Project 4: Data Wrangling with Json

1 Background

JavaScript Object Notation (JSON) is a format for storing and exchanging data. Roughly speaking, a JSON value is any JavaScript value with the exception of functions. For example, the following text is JSON:

```
[
 {
   "Department": "Computer Science",
   "Course Number": 220,
   "Instructor": "Joydeep Biswas",
   "Offered": true
 },
 {
   "Department": "Computer Science",
   "Course Number": 589,
   "Instructor": "Joydeep Biswas",
   "Offered": false
 }
]
```

This example shows a JSON array with two elements, where each element is a JSON dictionary. Each JSON dictionary has a set of keys—which are strings—and values—which are JSON values themselves. The example shows that JSON values include strings, numbers, and booleans. Note that arrays and objects are JSON values themselves. Therefore, arrays and dictionaries may be nested:

```
[
 {
   "x": {
"y": [ 1, 2, 3 ]
   }
 },
 "Something else"
]
```

The example above also shows that JSON arrays may be *heterogeneous*: the two elements of the array are an object and a string.

Ocelot has a function called `lib220.loadJSONFromURL` that takes a URL for a JSON file as a string and returns the file as a JavaScript object. This function also takes care of parsing the

JSON string notation, and providing you directly with an object. Here is an example of how to call `lib220.loadJSONFromURL()` , as executed in the Ocelot terminal:

```
> lib220.loadJSONFromURL('https://people.cs.umass.edu/~joydeepb/yelp-tiny.json');
< [
   {
     name: "China Garden",
     city: "Stanley",
     state: "NC",
     stars: 3,
     review_count: 3,
     attributes: {
       RestaurantsAttire: "casual",
       Alcohol: "none",
       OutdoorSeating: false
     },
     categories: [
       "Chinese",
       "Restaurants",
     ]
   },
   {
     name: "Enterprise Rent-A-Car",
     city: "Mesa",
     state: "AZ",
     stars: 4,
     review_count: 3,
     attributes: {},
     categories: [
       "Hotels & Travel",
       "Car Rental",
     ]
   },
  ]
```

2 The Yelp Dataset

The business review site Yelp releases a large dataset of restaurants (and other businesses) in a JSON format. In this assignment, you will use this dataset to answer vital questions such as “What is the most popular restaurant in California?”. Unfortunately, the full dataset is nearly 7GB, which is too large. Therefore, you will instead use a subset of the Yelp data. Each entry in the dataset is an array of JSON objects and each JSON object looks like this:

```
{
  name: "China Garden",
  city: "Stanley",
  state: "NC",
  stars: 3,
  review_count: 3,
  attributes: {
    RestaurantsAttire: "casual",
    Alcohol: "none",
    OutdoorSeating: false
  },
  categories: [
    "Chinese",
    "Restaurants"
  ]
}
```

The following url can be used to load the JSON file for this dataset into Ocelot:
https://people.cs.umass.edu/~joydeepb/yelp.json

For the scope of this project, you can assume the type of a `Restaurant` object to be like so:

```
type Restaurant = {
  name: string,
  city: string,
  state: string,
  stars: number,
  review_count: number,
  attributes: {} | {
    Ambience: {
      [key: string]: boolean
    }
  },
  categories: string[]
}
```

`Ambience: {[key: string]: boolean}` denotes that there are variable number of key-value pairs where the key is a string and the value is a boolean for the `Ambience` property.

3 Programming Task

Overview

The goal of the programming task is to define a class `FluentRestaurants` that supports the fluent design pattern to filter the dataset. We can use this class to perform queries such as "What vegan restaurants are in Wyoming?" Or "Which Mexican restaurants in NY are rated below 2 stars?" The fluent design thus allows the queries to be chained in arbitrary orders, with specified constraints, much like a user might wish to, on the Yelp website, to find specific restaurants of interest. For example, here is how you would use the class `FluentRestaurants` to run two queries:

```
let data = lib220.loadJSONFromURL(
  'https://people.cs.umass.edu/~joydeepb/yelp.json');
let f = new FluentRestaurants(data);

f.ratingLeq(5)
  .ratingGeq(3)
  .category('Restaurants')
  .hasAmbience('casual')
  .fromState('NV')
  .bestPlace().name;

f.ratingLeq(4)
  .ratingGeq(2)
  .category('Restaurants')
  .hasAmbience('romantic')
  .fromState('AZ')
  .bestPlace().name;
```

The key idea is that you can compose these functions together to pose complex data queries. In the above snippet, the first query determines the best "casual" restaurant in Nevada with at least 3 stars and at most 5 stars. The second query determines the best "romantic" restaurant in Arizona that has a rating of at least 2 stars and at most 4 stars. Although this is a dataset of restaurants and businesses, you can assume all objects in the dataset are restaurants.

