the JSON language

- value is valid if the values it is representing are valid.
- pair is valid if the key is not a reserved keyword or empty **(type 2 error, type 4 error)**, and if the value is valid, which is recursively checked by the value Attribute grammars.
- list is valid if the values within it are valid. If there is more than one value, hence the $n, they are valid if their value is valid AND they are the same data type as the first value, $1 **(type 6 error)**.
- dictionaries are valid if the pairs are valid, and the key values of said pairs are all unique, meaning no pairs have the same keys **(type 5 error)**.
- numbers are valid if the first character is numeric or it is a - sign (inherently, this checks for **type 3 errors**). However, if that numeric value is a 0, it is invalid unless it is directly followed by a decimal “.” **(type 1 error)**.
- Strings are valid if they are not reserved keywords like true or false (case sensitive) **(type 7 error)**

## Code explanation

This Parser still follows the same recursive descent algorithm as my Parser from part 2. However, this Parser then checks for semantic errors *after* having Parsed each token, before outputting the resulting Abstract syntax tree to a text file.
At a high level, the program checks every token in the input file, and returns the grammarType of each token to a list. This is done when the parse() method calls recognizeType(). Once this list is returned, the list is then "cleaned" by the clean() method, because there may be nested lists within the list due to the recursive nature of Parsing. Next, the pruneList() method is called to remove all non-terminal values as well as terminal labels, since they aren't needed when building an abstract syntax tree.

Finally, semanticCheck() is then called, to check for all possible error types. At it's core, semanticCheck() has a while loop that looks at the current item's label, and calls the appropriate error checking methods for that label.



```
if item.label == GrammarType.NUMBER:
    returnMessage = type1check(item)
    if returnMessage != "clear":
        break
```

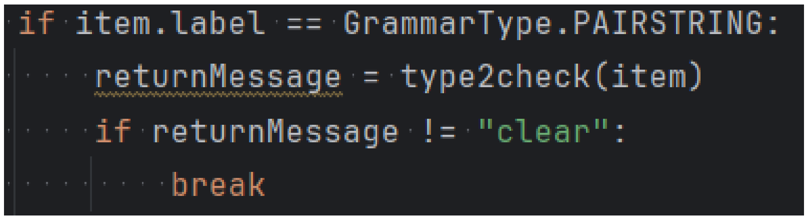
```
def semanticCheck(self, list):
    keyList = []
    listType = []
    Eliote
    def type1check(item):
        word = item.value
        if word[0] == "." or word[-1] == ".":
            semanticMessage = f"Type 1 Error at '{item}': Invalid Decimal Number"
        else:
            semanticMessage = "clear"
        return semanticMessage
```

Let's start by looking at how type 1 errors are detected.
When the item's label is a number, the type1check() method is called. This method ensures that there is no "." at the beginning or the end of a number. If there is, an error message is returned. Otherwise, the message "clear" is returned.

## An aside on what happens if an error is detected:

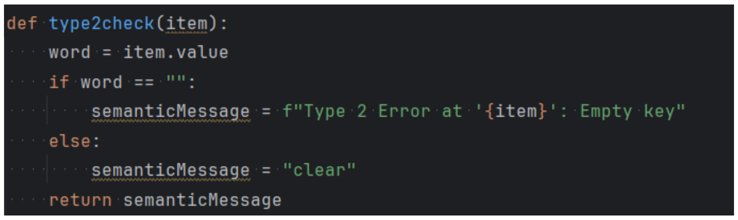
All the  typeNcheck() methods work in the way type1check() returns an error. If there is an error, a message related to that error is returned, and if there isn't "clear" is returned. This is because if the return message is not "clear" at any point, the while loop in semanticCheck() breaks early, and the error message is outputted to the text file instead of building an abstract syntax tree. This occurs because the semanticCheck() method returns the string containing the error message, or "clear" to the output() method. If the message isn't "clear", the output() method returns that error to the parse() method instead of building an abstract syntax tree. That message is then the contents of the output file created by the program.

