CIS 425, Fall 2020, Assignment 2

Important Introduction:

In this homework, we will work with JSON data in SML. We will manipulate it and print it. We will also play with some real data.

You need to download a .zip file containing several files from the course website, but the only files you should edit are `hw2.sml` and `hw2tests.sml`.

Your solutions must use pattern-matching. You may not use the functions `null`, `hd`, `tl`, `isSome`, or `valOf`, nor may you use anything containing a `#` character or features not used in class (such as mutation). Note that list order does not matter unless specifically stated in the problem.

This homework is more difficult than Homework 1. There are 19 required problems though several require just one line of code and none require more than 25 lines of code.

JSON is a human-readable data serialization format, originally inspired by Javascript's Object Notation. JSON is commonly used to represent data, for example Seattle Police Department crime report data near the U-district used in this assignment.

No previous experience with JSON is expected or needed, but if you are curious or confused, the ultimate reference is at `https://tools.ietf.org/html/rfc7159`. We will make some modest simplifying assumptions about JSON. Therefore, we will not quite be RFC compliant, but we will be close, and it wouldn't be too hard to get there.

A JSON value is one of seven things:
1. a number (floating point, e.g., `3.14` or `-0.2`)
2. a string (e.g., `"hello"`)
3. false
4. true
5. null
6. an array of JSON values, written between square brackets and commas (e.g., `[1, "world", null]`)
7. an "object," which is a sequence of field/value pairs
   - a field is always a string literal (e.g., `"foo"`)
   - a value is any JSON value
   - objects are written with curly braces, commas, and colons, e.g.,
     `{ "foo" : 3.14, "bar" : [1, "world", null], "ok" : true }`

However, our ML code will not use this *concrete* JSON notation, but rather use this `datatype` binding to represent JSON values:

```
datatype json =
         Num of real
       | String of string
       | False
       | True
       | Null
       | Array of json list
       | Object of (string * json) list
```

Remember that for any problem, if a type is required, then a *more general* type is acceptable provided your code is correct.

About the data files:

The crime-data files are needed only for problems 10–15, but you may find them useful for testing in other places as well. In case you are curious, the data is real and comes from the Seattle Police Department, it was originally retrieved via Socrata (www.socrata.com), but may no longer be available. For use in this assignment, we have selected all records in the dataset that took place within 1km of the Paul G. Allen Center for Computer Science & Engineering. The resulting data can be found in JSON format in the file `complete_police.json`.

Since that file contains over 10000 records, we have also included small, medium, and large subsets of the data, containing 10, 100, and 1000 records, respectively, available in the files `small_police.json`, `medium_police.json`, and `large_police.json`.

Converting the textual representation of JSON data into the structured representation given by the SML type `json` is the problem of parsing.

To facilitate playing with the data without needing to parse it, we have included the pre-parsed data files `parsed_small_police.sml`, `parsed_medium_police.sml`, and `parsed_large_police.sml`, each of which is a valid SML file that binds a single variable name to a value of type `json`. Thus they can be loaded into the REPL or other files with `use` function, as usual. Unfortunately, the complete dataset is too large for SML to load in a reasonable amount of time when it is represented like this, so it is not available in the pre-parsed format.

We will explore the data further below, but for now it suffices to say that it consists of a JSON array of event records, each of which has several fields, such as the kind of event (e.g., shoplifting or noise complaint) and the location (e.g., 4500 block of 15th avenue NE).

The Actual Problems

1. Write a function `make_silly_json` that takes an int `i` and returns a `json`. The result should represent a JSON array of JSON objects where every object in the array has two fields, `"n"` and `"b"`. Every object's `"b"` field should hold true (i.e., `True`). The first object in the array should have a `"n"` field holding the JSON number i.0, the next object should have an `"n"` field holding (($i-1$).0 and so on where the last object in the array has an `"n"` field holding 1.0. If `i` is 0, produce an array with 0 objects; assume `i` is not negative. Sample solution is less than 10 lines. Hints: We have provided a function `int_to_real`. You'll want a helper function that does most of the work.

2. Write a function `assoc` of type `''a * (''a * 'b) list -> 'b option` that takes two arguments `k` and `xs`. It should return `SOME v1` if `(k1,v1)` is the pair in the list closest to the beginning of the list for which `k` and `k1` are equal. If there is no such pair, `assoc` returns `NONE`. Sample solution is a few lines.

3. Write a function `dot` that takes a `json` (call it `j`) and a `string` (call it `f`) and returns a `json option`. If `j` is an object that has a field named `f`, then return `SOME v` where `v` is the contents of that field. If `j` is not an object or does not contain a field `f`, then return `NONE`. Sample solution is 4 short lines thanks to an earlier problem.

4. Write a function `one_fields` that takes a `json` and returns a `string list`. If the argument is an object, then return a list holding all of its field *names* (not field *contents*). Else return the empty list. Use a tail-recursive, locally defined helper function. It is easiest to have the result have the strings in reverse order from how they appear in the object and this reverse order is fine/expected. Sample solution is about 11 lines.

5. Write a function `no_repeats` that takes a `string list` and returns a `bool` that is true if and only if no string appears more than once in the input. Do *not* (!) use any explicit recursion. Rather, use helper functions `length` (pre-defined in SML) and `dedup` (provided earlier in the file) to complete this problem in one line.

6. Write a function `recursive_no_field_repeats` that takes a `json` and returns a `bool` that is true if and only if no object anywhere "inside" (arbitrarily nested) the `json` argument has repeated field

names. (Notice the proper answer for a `json` value like `False` is `true`. Also note that it is not relevant that different objects may have field names in common.) In addition to using some of your previous functions, you will want two locally defined helper functions for processing the elements of a JSON array and the contents of a JSON object's fields. By defining these helper functions locally, they can call `recursive_no_field_repeats` in addition to calling themselves recursively. Sample solution is 16 lines.

