CSE 341, Autumn 2020, Homework 2
Due: Friday October 23, 5:59PM

In this homework, we will work with JSON data in OCaml. We will manipulate it, print it, and (optionally, as a challenge problem) parse it. We will also play with some real data.

You need to download the starter code including several data files from the course website. Turn-in instructions are at the bottom of this document.

Your solutions must use pattern-matching. You may not use the functions `List.hd`, `List.tl`, `Option.is_some`, or `Option.get`, nor may you use any features not used in class.

Note that list order does not matter unless specifically stated in the problem.

This homework is more difficult than Homework 1. There are 26 required problems and 12 challenge problems, though several (both required and challenge) require just one line of code and none require more than 25 lines of code.

Background: JSON

JSON is a human-readable data serialization format, originally inspired by Javascript's Object Notation. JSON is commonly used to represent data, for example the King County Metro real-time transit data 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 OCaml code will not use this *concrete* JSON notation, but rather use this datatype binding to represent JSON values:

```
type json =
    | Num of float
    | 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 bus-data files are needed only for problems 21 through 26, 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 King County Metro. The dataset is real-time and is the same feed used by Google Maps and OneBusAway to display route timing information. The most recent data can be viewed at `https://s3.amazonaws.com/kcm-alerts-realtime-prod/vehiclepositions_pb.json`. Note that this data updates every few seconds. We provide a recent snapshot so that we're all on the same page. This snapshot can be found in JSON format in the file `complete_bus.json`.

Since that file contains around 1000 records, we have also included small and medium subsets of the data, containing 10 and 100 records, respectively, available in the files `small_bus.json` and `medium_bus.json`.

Converting the textual representation of JSON data into the structured representation given by the OCaml type `json` is the problem of parsing. Parsing JSON is an interesting problem, which we encourage you to explore (it's easier than you might think!) with the challenge problems.

To facilitate playing with the data without needing to parse it, we have included the pre-parsed data files `parsed_small_bus.ml`, `parsed_medium_bus.ml`, and `parsed_complete_bus.ml`, each of which is a valid OCaml 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 OCaml's `#use` directives, as usual.

We will explore the data further below, but for now it suffices to say that it consists of a JSON object that contains an array of vehicle records, each of which has several fields, such as the route name and number, and the position in latitude and longitude.

Part 0: Warm-up

This first problem will have you construct a silly, useless JSON value to get you used to working with the `json` datatype.

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 (in other words, the integer `i` converted to a floating point JSON number), 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. Sample solution is less than 10 lines. Hints: There's a function in OCaml's standard library called `float_of_int` that converts an integer to a float. You'll want a helper function that does most of the work.

Part 1: Printing JSON values

In the next three problems, you will write two useful helper functions, then finally, write a function to convert from JSON to string, the key step in printing JSON values

2. 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 OCaml's `^` operator for concatenation (e.g., `"hello" ^ "world"` evaluates to `"helloworld"`). Sample solution is 5 lines.

3. 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.

4. Write a function `string_of_json` that converts a `json` into the proper string encoding in terms of the syntax described on the first page of this assignment. The two previous problems are both 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`). In the `Num` case, use the provided `json_string_of_float` function. Sample solution is 25 lines.

Part 2: Processing JSON values

Next, you will write 14 functions (and 2 comments!) to allow you to manipulate and process JSON values that have already been parsed. These functions will allow us to analyze the JSON data in the next section.

5. Write a function `take` of type `int * 'a list -> 'a list` that takes an `int` called `n` and a list called `l` and returns the first `n` elements of `l` in the same order as `l`. You may assume that `n` is non-negative and does not exceed the length of `l`. Sample solution is about 5 lines.

6. Write a function `firsts` of type `('a * 'b) list -> 'a list` that takes a list of pairs and returns a list of all the first components of the pairs in the same order as the argument list. Sample solution is about 4 lines.

7. Write a *comment* in your file after your definition of `firsts` answering the following questions. Suppose `l` has type `(int * int) list`, and let `n` be an integer between 0 and the length of `l` (inclusive), and consider the expressions `firsts (take (n, l))` and `take (n, firsts l)`. Either (1) write one sentence explaining in informal but precise English why these two expressions always evaluate to the same value; or (2) give example values of `l` and `n` such that the two expressions evaluate to different values. Regardless of whether you decide option (1) or option (2) is correct, also write one sentence explaining which of the two expressions above might be faster to evaluate and why.

8. 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. (Note: There's a function with similar functionality in the OCaml standard library, but calling it requires features we haven't covered yet. Do not use that function or you will not receive credit.)

9. 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.

10. Write a function `dots` that takes a `json` called `j` and a `string list` called `fs` that represents an *access path*, or in other words, a list of field names. The function `dots` returns a `json option` by recursively accessing the fields in `fs`, starting at the beginning of the list. If any of the field accesses occur on non-objects, or to fields that do not exist, return `None`. Otherwise, return `Some v` where `v` is the value of the field "pointed to" by the access path. (Hint: Use recursion on `fs` plus your solution to the previous problem.) Sample solution is about 7 lines.

11. 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. The list you return can be in any order, but it is probably easiest to have the results in reverse order from how they appear in the object, and this reverse order is fine/expected. Sample solution is about 10 lines.

