PPA-S25 hw3

{"he][o":[{"wo{ld":1},[1,[2,3],4]]}

Logistics

This homework consists of 100 points total, with points for individual tasks as indicated below. There are 2 coding tasks in this assignment.

You should submit your work on Brightspace.

Package your solutions as follows:

```
# replace NYU_NET_ID with your ID, e.g., shw8119.tgz
$ tar czf NYU_NET_ID.tgz JsonFindStrings.sml JsonNesting.sml
```

The resulting `.tgz` archive should contain only `JsonFindStrings.sml` and `JsonNesting.sml`. To submit, upload this `.tgz` file.

Due Date: This submission is due at **5:00pm EST on Tuesday, Feb 18**. Course policy on late submissions is available on the course website. (https://cs.nyu.edu/~shw8119/courses/s25/3033-121-ppa/)

Setup

- Follow the instructions on the course website to access one of the `crunchy` compute servers.
- Install MaPLe. For the `crunchy` servers, you can:
  - Download https://cs.nyu.edu/~shw8119/courses/s25/3033-121-ppa/resources/mpl-v053.tgz and copy it to the server
  - Unpack by running `tar xzf mpl-v053.tgz`. This will create a directory `mpl-v053/`
  - The compiler, `mpl`, is located at `mpl-v053/bin/mpl`
  - Add `mpl-v053/bin` to your `PATH` so that you can access it easily: `export PATH="$(pwd -P)/mpl-v053/bin:$PATH"`. We recommend updating your `~/.bashrc` file or other configuration file as appropriate.

JSON Rainbows

JSON files are ubiquitous in modern computing. With billions of devices across the globe, it is safe to assume that literally *billions* of JSON parsers are currently running right now, as you are reading this.

Commonly, JSON is parsed sequentially, but it is possible to parse large JSON files in parallel. In this assignment, you will develop a parallel JSON parsing algorithm. The goal is not to produce the fastest parser in practice, but rather

to understand some of the algorithmic techniques that are useful for parallel parsing.

(Actually, the real goal of the assignment is to flex your understanding of **parallel prefix sums** via the `scan` primitive.)

We'll focus on two subproblems in particular:

- Finding where all of the strings in a JSON file begin and end.
- Identifying the *bracket nesting depth* of every curly and square bracket in a JSON file. We'll use this to assign "rainbow colors" to the brackets of a file, where matching brackets are assigned the same color, as modern code editors and IDEs typically do.

Marking Strings

Task 1 (50 points): In `JsonFindStrings.sml`, implement the function `in_string_flags(chars)`, where `chars` is a sequence of characters from a valid JSON file. The output should be a sequence of booleans, indicating for each character of the input whether or not that character is part of a JSON string. You should include the indices of the start and end quotes.

On an input of length N, your solution to Task 1 must have $O(N)$ work and $O(\mathrm{polylog}(N))$ span.

Consider the input `{"hello":[{"world":1},[1,["2",3],4]]}`. The output of `in_string_flags` should be as follows, where 1 means `true` and 0 means `false`. Notice that each character of a string is marked with a run of `true` booleans, and all other characters are marked `false`.

```
input:    {"hello":[{"world":1},[1,["2",3],4]]}
output:   0111111100011111110000000011100000000
```

Every JSON string begins and ends with a double quote character: ". Inside of a JSON string, there can be escape sequences. Valid JSON escape sequences are:

```
\"  \\  \/  \b  \f  \n  \r  \t  \uXXXX   (where X is a hex digit)
```

The tricky ones to handle in your solution are going to be the escape sequences `\"` and `\\`. Any escaped quote character does *not* terminate the string.

It might be tempting to check if a quote character is escaped simply by checking if the character immediately preceding the quote is a backslash. However, it's not quite so simple, because the backslash might itself have been escaped! For example, all of these are valid JSON strings:

```
"\""
"\\"
"\\\""
"\\\\\\\"\\\\\\"
```

Notice that, in any contiguous run of backslashes, it is important whether or not the length is odd or even.

