Beauty will save the world.
Fyodor Dostoevsky




**Popular resources**

Time and Computers
Unicode Talk
Unicode Hacks
Unicode Poster
Abusing Twitter API
Twitter API Visual Doc.
Hello Mach-O
Parsing JSON is a Minefield
PyCairo Visual Doc

**External profiles**

Twitter
LinkedIn
Facebook
GitHub
Strava
ITRA


# Parsing JSON is a Minefield

**[2016-10-26]** First version of the article
**[2016-10-28]** Presentation at Soft-Shake Conference, Geneva (slides)
**[2016-11-01]** Article and comments in The Register
**[2017-11-16]** Presentation at Black Alps Security Conference, Yverdon (slides)
**[2018-03-09]** Presentation at Toulouse Hacking Conference (slides)
**[2018-03-30]** Updated this article considering RFC 8259

Feel free to comment on Hacker News (2016-10), Hacker News (2018-04) or reddit.

## Session Description

*JSON is the de facto standard when it comes to (un)serialising and exchanging data in web and mobile programming. But how well do you really know JSON? We'll read the specifications and write test cases together. We'll test common JSON libraries against our test cases. I'll show that JSON is not the easy, idealised format as many do believe. Indeed, I did not find two libraries that exhibit the very same behaviour. Moreover, I found that edge cases and maliciously crafted payloads can cause bugs, crashes and denial of services, mainly because JSON libraries rely on specifications that have evolved over time and that left many details loosely specified or not specified at all.*

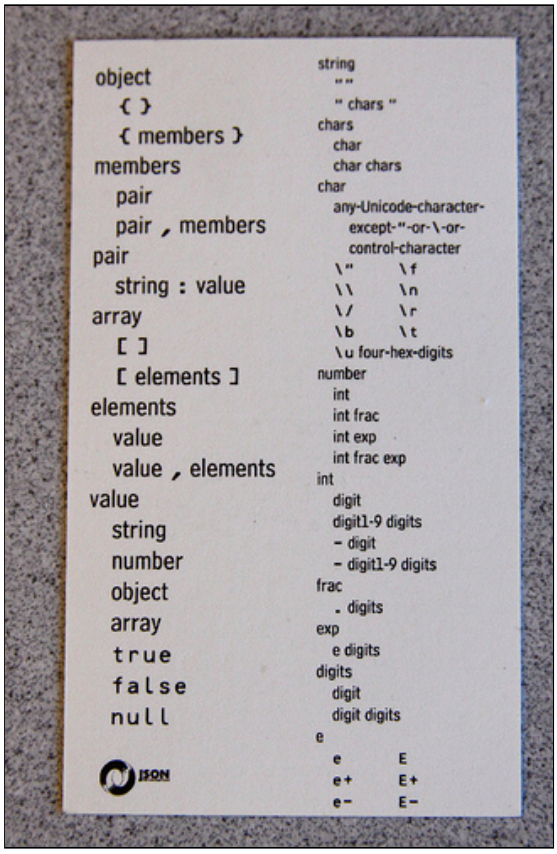
## Table of Contents



## 1. JSON Specifications

JSON is the de facto serialization standard when it comes to sending data over HTTP, the *lingua franca* used to exchange data between heterogeneous software, both in modern web sites and mobile applications.

"Discovered" in 2001 Douglas Crockford, JSON specification is so short and simple that Crockford created business cards with the whole JSON grammar on their back.




Pretty much all Internet users and programmers use JSON, yet few do actually agree on how JSON should actually work. The conciseness of the grammar leaves many aspects undefined. On top of that, several specifications exist, and their various interpretations tend to be murky.

Crockford chose not to version JSON definition:

> *Probably the boldest design decision I made was to not put a version number on JSON so there is no mechanism for revising it. We are stuck with JSON: whatever it is in its current form, that's it.*

Yet JSON is defined in at least seven different documents:

1. 2002 - json.org, and the business card
2. 2006 - IETF RFC 4627, which set the `application/json` MIME media type
3. 2011 - ECMAScript 262, section 15.12
4. 2013 - ECMA 404 according to Tim Bray (RFC 7159 editor), ECMA rushed out to release it because:

   > *"Someone told the ECMA working group that the IETF had gone crazy and was going to rewrite JSON with no regard for compatibility and break the whole Internet and something had to be done urgently about this terrible situation. (...) It doesn't address any of the gripes that were motivating the IETF revision.*

5. 2014 - IETF RFC 7158 makes the specification "Standard Tracks" instead of "Informational", allows scalars (anything other than arrays and objects) such as `123` and `true` at the root level as ECMA does, warns about bad practices such as duplicated keys and broken Unicode strings, without explicitely forbidding them, though.
6. 2014 - IETF RFC 7159 was released to fix a typo in RFC 7158, which was dated from "March 2013" instead of "March 2014".
7. 2017 - IETF RFC 8259 was released in December 2017. It basically adds two things: 1) outside of closed eco-systems, JSON MUST be encoded in UTF-8 and 2) JSON text that is not networked transmitted MAY now add the byte ordrer mark `U+FEFF`, although this is not stated explicitely.

Despite the clarifications they bring, RFC 7159 and 8259 contain several approximations and leaves many details loosely specified.

For instance, RFC 8259 mentions that a design goal of JSON was to be "a subset of JavaScript", but it's actually not. Specifically, JSON allows the Unicode line terminators `U+2028 LINE SEPARATOR` and `U+2029 PARAGRAPH SEPARATOR` to appear unescaped. But JavaScript specifies that strings cannot contains line terminators (ECMA-262 - 7.8.4 String Literals), and line terminators include... `U+2028` and `U+2029` (7.3 Line Terminators). The single fact that these two characters are allowed without escape in

JSON strings while they are not in JavaScript implies that JSON is **not** a subset of JavaScript, despite the JSON design goals.

Also, RFC 7159 is unclear about how a JSON parser should treat extreme number values, malformed Unicode strings, similar objects or handle recursion depth. Some corner cases are explicitely left free to implementations, while others suffer from contradictory statements.

To illustrate the poor precision of RFC 8259, I wrote a corpus of JSON test files and documented how selected JSON parsers chose to handle these files. You'll see that deciding if a test file should be parsed or not is not always straightforward. In my findings, there were no two parsers that exhibited the same behaviour, which may cause serious interoperability issues.

## 2. Parsing Tests

In this section, I explain how to create test files to validate parsers behaviour, discuss some interesting tests, and the rationale to decide if they should be accepted or rejected by RFC 8259 compliant parsers, or if parsers should be free to accept or reject the contents.

File names start with a letter which tells the expected result: `y` (yes) for parsing success, `n` (no) for parsing error, and `i` for implementation defined. They also give clues about which component of the parser is specifically tested.

For instance, `n_string_unescaped_tab.json` contains `["09"]`, which is an array containing a string, which consists in the `TAB 0x09` character, which MUST be u-escaped according to JSON specifications. Note how the underlined values represent the hex values of the bytes. This file specifically tests string parsing, hence the `string` in file name, and not `structure`, `array` or `object`. According to RFC 8259, this is not a valid JSON string, hence the `n`.

Note that since several parsers don't allow scalars at the top level (`"test"`), I embed strings into arrays (`["test"]`).

You'll find more that 300 tests in the JSONTestSuite GitHub repository.

The test files were mostly handcrafted while reading specifications, trying to pay attention to edge cases and ambiguous parts. I also tried to reuse other test suites found on the Internet (mainly json-test-suite and JSON Checker), but I found that most test suites did only cover basic cases.

Finally, I also generated JSON files with the fuzzing software American Fuzzy Lop. I then removed redundant tests that produced the same set of results, and then reduced the remaining ones to the keep the least number of characters that triggered these results (see section 3).

