Json

Sven Eric Panitz

27. April 2021

Inhaltsverzeichnis

1. Vorbemerkung

In dieser Aufgabe werden mehrere Tools in Bibliotheken verwendet, die nicht im Standard JDK enthalten sind.

Diese Bibliotheken werden wir als jar-Dateien einbinden. Hierzu müssen Sie in Ihrer integrierten Entwicklungsumgebung die entsprechenden jar-Datein in Ihr Projekt importieren.

Arbeiten Sie von der Kommandozeile, so müssen Sie dem Javacompiler diese jar-Dateien als Klassenpfad mit angeben.

So könnte dann ein kompletter Aufruf des Compilers wie folgt aussehen:

Shell

```
javac --enable-preview --source 15 --class-path
↪  .:antlr-4.9.1-complete.jar:bson-4.2.3.jar:mongodb-driver-sync-4.2.3.ja..
↪  -encoding utf-8  -d . Json.java
```

1.1. MongoDb Java Driver

Wir verwenden den MongoDb Java Driver. Dieses ist zum einen eine Bibliothek, die es ermöglicht, dass Java-Programme mit der Dokumenten basierten Datenbank MongoDB[1] interagieren können.

Wir verwenden in dieser Aufgabe drei jar-Dateien, die diese Bibliothek komplett enthalten. Es sind die Dateien:

`bson-4.2.3.jar`
`mongodb-driver-sync-4.2.3.jar`
`mongodb-driver-core-4.2.3.jar`

Diese lassen sich zum Beispiel auf folgender Webseite herunterladen:

`https://mvnrepository.com/artifact/org.mongodb/mongo-java-driver`

Dokumentation Wir brauchen zum Lösen der Aufgabe primär die Klasse `org.bson.Document`.

Deren Dokumentation ist über folgende Webseite einsehbar.

`https://mongodb.github.io/mongo-java-driver/4.2/apidocs/bson/`

1.2. Parser-Generator Tool ANTLR

In dieser Aufgabe wird auch das Parser-Generator Tool ANTLR verwendet. Wir arbeiten mit der jar-Datei, die alles enthält, das wir benötigen:

`antlr-4.9.1-complete.jar`

Diese lässt sich auf folgender Webseite herunterladen:

`https://www.antlr.org/download.html`

Sie brauchen sich weder mit ANTLR im Detail noch mit dem API von ANTLR im Spezielles auseinder setzen. Sie müssen zum einen die jar-Datei im Klassenpfad haben, zum anderen müssen Sie es verwenden, um aus der Grammatik-Datei

`name/panitz/json/JsonGrammar.g4`

[1] Dieser Name ist äußerst unglücklich gewählt, weil es ein Wort ist, dass im Deutschen als sehr diskriminierendes Schimpfwort bekannt ist. Ich gebe zu, dass ich jedes Mal schlucken muss, wenn ich es hier schreibe.

den Java Quelltext zu generieren.

Der entsprechende Aufruf ist:

Shell

```
java -jar antlr-4.9.1-complete.jar name/panitz/json/JsonGrammar.g4
```

Dadurch werden folgende Dateien generiert:

`JsonGrammarBaseListener.java`, `JsonGrammarListener.java` `JsonTreeBuilder.java` `JsonGrammarLexer.java` `JsonGrammarParser.java`

2. Json

Neben dem Format XML, hat sich ein zweites Format zur Serialisierung, textuellen Beschreibung von hierarchischen Strukturen etabliert und weit verbreitet, das Json-Format. Json steht für JavaScript Object Notation und zeigt damit an, woher das Format kommt: aus der Programmierung mit Javascript.

Javascript selbst stellt schon eine Kuriosität in der Welt der Programmierung dar. Ursprünglich entwickelt 1995 unter den Namen *Livescript* von der Firma *Netscape* als Sprache, um im Browser kleine dynamische Änderungen im HTML aufgrund von Anwendereingaben zu programmieren, ist es heute die Hauptsprache innerhalb des Browsers. Es gibt eine Reihe von Compilern, die Javascript als Zielsprache haben. Damit ist Javascript heutzutage auch so etwas wie eine Assemblersprache auf Browsern.

