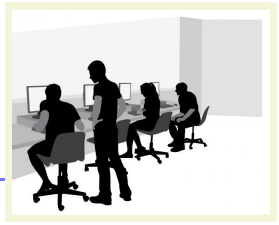
Lab 8 JSON



Purpose This lab aims to provide you a practical example of the usefulness of nested recursive data structures. Instead of dealing with the seemingly esoteric SExpr we'll be working with JSON.

Installing the Teachpack

This lab relies on the ability for us to parse JSON, and get JSON from the internet. Parsing JSON is *not* trivial, so instead of making you all figure that whole mess out yourself we're providing a teachpack that does it for you.

If you are interested in learning more about parsing, context free grammars, and simple web requests you should read through the source code of this teachpack.

To get the parser, download the teachpack and save it in the same folder as your source code for this lab. Then add the following line to the top of your source code.

```
(require "lab8-teachpack.rkt")
```

JSON

TAs should explain escaping strings briefly. It's unfortunately unavoidable, so students need to know that "\"" is the " string in Racket.

JSON is a lightweight data-interchange format. Originally developed as part of JavaScript, it is now one of the most popular formats for exchanging data on the web. Unlike many other formats, JSON aims to be easy for humans to read, making it nicer to work with.

To get an idea of what JSON is exactly, and how we're going to represent it in our code take a look at the official spec. We'll base our Data Definition off of this spec.

```
; A RawJSON is a String.
; This is the string containing actual JSON text as found in the wild.
; for example: {"foo":123} is valid JSON, so the string "{\"foo\":123}"
; in ISL is a valid RawJSON.
;
; Note: In order to have a " within a string in *SL we must escape it
;       with the backslash (\), so "\"" is the string ".

(define rjson1 "\"Hello World!\"")
(define rjson2 "[{\"name\":\"Nate\",\"email\":\"nathanl@ccs.neu.edu\"}]")
```

These strings are not very useful without some way to interpret them however. We need to "parse" them into some other data structure in

our language that we can deal with.

```
; A JSON is one of:
; - String
; - Number
; - Boolean
; - 'null
; - JSONArray
; - JSONObject

; A JSONArray is a [Listof JSON], representing an ordered
; collection of values.

; A JSONObject is a [Listof JSONPair], representing a set
; of named values, similar to a structure.

; A JSONPair is a (make-pair String JSON),
; representing a named value.
; (define-struct pair (name value))

; BASIC EXAMPLES
(define json-string1 "") ; Empty string
(define json-string2 "Hello World!")
(define json-string3 "CamBot")
(define json-string4 "Please mind the jump.")
(define json-number1 0)
(define json-number2 392699/125000)
(define json-number3 1/40)
(define json-value1 true)
(define json-value2 false)
(define json-value3 'null)

; ARRAY EXAMPLES
(define json-array1 empty) ; Empty array
(define json-array4 (list 1))
(define json-array2 (list 1 2 3 4 5))
(define json-array3 (list 1 "hi" false (list "foo" 10)))
(define json-array5 (list empty))
(define json-array6 (list (list 1 2 3)
                          (list 4 5 6)))

; OBJECT EXAMPLES
(define json-object1 empty) ; Empty object
(define json-object2 (list (make-pair "something" 10)))
(define json-object3 (list (make-pair "foo" 1337)
                           (make-pair "bar" "yo!")))
(define json-object4 (list (make-pair "life" 42)
                           (make-pair "death" 666)))
(define json-object5 (list (make-pair "cats" (list "Hobbes"
                                                   "Garfield"
                                                   "Mittens"))
                           (make-pair "dogs" (list "Snoopy"
                                                   "Spot"
                                                   "Moxie"))))
(define json-object6 (list (make-pair "foo" (list (make-pair "bar" 1)
                                                  (make-pair "baz" 'null)))
                           (make-pair "fizz" "buzz")))
```

You'll notice these definitions follow the spec very directly.

Parsing JSON

Parsing RawJSON is not trivial, and could make for an interesting project, however since that is not the focus of this lab we've written a basic RawJSON parser for you.

```
; string->json : RawJSON -> JSON

(string->json "[1,2,3]")
; => (list 1 2 3)
(string->json "{\"foo\":[1,2,3]}")
; => (list (make-pair "foo" (list 1 2 3)))
```

> **Exercise 1** Use `string->json` to define 3 of your own examples of some JSON data. Remember to escape strings.

Using JSON

Here's where you start having some fun with JSON. I've come up with most of these exercises out of tasks I've needed to do with JSON personally.

But before that, of course, we must...

> **Exercise 2** Write a template for JSON. Note that JSONObjects and JSONArrays both contain JSON, so you must write a template that is mutually recursive. Since the template is just a comment, feel free to assume you've already completed the next exercise.
>
> **Exercise 3** Write the functions `json-pair?`, `json-object?`, `json-array?`, and `json?` that when given any input return a Boolean. These functions should only return `true` when the given input is valid. Be sure to check the whole value (i.e. all values in an array).
>
> **Exercise 4** Design the function `json=?` that will determine if two JSONs are equal. Note that two JSONObjects are equal if all of their pairs are equal. For the sake of simplicity, you can assume order matters here.
>
> **Challenge** Design `json=?.v2` which doesn't care about the order of pairs in JSONObjects.