Going back to how each Error type is detected, shown below is how a type 2 error is handled:



```python
if item.label == GrammarType.PAIRSTRING:
    returnMessage = type2check(item)
    if returnMessage != "clear":
        break
```

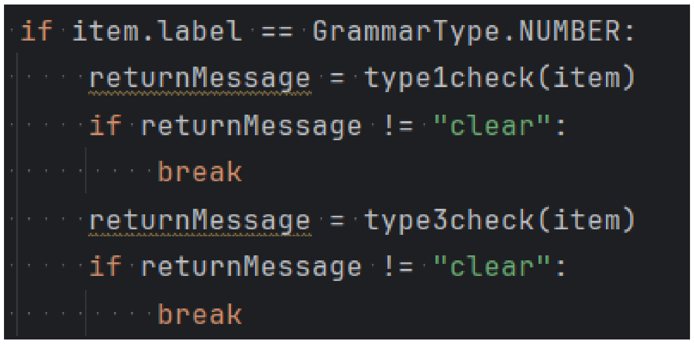
If the item's label is of type PAIRSTRING, meaning it is a String for a key value of a pair, the type2check() method is called.



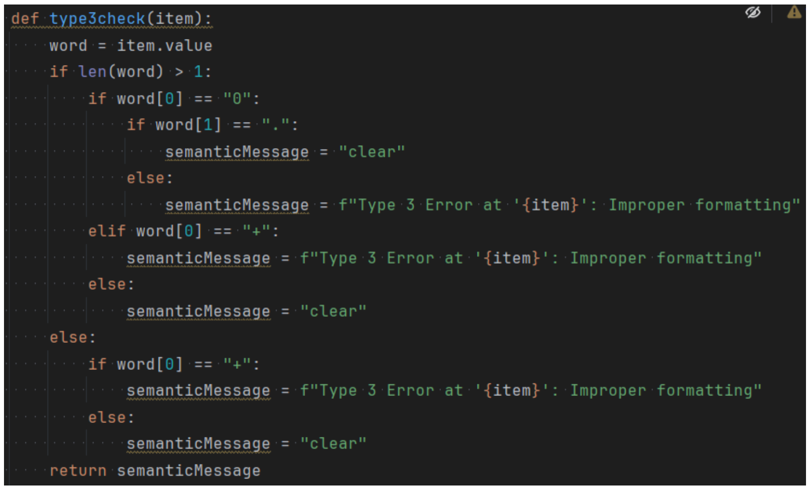
```python
def type2check(item):
    word = item.value
    if word == "":
        semanticMessage = f"Type 2 Error at '{item}': Empty key"
    else:
        semanticMessage = "clear"
    return semanticMessage
```

This method then checks and ensures that the value of that string is not empty. If it is, it returns an error message, otherwise it returns clear.

A type 3 error is checked when a number is detected, where type3check() is then called:



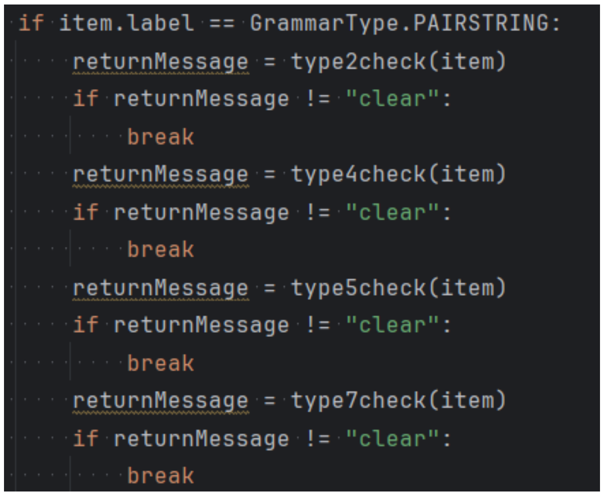
```python
if item.label == GrammarType.NUMBER:
    returnMessage = type1check(item)
    if returnMessage != "clear":
        break
    returnMessage = type3check(item)
    if returnMessage != "clear":
        break
```