Im Browser gab es eine zweite Programmiersprache, die eine Programmausführung auf dem Client in Webseiten ermöglichte: Java. Dies allein führte aus Marketing-Sicht zum Namen *Javascript*, obwohl beide Sprache nichts gemeinsam haben, weder in Syntax, Typisierung noch im Ausführungsmodell.

Der eigentliche Sprachkern von *Javascript* heißt offiziell auch schon lange nicht mehr *Javascript* sondern ist unter den Namen *ECMAScript*[ECM20] standartisiert. Der Standard wird von der Ecma International (Ecma) herausgegeben. Die Ecma ist eine Organisation zur Normung mit Sitz in Genf. Absurder Weise spielt *Javascript* in der Webprogrammierung eine immer größere Rolle und wird nicht mehr nur clientseitig im Browser ausgeführt, sondern auch serverseitig mit der Node.js Technologie.

Da in *Javascript* nicht mehr nur innerhalb des Browsers lokal gearbeitet wird, sondern Daten an den Server geschickt werden und von Webservices erfragt werden, gibt es einen hohen Bedarf nach einer Serialisierung von Objekten. Json ist die übliche Wahl, zur Serialisierung in der *Javascript* Programmierung. Das Json Format erfüllt dabei folgende Entwurfskriterien:

- es ist ebenso wie XML kein binäres sondern ein textuelles Format.
- Json kann hierarchische Strukturen also Baumstrukturen darstellen.

- anders als XML hat Json wenig Overhead an Daten. Es werden nur wenige Steuerungs- und Strukturierungsinformationen benötigt.
- Json ist für einen Menschen gut zu lesen.
- Json ist leicht von Programmen zu lesen.
- Json ist in einer Syntax, die sich in der *Javascript*-Syntax einfügt, formuliert.

Anders als bei XML spielt die Dokumentensicht in Json keine Rolle. Dieser Lehrbrief ist direkt als XML Dokument formuliert und manche Leute schreiben HTML Seiten direkt als Tetdokument. Hierfür ist Json als Datenformat schlecht geeignet und es käme kaum jemand auf die Idee, ein Textdokument mit Struktur- und Formatierungsinformation als Json-Dokument zu formulieren. Bei Json geht es primär um den Transport und die Speicherung von Daten.

Json wurde ursprünglich Anfang der 2000er von Douglas Crockford spezifiziert. Erstmals offiziell standartisiert wurde es 2013 von der ECMA als ECMA-404[ECM17].

In dieser Aufgabe werden wir und mit unterschiedlichen Bibliotheken ein kleines Api für Json entwickeln. Hierzu zunächst die nötigen Imports.

Java: Json

```
package name.panitz.json;
import java.util.*;
import java.util.function.Consumer;
import java.util.stream.Collectors;

import java.io.*;

import org.antlr.v4.runtime.*;
import org.antlr.v4.runtime.tree.ParseTreeWalker;
import org.bson.Document;
import com.mongodb.client.MongoClients;
import com.mongodb.client.MongoCollection;
import com.mongodb.client.model.Filters;
```

Die Funktionalität wird innerhalb einer Schnittstelle `Json` entwickelt.

Java: Json

```
public interface Json{
```

Diese Schnittstelle dient gleichzeitig als gemeinsame Schnittstelle für Json-Datentypen unterschiedlicher Ausprägung mit einigen default-Methoden für Json-Objekten, als auch als Rahmen, in dem einige statische Methoden und Klassen enthalten sind.

2.1. Schlüssel-Wert-Paare

Zunächst einmal ist ein Json Objekt eine Abbildung von Schlüsseln nach Werten. Die Schlüssel-Wert Paare werden in geschweiften Klammern zusammen gefasst und durch Kommata separiert.

Die Schlüssel sind Strings, die in doppelten Anführungszeichen eingeschlossen sind. Nach dem Schlüssel folgt ein Doppelpunkt. Dann kommt der Wert. Der Wert kann wieder ein String sein, aber auch eine Zahlenkonstanten, einer der beiden bool'schen Literale `true` und `false` oder das Schlüsselwort `null`.

Hier ein Beispiel eines Strings in Json Notation. Es gibt die Schlüssel `vorname`, `name` und `gebJahr` mit konkreten Werten.