### 2.1 Structure

**Scalars** - Clearly, scalars such as `123` or `"asd"` must be parsed. In practice, many popular parsers do still implement RFC 4627 and won't parse lonely values. So there are basic tests such as:

| | |
|---|---|
| y_structure_lonely_string.json | "asd" |

**Trailing commas** - Trailing commas such as in `[123,]` or `{"a":1,}` are not part of the grammar, so these files should not pass, right? The thing is that RFC 8259 allows parsers to support "extensions" (section 9), although it does not elaborates about extensions. In practice, allowing trailing commas is a common extension. Since it's not part of JSON grammar, parser *don't have* to support it, so the file name starts with `n`.

| | |
|---|---|
| n_object_trailing_comma.json | {"id":0,} |
| n_object_several_trailing_commas.json | {"id":0,,,,,} |

**Comments** - Comments are not part of the grammar. Crockford removed them from early specifications. Yet, they are still another common extension. Some parsers allow trailing comments `[1]//xxx`, or even inline comments `[1,/*xxx*/2]`.

| | |
|---|---|
| y_string_comments.json | ["a/*b*/c/*d//e"] |
| n_object_trailing_comment.json | {"a":"b"}/**/ |
| n_structure_object_with_comment.json | {"a":/*comment*/"b"} |

**Unclosed Structures** - These tests cover everything that is opened and not closed or the opposite, such as `[` or `[1,{,3]`. They are clearly invalid and must fail.

| | |
|---|---|
| n_structure_object_unclosed_no_value.json | {"": |
| n_structure_object_followed_by_closing_object.json | {}} |

**Nested Structures** - Structures may contain other structures. An array may contain other arrays. The first element can be an array, whose first element is also an array, etc, like russian dolls `[[[[[]]]]]`. RFC 8259 allows parsers to set limits to the maximum depth of nesting (section 9).

In practice, several parsers don't set a depth limit and crash after a certain threshold. For example, Xcode itself will crash when opening a `.json` file made the character `[` repeated 10000 times, most probably because the JSON syntax highlighter does not implement a depth limit.

```
$ python -c "print('['*100000)" > ~/x.json
$ ./Xcode ~/x.json
Segmentation fault: 11
```

**White Spaces** - RFC 8259 grammar defines white spaces as `0x20` (space), `0x09` (tab), `0x0A` (line feed) and `0x0D` (carriage return). It allows white spaces before and after "structural characters" `[]{}:,`. So, we'll write passing tests like `20[090A]0D` and failing ones including all kinds of white spaces that are not explicitely allowed, such as `0x0C` form feed or `[E281A0]`, which is the UTF-8 encoding for `U+2060 WORD JOINER`.

| | |
|---|---|
| n_structure_whitespace_formfeed.json | [0C] |
| n_structure_whitespace_U+2060_word_joiner.json | [E281A0] |
| n_structure_no_data.json | |

### 2.2 Numbers

**NaN and Infinity** - Strings that describe special numbers such as `NaN` or `Infinity` are not part of the JSON grammar. However, several parsers accept them, which can be considered as an "extension" (section 9). Test files also test the negative forms `-NaN` and `-Infinity`.

| | |
|---|---|
| n_number_NaN.json | [NaN] |
| n_number_minus_infinity.json | [-Infinity] |

**Hex Numbers** - RFC 8259 doesn't allow hex numbers. Tests will include numbers such as `0xFF`, and these files should not be parsed.

| | |
|---|---|
| n_number_hex_2_digits.json | [0x42] |

**Range and Precision** - What about numbers with a huge amount of digits? According to RFC 8259, "A JSON parser MUST accept all texts that conform to the JSON grammar" (section 9). However, according to the same paragraph, "An implementation may set limits on the range and precision of numbers.". So, it is unclear to me whether parsers are allowed to raise errors when they meet extreme values such `1e9999` or `0.000000000000000000000000000000001`.

| | |
|---|---|
| i_number_very_big_negative_int.json | [-237462374673276894279832(...) |

**[Update 2016-11-02]** The original version of this article classified the "Range and Precision" tests as `y_` (must pass). This classification was challenged and I eventually changed the tests into `i_` (implementation defined).

**Exponential Notation** - Parsing exponential notation can be surprisingly hard (see the results section). Here are some valid contents `[0E0]`, `[0e+1]` and invalid ones `[1.0e+]`, `[0E]` and `[1eE2]`.

| | |
|---|---|
| n_number_0_capital_E+.json | [0E+] |
| n_number_.2e-3.json | [.2e-3] |
| y_number_double_huge_neg_exp.json | [123.456e-789] |

### 2.3 Arrays

Most edge cases regarding arrays are opening/closing issues and nesting limit. These cases were discussed in section 2.1 Structure. Passing tests will include `[[],[[]]]`, while failing tests will be like `]` or `[[[[]`.

| | |
|---|---|
| n_array_comma_and_number.json | [,1] |
| n_array_colon_instead_of_comma.json | [": 1] |
| n_array_unclosed_with_new_lines.json | [1,0A10A,1 |

### 2.4 Objects

**Duplicated Keys** - RFC 8259 section 4 says that "The names within an object should be unique.". It does not prevent parsing objects where the same key does appear several times `{"a":1,"a":2}`, but lets parsers decide what to do in this case. The same section 4 even mentions that "(some) implementations report an error or fail to parse the object", without telling clearly if failing to parse such objects is compliant or not with the RFC and especially section 9: "A JSON parser MUST accept all texts that conform to the JSON grammar.".

Variants of this special case include same key - same value `{"a":1,"a":1}`, and similar keys or values, where the similarity depends on how you compare strings. For example, the keys may be binary different but equivalent according to Unicode NFC normalization, such as in `{"C3A9":"NFC","65CC81":"NFD"}` where boths keys encode "é". Tests will also include `{"a":0,"a":-0}`.

| | |
|---|---|
| y_object_empty_key.json | {"":0} |
| y_object_duplicated_key_and_value.json | {"a":"b","a":"b"} |
| n_object_double_colon.json | {"x"::"b"} |
| n_object_key_with_single_quotes.json | {key: 'value'} |
| n_object_missing_key.json | {:"b"} |
| n_object_non_string_key.json | {1:1} |

### 2.5 Strings

**File Encoding** - Former RFC 7159 did only recommend UTF-8, and said that "JSON text SHALL be encoded in UTF-8, UTF-16, or UTF-32".

Now RFC 8259 section 8.1 says that "JSON text exchanged between systems that are not part of a closed ecosystem MUST be encoded using UTF-8".

Still, passing tests should include text encoded in these three encodings. UTF-16 and UTF-32 texts should also include both their big-endian and little-endian variants.

The parsing of invalid UTF-8 will be implementation defined.

| | |
|---|---|
| y_string_utf16.json | FFFE[00"00E900"00]00 |
| i_string_iso_latin_1.json | ["E9"] |

**[Update 2016-11-04]** The first version of this article considered invalid UTF-8 as `n_` tests. This classification was challenged and I eventually changed these tests into `i_` tests.

**Byte Order Mark** - Former RFC 8259 section 8.1 stated "Implementations MUST NOT add a byte order mark to the beginning of a JSON text", "implementations (...) MAY ignore the presence of a byte order mark rather than treating it as an error".

Now, RFC 8259 section 8.1 adds: "Implementations MUST NOT add a byte order mark (`U+FEFF`) to the beginning *of a networked-transmitted JSON text*.", which seems to imply that implemenatations may now add a BOM when JSON is not sent over the network.