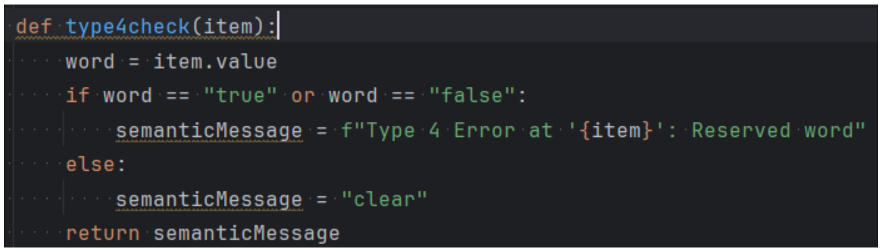
```python
def type3check(item):
    word = item.value
    if len(word) > 1:
        if word[0] == "0":
            if word[1] == ".":
                semanticMessage = "clear"
            else:
                semanticMessage = f"Type 3 Error at '{item}': Improper formatting"
        elif word[0] == "+":
            semanticMessage = f"Type 3 Error at '{item}': Improper formatting"
        else:
            semanticMessage = "clear"
    else:
        if word[0] == "+":
            semanticMessage = f"Type 3 Error at '{item}': Improper formatting"
        else:
            semanticMessage = "clear"
    return semanticMessage
```

If the length of the number is greater than 1, then the method ensures that if there is a 0, it is immediately followed by a ".". It also checks if there is a leading + sign at the beginning of the number. If there is a leading 0 without a decimal point right after it, or a leading + sign, then an error is thrown. The else block checks only for a leading + sign, since a 0 on its own is ok to have. If there is no error, the method returns “clear”

A type 4 error is checked when a key value String is detected:



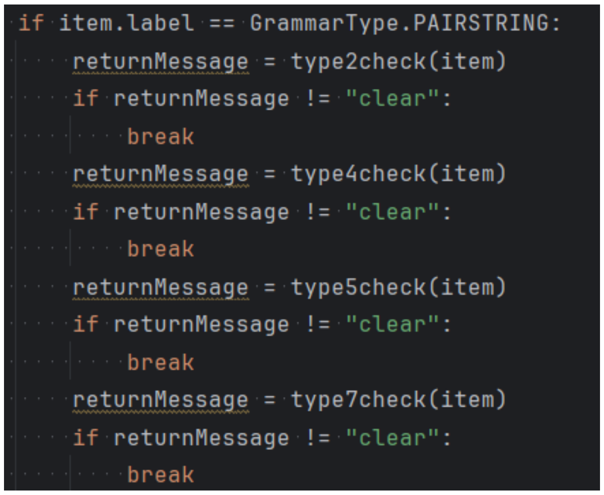
```python
if item.label == GrammarType.PAIRSTRING:
    returnMessage = type2check(item)
    if returnMessage != "clear":
        break
    returnMessage = type4check(item)
    if returnMessage != "clear":
        break
    returnMessage = type5check(item)
    if returnMessage != "clear":
        break
    returnMessage = type7check(item)
    if returnMessage != "clear":
        break
```



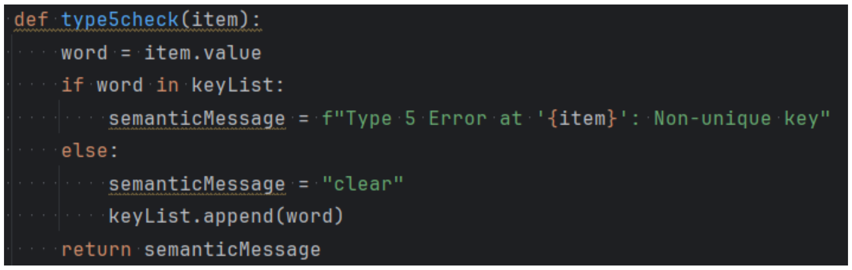
```python
def type4check(item):
    word = item.value
    if word == "true" or word == "false":
        semanticMessage = f"Type 4 Error at '{item}': Reserved word"
    else:
        semanticMessage = "clear"
    return semanticMessage
```

type4check() looks at the item’s value, and ensures it is not equal to “true” or “false”. If it is, it returns an error message, otherwise, it returns “clear”.