Java: Json

```
  static String ex1 = """
{ "vorname": "Eva"
, "name": "Luator"
, "gebJahr": 1962
}
""";
```

Das entsprechende Dokument sieht in XML wie folgt aus.

xml: ex1

```
<person>
  <vorname>Eva</vorname>
  <name>Luator</name>
  <gebJahr>1962</gebJahr>
</person>
```

Ein semantischer Unterschied zwischen dem hier dargestellten Json-Dokument und XML-Dokument ist, dass die Reihenfolge der Schlüssel-Wert-Paare in einem Json-Dokument irrelevant ist, anders als die Kindelemente in einem XML-Dokument.

Damit ist das Beispieldokument eher vergleichbar mit folgenden XML-Dokument, in dem die Werte als Attribute noriert sind.

xml: ex1

```
<person
  vorname="Eva"
  name="Luator"
  gebJahr="1962" />
```

2.2. Rekusive Struturen

Da die Werte eines Schlüssel-Wert Paares auch wieder Json-Objekte sein können, lassen sich auch rekursive Objekte in Json darstellen. Hier als Beispiel eine Listenstruktur aus vier Elementen.

Java: Json

```
  static String xs = """
{ "head": 1
, "tail":
  { "head": 2
  , "tail":
    { "head": 3
    , "tail":
       { "head": 4
       , "tail": null
       }
    }
  }
}
""";
```

2.3. Arrays

Zusätzlich können die Werte in einem Json-Objekt Reihungen von weiteren Objekten sein. Diese werden dann in eckigen Klammern und durch Komma separiert aufgelistet. In diesen Reihungen ist anders als bei den Schlüssel-Wert-Paaren die Reihenfolge relevant.

Abbildung 1 zeigt ein kleines Beispieldokument, das das englische Königshaus repräsentiert.

Java: Json

```
  static String windsor = """
{ "name": "George"
, "children":
  [{ "name": "Elizabeth"
   , "children":
     [{ "name": "Charles"
      , "children":
        [{ "name": "William"
         , "children":
           [{"name":"George"},{"name":"Charlotte"},{"name":"Louise"}]}
        ,{ "name": "Henry","children": [{ "name": "Archie"}]}
      ]}
     ,{ "name": "Anne"
      , "children" :
         [{ "name": "Peter"
          , "children": [{ "name": "Savannah"},{ "name": "Isla"}]}
         ,{ "name": "Zara","children":[{ "name": "Mia"},{ "name": "Lena"}]}
         ]}
     ,{ "name": "Andrew"
      , "children": [{"name":"Beatrice"},{"name": "Eugenie"}]}
     ,{ "name": "Edward"
      , "children": [{"name":"Louise"},{ "name": "James"}]}]}
  ,{ "name": "Magaret"
   , "children":
     [{ "name": "David"
      , "children": [{ "name": "Charles"},{ "name": "Margarita"}]}
     ,{ "name": "Sarah"
      , "children": [{ "name": "Samuel"},{ "name": "Arthur"}]}
     ]}
  ]}
""";
```

Abbildung 1: Eine Baumstruktur im Json-Fomrat

3. Json für Java

In der Java-Stadard Edition gibt es derzeit kein API zur Json-Verarbeitung. Es gibt also noch kein allgemeines API zum Arbeiten mit Json. Es gibt hingegen ein API in der Java Enterprise Edition (javaee).

Ursprünglich von Google für interne Projekte entwickelt gibt es die Bibliothek *Gson*[gso], mit deren Hilfe beliebige Java-Objekte als Json serialisiert und deserialisiert werden können.

Wir wollen in dieser Aufgabe ein eigenes kleines nützliches API für Json selbst entwickeln.

3.1. Ein Datentyp für Json

Für jedes Json-Element schreiben wir eine Record-Klasse, die die Schnittstelle `Json` implementiert.

Zahlenwerte Die Werte in einem Json-Objekt können Zahlen sein. So sehen wir eine Klasse für ganze Zahlen vor:

Java: Json

```
public static record JsonLong(long n) implements Json{}
```

Eine zweite Klasse repräsentiere Fließkommazahlen:

Java: Json

```
public static record JsonDouble(double n) implements Json{}
```

Wahrheitswerte In Json-Objekte können Wahrheitswerte notiert sein. Auch hierfür sei eine Record-Klasse definiert.

