Final Project

Part 3

Written Document

# Vaishnavi Bharti B009807221 CSCI 2115 01

**Language Description:**

The Lexical Scanner is taking in the JASON as input and uses this Grammar:

How do you build the abstract syntax trees?

I knew it worked on Jason's grammar description and was discussed in the tutorial.

So, I knew the way it encountered characters to create a parse tree I have to do the same for AST.

And

So, when {encountered Dictionary Parser initially creates an object node that calls a pair or list parser that calls primitive parsers and other parsers to develop children’s nodes and create key and values parent node.

The primitive parsers like null, Boolean, string, and number consume the token and create value nodes.

The List parser creates a list node that creates smaller value nodes.

Semantic Error:

1. Invalid Decimal Numbers (Type 1 Error)
2. Empty Key (Type 2 Error)
3. Invalid Numbers (Type 3 Error)
4. Reversed Words (Type 4 Error)
5. Consistent Types for List Elements (Type 6 Error)
6. Reversed Words as String (Type 7 Error)

Real Input to Parser

Test:

token(CurlyOpen, '{')

token(String, 'valid\_number')

token(Colon, ':')

token(Number, 3.14)

token(Comma, ',')

token(String, 'invalid\_number\_1')

token(Colon, ':')

token(Number, 3.)

token(Comma, ',')

token(String, 'invalid\_number\_2')

token(Colon, ':')

token(Number, .14)

token(CurlyClose, '}')

Test2:

token(CurlyOpen, '{')

token(String, 'name')

token(Colon, ':')

token(String, 'Alice')

token(Comma, ',')

token(String, '')

token(Colon, ':')

token(Number, 30)

token(Comma, ',')

Test3

token(CurlyOpen, '{')

token(String, 'valid\_number\_1')

token(Colon, ':')

token(Number, -12.001)

token(Comma, ',')

token(String, 'valid\_number\_2')

token(Colon, ':')

token(Number, 1.23e+10)

token(Comma, ',')

token(String, 'invalid\_number\_1')

token(Colon, ':')

token(Number, 0123)

token(Comma, ',')

token(String, 'invalid\_number\_2')

token(Colon, ':')

token(Number, +2)

token(CurlyClose, '}')

Test4

token(CurlyOpen, '{')

token(String, 'true')

token(Colon, ':')

token(String, 'value1')

token(Comma, ',')

token(String, 'false')

token(Colon, ':')

token(String, 'value2')

token(String, 'null')

token(Colon, ':')

token(Number, 34)

token(CurlyClose, '}')

test5

token(CurlyOpen, '{')

token(String, 'key1')

token(Colon, ':')

token(Number, 1)

token(Comma, ',')

token(String, 'key1')

token(Colon, ':')

token(Number, 2)

token(CurlyClose, '}')

test6

token(CurlyOpen, '{')

token(String, 'valid\_list')

token(Colon, ':')

token(SquareOpen, '[')

token(Number, 1)

token(Comma, ',')

token(Number, 2)

token(Comma, ',')

token(Number, 3)

token(SquareClose, ']')

token(Comma, ',')

token(String, 'invalid\_list')

token(Colon, ':')

token(SquareOpen, '[')

token(Number, 1)

token(Comma, ',')

token(String, 'two')

token(Comma, ',')

token(False, false)

token(SquareClose, ']')

token(CurlyClose, '}')

test7

token(CurlyOpen, '{')

token (String, 'key')

token (Colon, ':')

token (String, 'true')

token (Comma, ',')

token (String, 'another\_key')

token (Colon, ':')

token (String, 'false')

token (Comma, ',')

token (String, 'key3')

token (Colon, ':')

token (String, 'null')

token (CurlyClose, '}')

ATTRIBUTE GRAMMAR:

Production Rule:

1. Value → Number | String | Null | Boolean | List | Dict
2. Number → Digit+ "." Digit+ | Digit+
3. String → '"' Char\* '"'
4. Null → "null"
5. Boolean → "true" | "false"
6. List → "[" Value ("," Value)\* "]"
7. Dict → "{" Pair ("," Pair)\* "}"
8. Pair → String ":" Value

Semantic Rules

So I have a Validator class that has methods that check semantic errors and passes the Jason if it follows production rules let’s go through it

**1.Number** → Digit+ "." Digit+ | Digit+

To check this, use Digit+ "." Digit+|Digit+

My number validator does semantic error for digits starting or ending with ‘.’ And that no + symbol is before Digit for Digit+

2.String → '"' Char\* '"'

This is checked by using a String parser that calls string validator so that only character passes that are not reserved words.

3.Dict → "{" Pair ("," Pair)\* "}"

This is checked by a Dictionary parser that checks out the grammar

4.Value → Number | String | Null | Boolean | List | Dict

This is checked by calling a value parser that checks out using a primitive parser like Boolean string, digits, null, etc

5.Boolean → "true" | "false"

The BooleanParser method checks if the token is either "true" or "false

6. List → "[" Value ("," Value)\* "]"

The ListParser method checks if the list is enclosed in square brackets, with values separated by commas.

The validate list method checks that all elements inside the list are of the same type. If there are mixed types, it adds an error (Type 6)

7. Pair → String ":" Value

The PairParser method checks that the pair consists of a valid string key followed by a colon and a valid value.

validate\_dict\_key method ensures that:

The string key is not empty.

The string key does not use reserved words (true, false, null).

The key is not duplicated in the dictionary.

Code Explanation:

Recursive Descent Parser:

This is done by using a series of parsers each using terminal and nonterminal parsers.

Terminal Parser: that calls other parsers to check if it is semantically correct json.

Non-terminal Parser: that doesn’t call other parsers to check if it is semantically correct json.

Each terminal parser calls the terminal and another non-terminal parser recursively in basically using top-down manner.

So initially lexical scanner tokenizes Jason for input for the semantic parser.

Then according to it call the value parser that handles the first production rule in the mentioned Attribution grammar production rule

And uses different parsers like digit parser that follows all production rules related to digits.

Similar to string, Boolean, null, list which check production rules related to it, and in case of finding romantic error calls validator methods related to it.

While dict calls other parsers that form no terminal and terminal for the production rules that it follows related to dict that individually checks out the.

While doing so it also checks out the AST rules if they find no semantic errors in each parser it adds a node to AST. If digit parsers don’t encounter any semantic error and it follows the production rule it adds a child node. If we encounter dict it add

Value

Object

{

And } after calling other terminal and non-terminal parsers that add their node to it.

This is kind different from part 2 where I create a parser tree as

How it is different from part2:

We didn’t check for semantic errors rather checked if it was syntactically right. Also, AST is very different from the parser tree as it does not include Syntactic elements of Jason like, :, etc.

HOW DID CLEANED UP?

I just used a parser with their original functioning and added a validator for them to check semantic errors when we are going through that token.

And deleted the parse tree class almost completely to add new AST logic which is way simpler than that and used it in parser to and nodes to AST.

Error Handling:

Test 2 file :

A screenshot of a computer

Description automatically generated

Test3

A screenshot of a computer error

Description automatically generated

Test 5

A screenshot of a computer

Description automatically generated