180: JSON

by Amirouche Boubekki

Status

This SRFI is currently in *final* status. Here is an explanation of each status that a SRFI can hold. To provide input on this SRFI, please send email to `srfi-180@srfi.schemers.org`. To subscribe to the list, follow these instructions. You can access previous messages via the mailing list archive.

- Received: 2020-01-17
- Draft #1 published: 2020-01-17
- Draft #2 published: 2020-03-02
- Draft #3 published: 2020-06-20
- Finalized: 2020-07-01
- Revised to fix errata:
  - 2020-08-07 (Fix `json-generator` to accept generators as well as ports.)

Abstract

This library describes a JavaScript Object Notation (JSON) parser and printer. It supports JSON that may be bigger than memory.

Rationale

JSON is a *de facto* industry standard for data exchange.

For best interoperability, the sample implementation is based on RFC 8259, and the tests are based on JSONTestSuite.

The mapping between JSON types and Scheme objects is not trivial because a given mapping might not be the best for every situation. That is the reason why this library makes public the procedure `json-fold`, inspired by Oleg Kiselyov's `foldts`.

Specification

`(json-error? obj) → boolean`

Returns `#t` if `OBJ` is an error object that is specific to this library.

`(json-error-reason obj) → string`

Return a string explaining the reason for the error. This should be human-readable.

`(json-null? obj) → boolean`

Return `#t` if `OBJ` is the Scheme symbol `'null`, which represents the JSON `null` in Scheme. In all other cases, return `#f`.

`json-nesting-depth-limit` parameter

Parameter holding a number that represents the maximum nesting depth of JSON text that can be read by `json-generator`, `json-fold`, and `json-read`. If the value returned by this parameter is reached, the implementation must raise an error that satisfies `json-error?`.

The default value of `json-nesting-depth-limit` is `+inf.0`.

A proper value should be set on a per-application basis to mitigate the risks of denial-of-service attacks.

`json-number-of-character-limit` parameter

Parameter holding a number that represents the maximum number of characters for a given JSON text that can be read by `json-generator`, `json-fold`, and `json-read`. If the value returned by this parameter is reached, the implementation must raise an error that satisfies `json-error?`.

The default value of `json-number-of-character-limit` is `+inf.0`.

A proper value should be set on a per-application basis to mitigate the risks of denial-of-service attacks.

`(json-generator [port-or-generator]) → generator`

Streaming event-based JSON reader. `PORT-OR-GENERATOR` default value is the value returned by `current-input-port`. It must be a textual input port or a generator of characters. `json-generator` returns a generator of Scheme objects, each of which must be one of:

- `'array-start` symbol denoting that an array should be constructed.
- `'array-end` symbol denoting that the construction of the array for which the last `'array-start` was generated and not closed is finished.
- `'object-start` symbol denoting that an object should be constructed. The object's key-value pairs are emitted in sequence like those in a property list (plist) where keys are strings. That is, the generation of a key is always followed by the generation of a value. Otherwise, the JSON would be invalid and `json-generator` would raise an error.
- `'object-end` symbol denoting that the construction of the object for which the last `object-start` was generated and not closed is finished.
- the symbol `'null`
- boolean
- number
- string

In the case where nesting of arrays or objects reaches the value returned by the parameter `json-nesting-depth-limit`, the generator must raise an object that satisfies the predicate `json-error?`

In cases where the JSON is invalid, the generator returned by `json-generator` should raise an object that satisfies the predicate `json-error?`.

Otherwise, if `PORT-OR-GENERATOR` contains valid JSON text, the generator returned by `json-generator` must yield an end-of-file object in two situations:

- The first time the generator returned by `json-generator` is called, it returns an object that is a boolean, a number, a string or the symbol `'null`.
- The first time the generator returned by `json-generator` is called, it returns a symbol that is not the symbol `'null`. When the underlying JSON text is valid, it should be the symbol starting a structure: `'object-start` or `'array-start`. The end-of-file object is generated when that structure is finished.

In other words, the generator returned by `json-generator` will parse at most one JSON value or one top-level structure. If `PORT` is not finished, as in the case of JSON lines, the user should call `json-generator` again with the same `PORT-OR-GENERATOR`.

Examples

```
(assume
  (equal?
    (call-with-input-string "42 101 1337" (lambda (port) (generator->list (json-generator port))))
    '(42)))

(assume
  (equal?
    (call-with-input-string "[42] 101 1337" (lambda (port) (generator->list (json-generator port))))
    '(array-start 42 array-end)))
```

`(json-fold proc array-start array-end object-start object-end seed [port-or-generator])`

Fundamental JSON iterator.

`json-fold` will read the JSON text from `PORT-OR-GENERATOR`, which has `(current-input-port)` as its default value. `json-fold` will call the procedures passed as argument:

- `(PROC obj seed)` is called when a JSON value is generated or a complete JSON structure is read. `PROC` should return the new seed that will be used to iterate over the rest of the generator. Termination is described below.
- `(OBJECT-START seed)` is called with a seed and should return a seed that will be used as the seed of the iteration over the key and values of that object.
- `(OBJECT-END seed)` is called with a seed and should return a new seed that is the result of the iteration over a JSON object.

`ARRAY-START` and `ARRAY-END` take the same arguments, and have similar behavior, but are called for iterating on JSON arrays.