Ok, so now we know how to identify JSON, and even compare them, but what else can we do with it? JSON is just a way to hold data, so the real question is what data do we want to work with.

One set of data that is particularly interesting is weather data. I check

the weather at least a few times a day. Generally I'm looking at forecasts, however I rarely look at the historical data. It would be interesting to get an idea of what weather has actually been like in the past.

To get some data we need to find a service who will give it to us. These are called APIs. openweathermap.org happens to have an API which will give us RawJSON back. Perfect!

The teachpack conveniently gives you a function for getting the data from a url.

```
; get-url : String -> String

(get-url "http://api.openweathermap.org/data/2.5/history/city?q=Boston")
; {
;   "message" : "",
;   "cod" : "200",
;   "city_id" : 2655138,
;   "calctime" : 0.5875,
;   "cnt" : 36,
;   "list" : [
;     {
;       "main" : {
;         "temp" : 282.94,
;         "pressure" : 1014,
;         "humidity" : 93,
;         "temp_min" : 282.15,
;         "temp_max" : 284.15
;       },
;       "wind" : {
;         "speed" : 3.1,
;         "deg" : 350
;       },
;       "clouds" : {
;         "all" : 75
;       },
;       "weather" : [
;         {
;           "id" : 310,
;           "main" : "Drizzle",
;           "description" : "light intensity drizzle rain",
;           "icon" : "09n"
;         }
;       ],
;       "dt" : 1414554644
;     },
;
;     ...
;
;   ]
; }
```

Exercise 5 I cheated a little and formatted the resulting string so as to make it readable. Define `weather-data` as the result from the `get-url` function above.

Try not to duplicate calls to `get-url` since this function is downloading data

from the internet. It's "slow", saving results into definitions and using those is preferable.

Now for the fun stuff! We can start looking at this data and answering questions about. Keep in mind JSON is a recursive data structure, so these functions need to properly traverse the structure.

Exercise 6 Design a function `find-value` that takes a JSON and a String and returns the *first* JSON `pair-value` from a `make-pair` where the `pair-name` is equal to the given string.

The `find-value` function is going to be very useful in the remaining exercises. Be sure it works with lots of tests. Here are a few to get you started.

```
(check-expect (find-value "a" "a")
              false)
(check-expect (find-value (list (pair "a" 1)) "a")
              1)
(check-expect (find-value (list (pair "a" (list (pair "b" 2)))) "b")
              2)
(check-expect (find-value (list (pair "a" 1) (pair "b" 2)) "b")
              2)
(check-expect (find-value (list (pair "a" (list (list (pair "b" 2))))) "b")
              2)
(check-expect (find-value (list (pair "a" 1) (pair "a" 2)) "a")
              1)
```

Exercise 7 Design a function `max-temp` which, given a RawJSON string, returns the maximum temperature from the data as a Number.

Exercise 8 Design a function `average-humidity` which, given a RawJSON string, returns the humidity from the data as a Number.

Exercise 9 Design a function `current-weather` which given a RawJSON string returns a [Listof String] of all the "description"s of the current weather from the data for the latest data point.

For example running this function right now yields `(list "Sky is Clear")`.

Exercise 10 Challenge Problem Design the function `flatten-json` that will return a *[Listof (list String JSON)]*, but the `second` of any element of the list will not be a `cons`.

Flatten will work as follows:

If a JSON `j` is a String, Number, Boolean, `'null`, or `empty`, `(flatten-json j)` will return `(list (list X j))`, where X is one of `"STRING"`, `"NUMBER"`, `"BOOLEAN"`, `"NULL"`, or `"EMPTY"`, depending on what j is.

If it is a JSONObject, then the names of its pairs will be string-appended onto the names of the flattening of the associated value.

If a JSONArray, then the indices of its values will be string-appended onto the names of the flattening of the associated value.

When string appending, append `"-"` between the two string values as to separate one field name from another.*

The following check should pass:

```
(check-expect (flatten-json
                (list 'null
                      '()
                      (list (make-pair "greeting"
                                       (list (make-pair "formal" "hello")
                                             (make-pair "informal" "howdy")
                                             (make-pair "garbage" (list true 1)))))))
              (list (list "0-NULL" 'null)
                    (list "1-EMPTY" '())
                    (list "2-greeting-formal-STRING" "hello")
                    (list "2-greeting-informal-STRING" "howdy")
                    (list "2-greeting-garbage-0-BOOLEAN" true)
                    (list "2-greeting-garbage-1-NUMBER" 1)))
```

*using "-" will be an issue if the field names use "-". In the real world, when implementing this function, one would have to either manually look at the data to find a separating a string that would leave no ambiguity in the results, or write a function that given a JSON would find an appropriate separating string. A similar issue may arise if the field names are also just 0, 1, 2, etc, for when naming by index, but a similar workaround could be used by finding a usable name along the lines of 0INDEX, 1INDEX, 2INDEX, etc.

Exercise 11 Challenge Problem Design the function `weather-graph` which given a RawJSON and a String will return an Image. The image is a bar graph of each value of the given string.