Lab 1: Parsing JSON in C

Due date: Monday, April 4, 11:59pm.

Lab Assignment

This is your first team assignment. It consists of two parts: synergistic activities and programming.

The purpose of the programming part of the assignment is for you to become comfortable with parsing and working with JSON files in C.

Synergistic Activities

This lab includes the following team-building activities.

Team name. Come up with a team name.

Team repository and wiki page setup. Set your team up for work using the CSC 468 github repository. We will discuss the specific details for how the repository should be set up, but essentially, each team needs to have the following:

- Workspace. Part of the repository where the team will upload/maintain project-related code.
- Team Wiki Page. Will contain team information and links to documents produced throughout the project.

The wiki page should contain the team name and the list of team members.

FLOPPY logo. Come up and create a drawing/rendering of FLOPPY logo. Put an image of your logo on your team's Wiki page.

FLOPPY backronym. Come up with a backronym for `FLOPPY`. Your backronym should be somewhat relevant to the nature of FLOPPY and shall be **PG**-rated.

JSON Parser

There is a good selection of JSON parsers for C. The list is available at

http://www.json.org

For this lab, and for future use, you can pick any of the parsers. I rely on each team's expertise with C and software development to (a) select and (b) succesfully use the JSON parser of your choice. Please note, that you will need, most likely, to include the source code for the parser (or the `.o` files) in your eventual FLOPPY distribution.

Overview

For this lab each team will create a collection of JSON generators written in C, and a C program that parses input JSON objects and converts them into relational tuples (outputed in the form of a collection of CSV files).

First, some theory.

JSON

JavaScript Object Notation, or JSON is an easy-to-understand, human-readable format for storing collections of key-value pairs (a.k.a., dictionaries) and for passing them from one application to another.

A somewhat simplified description of the JSON data format is as follows (we are using Backus-Naur notation here):

```
<JSONObject> ::= '{' '}' |
                 '{' <key> ':' <value>
                    (',' <key> ':' <value>)* '}'

<key> ::= <string>

<value> ::= <JSONObject>|
            <array>     |
            <string> | <number> | true | false | null

<array> ::= '[' ']' |
            '[' <value> (',' <value>)* ']'
```

In a nutshell, a JSON object is a collection of zero or more comma-separated key-value pairs enclosed in curly braces. Keys are strings (all JSON strings must be enclosed in double quotes), values can be either full JSON objects, JSON arrays, or atomic values. Atomic values can be strings, numbers, or constants `true, false` or `null`. A JSON array is a sequence of zero or more comma-separated values enclosed in square brackets.

The fact that JSON objects can contain other JSON objects as values of keys, or as components of arrays (which in turn are values of keys) makes JSON a recursive data format.

A **proper JSON file** is a file containing either a single JSON object, or a file containing an array of JSON objects. Additionally, input files that contain multiple JSON objects, not enclosed in square brackets (and possibly not comma-separated) are possible. You need to check with the JSON parser you chose what types of input files it accepts.

Example. Here is an example of a JSON object containing all features of JSON syntax:

```
{
 "id" : 2,
 "name": { "first": "Bryce",
           "last": "Gordon"
         },
 "classes" : ["CSC 101", "CSC 102", "CSC 103"],
 "status" : "sophomore",
 "notes" : [ {"date": "August 3", "content" : "need to buy cookies", "completed": true},
             {"date": "August 5", "content" : "full moon tooday"},
             {"date": null, "content": "laundry day",  "completed": "false"}
           ]
}
```

Converting JSON Objects into Relational Tables on the Fly

For this assignment, you will have to implement the procedure for converting a collection of JSON objects into a collection of relational tables described below.

The conversion procedure assumes the following about the input JSON object collection.

- **Multiple Objects.** The collection consists of multiple objects.
- **Object format.** All JSON objects in the collection describe similar entities. At the same time, the actual format of individual JSON objects may vary from object to object: individual objects may omit some key-value pairs other objects possess, for example.
- **First Object Rule.** We assume that the first JSON object in the input collection that you encounter contains **all possible key-value pairs.** All following objects may omit some of the key-value pairs, but no object after the first object may contain a key-value pair (either at the top level, or as part of an embedded JSON object) that is not found in the first object. (This restriction will allow you to infer the schema of all tables you need to create based on the first JSON object you read. This is a *very simplifying assumption on our part*.
- **Value match.** The values of the keys with the same name will always be of compatible types (largely this means that we won't have an atomic value assigned to a key in one object, and a JSON object assigned to that key in a different object).
- **Id.** Each JSON object in a collection will have a key-value pair for a key named "id". The value of this key will be unique in the collection.

The conversion procedure operates as follows.

Conversion of flat JSON objects. If all key-value pairs in the JSON objects in a collection are atomic values, then the relational representation of the collection is a single relational tables that has as attributes all the keys found in the JSON collection. The id attribute will be the *primary key* of the tables.

The relational table can be named after the name of the file in which the JSON collection is stored. Tuples in the table can have missing values.

