The Unicon JSON Library

Gigi Young and Clinton Jeffery

Unicon Technical Report: 20

October 16, 2020

Abstract

This report describes a library for supporting conversion to and from JSON (JavaScript Object Notation) and their equivalent Unicon structures.

Unicon Project
http://unicon.org

University of Idaho
Department of Computer Science
Moscow, ID, 83844, USA

1 Introduction

JSON is a widely used lightweight data-interchange format [Crawford] that supports nested, sequential and associative data structures. This report describes two Unicon library functions that allow for convenient data conversion between JSON and Unicon.

2 An Overview of the JSON Grammar

The JSON grammar is straightforward. A JSON data structure is composed of one or more JSON values, which can be either a string, number, boolean true, boolean false, null, array, or object. An array contains a list of JSON values, while an object is a set of string-value pairs. In other words, an array is a list of values and an object is a table (dictionary) that requires keys to be strings. Arrays and objects may be arbitrarily nested within each other.

The manner in which the tokens of the JSON grammar are defined allows a token's category to be determined from its first character. The JSON tokens are string, number, true, false, null, and the operators used for array, object, and their productions ({}`[]:,`). The following table gives the possible first characters of each non-operator JSON token.

possible first characters	JSON token
{	object
`[`	array
-0123456789	number
"	string
t	true
f	false
n	null

3 The Unicon JSON API

The JSON library contains four thread-safe functions: `jtou()`, `jtous()`, `jtouf()` for converting from JSON to Unicon and `utoj()` for converting from Unicon to JSON. The data equivalencies are as follows:

JSON to Unicon data type equivalency

JSON data type	Unicon data type
object	table
array	list
string	string
number	int, real
true	string - `"__true__"`
false	string - `"__false__"`
null	null value - `&null`

Unicon to JSON data equivalency

Unicon data type	JSON data type
table, set, cset, record, class	object
list	array
string	string
int, real	number
string - `"__true__"`	true
string - `"__false__"`	false
null value - `&null`	null

There are more Unicon data structures than JSON was designed to represent. Consequently, string encodings (`"__unitable__"`, `"__uniset__"`, `"__unicset__"`, `"__unirecord__"`, and `"__uniclass__"`) have been defined to differentiate the different types of Unicon structures represented by a JSON object (tables, sets, csets, records, and classes respectively). The JSON object Unicon-encodings are listed in the table below:

JSON object Unicon-encodings

Unicon data type	JSON specification
table	`{"__unitable__":1, ...}`
set	`{"__uniset__":[...]}`
cset	`{"__unicset__":"..."}`
record	`{"__unirecord__":"RecordName","Field1":"Value1", ...}`
class	`{"__uniclass__":"ClassName","Field1":"Value1", ...}`

These encodings are strict and no type coercions will be performed. For example,

```
{"__unitable__":"1"}
```

would produce a Unicon table with key-value pair

```
T["__unitable__"] := "1"
```

instead of an empty table. Any JSON object that does not adhere to these Unicon-encodings will default to a Unicon table. Furthermore, records and classes that adhere to Unicon-encoding syntax but whose constructors do not exist in the calling program will also be converted to Unicon tables.

The Unicon table is a superset of what the JSON object can represent. A JSON object is a dictionary which only supports string keys, while a Unicon table supports any valid Unicon type as a key. To support JSON conversion of Unicon tables, the grammar of a JSON object is optionally extended to accept any JSON value as a key. The extended JSON object becomes a set of value-value pairs instead of string-value pairs.

Although the aim of this library is to support inter-language data exchange, it is just as important to support all Unicon data types. The choosen compromise allows Unicon users the decision of whether or not to adhere to strict JSON with the parameter `strict`.

`strict` is a parameter for every function in the JSON library API and defaults to `&null`. If `strict` is non-null, then library functions adhere to the JSON specification. Otherwise, an extended-JSON specification which allows for non-string key values for JSON objects is used.