Tests with implementation defined will include a plain UTF-8 BOM with no other content, a UTF-8 BOM with a UTF-8 string, but also a UTF-8 BOM with a UTF-16 string, and a UTF-16 BOM with a UTF-8 string.

| | |
|---|---|
| n_structure_UTF8_BOM_no_data.json | EFBBBF |
| n_structure_incomplete_UTF8_BOM.json | EFBB{} |
| i_structure_UTF-8_BOM_empty_object.json | EFBBBF{} |

**Control Characters** - Control characters must be escaped, and are defined as `U+0000` through `U+001F` (section 7). This range does not include `0x7F DEL`, which may be part of other definitions of control characters (see section 4.6 Bash JSON.sh). That is why passing tests include `["7F"]`.

| | |
|---|---|
| n_string_unescaped_ctrl_char.json | ["a09a"] |
| y_string_unescaped_char_delete.json | ["7F"] |
| n_string_escape_x.json | ["\x00"] |

**Escape** - "All characters may be escaped" (section 7) like `\uXXXX`, but some MUST be escaped: quotation mark, reverse solidus and control characters. Failing tests should include the escape character without the escaped value, or with an incomplete escaped value. Examples: `["\"]`, `["\`, `[\`.

| | |
|---|---|
| y_string_allowed_escapes.json | ["\"\\/\b\f\n\r\t"] |
| n_structure_bad_escape.json | ["\ |

The escape character can be used to represent codepoints in the Basic Multilingual Plane (`\u005C`). Passing tests will include the zero character `\u0000`, which may cause issues in C-based parsers. Failing

tests will include capital U `\U005C`, non-hexadecimal escaped values `\u123Z` and incomplete escaped values `\u123`.

| | |
|---|---|
| y_string_backslash_and_u_escaped_zero.json | ["\u0000"] |
| n_string_invalid_unicode_escape.json | ["\uqqqq"] |
| n_string_incomplete_escaped_character.json | ["\u00A"] |

**Escaped Invalid Characters**

Codepoints outside of the BMP are represented by their escaped UTF-16 surrogates: `U+1D11E` becomes `\uD834\uDD1E`. Passing tests will include single surrogates, since they are valid JSON according to the grammar. RFC 7159 errata 3984 raised the issue of grammatically correct escaped codepoints that don't encode Unicode characters.

> *The ABNF cannot at the same time allow non conformant Unicode codepoints (section 7) and states conformance to Unicode (section 1).*

The editors considered that the grammar should not be restricted, and that warning users about the fact that parsers behaviour was "unpredictable" (RFC 8259 section 8.2) was enough. In other words, parsers MUST parse u-escaped invalid codepoints, but the result is undefined, hence the `i_` (implementation definded) prefix in the file name. According to the Unicode standard, invalid codepoints should be replaced by `U+FFFD REPLACEMENT CHARACTER`. People familiar with Unicode complexity won't be surprised that this replacement is not mandatory, and can be done in several ways (see Unicode PR #121: Recommended Practice for Replacement Characters). So several parsers use replacement characters, while other keep the escaped form or produce an non-Unicode character (see Section 5 - Parsing Contents).

**[Update 2016-11-03]** In the first version of this article, I treated non-characters such as `U+FDD0` to `U+10FFFE` the same was as invalid codepoints (`i_` tests). This classification was challenged and I eventually changed the non-characters tests into `y_` tests.

| | |
|---|---|
| y_string_accepted_surrogate_pair.json | ["\uD801\udc37"] |
| n_string_incomplete_escaped_character.json | ["\u00A"] |
| i_string_incomplete_surrogates_escape_valid.json | ["\uD800\uD800\n"] |
| i_string_lone_second_surrogate.json | ["\uDFAA"] |
| i_string_1st_valid_surrogate_2nd_invalid.json | ["\uD888\u1234"] |
| i_string_inverted_surrogates_U+1D11E.json | ["\uDd1e\uD834"] |

**Raw non-Unicode Characters**

The previous section discussed non-Unicode codepoints that appear in strings, such as `"\uDEAD"`, which is valid Unicode in its u-escaped form, but doesn't decode into a Unicode character.

Parsers also have to handle raw bytes that don't encode Unicode characters. For instance, the byte `FF` does not represent a Unicode character in UTF-8. As a consequence, a string containing `FF` is not an UTF-8 string. In this case, parsers should simply refuse to parse the string, because "A string is a sequence of zero or more Unicode characters" RFC 8259 section 1 and "JSON text (...) MUST be encoded using UTF-8 RFC 8259 section 8.1.

| | |
|---|---|
| y_string_utf8.json | ["€𝄞"] |
| n_string_invalid_utf-8.json | ["FF"] |
| n_array_invalid_utf8.json | [FF] |

**2.6 RFC 8259 Ambiguities**

Beyond the specific cases we just went through, finding out if a parser is RFC 8259 compliant or not is next to impossible because of section 9 "Parsers":

> *A JSON parser MUST accept all texts that conform to the JSON grammar. A JSON parser MAY accept non-JSON forms or extensions.*

To this point, I perfectly understand the RFC. All grammatically correct inputs MUST be parsed, and parsers are free to accept other contents as well.

> *An implementation may set limits on the size of texts that it accepts. An implementation may set limits on the maximum depth of nesting. An implementation may set limits on the range and precision of numbers. An implementation may set limits on the length and character contents of strings.*

All these limitations sound reasonable (except maybe the one about "character contents"), but contradict the "MUST" from the previous sentence. RFC 2119 is crystal-clear about the meaning of "MUST":

> *MUST - This word, or the terms "REQUIRED" or "SHALL", mean that the definition is an absolute requirement of the specification.*

RFC 8259 allows restrictions, but does not set minimal requirements, so technically speaking, a parser that cannot parse strings longer than 3 characters is still compliant with RFC 8259.

Also, RFC 8259 section 9 should require the parsers to document the restrictions clearly, and/or allow configuration by the user. These configurations would still cause interoperability issues, that's why minimal requirements should be preferred.

This lack of precision regarding allowed restrictions makes it almost impossible to say if a parser is actually RFC 8259 compliant. Indeed, parsing contents that don't match the grammar is not wrong (it's an "extension") and rejecting contents that does match the grammar is allowed (it's a parser "limit").

## 3. Testing Architecture

Independently from how parsers should behave, I wanted to observe how they actually behave, so I picked several JSON parsers and set up things so that I could feed them with my test files.

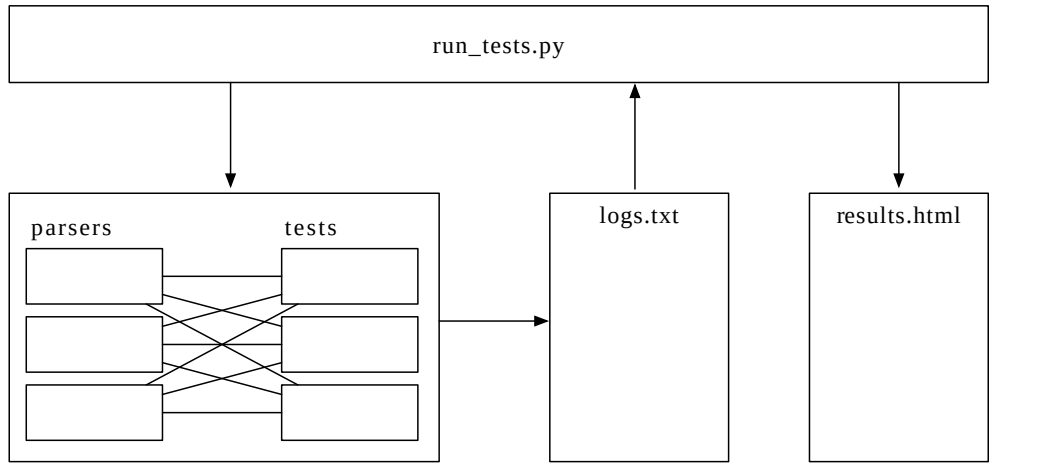
As I'm a Cocoa developer, I included mostly Swift and Objective-C parsers, but also C, Python, Ruby, R, Lua, Perl, Bash and Rust parsers, chosen pretty arbitrarily. I mainly tried to achieve diversity in age, popularity and languages.