Specifications

Create a JavaScript file in Ocelot, define a `FluentRestaurants` class, and then implement the class methods enumerated below utilizing the Fluent Pattern discussed in lecture. The constructor should be defined as follows:

```
constructor(jsonData) {
   this.data = jsonData;
 }
```

To work with JSON objects, you will need to use the `lib220.getProperty(jsonObj, memberStr)` library function.

```
getProperty(obj: Object, memberStr: string):
  { found: true, value: any } | { found: false }
```

`lib220.getProperty` takes in a parsed JSON object and the string name of an object member and returns another object. The returned object has two member variables, `found` and `value`, to indicate whether the property with the specified string was found in the object or not, and if so, to return its value. Here is an example usage:

```
test("Usage for getProperty", function() {
  let obj = { x: 42, y: "hello"};
  assert(lib220.getProperty(obj, 'x').found === true);
  assert(lib220.getProperty(obj, 'x').value === 42);
  assert(lib220.getProperty(obj, 'y').value === "hello");
  assert(lib220.getProperty(obj, 'z').found === false);
});
```

The `FluentRestaurants` class must implement the following methods:

Method 1:

```
fromState(stateStr: string): FluentRestaurants
```

It takes a string, `stateStr`, and returns a new `FluentRestaurants` object in which all restaurants are located in the given state, `stateStr`.

Method 2:

```
ratingLeq(rating: number): FluentRestaurants
```

It takes a number, `rating`, and returns a new `FluentRestaurants` object that holds restaurants with ratings less than or equal to `rating.`

Method 3:

```
ratingGeq(rating: number): FluentRestaurants
```

It takes a number, `rating`, and returns a new `FluentRestaurants` object that holds restaurants with ratings which are greater than or equal to `rating.`

Method 4:

```
category(categoryStr: string): FluentRestaurants
```

It that takes a string, `categoryStr`, and produces a new `FluentRestaurants` object that holds only those restaurants that have the provided category, `categoryStr`.

Method 5:

```
hasAmbience(ambienceStr: string): FluentRestaurants
```

It takes a string, `ambienceStr`, and produces a new `FluentRestaurants` object with restaurants that have the provided ambience, `ambienceStr`. Each restaurant object contains an `‘attributes’` key that **may or may not** contain an `Ambience` key, which itself is an object:

```
{ ...
  attributes: {
    ... Ambience: {
            hipster: false,
            trendy: false,
            upscale: false,
            casual: false
         }
    }
}
```

Each member of the `Ambience` object has a key-value pair for ambience types, and whether the restaurant has that ambience or not. For a restaurant object to have a given ambience, the value for that particular ambience must be true.

Method 6:

```
bestPlace(): Restaurant | {}
```

It returns the “best” restaurant. The “best” restaurant is the highest rated restaurant. If there is a tie, pick the one with the most reviews. If there’s a tie with the most reviews, pick the first restaurant. If there is no matching result, it should return an empty object.

4 Testing Your Code

You will have to test your code thoroughly to ensure that it robustly handles the challenges of working with real-world data, including missing / optional fields and variable structure. To help you get started, we have provided a few test cases here. It is up to you to define additional tests to check your solution for correctness.

```
const testData = [
 {
    name: "Applebee's",
    state: "NC",
    stars: 4,
    review_count: 6,
  },
  {
    name: "China Garden",
    state: "NC",
    stars: 4,
    review_count: 10,
  },
  {
    name: "Beach Ventures Roofing",
    state: "AZ",
    stars: 3,
    review_count: 30,
  },
  {
    name: "Alpaul Automobile Wash",
    state: "NC",
    stars: 3,
    review_count: 30,
  }
]

test('fromState filters correctly', function() {
    let tObj = new FluentRestaurants(testData);
    let list = tObj.fromState('NC').data;
    assert(list.length === 3);
    assert(list[0].name === "Applebee's");
    assert(list[1].name === "China Garden");
    assert(list[2].name === "Alpaul Automobile Wash");
});

test('bestPlace tie-breaking', function() {
    let tObj = new FluentRestaurants(testData);
    let place = tObj.fromState('NC').bestPlace();
    assert(place.name === 'China Garden');
});
```