---

jtou(s, strict:&null, mode:&null, error:&errout) **JSON to Unicon**

`jtou(s)` generates the equivalent Unicon data structure(s) to parameter `s`, a JSON-encoded string or JSON filename. Parameter `strict` enforces strict adherence to the JSON specification if non-null. Parameter `mode` can be specified as `"s"` (see `jtous()`) to force JSON-encoded string conversion or as `"f"` (see `jtouf()`) to force JSON file conversion. `jtou(s)` tries to first process `s` as a string, then as a filename if `mode` is not specified. Parameter `error` allows the user to specify a file or filestream other than standard error for error output. This function fails if `s` is not valid JSON, depending on parameter `strict`. It should be noted that error messages may be misleading if `mode` is not specified.

To conform with other JSON libraries, trailing commas are allows for JSON arrays and objects.

---

jtous(s, strict:&null, error:&errout) **JSON string to Unicon**

`jtous(s)` (equivalent to `jtou(s,"s")`) generates the equivalent Unicon data structure(s) to parameter `s`, a JSON-encoded string. Parameter `strict` enforces strict adherence to the JSON specification if non-null. Parameter `error` allows the user to specify a file or filestream other than standard error for error output. This function fails if `s` is not valid JSON, depending on parameter `strict`.

To conform with other JSON libraries, trailing commas are allows for JSON arrays and objects.

---

jtouf(s, strict:&null, error:&errout) **JSON file to Unicon**

`jtouf(s)` (equivalent to `jtou(s,"f")`) generates the equivalent Unicon data structure(s) to parameter `s`, a JSON filename. Parameter `strict` enforces strict adherence to the JSON specification if non-null. Parameter `error` allows the user to specify a file or filestream other than standard error for error output. This function fails if `s` is not valid JSON, depending on parameter `strict`.

To conform with other JSON libraries, trailing commas are allows for JSON arrays and objects.

---

utoj(u, strict:&null, error:&errout) **Unicon to JSON**

`utoj(u)` takes a Unicon value, `u`, and returns a corresponding JSON-formatted string. Parameter `strict` enforces strict adherence to the JSON specification if non-null. Parameter `error` allows the user to specify a filestream other than standard error for error output.

4 Examples

Unicon's JSON library is straightforward. `jtou()`, `jtous()`, or `jtouf()` are used to convert JSON to Unicon data types and `utoj()` is used to convert Unicon data types to JSON. `jtou(s,"s")` is equivalent to `jtous(s)` and `jtou(s,"f")` is equivalent to `jtouf(s)`. `jtou(s)` will convert `s` whether it is a JSON-encoded string or JSON filename as long as the JSON is valid. However, one may want to use the specific functions `jtous()` and `jtouf()` in the case of error handling accuracy.

The following program converts a JSON data structure which consists of an object and an array to a Unicon list containing a table and list. The `jtou*()` family of functions are generators, which allows the use of the list constructor.

```
link json
procedure main()
   u := [: jtou("{\"one\":1, \"two\":2, \"three\":3}_
       [true, false, null, 1.23e3]") :]
   x := u[1]
   y := u[2]
   write(x["one"])
   write(x["two"])
   write(x["three"])
   every i := 1 to *y do write(y[i])
end
```

The output of which is:

```
1
2
3
__true__
__false__
 1230.0
```

This simple program converts a Unicon table to a JSON object

```
link json
procedure main()
   T := table()
   T["one"] := 1
   T["two"] := 2
   T["list"] := [1,2,3]
   T["table"] := table()
   write(utoj(T))
end
```

and outputs:

```
{"__unitable__":1,"two":2,"table":{"__unitable__":1},"one":1,"list":[1,2,3]}
```

The extraction of keys from Unicon tables occurs in a random manner, resulting in JSON data that does not follow the order in which the Unicon table was constructed. However, lists are sequential in Unicon, so the resulting ordering of the JSON array is retained.