Type 5 error is checked when a key value String is detected:



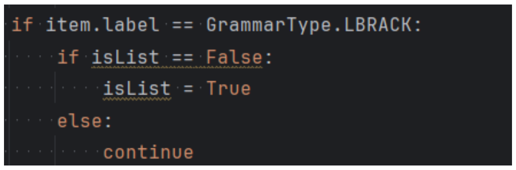
```python
if item.label == GrammarType.PAIRSTRING:
    returnMessage = type2check(item)
    if returnMessage != "clear":
        break
    returnMessage = type4check(item)
    if returnMessage != "clear":
        break
    returnMessage = type5check(item)
    if returnMessage != "clear":
        break
    returnMessage = type7check(item)
    if returnMessage != "clear":
        break
```



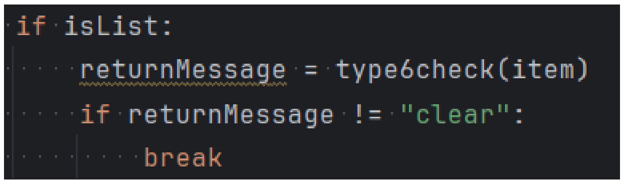
```python
def type5check(item):
    word = item.value
    if word in keyList:
        semanticMessage = f"Type 5 Error at '{item}': Non-unique key"
    else:
        semanticMessage = "clear"
        keyList.append(word)
    return semanticMessage
```

type5check() ensures that the key is unique in the dictionary. If it isn't it returns an error message, otherwise, the "clear" message is returned, and that key's value is appended to the list of used key values. This list is then iterated through when the next key value is detected, to ensure it is not using an already used key.

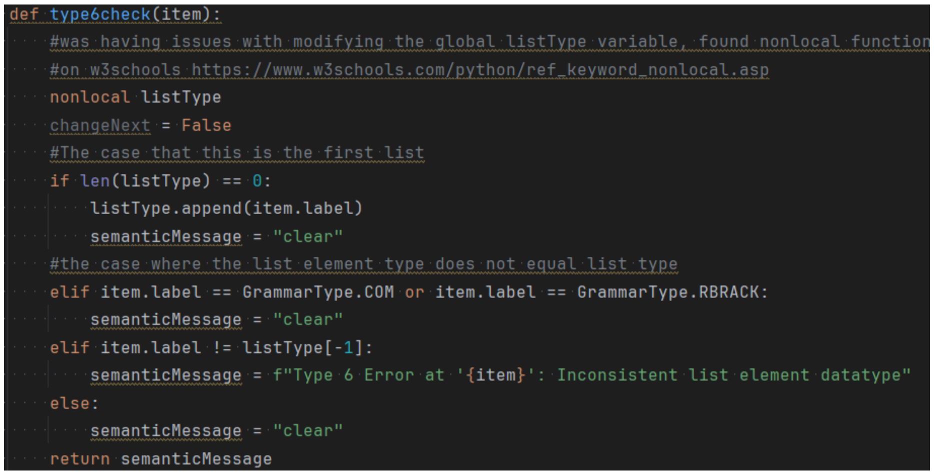
Type 6 errors are checked when the isList boolean is set to true, which occurs when a left bracket is detected. When isList is true, the type6check() method is called.



```python
if item.label == GrammarType.LBRACK:
    if isList == False:
        isList = True
else:
    continue
```

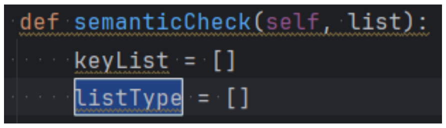


```python
if isList:
    returnMessage = type6check(item)
    if returnMessage != "clear":
        break
```



```
def type6check(item):
    #was having issues with modifying the global listType variable, found nonlocal function
    #on w3schools https://www.w3schools.com/python/ref_keyword_nonlocal.asp
    nonlocal listType
    changeNext = False
    #The case that this is the first list
    if len(listType) == 0:
        listType.append(item.label)
        semanticMessage = "clear"
    #the case where the list element type does not equal list type
    elif item.label == GrammarType.COM or item.label == GrammarType.RBRACK:
        semanticMessage = "clear"
    elif item.label != listType[-1]:
        semanticMessage = f"Type 6 Error at '{item}': Inconsistent list element datatype"
    else:
        semanticMessage = "clear"
    return semanticMessage
```