Several parsers have options to increase or decrease strictness, tweak Unicode support or allow specific extensions. I strived to always configure the parsers so that they behave as close as possible to the most strict interpretation of RFC 8259.

A Python script `run_tests.py` runs each parser with each test file (or a single test when the file is passed as an argument). The parsers are generally wrapped so that the process returns 0 in case of success, 1 in case of parsing error, yet another status in case of crash, a 5-second delay being considered as a timeout. Basically, I turned JSON parsers into JSON validators.

`run_tests.py` compares the return value of each test with the expected result indicated by the file name prefix. When the value doesn't match, or when this prefix is `i` (implementation defined), `run_tests.py` writes a line in a log file (`results/logs.txt`) in a specific format such as:

```
Python 2.7.10   SHOULD_HAVE_FAILED  n_number_infinity.json
```

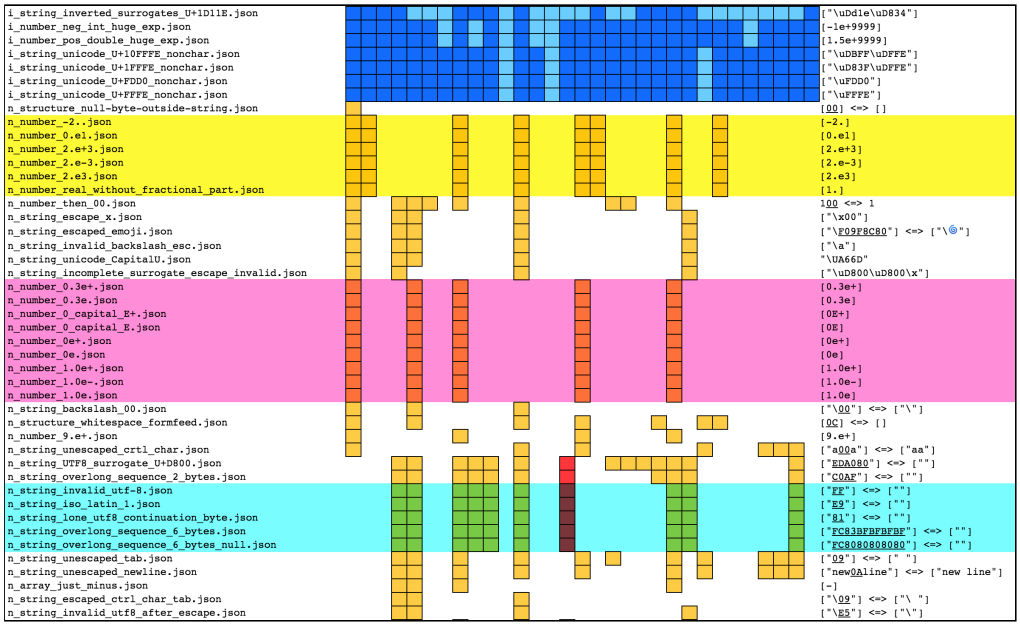



`run_tests.py` then reads the log file and generates HTML tables with the results (`results/parsing.html`).

The results show one row per file, one column per parser, and one color per unexpected result. They also show detailed results by parser.

| |
|---|
| parsing should have succeeded but failed |
| parsing should have failed but succeeded |
| result undefined, parsing succeeded |
| result undefined, parsing failed |
| Crash |
| Timeout |

Tests are sorted by results equality, making easy to spot sets of similar results and remove redundant tests.



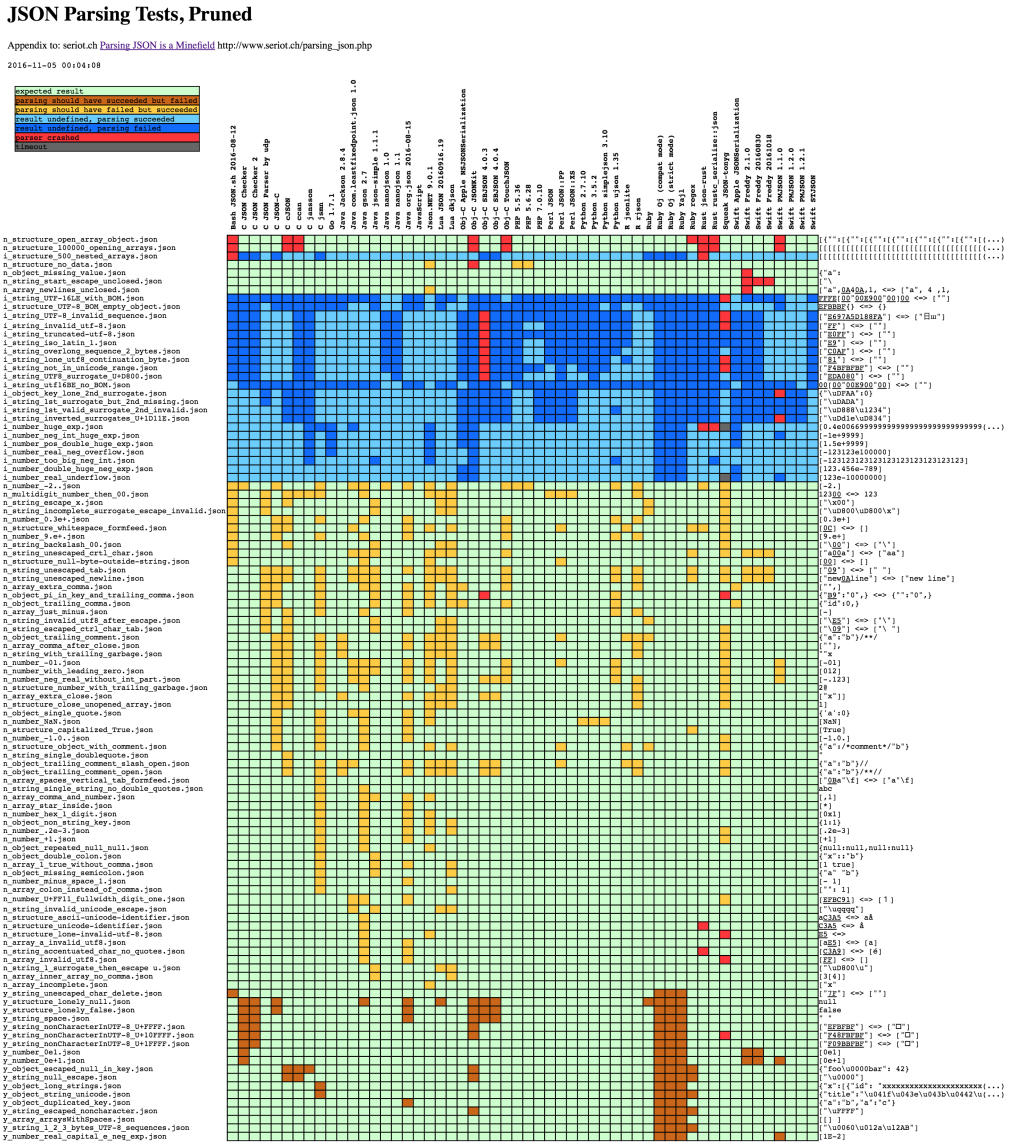

## 4. Results and Comments



### 4.1 Full Results

Full results are presented in http://seriot.ch/json/parsing.html. The tests are vertically sorted by similar results, so it is easy to prune similar tests. An option in `run_tests.py` will produce "pruned results": when a set of tests yields the same results, only the first one is kept. Pruned results HTML file is available here: http://www.seriot.ch/json/parsing_pruned.html.