12. 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 provided helper function `dedup` (which returns its argument without duplicates) together with standard library function `List.length` to complete this problem in one line.

13. 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, rather than at the top level, they can call `recursive_no_field_repeats` in addition to calling themselves recursively. Sample solution is about 15 lines.

14. Write a function `count_occurrences` of type `string list * exn -> (string * int) list`. If the `string list` argument is sorted (using OCaml's built-in comparison operator, `<`), 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 about 12 lines.

15. Write a function `string_values_for_access_path` of type

    `(string list) * (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 available via the given "access path" (`string list`), 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). Sample solution is 6 lines thanks to `dots`.

16. Write a function `filter_access_path_value` of type

    `string list * 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 available via the given access path, and *that field's* contents are a JSON string equal to the second argument. Sample solution uses `dots` and is less than 10 lines.

17. Some of the bus data uses latitude and longitude positions to describe the location of vehicles in real time. To narrow our focus onto a particular geographical area, we will use the record types `rect` and `point`, which represent a rectangle and a point, respectively. The types are defined in the starter code, but copied here for completeness.

    ```
    type rect = { min_latitude: float; max_latitude: float;
                  min_longitude: float; max_longitude: float }
    type point = { latitude: float; longitude: float }
    ```

    Write a function `in_rect` of type `rect * point -> bool` that determines whether a given point falls inside a given rectangle (inclusive). Solution is two lines and uses a lot of conjunction (`&&`).

18. Write a function `point_of_json` of type `json -> point option`. If the argument is a `json` object that contains fields named `"latitude"` and `"longitude"`, both of which are `json` numbers, then `point_of_json` returns `Some p` where `p` is the point represented by these coordinates. Otherwise, it returns `None`. Solution is 5 lines and uses `dot` and nested patterns.

19. Write a function `filter_access_path_in_rect` of type

    `string list * rect * json list -> json list`.

    The output should be a subset of the third argument, containing exactly those elements of the input list that (1) have a field available via the given access path, (2) *that field's* contents are a JSON object that can be converted to a `point` via `point_of_json`, and (3) the resulting `point` is within the rectangle specified by the second argument. Sample solution is less than 15 lines.

20. After your definition of `filter_access_path_in_rect`, write a comment containing 1-3 sentences describing the similarities with `filter_access_path_value`. Can you think of any way to refactor these two function to use a common, more general function? (Do not actually do this refactoring.) On a scale from 1 to "run-time error", how annoyed are you about having to write both of these functions on the same homework?

Part 3: Analyzing the data

Finally, we will use the functions you wrote in the part 2 to create some variable bindings to help us analyze the JSON data. We will analyze the complete subset of the data stored in `complete_bus_positions_list`, which you will need to uncomment in the provided code. All of the problems in part 3 have one-line solutions and use one or more functions defined in previous parts.

21. Bind to the variable `route_histogram` a histogram (using `histogram_for_access_path`) of the objects in `complete_bus_positions_list` based on the `"route_num"` field. (Hint: since this field is nested inside several objects, you need to look at the data to figure out the rest of the access path that comes before this field.)

22. Bind to the variable `top_three_routes` a list containing the three most frequently appearing route numbers in `route_histogram`. (Hint: use `take` and another function from part 2.)

23. Bind to the variable `buses_in_ud` a list containing all records referring to buses in the U district. For the purposes of this problem, the U district is defined by the rectangle bound to the variable `u_district` in the provided code.

24. Bind to the variable `ud_route_histogram` a histogram of the objects in `buses_in_ud` based on the `"route_num"` field, as in problem 21.

25. Bind to the variable `top_three_ud_routes` a list containing the three most frequently appearing route numbers in `ud_route_histogram`, as in problem 22.

26. Bind to the variable `all_fourty_fours` a list containing all records from `complete_bus_positions` that refer to a vehicle whose (suitably nested, as in problem 21) `route_num` field is `"44"`.

The remaining problems are challenge problems that build to a parser for JSON. They are described in `hw2challenge.ml`.

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 : int -> json = <fun>
val concat_with : string * string list -> string = <fun>
val quote_string : string -> string = <fun>
val string_of_json : json -> string = <fun>
val take : int * 'a list -> 'a list = <fun>
val firsts : ('a * 'b) list -> 'a list = <fun>
val assoc : 'a * ('a * 'b) list -> 'b option = <fun>
val dot : json * string -> json option = <fun>
val dots : json * string list -> json option = <fun>
val one_fields : json -> string list = <fun>
val no_repeats : 'a list -> bool = <fun>
val recursive_no_field_repeats : json -> bool = <fun>
val count_occurrences : 'a list * exn -> ('a * int) list = <fun>
val string_values_for_access_path : string list * json list -> string list = <fun>
val filter_access_path_value : string list * string * json list -> json list = <fun>
val in_rect : rect * point -> bool = <fun>
val point_of_json : json -> point option = <fun>
val filter_access_path_in_rect : string list * rect * json list -> json list = <fun>
val route_histogram : (string * int) list = (* ... *)
val top_three_routes : string list = (* ... *)
val buses_in_ud : json list = (* ... *)
val ud_route_histogram : (string * int) list = (* ... *)
val top_three_ud_routes : string list = (* ... *)
val all_fourty_fours : json list = (* ... *)
```

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.

Turn-in Instructions

Put your solutions to the problems in `hw2.ml`. Put your additional tests in `hw2tests.ml`. Put your solutions to the challenge problems in `hw2challenge.ml`. Upload these files to Gradescope.