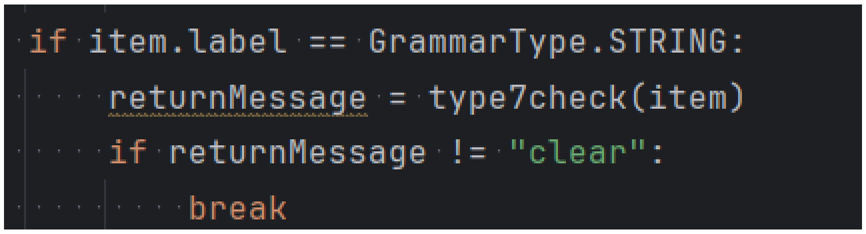
The type6check() method uses the variable listType that was declared at the very beginning of the semanticCheck() method. This is because a list's type must be able to be referenced for



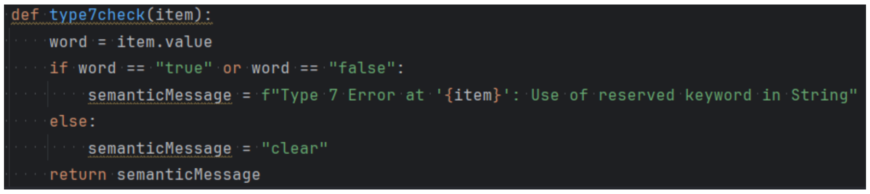
```
def semanticCheck(self, list):
    keyList = []
    listType = []
```

future elements in the list, so the variable can not be declared within type6check(). If the length of the list is 0, that means a type for the list hasn't been determined yet, so the type of the current item is added to the list, and type6check() returns "clear", otherwise, if the item's label is COM, a comma, or RBRACK, a right bracket, the method also returns clear, as those are the values of the list, but rather terminal values that either separate the elements or closes the list. Else, if the item's type does not match the type at the last index of the listType list, then an error message is returned.

Type 7 is checked when a String is detected:



```
if item.label == GrammarType.STRING:
    returnMessage = type7check(item)
    if returnMessage != "clear":
        break
```



```
def type7check(item):
    word = item.value
    if word == "true" or word == "false":
        semanticMessage = f"Type 7 Error at '{item}': Use of reserved keyword in String"
    else:
        semanticMessage = "clear"
    return semanticMessage
```

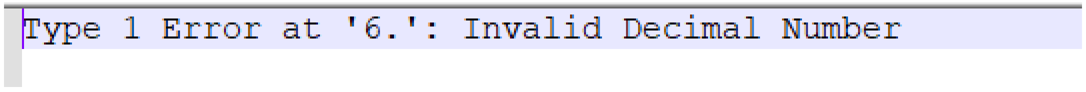
The type7check() method is called, and all it does is return an error message if the value of the string is either "true" or "false". Otherwise, it returns "clear".

I've explained how errors are handled in the aside section after the type1check() explanation, but I haven't explained what happens if no error is detected. If there is no error, the output() method, which called the semanticCheck() method, can finally move on to building the abstract syntax tree, using the same logic the part 2 Parser used to build a tree, only this time there are no non-terminal values or terminal labels to add to the tree. Instead, indentation is based on Whether the item is a bracket or brace, or a value within brackets or braces. If the terminal is a bracket or brace, its current indentation level becomes the minIndent amount. Commas within that dictionary or list are then printed at that same indentation level. If it is a value or a pair within a list or a dictionary, their indentation is one higher than the braces, brackets or commas, as they are contents of the list or dictionary. Once output() has iterated through the entire cleanList, the list created, where each index contains the value of each token with the appropriate indentation level is sent back to the parse() method, which initially called the output() method. parse() in turn returns this list to the main method, where it is iterated through, with each element being printed on a new line in a text file, which is then the resulting output file that the user can view.