Java: Json

```
public static record JsonBoolean(boolean b) implements Json{}
```

Texte Der häufigste Wert dürften Strings sein, für die es eine weitere Klasse gibt.

Java: Json

```
public static record JsonString(String s) implements Json{}
```

Json-Objekte Ein Json-Objekt ist schließlich ein Abbildung, die String-Schlüssel auf Json-Werte abbildet. Hierzu können wir die Standard-Abbildungen aus `java.util.Map` verwenden und benötigen ein Objekt des Typs: `Map<String,Json>`.

Somit erhalten wir folgende Record-Klasse:

Java: Json

```
public static record JsonObject(Map<String,Json> map) implements Json{}
```

Reihungen Das letzte Konstrukt in Json sind die Reihungen von Json-Objekten. Hierzu verwenden wir intern die Standardlisten aus Java:

Java: Json

```
public static record JsonArray(List<Json> elements) implements Json{}
```

Jetzt lassen sich mit diesen Record-Klassen schon einmal Json-Objekte erzeugen.

Hierzu ein kleines Beispiel:

Shell

```
jshell> import static name.panitz.json.Json.*
jshell> import java.util.*

jshell> new JsonObject(Map.of("name",new
↪  JsonString("Luator"),"gebDatum",new JsonLong(1962)))
$4 ==> JsonObject[map={gebDatum=JsonLong[n=1962],
↪  name=JsonString[s=Luator]}]
```

3.1.1. Json Einlesen

Wir haben eine Datenstruktur, die in der Lage ist Json-Objekte strukturiert darzustellen. In der Regel werden die Json-Objekte als Strings vorliegen, zum Beispiel weil sie über Netzwerkanfragen erhalten wurden. Diese als String serialisierte Form eines Json-Dokuments ist erst in unsere Datenstruktur zu transformieren. Ein Programm, das einen String entsprechend einer Grammatik einliest und in strukturierter Form darstellt, wird als Parser bezeichnet. Das Entwickeln von Parsern ist eine Teildisziplin des Compilerbaus. Die Theorie hierfür werden Sie im Laufe Ihres Studiums im Modul »Automaten und formale Sprachen« kennenlernen.

In der Regel wird ein Parser nicht von Hand codiert, sondern es wird ein Tool verwendet, das aus einer Grammatikdefinition ein Programm generiert, das den Parser für die Grammatik realisiert. Im Anhang A geben wir die Eingabe der Json Grammatik für das Parsergeneratortool Antlr. Dadurch erhalten wir Klassen zum Einlesen der Json-Objekte.

Die mit diesem Tool generierte Funktionalität wird in folgender Funktion verwendet, die es erlaubt, ein Json-Objekt für einen String in unsere Json-Datenstruktur einzulesen.

Java: Json

```java
public static JsonObject parse(Reader in) throws Exception{
  var codePointCharStream = CharStreams.fromReader(in);
  var lexer = new JsonGrammarLexer(codePointCharStream);
  var parser = new JsonGrammarParser(new CommonTokenStream(lexer));

  var tree = parser.obj();
  ParseTreeWalker.DEFAULT.walk(new JsonTreeBuilder(), tree);
  return tree.result;
}
```

Unterschiedliche Fehler, die auftreten können, weil der Reader nicht lesbar ist oder darin kein korrektes Json-Dokument zu lesen ist, werden zu Ausnahmen führen, die der Einfachheit halber nicht differenziert aufgeführt sind.

Um jetzt für einen String ein Json-Objekt zu erhalten, kann dieser als `StringReader`-Objekt der Funktion `parse` übergeben werden.

Shell

```
jshell> import static name.panitz.json.Json.*

jshell> parse(new StringReader(ex1))
$2 ==> JsonObject[map={gebJahr=JsonLong[n=1962],
↪  vorname=JsonString[s=Eva], name=JsonString[s=Luator]}]

jshell>
```

4. Aufgaben

Wir haben tatsächlich mit wenigen Handgriffen eine Datenstruktur für Json entwickelt. In dieser Aufgabe sollen weitere Methoden für diese Struktur entwickelt werden. Insbesondere werden wir auch eine Methode zum Speichern der Objekte in eine Datenbank schreiben.