The most serious issues are crashes (in red), since parsing an uncontrolled input may put the whole process at risk. The "should have passed" tests (in brown) are also very dangerous, because an uncontrolled input may prevent the parser to parse a whole document. The "should have failed" tests (in yellow) are more benign. They indicate a JSON "extension" that can be parsed. Everything will work fine, until the parser is replaced with another parser which does not parse the same "extensions"...




This section highlights and comments some noticeable results.

### 4.2 C Parsers

Here are the five C parsers considered:

- https://github.com/zserge/jsmn
- https://github.com/akheron/jansson
- https://github.com/rustyrussell/ccan/
- https://github.com/DaveGamble/cJSON
- https://github.com/udp/json-parser

And here is a quick comparison between them:

| | jsmn | jansson | ccan | cJSON | json-parser |
|---|---|---|---|---|---|
| Parses `["\u0000"]` | YES | YES | NO | NO | YES |
| Too liberal | YES | NO | NO | YES | YES |
| Crash on nested structs. | NO | NO | YES | YES | NO |
| Rejects big numbers | YES | YES | NO | NO | NO |

You can refer to the full results for more details.

### 4.3 Objective-C Parsers

Here are a couple of Objective-C parsers that used to be very popular in the early days of iOS development, especially because Apple waited until iOS 5 to release NSJSONSerialization. They are still interesting to test, since they are used in production in many applications. Let's consider:

- https://github.com/johnezang/JSONKit

- https://github.com/TouchCode/TouchJSON
- https://github.com/stig/json-framework aka SBJSON

And here is a quick comparison between them:

| | JSONKit | TouchJSON | SBJSON |
|---|---|---|---|
| Crash on nested structs. | YES | NO | YES |
| Crash on invalid UTF-8 | NO | NO | YES |
| Parses trailing garbage `[]x` | NO | NO | YES |
| Rejects big numbers | NO | YES | NO |
| Parses bad numbers `[0.e1]` | NO | YES | NO |
| Treats `0x0C FORM FEED` as white space | NO | YES | NO |
| Parses non-char. `["\uFFFF"]` | NO | YES | YES |

SBJSON survived after the arrival of NSJSONSerialization, is still maintained and is available through CocoaPods, so I reported the crash when parsing non UTF-8 strings such as `["FF"]` in issue #219.

```
*** Assertion failure in -[SBJson4Parser parserFound:isValue:], SBJson4Parser
*** Terminating app due to uncaught exception 'NSInternalInconsistencyExcepti
*** First throw call stack:
(
    0   CoreFoundation                      0x00007fff95f4b4f2 __exceptionPrep
    1   libobjc.A.dylib                     0x00007fff9783bf7e objc_exception_
    2   CoreFoundation                      0x00007fff95f501ca +[NSException ra
    3   Foundation                          0x00007fff9ce86856 -[NSAssertionHa
    4   test_SBJSON                         0x00000001000067e5 -[SBJson4Parse
    5   test_SBJSON                         0x00000001000073f3 -[SBJson4Parse
    6   test_SBJSON                         0x0000000100004289 -[SBJson4Strea
    7   test_SBJSON                         0x0000000100007989 -[SBJson4Parse
    8   test_SBJSON                         0x0000000100005d0d main + 221
    9   libdyld.dylib                       0x00007fff929ea5ad start + 1
)
libc++abi.dylib: terminating with uncaught exception of type NSException
```

#### 4.4 Apple (NS)JSONSerialization

https://developer.apple.com/reference/foundation/nsjsonserialization

NSJSONSerialization was introduced with iOS 5 and is the standard JSON parser on OS X and iOS since then. It is available in Objective-C, and was rewritten in Swift: NSJSONSerialization.swift. The NS prefix was then dropped in Swift 3.

**Restrictions and Extensions**

JSONSerialization has the following, undocumented restrictions:

- it won't parse big numbers: `[123123e100000]`
- it won't parse u-escaped invalid codepoints: `["\ud800"]`

JSONSerialization has the following, undocumented extension:

- it does parse trailing commas: `[1,]` and `{"a":0,}`

I find the restriction about invalid codepoints to be especially problematic, especially in such a high-profile parser, because trying to parse uncontrolled contents may result in a parsing failure.

**Crash on Serialization**

This article is more about JSON parsing than JSON producing, yet I wanted to mention this crash that I found in JSONSerialization when writing `Double.nan`. Remember that `NaN` does not conform to JSON grammar, so in this case, JSONSerialization should throw an error, but not crash the whole process.

```
do {
    let a = [Double.nan]
    let data = try JSONSerialization.data(withJSONObject: a, options: [])
} catch let e {
}

SIGABRT
```

```
*** Terminating app due to uncaught exception 'NSInvalidArgumentException', re
```

**[Update 2016-10-27]** The original version of this article erroneously said that `JSONSerialization.isValidJSONObject(["x":"x", "x":"x"])` would crash because of a bug in the method. @H2CO3_iOS found that the crash is not related to JSONSerialization but to Swift dictionaries, that just cannot be build with the same key appearing twice.

#### 4.5 Freddy (Swift)

Freddy (https://github.com/bignerdranch/Freddy) is a real JSON Parser written is Swift 3. I say real because several GitHub projects claim to be Swift JSON parsers, but actually use Apple JSONSerialization and just map JSON contents with model objects.

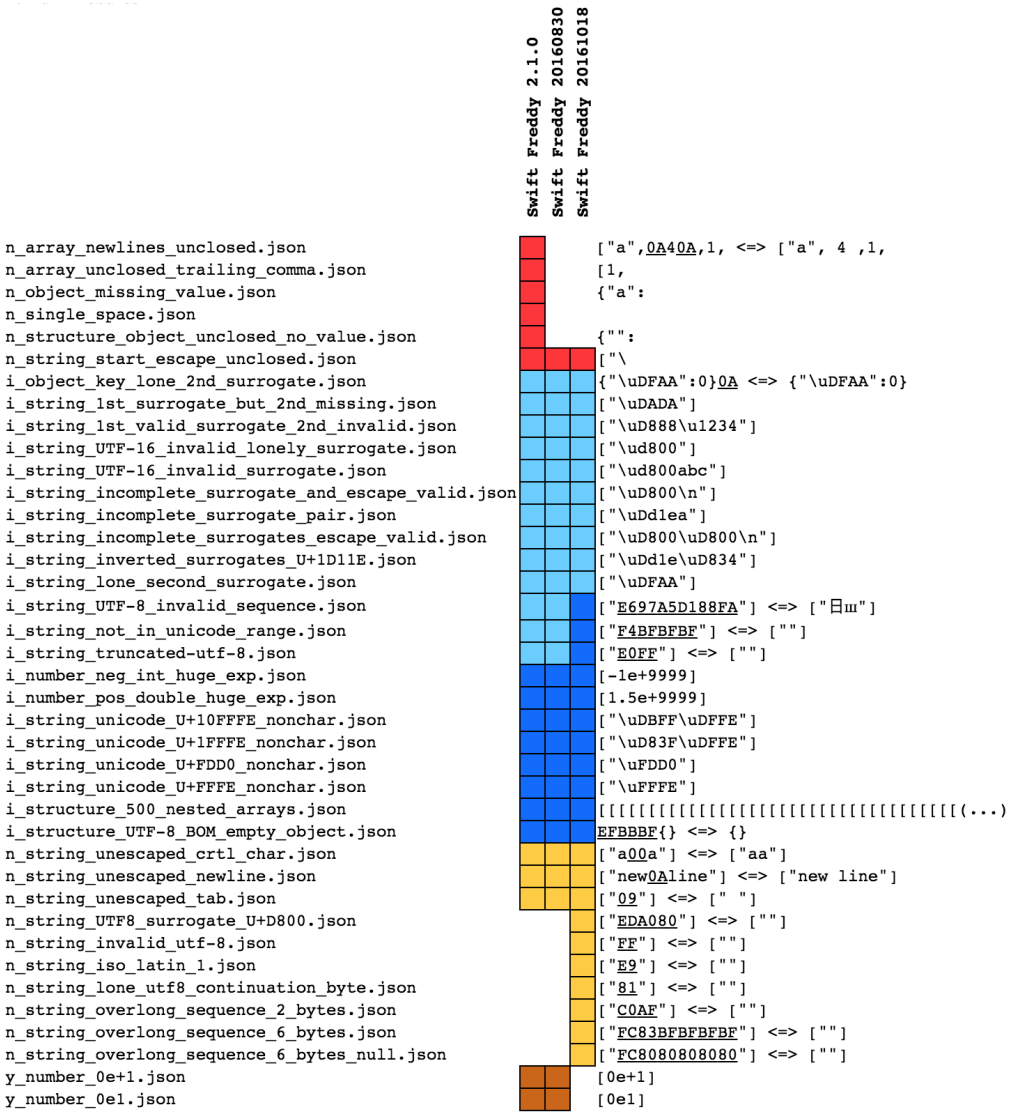
Freddy is interesting because it is written by a famous organization of Cocoa developers, and does leverage Swift type safety by using a Swift enum to represent the different kind of JSON nodes (Array, Dictionary, Double, Int, String, Bool and Null).