## Error handling:

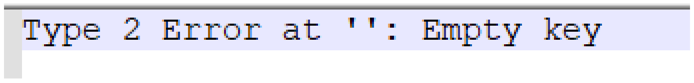
As explained in the short aside on what happens when an error occurs within the semanticCheck() method, if the returning message from any of the error type checking methods explain above are not “clear”, then the while loop within semantic check breaks, and it returns the error message generated by one of the error type checking methods. The output method also checks to see if the message returned by semanticCheck() is “clear” or not. If it isn’t, it returns before even attempting to create an abstract syntax tree, and just returns the error message instead. The parse method then returns this message back to the main method, which prints it to a file.

When a type 1 error occurs, the output file looks something like this:



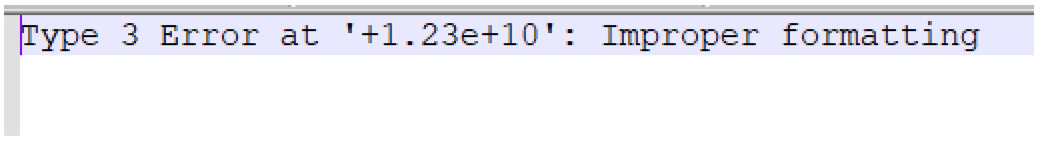
```
Type 1 Error at '6.': Invalid Decimal Number
```

When a type 2 error occurs, the output file looks something like this:



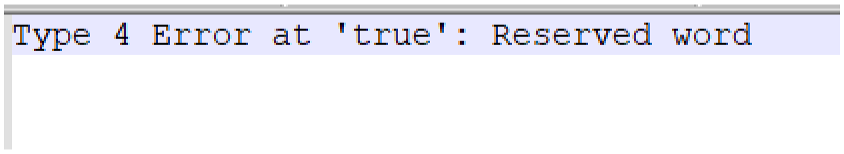
```
Type 2 Error at '': Empty key
```

When a type 3 error occurs, the output file looks something like this:



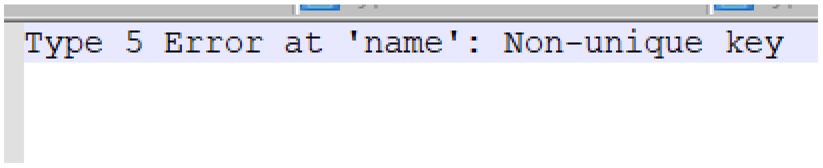
```
Type 3 Error at '+1.23e+10': Improper formatting
```

When a type 4 error occurs, the output file looks something like this:



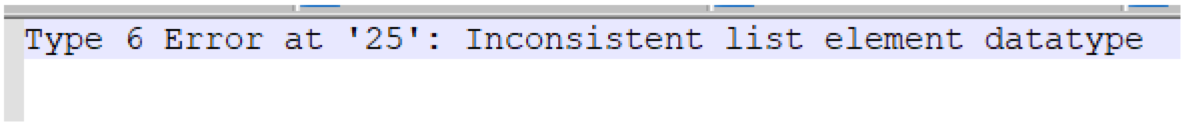
```
Type 4 Error at 'true': Reserved word
```

When a type 5 error occurs, the output file looks something like this:



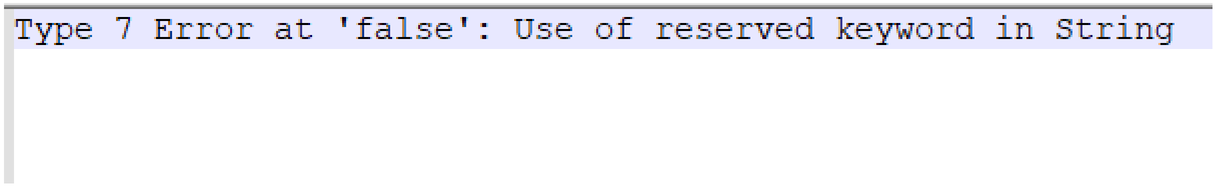
```
Type 5 Error at 'name': Non-unique key
```

When a type 6 error occurs, the output file looks something like this:



```
Type 6 Error at '25': Inconsistent list element datatype
```

When a type 7 error occurs, the output file looks something like this:



```
Type 7 Error at 'false': Use of reserved keyword in String
```