Aufgabe 1 In dieser Aufgabe sind weitere default-Methoden für Json-Objekte zu entwickeln.

a) Schreiben Sie jetzt eine Funktion `forEach`, die auf alle in einem Json-Objekt rekursiv enthaltenen Json-Dokumente die übergebene Funktion anwendet.

Java: Json

```
default void forEach(Consumer<JsonObject> consume){
}
```

b) Schreiben Sie eine Funktion, die für das Json-Objekt alle Werte des Attributes `name` in einer Menge sammelt.

Java: Json

```
default Set<String> collectNames(){
  var result = new HashSet<String>();
  collectNames(result);
  return result;
}
```

Java: Json

```
default void collectNames(Set<String> result){
  /* ToDo */
}
```

Hier ein Beispielaufruf für das englische Königshaus:

Shell

```
jshell> parse(new StringReader(ex1)).collectNames()
$2 ==> [Luator]

jshell> parse(new StringReader(windsor)).collectNames()
$2 ==> [Margarita, Charlotte, Arthur, Louise, Eugenie, Edward,
↪  Magaret, Samuel, George, Sarah, Anne, Zara, Isla, Henry,
↪  Charles, Savannah, Mia, James, Andrew, Lena, William, Beatrice,
↪  Archie, Elizabeth, David, Peter]
```

Versuchen Sie die Lösung unter Verwendung der Methode `forEach`.

c) Schreiben Sie in dieser Aufgabe eine Funktion, die das Json-Objekt serialisiert in einen Writer schreibt.

Java: Json

```
default void write(Writer out) throws Exception{
  write("\n",out);
}
```

Die eigentliche Methode bekommt einen String übergeben, mit dem sich neue Zeilen und eine Einrückung je nach Baumtiefe realisieren lassen. Bei jedem rekursiven Aufruf empfiehlt es sich, diese Einrückung zu vergrößern (`indent+"  "`).

Java: Json

```
default void write(String indent, Writer out) throws Exception{
  if (this instanceof JsonString s) out.write("\""+s.s+"\"");

  /* ToDo missing Json record instances.  */

  else
    throw new RuntimeException("Missing pattern: "+this.getClass());
}
```

Mit dieser Funktion lässt sich dann auch eine einfache Funktion schreiben, die ein Json-Objekt in einen String serialisiert:

Java: Json

```
default String show() {
  try{
    var out = new StringWriter();
    write(out);
    return out.toString();
  }catch(Exception e){ return "";}
}
```

Somit gibt es eine Art Rundreise: einen String als Json-Objekt parsen und dann wieder als String serialisieren:

Shell

```
jshell> import static name.panitz.json.Json.*

jshell> System.out.println(parse(new StringReader(ex1)).show())

{"gebJahr": 1962
,"vorname": "Eva"
,"name": "Luator"
}
```

Aufgabe 2 In dieser Aufgabe geht es darum, für Java-Objekte einer bestimmten Klasse Json-Objekte zu erzeugen und aus den Json-Objekten wieder die Java-Objekte zu bekommen.

Hierzu sei als Beispielklasse eine kleine Record-Klasse für einfache Personendaten definiert:

Java: Json

```
static record Person(String name,String vorname,int gebJahr){
```

a) Schreiben Sie für die Record-Klasse `Person` eine Methode, die die Daten des Personen-Objektes als ein Json-Objekt darstellt:

Java: Json

```
  Json toJson(){
    var map = new HashMap<String,Json>();
    /* ToDo: Fill map with correct data. */
    return  new JsonObject(map);
  }
}
```

b) Schreiben Sie die inverse Funktion, die das this-Json-Objekt zu einem Objekt der Klasse `Person` transformiert. Hierzu muss das Json-Objekt natürlich die Felder für eine Person enthalten. Wenn das Json-Objekt keine Objekt der Klasse `Person` codiert, dann wird eine Ausnahme geworfen.

Java: Json

```
default Person getPerson(){
  if (this instanceof JsonObject o){
     /* ToDo build the person object and return it. */
  }
  throw new RuntimeException("json not a Person: "+this.toString());
}
```

Aufgabe 3 In dieser Aufgabe sollen Json-Objekte in eine Datenbank geschrieben werden. Hierzu verwenden wir die Datenbank MongoDB (`www.mongodb.com`), die im Prinzip Json-Objekte speichert.