But, being released in January 2016, Freddy is still young, and buggy. My test suite showed that unclosed structures such as `[1,` and `{"a":` used to crash the parser, as well as a string with a single space `" "`, so I opened issue #199 that was fixed within 1 day!

Additionnally, I found that `"0e1"` was incorrectly rejected by the parser, so I opened issue #198, which was also fixed within 1 day.

However, Freddy does still crash on 2016-10-18 when parsing `["\`. I reported the bug in (issue #206).

The following table does summarize the evolution of Freddy's behaviour:




#### 4.6 Bash JSON.sh

I tested https://github.com/dominictarr/JSON.sh/ from 2016-08-12.

This Bash parser relies on a regex to find the control characters, which MUST be backslash-escaped according to RFC 8259. But Bash and JSON don't share the same definition of control characters.

The regex uses the `:cntlr:` syntax to match control characters, which is a shorthand for `[\x00-\x1F\x7F]`. But according to JSON grammar, `0x7F DEL` is not part of control characters, and may appear unescaped.

```
 00 nul   01 soh   02 stx   03 etx   04 eot   05 enq   06 ack   07 bel
 08 bs    09 ht    0a nl    0b vt    0c np    0d cr    0e so    0f si
 10 dle   11 dc1   12 dc2   13 dc3   14 dc4   15 nak   16 syn   17 etb
 18 can   19 em    1a sub   1b esc   1c fs    1d gs    1e rs    1f us
 20 sp    21  !    22  "    23  #    24  $    25  %    26  &    27  '
 28  (    29  )    2a  *    2b  +    2c  ,    2d  -    2e  .    2f  /
 30  0    31  1    32  2    33  3    34  4    35  5    36  6    37  7
 38  8    39  9    3a  :    3b  ;    3c  <    3d  =    3e  >    3f  ?
 40  @    41  A    42  B    43  C    44  D    45  E    46  F    47  G
 48  H    49  I    4a  J    4b  K    4c  L    4d  M    4e  N    4f  O
 50  P    51  Q    52  R    53  S    54  T    55  U    56  V    57  W
 58  X    59  Y    5a  Z    5b  [    5c  \    5d  ]    5e  ^    5f  _
 60  `    61  a    62  b    63  c    64  d    65  e    66  f    67  g
 68  h    69  i    6a  j    6b  k    6c  l    6d  m    6e  n    6f  o
 70  p    71  q    72  r    73  s    74  t    75  u    76  v    77  w
 78  x    79  y    7a  z    7b  {    7c  |    7d  }    7e  ~    7f del
```

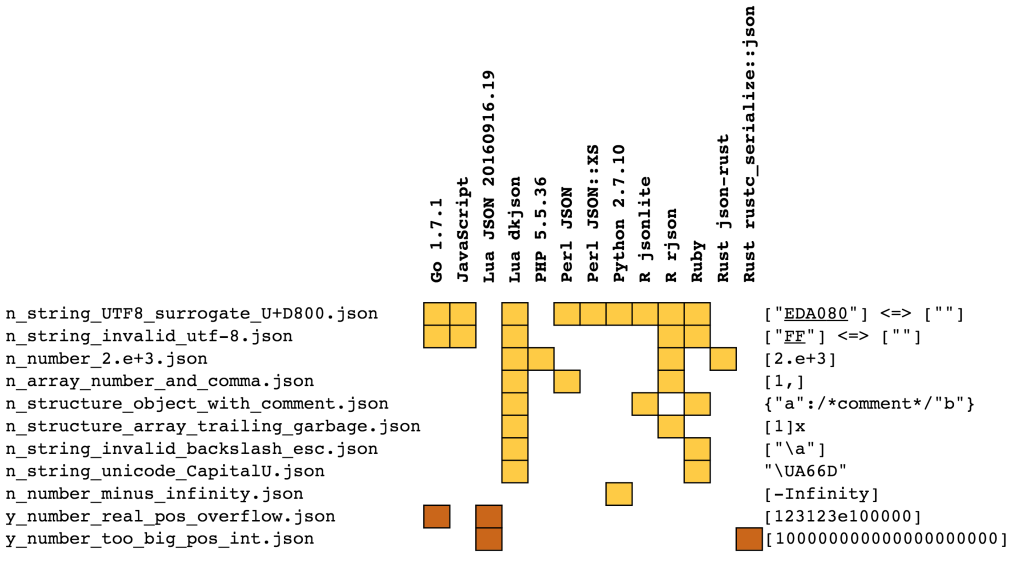
As a consequence, JSON.sh cannot parse `["7F"]`. I reported this bug in issue #46.

Additionally, JSON.sh does not limit the nesting level, and will crash when parsing 10000 times the open array character `[`. I reported this bug in issue #47.

```
$ python -c "print('['*100000)" | ./JSON.sh
./JSON.sh: line 206: 40694 Done                    tokenize
     40695 Segmentation fault: 11  | parse
```

#### 4.7 Other Parsers

Besides C / Objective-C and Swift, I also tested parsers from other environments. Here is a synthetic review of their extensions and restrictions, with a subset of the full tests results. The goal of this table is to demonstrate that there are no two parsers that agree on what is wrong and what is right.