The following program performs a class conversion

```
link json

class Person(name, age, gender)
   #
   # some methods here
   #
end

procedure main()
   person := Person("Joe", 42, "Male")
   write(utoj(person))
end
```

which outputs:

```
{"__uniclass__":"Person","name":"Joe","age":42,"gender":"Male"}
```

The following program performs a Unicon-to-JSON-back-to-Unicon conversion and is more complicated than the previous examples. This shows that the JSON library can be used to store Unicon data. Unicon's JSON library does not store object references, but can recreate structually equivalent objects.

```
link json
link ximage

record R(a,b,c)

class S(d,e,f)
end

procedure main()
   c := S()
   c.d := "a"
   c.e := "b"
```

```
   c.f := "c"
   r := R(1,2,3)
   s := set(["abc"])
   s2 := set(["__unicset__","__uniset__","__uniclass__","__unirecord__"])
   t := table()
   t['a'] := 'bc'
   t["\177b"] := 2
   t["c"] := table()
   t["c"]["\^cd"] := 3.89e-4
   t[1] := r
   t[2] := c
   t[s] := set(["I'm a set element"])
   t["__uniset__"] := s2
   l := [t, s]

   write("Before encoding:",ximage(l))
   X := utoj(l)
   write("\nEncoded JSON: ",X)
   y := jtous(X)
   write("\nAfter encoding:",ximage(y),"\n")
end
```

This program produces the output:

```
Before encoding:L4 := list(2)
   L4[1] := T1 := table(&null)
      T1[1] := R_R_1 := R()
         R_R_1.a := 1
         R_R_1.b := 2
         R_R_1.c := 3
      T1[2] := S_1 := S()
         S_1.d := "a"
         S_1.e := "b"
         S_1.f := "c"
      T1["__uniset__"] := S2 := set()
         insert(S2,"__uniclass__")
         insert(S2,"__unicset__")
         insert(S2,"__unirecord__")
         insert(S2,"__uniset__")
      T1["c"] := T2 := table(&null)
         T2["\x03d"] := 0.000389
      T1["\db"] := 2
      T1['a'] := 'bc'
      T1[{S1 := set()
```

```
            insert(S1,"abc")
         S1}] := S3 := set()
         insert(S3,"I'm a set element")
   L4[2] := S1
```

Encoded JSON: [{"__unitable__":1,"\u007Fb":2,2:{"__uniclass__":"S","d":"a","e":"b","f":"c"},"__uniset__":{"__uniset__":["__uniclass__","__uniset__","__unirecord__","__unicset__"]},1:{"__unirecord__":"R","a":1,"b":2,"c":3},{"__uniset__":["abc"]}:{"__uniset__":["I'm a set element"]},{"__unicset__":"a"}:{"__unicset__":"bc"},"c":{"__unitable__":1,"\u0003d":0.000389}},{"__uniset__":["abc"]}]

```
After encoding:L24 := list(2)
   L24[1] := T7 := table(&null)
      T7[1] := R_R_3 := R()
         R_R_3.a := 1
         R_R_3.b := 2
         R_R_3.c := 3
      T7[2] := S_2 := S()
         S_2.d := "a"
         S_2.e := "b"
         S_2.f := "c"
      T7["__uniset__"] := S4 := set()
         insert(S4,"__uniclass__")
         insert(S4,"__unicset__")
         insert(S4,"__unirecord__")
         insert(S4,"__uniset__")
      T7["c"] := T13 := table(&null)
         T13["\x03d"] := 0.000389
      T7["\db"] := 2
      T7['a'] := 'bc'
      T7[{S5 := set()
            insert(S5,"abc")
         S5}] := S6 := set()
         insert(S6,"I'm a set element")
   L24[2] := S7 := set()
      insert(S7,"abc")
```

References

[Crawford] anonymous, but owing to Douglas Crawford. "Introducing JSON", json.org.