Intern wird ein binäres Datenformat für die Json-Objekte verwendet. Dieses wird als `BSON`[Incb] bezeichnet. Hierfür steht ein Java API zur Verfügung.

Es gibt ein Java Api, um mit Bson-Objekten zu arbeiten[Inca]. Für unsere Zwecke reicht es aus, hier allein mit der Klasse `org.bson.Document` zu arbeiten.

Um also unsere Json-Objekte in die Datenbank zu schreiben, benötigen wir Methoden, um Json-Objekte in BSON-Objekte zu transformieren und umgekehrt.

a) Beginnen wir damit, aus unseren Json-Objekten ein BSON-Objekt zu machen.

   Nur Objekte der Klasse `JsonObject` lassen sich in ein BSON Dokument transformieren. Ein BSON Dokument ist ein `Map`. Die Schlüssel sind Strings, die Werte

beliebige Objekte, also zum Beispiel Strings, Zahlen, Boolean, Listen oder wieder vom Typ `Document`.

Java: Json

```
default Document toBSON(){
  if (this instanceof JsonObject o){
    return (Document)toBSON2();
  }
  return null;
}
```

Die eigentliche Methode erzeugt ein BSON Dokument und füllt es mit den korrekten Schlüssel-Wert Paaren.

Java: Json

```
default Object toBSON2(){
  if (this instanceof JsonObject o){
    return null; /*ToDo*/
  }
  else if (this instanceof JsonDouble d) return d.n;
  return null;
}
```

b) Nun sollen die BSON Dokumente der Datenbank wieder in Json Objekte unserer Implementierung umgewandelt werden.

Java: Json

```
static Json toJson(Document bson){
  try{
    return null;/*ToDo*/
  }catch(Exception e){
    throw new RuntimeException(e);
  }
}
```

Tipp: die einfachste Lösung dieser Methode nutzt aus, dass die BSON-Objekte sich mit der Methode `toJson()` als Json-String serialisieren lassen. Diesen können wir als ein Objekt unserer Json-Implementierung parsen.

c) Wir können jetzt für unsere Json-Objekte eine Methoden schreiben, um diese in eine Datenbank zu schreiben. Datenbanken haben einen Namen und enthalten Collections, in denen die Dokumente gespeichert werden können.

Java: Json

```
default void saveToCollection(String dbname,String collectionName){
  var mongoClient = MongoClients.create();
  var database = mongoClient.getDatabase(dbname);
  var collection = database.getCollection(collectionName);
  collection.insertOne(toBSON());
}
```

Interessanter ist nun der andere Weg, nämlich die Dokumente aus einer Datenbank wieder abzufragen.

Java: Json

```
static MongoCollection<Document>
            getCollection(String dbname,String collectionName){
  var mongoClient = MongoClients.create();
  var database = mongoClient.getDatabase(dbname);
  return database.getCollection(collectionName);
}
```

Das Ergebnis ist ein Objekt der Klasse:

`com.mongodb.client.MongoCollection`

Diese stellt viele Methoden zum Arbeiten mit der Datenbank Collection zur Verfügung.

Schreiben Sie eine Funktion, die durch alle Dokumente, die in einer Collection in der Datenbank gespeichert sind, für alle toplevel Attribute mit Schlüsselnamen `name`, die einen String als Wert haben, diesen in einer Ergebnismenge sammeln.

Java: Json

```
static Set<String> getNames(String dbname,String collectionName){
  var result = new HashSet<String>();
  /* ToDo */
  return result;
}
```

Soweit unser erster Ausflug in die Welt von Json und dokumentenbasierten Datenbanken. Dieses können Sie in weiterführenden Wahlmodulen und Wahlprojekten im Laufe des Studiums weiter vertiefen und anwenden.

Java: Json

```
}
```

A. Ein Parser für Json

In diesem Anhang wird der Parser für Json-Dokumenten entwickelt. Hierzu wird zunächst die Grammatik, die die Syntax von Json-Dokumenten definiert, für das Tool ANTLR formuliert.