Conversion Example 1. Consider the JSON collection from file `students.json` shown below.

```
{
 "id":1,
 "name": "John Smith",
 "major": "CS",
 "gpa": 3.5
}
{
 "id":2,
 "name": "Mary Brown",
 "major": "CS",
}
```

This collection is converted to a relational table Students, which can be represented, as a CSV file as follows:

```
id, name, major, gpa
1, "John Smith", "CS", 3.5
2, "Mary Brown", "CS",
```

Conversion of embedded JSON objects. A JSON collection where certain key-value pairs have embedded JSON objects as values gets converted into a relational database as follows:

- Following the conversion of the flat JSON objects procedure, a relational table is created for all key-value pairs of the outer JSON objects that have atomic values. The `id` attribute will the be *primary key* of this table, and the table name will follow the name of the file containing the JSON collection.
- For each key that has embedded JSON objects as values, create a relational table named after the key. This table will contain a *foreign key* to the original table (the id key-value pair), which will also serve as its primary key. All other attributes of this table will be the keys of the embedded JSON object. If any values of such keys are embedded objects themselves, **recurse** this construction.

Conversion Example 2. Consider the JSON collection from file `students.json` shown below.

```
{
 "id":1,
 "name": {"first:"John", "last": Smith"},
 "major": "CS",
 "gpa": {"overall": 3.5, "CalPoly":3.4}
}
{
 "id":2,
 "name": {"first":"Mary", "last": "Brown"},
 "major": "CS",
 "gpa" : {"overall": 3.3}
}
```

This collection can be represented as three relational tables: Students, StudentsName, Students-Gpa[1] as follows:

[1] Yeah, I know, grammar...

```
Studens.csv:
id, major
1, "CS"
2, "CS"

StudentsName.csv:
StudentsId, first, last
1, "John", "Smith"
2, "Mary", "Brown"

StudentsGpa.csv:
StudentsId, overall, CalPoly
1, 3.5, 3.4
1, 3.3,
```

Conversion of arrays. A collection of JSON objects where a value for a certain key is a JSON array gets reoresented as a relational database as follows.

- Following the conversion of the flat JSON objects procedure, a relational table is created for all key-value pairs of the outer JSON objects that have atomic values. The `id` attribute will the be *primary key* of this table, and the table name will follow the name of the file containing the JSON collection.
- For each key that has a JSON array as a value, a new table, named after the key is created. The new table will have the following attributes:
  - **Atomic array values.** If all values in the array are atomic, then the schema for the table is:
    Id: the id attribute of the parent object.
    Position: the position of the current value in the array (1,2,3,...)
    Value: the value at the given position
  - **Embedded JSON objects as array values.** If the values of the array are embedded JSON objects, then the relational table will have the Id and Position attribute as specified above, with the remaining attributes determined by the conversion of embedded JSON objects process described above.

The *primary key* of the table representing the array values will be the pair Id, Position.

Conversion Example 3. Consider the JSON collection from file `students.json` shown below.

```
{
 "id":1,
 "name": {"first:"John", "last": Smith"},
 "major": "CS",
 "courses": ["CPE 101", "CPE 102", "CPE 102"]
 "gpas": [{"quarter":"spring", "gpa":3.0}, {"quarter":"fall", "gpa":3.5}]
}
{
 "id":2,
```

```
"name": {"first":"Mary", "last": "Brown"},
"major": "CS",
"courses": ["CPE 101", "STAT 312", "BUS 101"]
"gpas": [{"quarter":"spring", "gpa":4.0}]
```

This collection can be represented as four relational tables: Students, StudentsName, StudentsCourses, and StudentsGpas as follows:

```
Studens.csv:
id, major
1, "CS"
2, "CS"

StudentsName.csv:
StudentsId, first, last
1, "John", "Smith"
2, "Mary", "Brown"

StudentsCourses.csv:
StudentsId, position, value
1, 1, "CPE 101"
1, 2, "CPE 102"
1, 3, "CPE 103"
2, 1, "CPE 101"
2, 2, "STAT 312"
2, 3, "BUS 101"

StudentsGpa.csv:
StudentsId, position, quarter, gpa
1, 1, "spring", 3.0
1, 2, "fall", 3.5
2,1, "spring", 4.0
```

JSON Generators

As part of this lab, you will implement four JSON generators. Each JSON generator shall generate JSON objects describing a meaningful data collection. Examples of meaningful data collections include, but are not limited to:

- A music CD catalog. (each JSON object describes the contents of one CD)
- A collection of course rosters. (each JSON object describes the information, including the list of students, taking one course)
- A collection of restaurant menus. (each JSON object describes one restaurant menu. Alternatively, each JSON object describes one dish and a JSON collection describes the menu of one restaurant)
- A collection of movie descriptions (each JSON object describes IMDB-style information about one movie)

- A collection of user accounts of an e-commerce store (each JSON object describes data for one customer complete with purchase history)

For each data format write a separate C program generating a given number (passed as a parameter) of JSON objects of this type.

Provide a `README` file describing all four data formats and explaining what data collections they represent.

JSON To Relational DB Converter

Write a C program that takes as input a JSON document collection (presented as a single file) and outputs a collection of CSV files representing the relational database encoded by the JSON document collection. The program shall implement the conversion algorithm described above.

Name your C program `Json2CSV.c`.

Submission instructions

Submit all your `.c`, `.h` files, as well as a `README` file and four JSON files - containing an example of *one* object in each data format that you generated as a single `.zip` or `.tar.gz` archive.

Use handin:

```
$ handin dekhtyar 468-lab1 <file>
```

Good Luck!