HINT. For any contiguous run of backslashes, consider computing the index of where that run of backslashes began.

```
    chars: " \ \ \ " _ \ \ _ "
    index: 0 1 2 3 4 5 6 7 8 9
run starts: . 1 1 1 . . 6 6 . .
```

Also, an important property of JSON is that backslashes can only appear inside of strings. This just happens to be true of the JSON grammar; backslashes aren't permitted elsewhere. More info about the JSON grammar can be found here: https://www.json.org/json-en.html

As usual, you will need functions on sequences:

```
Seq.length: 'a Seq.t -> int      (* O(1) work, O(1) span *)
Seq.nth: 'a Seq.t -> int -> 'a   (* O(1) work, O(1) span *)
```

We also encourage using functions such as the following:

```
Parallel.tabulate:
  (int * int) -> (int -> 'a) -> 'a Seq.t

Parallel.reduce:
  ('a * 'a -> 'a) -> 'a -> (int * int) -> (int -> 'a) -> 'a

(* note: output sequence of scan is length N+1 *)
Parallel.scan:
  ('a * 'a -> 'a) -> 'a -> (int * int) -> (int -> 'a) -> 'a Seq.t

(* `filter (lo, hi) f p` computes the sequence
 * `[f(i) : lo <= i < hi for each i satisyfing p(i)]`
 * i.e., p(i) says whether or not to include f(i) in the output *)
Parallel.filter:
  (int * int) -> (int -> 'a) -> (int -> bool) -> 'a Seq.t
```

Assuming the functions **f**, **g**, and **p** all require $O(1)$ work, you may assume the following cost bounds for these functions:

```
tabulate (lo, hi) f     (* O(hi-lo) work, O(log(hi-lo)) span *)
reduce g z (lo, hi) f   (* O(hi-lo) work, O(log(hi-lo)) span *)
scan g z (lo, hi) f     (* O(hi-lo) work, O(log(hi-lo)) span *)
filter (lo, hi) f p     (* O(hi-lo) work, O(log(hi-lo)) span *)
```

Time for Rainbows

Task 2 (50 points): In the file **JsonNesting.sml**, implement the function **bracket_depths chars string_flags**, where **chars** are characters from a valid

JSON file, and `string_flags` are flags indicating whether or not each character is part of a JSON string. That is, you can assume that the input `string_flags` is a correct result from the first task.

(Tasks 1 and 2 will be graded separately. Your grade on Task 2 will not depend on the correctness of your solution to Task 1.)

On inputs of length N, your solution to Task 2 must have $O(N)$ work and $O(\mathbf{polylog}(N))$ span. You can assume the same costs as shown above for common library functions.

Your goal is to compute the nesting depth of every bracket in the JSON file. Brackets can either be curly or square:

```
{ } [ ]
```

The output should be a sequence of tuples (i, d_c, d_s), each of which corresponds to one bracket in the file, where:

- i is the index (in `chars`) of the bracket
- d_c is the "curly depth", i.e., the number of curly bracket pairs that surround index i (**not** including the bracket at position i)
- d_s is the "square depth", i.e., the number of square bracket pairs that surround index i (**not** including the bracket at position i)

For example, consider the input `{"he][o":[{"wo{ld":1},[1,[2,3],4]]}`. This input has 10 brackets:

```
string_flags:   01111111000111111100000000000000000
       chars:   {"he][o":[{"wo{ld":1},[1,[2,3],4]]}
```

The output for this example should as follows:

```
(0,  0, 0),    (* open { at index 0; no surrounding brackets *)
(9,  1, 0),    (* open [ at index 9; 1 surrounding curly pair *)
(10, 1, 1),    (* open { at index 10; 1 surrounding curly pair,
                * and 1 surrounding square pair *)
(20, 1, 1),    (* close } at index 20; 1 surrounding curly,
                * and 1 surrounding square *)
(22, 1, 1),    (* etc... *)
(25, 1, 2),    (* etc... *)
(29, 1, 2),
(32, 1, 1),
(33, 1, 0),
(34, 0, 0)
```

Notice that, in the example above, there are strings that contain bracket characters, but these are part of the string and should **not** be counted as actual brackets of the JSON file. (This is why we computed the `string_flags` separately; you

can refer to these booleans to figure out which characters are and are not inside of a string.)

Testing

You can test your code as follows.

```
$ make test
$ ./test
```

This will run your code on a few test JSON files inside of `test-json/`. Feel free to add more tests, and remember to add these to the list at the top of `test.sml` and then recompile.

For reference and to help with testing for correctness, there are reference solutions available inside of `ReferenceSolutions.sml`. Note that these solutions are entirely sequential and therefore do not meet the work and span requirements of Tasks 1 and 2.

See the Rainbow, Taste the Rainbow

Using the results of your `in_string_flags` and `bracket_depths` functions, we can parse a JSON file and assign colors to the strings and brackets. Brackets of the same nesting depth (here computed as `d_c + d_s`) will be assigned the same color.

Run the following to see the rainbow! (Hopefully, the terminal you are using supports colors!)

```
$ make main
$ ./main test-json/3.json
```

{"he][o":[{"wo{ld":1},[1,[2,3],4]]}