Here are the references for the tested parsers:

- Lua JSON 20160728.17 http://regex.info/blog/lua/json (enjoy quality of comments in source code)
- Lua dkjson 2.5.1 https://github.com/LuaDist/dkjson
- Go 1.7.1, json module https://golang.org/pkg/encoding/json/
- Python 2.7.10, json module https://docs.python.org/2.7/library/json.html
- JavaScript, macOS 10.12

- Perl JSON https://metacpan.org/pod/JSON
- Perl JSON::XS https://metacpan.org/pod/JSON::XS
- PHP 5.6.24, macOS 10.12
- R rjson https://cran.r-project.org/web/packages/rjson/index.html
- R jsonlite https://github.com/jeroenooms/jsonlite
- Rust json-rust https://github.com/maciejhirsz/json-rust
- Rust rustc_serialize::json https://doc.rust-lang.org/rustc-serialize/rustc_serialize/json/

Upon popular request, I also added the following Java parsers, which are not shown on this image but that appear in the full results:

- Java Gson 2.7 https://github.com/google/gson
- Java Jackson 2.8.4 https://github.com/FasterXML/jackson
- Java Simple JSON 1.1.1 https://code.google.com/archive/p/json-simple/

The Python JSON module will parse `NaN` or `-Infinity` as numbers. While this behaviour can be fixed by setting the `parse_constant` options to a function that will raise an Exception as shown below, it's such an uncommon practice that I didn't use it in the tests, and let the parser erroneously parse these number constants.

```
def f_parse_constant(o):
    raise ValueError

o = json.loads(data, parse_constant=f_parse_constant)
```

#### 4.8 JSON Checker

A JSON parser transforms a JSON document into another representation. If the input is invalid JSON, the parser returns an error.

Some programs don't transform their input, but just tell if the JSON is valid or not. These programs are JSON validators.

json.org has a such a program, written in C, called JSON_Checker http://www.json.org/JSON_checker/, that even comes with a (small) test suite:

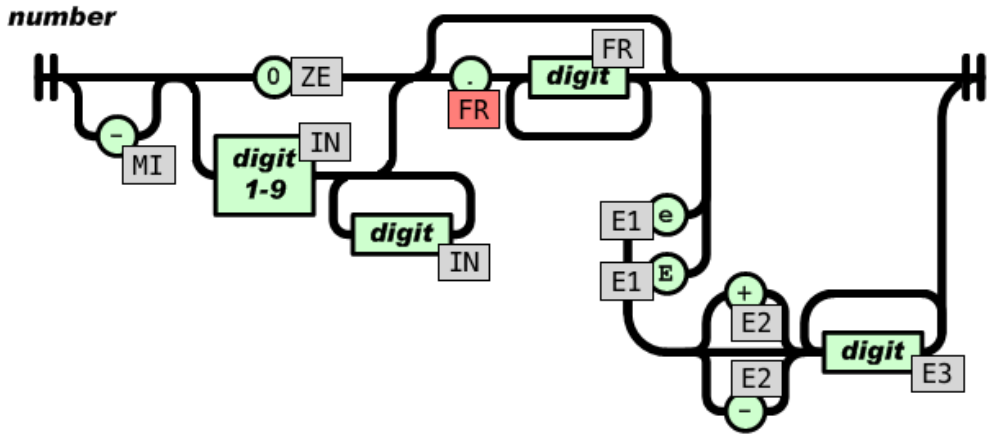
> *JSON_Checker is a Pushdown Automaton that very quickly determines if a JSON text is syntactically correct. It could be used to filter inputs to a system, or to verify that the outputs of a system are syntactically correct. It could be adapted to produce a very fast JSON parser.*

Even if JSON_Checker is not a formal reference implementation, one could expect JSON_Checker to clarify JSON specifications or at least implement them correctly.

Unfortunately, JSON_Checker violates the specifications defined on same web site. Indeed, JSON_Checker will parse the following inputs: `[1.]`, `[0.e1]`, which do not match JSON grammar.

Moreover, JSON_Checker will reject `[0e1]` which is a perfectly valid JSON number. This last bug is even more serious because a whole document can be rejected as long as it contains the number `0e1`.

The elegance of the JSON_Checker implementation as a pushdown automaton doesn't prevent the code from being wrong, but at least the state transition table makes it easy to spot the errors, especially when you add the states onto the schema of what is a number.




**Bug 1: rejection of 0e1** In the code, the state `ZE`, reached after parsing `0`, just lacks transitions to `E1` by reading `e` or `E`. We can fix this case by adding the two missing transitions.

**Bug 2: acceptance of [1.]** In one case, like after `0.`, the grammar requires a digit. In the other case, like after `0.1` the grammar doesn't. And yet JSON_Checker defines a single state `FR` instead of two. We can fix this case by replacing the `FR` state in red on the schema with a new state `F0` or `frac0`. With this fix, the parser will require a digit after `1.`.

```
static int state_transition_table[NR_STATES][NR_CLASSES] = {
/*
    The state transition table takes the current state and the current symbol,
    and returns either a new state or an action. An action is represented as a
    negative number. A JSON text is accepted if at the end of the text the
    state is OK and if the mode is MODE_DONE.

                 white                                      1-9                                   ABCDF  etc
             space |  {  }  [  ]  :  ,  "  \  /  +  -  .  0  |  a  b  c  d  e  f  l  n  r  s  t  u  |  E  |*/
/*start  GO*/ {GO,GO,-6,__,-5,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__},
/*ok     OK*/ {OK,OK,__,-8,__,-7,__,-3,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__},
/*object OB*/ {OB,OB,__,-9,__,__,__,__,ST,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__},
/*key    KE*/ {KE,KE,__,__,__,__,__,__,ST,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__},
/*colon  CO*/ {CO,CO,__,__,__,__,-2,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__},
/*value  VA*/ {VA,VA,-6,__,-5,__,__,__,ST,__,__,__,MI,__,ZE,IN,__,__,__,__,__,F1,__,N1,__,__,T1,__,__,__},
/*array  AR*/ {AR,AR,-6,__,-5,-7,__,__,ST,__,__,__,MI,__,ZE,IN,__,__,__,__,__,F1,__,N1,__,__,T1,__,__,__},
/*string ST*/ {ST,__,ST,ST,ST,ST,ST,ST,-4,ES,ST,ST,ST,ST,ST,ST,ST,ST,ST,ST,ST,ST,ST,ST,ST,ST,ST,ST,ST,ST},
/*escape ES*/ {__,__,__,__,__,__,__,__,ST,ST,ST,__,__,__,__,__,__,__,ST,__,__,__,ST,__,ST,ST,__,ST,U1,__,__,__},
/*u1     U1*/ {__,__,__,__,__,__,__,__,__,__,__,__,__,__,U2,U2,U2,U2,U2,U2,U2,U2,__,__,__,__,__,__,U2,U2,__},
/*u2     U2*/ {__,__,__,__,__,__,__,__,__,__,__,__,__,__,U3,U3,U3,U3,U3,U3,U3,U3,__,__,__,__,__,__,U3,U3,__},
/*u3     U3*/ {__,__,__,__,__,__,__,__,__,__,__,__,__,__,U4,U4,U4,U4,U4,U4,U4,U4,__,__,__,__,__,__,U4,U4,__},
/*u4     U4*/ {__,__,__,__,__,__,__,__,__,__,__,__,__,__,ST,ST,ST,ST,ST,ST,ST,ST,__,__,__,__,__,__,ST,ST,__},
/*minus  MI*/ {__,__,__,__,__,__,__,__,__,__,__,__,__,__,ZE,IN,__,__,__,__,__,__,__,__,__,__,__,__,__,__},

-/*zero   ZE*/ {OK,OK,__,-8,__,-7,__,-3,__,__,__,__,__,FR,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__},
+/*zero   ZE*/ {OK,OK,__,-8,__,-7,__,-3,__,__,__,__,__,F0,__,__,__,__,__,__,__,__,__,__,__,__,__,__,E1,__},

-/*int    IN*/ {OK,OK,__,-8,__,-7,__,-3,__,__,__,__,__,FR,IN,IN,__,__,__,__,E1,__,__,__,__,__,__,__,__,E1,__},
+/*int    IN*/ {OK,OK,__,-8,__,-7,__,-3,__,__,__,__,__,F0,IN,IN,__,__,__,__,E1,__,__,__,__,__,__,__,__,E1,__},

+/*frac0  F0*/ {__,__,__,__,__,__,__,__,__,__,__,__,__,__,FR,FR,__,__,__,__,__,__,__,__,__,__,__,__,__,__},

 /*frac   FR*/ {OK,OK,__,-8,__,-7,__,-3,__,__,__,__,__,__,FR,FR,__,__,__,__,E1,__,__,__,__,__,__,__,__,E1,__},
 /*e      E1*/ {__,__,__,__,__,__,__,__,__,__,__,__,__,__,E2,E2,__,__,__,__,__,__,__,__,__,__,__,__,__,__},
 /*ex     E2*/ {__,__,__,__,__,__,__,__,__,__,__,__,__,__,E3,E3,__,__,__,__,__,__,__,__,__,__,__,__,__,__},
 /*exp    E3*/ {OK,OK,__,-8,__,-7,__,-3,__,__,__,__,__,__,E3,E3,__,__,__,__,__,__,__,__,__,__,__,__,__,__},
 /*tr     T1*/ {__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,T2,__,__,__},
 /*tru    T2*/ {__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,T3,__},
 /*true   T3*/ {__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__},
 /*fa     F1*/ {__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,F2,__,__,__,__,__,__,__,__,__,__,__,__,__},
 /*fal    F2*/ {__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__},
 /*fals   F3*/ {__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__},
 /*false  F4*/ {__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__},
 /*nu     N1*/ {__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__},
 /*nul    N2*/ {__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__},
 /*null   N3*/ {__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__,__},
 };
```