`json-fold` must return the seed when:

- `PORT-OR-GENERATOR` yields an object that satisfies the predicate `eof-object?`
- All structures, array or object, that were started have ended. The returned object is `(PROC obj SEED)` where `obj` is the object returned by `ARRAY-END` or `OBJECT-END`

Example

`json-read` can be defined in terms of `json-fold`:

```

(define (%json-read port-or-generator)

  (define %root '(root))

  (define (array-start seed)
    ;; array will be read as a list, then converted into a vector in
    ;; array-end.
    '())

  (define (array-end items)
    (list->vector (reverse items)))

  (define (object-start seed)
    ;; object will be read as a property list, then converted into an
    ;; alist in object-end.
    '())
  (define (plist->alist plist)
    ;; PLIST is a list of an even number of items.  Otherwise,
    ;; json-generator would have raised a json-error.
    (let loop ((plist plist)
               (out '()))
      (if (null? plist)
          out
          (loop (cddr plist) (cons (cons (string->symbol (cadr plist)) (car plist)) out)))))

  (define object-end plist->alist)

  (define (proc obj seed)
    ;; proc is called when a JSON value or structure was completely
    ;; read.  The parse result is passed as OBJ.  In the case where
    ;; what is parsed is a simple JSON value, OBJ is simply
    ;; the token that is read.  It can be 'null, a number or a string.
    ;; In the case where what is parsed is a JSON structure, OBJ is
    ;; what is returned by OBJECT-END or ARRAY-END.
    (if (eq? seed %root)
        ;; This is toplevel.  A complete JSON value or structure was
        ;; read, so return it.
        obj
        ;; This is not toplevel, hence json-fold is called recursively
        ;; to parse an array or object.  Both ARRAY-START and
        ;; OBJECT-START return an empty list as a seed to serve as an
        ;; accumulator.  Both OBJECT-END and ARRAY-END expect a list
        ;; as argument.
        (cons obj seed)))

  (let ((out (json-fold proc
                        array-start
                        array-end
                        object-start
                        object-end
                        %root
                        port-or-generator)))
    ;; if out is the root object, then the port or generator is empty.
    (if (eq? out %root)
        (eof-object)
        out)))
```

(json-read [port-or-generator]) → object

JSON reader procedure. `PORT-OR-GENERATOR` must be a textual input port or a generator of characters. The default value of `PORT-OR-GENERATOR` is the value returned by the procedure `current-input-port`. The returned value is a Scheme object. `json-read` must return only the first toplevel JSON value or structure. When there are multiple toplevel values or structures in `PORT-OR-GENERATOR`, the user should call `json-read` several times to read all of it.

The mapping between JSON types and Scheme objects is the following:

- `null` → the symbol `'null`
- `true` → `#t`
- `false` → `#f`
- number → number
- string → string
- array → vector
- object → association list with keys that are symbols

In the case where nesting of arrays or objects reaches the value returned by the parameter `json-nesting-depth-limit`, `json-read` must raise an object that satisfies the predicate `json-error?`

(json-lines-read [port-or-generator]) → generator

JSON reader of jsonlines or ndjson. As its first and only argument, it takes a generator of characters or a textual input port whose default value is the value returned by `current-input-port`. It will return a generator of Scheme objects as specified in `json-read`.

(json-sequence-read [port-or-generator]) → generator

JSON reader of JSON Text Sequences (RFC 7464). As its first and only argument, it takes a generator of characters or a textual input port whose default value is the value returned by `current-input-port`. It will return a generator of Scheme objects as specified in `json-read`.

(json-accumulator port-or-accumulator) → procedure

Streaming event-based JSON writer. `PORT-OR-ACCUMULATOR` must be a textual output port or an accumulator that accepts characters and strings. It returns an accumulator procedure that accepts Scheme objects as its first and only argument and that follows the same protocol as described in `json-generator`. Any deviation from the protocol must raise an error that satisfies `json-error?`. In particular, objects and arrays must be properly nested.

Mind the fact that most JSON parsers have a nesting limit that is not documented by the standard. Even if you can produce arbitrarily nested JSON with this library, you might not be able to read it with another library.

(json-write obj [port-or-accumulator]) → unspecified

JSON writer procedure. `PORT-OR-ACCUMULATOR` must be a textual output port, or an accumulator that accepts characters and strings. The default value of `PORT-OR-ACCUMULATOR` is the value returned by the procedure `current-output-port`. The value returned by `json-write` is unspecified.

`json-write` will validate that `OBJ` can be serialized into JSON before writing to `PORT`. An error that satisfies `json-error?` is raised in the case where `OBJ` is not an object or a composition of the following types:

- symbol `'null`
- boolean
- number. Must be integers or inexact rationals. (That is, they must not be complex, infinite, NaN, or exact rationals that are not integers.)
- string
- vector
- association list with keys as symbols

Implementation

The sample implementation is available in this Git repo.

Acknowledgements

Thanks to the participants on the SRFI 189 mailing list: Lassi Kortela, Duy Nguyen, Shiro Kawai, Alex Shinn, Marc Nieper-Wißkirchen.

Thanks to Arthur A. Gleckler and John Cowan.

Thanks to Oleg Kiselyov.

Copyright

Editor: Arthur A. Gleckler