7. Write a function `count_occurrences` of type `string list * exn -> (string * int) list`. If the `string list` argument is sorted (in terms of the ordering implied by the `strcmp` function we provided you), then the function should return a list where each string is paired with the number of times it occurs. (The order in the output list does not matter.) If the list is not sorted, then raise the `exn` argument. Your implementation should make a single pass over the `string list` argument, primarily using a tail-recursive helper function. You will want the helper function to take a few arguments, including the "current" string and its "current" count. Sample solution is 14 lines.

8. Write a function `string_values_for_field` of type `string * (json list) -> string list` (the parentheses in this type are optional, so the REPL won't print them). For any object in the `json list` that has a field equal to the `string` and has *contents* that are a JSON string (e.g., `String "hi"`) put the contents string (e.g., `"hi"`) in the output list (order does not matter; the output should have duplicates when appropriate). Assume no single object has repeated field names. Sample solution is 6 lines thanks to `dot`.

9. Write a function `filter_field_value` of type `string * string * json list -> json list`. The output should be a subset of the third argument, containing exactly those elements of the input list that have a field with name equal to the first string argument and *that field's* contents are a JSON string equal to the second string argument. Sample solution uses `dot` and is less than 10 lines.

The next six problems have you create *variable bindings*. Use either `histogram_for_field` or `filter_field_value` as appropriate for analyzing the data in `large_incident_reports_list` — none of these problems require very many lines of code.

10. Bind to the variable `large_event_clearance_description_histogram` a histogram (using `histogram_for_field`) of the objects in `large_incident_reports_list` based on the `"event_clearance_description"` field.

11. Bind to the variable `large_hundred_block_location_histogram` a histogram (using `histogram_for_field`) of the objects in `large_incident_reports_list` based on the `"hundred_block_location"` field.

12. Bind to the variable `forty_third_and_the_ave_reports` a `json list` containing the elements of `large_incident_reports_list` whose `"hundred_block_location"` field contains the JSON string `"43XX BLOCK OF UNIVERSITY WAY NE"`.

13. Bind to the variable `forty_third_and_the_ave_event_clearance_description_histogram` a histogram based on the `"event_clearance_description"` field (like in problem 10), but considering only the objects whose `"hundred_block_location"` field contains `"43XX BLOCK OF UNIVERSITY WAY NE"` (as in problem 12).

14. Bind to the variable `nineteenth_and_forty_fifth_reports` a `json list` containing the elements of `large_incident_reports_list` whose `"hundred_block_location"` field contains the JSON string `"45XX BLOCK OF 19TH AVE NE"`.

15. Bind to the variable `nineteenth_and_forty_fifth_event_clearance_description_histogram` a histogram based on the `"event_clearance_description"` field (like in problem 10), but considering only the objects whose `"hundred_block_location"` field contains `"45XX BLOCK OF 19TH AVE NE"` (as in problem 14).

In the next four problems we will convert from `json` to `string`, the key step in printing JSON values.

16. Write a function `concat_with` that takes a separator string and a list of strings, and returns the string that consists of all the strings in the list concatenated together, separated by the separator. The separator should be only *between* elements, not at the beginning or end. Use ML's `^` operator for concatenation (e.g., `"hello" ^ "world"` evaluates to `"helloworld"`). Sample solution is 5 lines.

17. Write a function `quote_string` that takes a `string` and returns a `string` that is the same except there is an additional `"` character at the beginning and end. Sample solution is 1 line.

18. Write a function `real_to_string_for_json` that takes a `real` and returns a `string`. You can't *quite* just use the `real_to_string` function we provided because in ML negative numbers start with `~`, but we want the more common `-` character. But we also provided `real_abs`. Sample solution is 1–2 lines.

19. Write a function `json_to_string` that converts a `json` into the proper string encoding in terms of the syntax described on the first page of this assignment. The three previous problems are all helpful, but you will also want local helper functions for processing arrays and objects (hint: in both cases, create a `string list` that you then pass to `concat_with`). Sample solution is 25 lines.

Summary

Evaluating a correct homework solution should generate these bindings *or more general types* in addition to the bindings from the code provided to you.

```
val make_silly_json = fn : int -> json
val assoc = fn : ''a * (''a * 'b) list -> 'b option
val dot = fn : json * string -> json option
val one_fields = fn : json -> string list
val no_repeats = fn : string list -> bool
val recursive_no_field_repeats = fn : json -> bool
val count_occurrences = fn : string list * exn -> (string * int) list
val string_values_for_field = fn : string * json list -> string list
val filter_field_value = fn : string * string * json list -> json list
val large_event_clearance_description_histogram = (*...*) : (string * int) list
val large_hundred_block_location_histogram = (*...*) : (string * int) list
val forty_third_and_the_ave_reports = (*...*) : json list
val forty_third_and_the_ave_event_clearance_description_histogram =
         (*...*) : (string * int) list
val nineteenth_and_forty_fifth_reports = (*...*) : json list
val nineteenth_and_forty_fifth_event_clearance_description_histogram =
         (*...*) : (string * int) list
val concat_with = fn : string * string list -> string
val quote_string = fn : string -> string
val real_to_string_for_json = fn : real -> string
val json_to_string = fn : json -> string
```

Of course, generating these bindings does not guarantee that your solutions are correct. *Test your functions: Put your testing code in a separate file. We will not grade it, but you must turn it in.*

Assessment

Your solutions should be correct, in good style, and use only features we have used in class.