A.1. Json Grammatik

Die Grammatik ist eine kontextfreie Grammatik mit den folgenden Regeln:

antlr: JsonGrammar

```
grammar JsonGrammar;
@header {
package name.panitz.json;
import static name.panitz.json.Json.*;
}
obj returns[JsonObject result]
   : '{' pair (',' pair)* '}'
   | '{' '}';
pair returns[Pair<String,Json> result]
   : stringB ':' json;
arr returns[Json result]
   : '[' json (',' json)* ']'
   | '[' ']' ;
json returns[Json result]
   : nullB
   | NUMBER
   | obj
   | arr
   | trueB
   | falseB
   | stringB ;
nullB: 'null';
trueB: 'true';
falseB: 'false';
stringB  returns[String result]: STRING;
```

A.2. Json Token

Für den Lexer des Json-Formats folgen hier die Definitionen für Stringliterale, Zahlen und Zwischenraum.

antlr: JsonGrammar

```
STRING: '"' (ESC | SAFECODEPOINT)* '"';
fragment ESC: '\\' (["\\/bfnrt] | UNICODE);
fragment UNICODE: 'u' HEX HEX HEX HEX;
fragment HEX: [0-9a-fA-F];
fragment SAFECODEPOINT: ~ ["\\\u0000-\u001F];
NUMBER:  ('-')?INT ('.' [0-9] +)? EXP?;
fragment INT: '0' |  [1-9] [0-9]*;
fragment EXP: [Ee] [+\-]? INT;
WS: [ \t\n\r] + -> skip;
```

A.3. Json Baum Erzeugen

Die folgende Klasse wird verwendet, um mit dem von ANTLR erstellten Parser Objekte unserer Datenstruktur `Json` zu erzeugen.

java: JsonTreeBuilder

```java
package name.panitz.json;
import org.antlr.v4.runtime.misc.Pair;
import java.util.*;
import java.util.stream.Collectors;
import name.panitz.json.JsonGrammarBaseListener;
import static name.panitz.json.JsonGrammarParser.*;
import static name.panitz.json.Json.*;

public class JsonTreeBuilder extends JsonGrammarBaseListener{
  @Override public void exitObj(ObjContext ctx) {
     var map = new HashMap<String,Json>();
     ctx.result = new JsonObject(map);
     for (var p:ctx.pair()) map.put(p.result.a,p.result.b);
  }
  @Override public void exitPair(PairContext ctx) {
    ctx.result = new Pair<>(ctx.stringB().result,ctx.json().result);
  }
  @Override public void exitArr(JsonGrammarParser.ArrContext ctx) {
    ctx.result = new JsonArray(ctx.json().stream()
            .map(c->c.result).collect(Collectors.toList()));
  }
  @Override public void exitStringB(StringBContext ctx) {
    var img = ctx.STRING().getText();
    ctx.result = img.substring(1,img.length()-1);
  }
  @Override public void exitJson(JsonContext ctx) {
    if (ctx.stringB()!=null){
      var img = ctx.stringB().result;
      ctx.result = new JsonString(img);
    }else if (ctx.NUMBER()!=null){
      ctx.result
        = ctx.NUMBER().getText().contains(".")
        ? new JsonDouble(Double.parseDouble(ctx.NUMBER().getText()))
        : new JsonLong(Long.parseLong(ctx.NUMBER().getText()));
    }else if (ctx.obj()!=null) ctx.result = ctx.obj().result;
    else if (ctx.arr()!=null) ctx.result = ctx.arr().result;
    else if (ctx.trueB()!=null) ctx.result = new JsonBoolean(true);
    else if (ctx.falseB()!=null) ctx.result = new JsonBoolean(false);
    else if (ctx.nullB()!=null) ctx.result = null;
  }
}
```

Literatur

[ECM17] ECMA. ECMA 404, The JSON Data Interchange Syntax. http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf, 12 2017.

[ECM20] ECMA. ECMA 262, ECMAScript® 2020 language specification. https://www.ecma-international.org/wp-content/uploads/ECMA-262.pdf, 6 2020.

[gso] Gson user guide. github.com/google/gson/blob/master/UserGuide.md.

[Inca] MongoDB Inc. Bson api documentation. mongodb.github.io/mongo-java-driver/4.3/apidocs/bson/index.html.

[Incb] MongoDB Inc. Bson specification. bsonspec.org/#/specification.