Several other parsers (Obj-C TouchJSON, PHP, R rjson, Rust json-rust, Bash JSON.sh, C jsmn and Lua dkjson) will also erroneously parse `[1.]`. One may wonder if, at least in some cases, this bug may have spread from JSON_Checker because parser developers and testers used it as a reference, as advised on json.org.

[Update 2017-11-18] The aforementioned bugs have been fixed, and JSON Checker is now published on Douglas Crockford's GitHub.

#### 4.9 Regex

We may wonder if a regex can validate the conformance to JSON grammar of a given input. See for instance this attempt to find the shortest regex on StackExchange: Write a JSON Validator. The problem is that it is very difficult to know if a regex does succeed or not without a solid test suite.

I found this Ruby regex to validate JSON on StackOverflow to be the best one:

```
JSON_VALIDATOR_RE = /(
    # define subtypes and build up the json syntax, BNF-grammar-style
    # The {0} is a hack to simply define them as named groups here but not mat
    # I added some atomic grouping to prevent catastrophic backtracking on in
    (?<number>  -?(?=[1-9]|0(?!\d))\d+(\.\d+)?([eE][+-]?\d+)?){0}
    (?<boolean> true | false | null ){0}
    (?<string>  " (?>[^"\\\\]* | \\\\ ["\\\\bfnrt\/] | \\\\ u [0-9a-f]{4} )* 
    (?<array>   \[ (?> \g<json> (?: , \g<json> )* )? \s* \] ){0}
    (?<pair>    \s* \g<string> \s* : \g<json> ){0}
    (?<object>  \{ (?> \g<pair> (?: , \g<pair> )* )? \s* \} ){0}
    (?<json>    \s* (?> \g<number> | \g<boolean> | \g<string> | \g<array> | \
    )
    \A \g<json> \Z
    /uix
```

Yet, it fails to parse valid JSON, such as:

- u-escaped codepoints, including valid ones: `["\u002c"]`

- backslash-escaped backslash: `["\\a"]`

Also, it does parse the following extensions, which is just a bug for a JSON validator:

- a capitalized True: `[True]`
- an unescaped control character: `["09"]`

## 5. Parsing Contents

RFC 8259 Section 9 says:

> *A JSON parser transforms a JSON text into another representation.*

All of the above testing architecture will only tell if a parser would parse a JSON document or not, but doesn't say anything about the representation of the resulting contents.

For example, a parser may parse the u-escaped invalid Unicode character (`"\uDEAD"`) without error, but what will the result be like? a replacement character, or something else, who knows? RFC 8259 is silent about it.

Similarily, extreme numbers such as `0.000000000000000000000001` or `-0` can be parsed, but what should the result be? RFC 8259 doesn't make a distinction between integers and doubles, or zero and -zero. It doesn't even say if numbers can be converted into strings or not.

And what about objects with the same keys (`{"a":1,"a":2}`)? Or same keys and same values (`{"a":1,"a":1}`)? And how should a parser compare object keys?? Should it use binary comparison or a Unicode normal form such as NFC? Here again, RFC is silent

In all these cases, parsers are free to output whatever they want, leading to interoperability issues (think of what could go wrong when you decide to change your usual JSON parser with another one).

With that in mind, let's create tests for which the representation after parsing is not clearly defined. These tests serve only to document how parsers output may differ.

Contrary to the parsing tests, these tests are hard to automate. Instead, the results shown here were observed via log statements and/or debuggers.

Below is an inexhaustive list of some striking differences between resulting representations after parsing. All results are shown in appendix "Parsing Contents".

**Numbers**

- `1.000000000000000005` is generally converted into the float `1.0`, but Rust 1.12.0 / json 0.10.2 will keep the original precision and use the number `1.000000000000000005`
- `1E-999` is generally converted into float or double `0.0`, but Swift Freddy yields the string `"1E-999"`. Swift Apple JSONSerializattion and Obj-C JSONKit will simply refuse to parse it and return an error.
- `10000000000000000999` may be converted into a double (Swift Apple JSONSerialization), an unsigned long long (Objective-C JSONKit) or a string (Swift Freddy). It is to be noted that C cJSON will parse this number as a double, but loses precision in the process, resulting in a new number `10000000000000002048` (note the last four digits).

**Objects**

- In `{"C3A9":"NFC","65CC81":"NFD"}` keys are NFC and NFD representations of "é". Most parsers will yield the two keys, except Swift parsers Apple JSONSerialization and Freddy, where dictionaries first normalize keys before testing them for equality.
- `{"a":1,"a":2}` does generally result in `{"a":2}` (Freddy, SBJSON, Go, Python, JavaScript, Ruby, Rust, Lua dksjon), but may also result in `{"a":1}` (Obj-C Apple NSJSONSerialization, Swift Apple JSONSerialization, Swift Freddy), or `{"a":1,"a":2}` (cJSON, R, Lua JSON).
- `{"a":1,"a":1}` does generally result in `{"a":1}`, but is parsed as `{"a":1,"a":1}` in cJSON, R and Lua JSON.
- `{"a":0,"a":-0}` is generally parsed as `{"a":0}`, but can also be parsed as `{"a":-0}` (Obj-C JSONKit, Go, JavaScript, Lua) or even `{"a":0, "a":0}` (cJSON, R).

**Strings**

- `["A\u0000B"]` contains the u-escaped form of the `0x00 NUL` character, which is likely to cause problems in C-based JSON parsers. Most parsers handle this payload gracefully, but JSONKit and

cJSON won't parse it. Interestingly, Freddy yields only `["A"]` (the string stop after unescaping byte `0x00`).

- `["\uD800"]` is the u-escaped form of `U+D800`, an invalid lone UTF-16 surrogate. Many parsers will fail and return an error, despite the string being perfectly valid according to JSON grammar. Python leaves the string untouched and yields `["\uD800"]`. Go and JavaScript replace the offending chacracter with "`&#65533;`" `U+FFFD REPLACEMENT CHARACTER` `["EFBFBD"]`, R rjson and Lua dkjson simply translate the codepoint into its UTF-8 representation `["EDA080"]`. R jsonlite and Lua JSON 20160728.17 replace the offending codepoint with a question mark `["?"]`.

- `["EDA080"]` is the non-escaped, UTF-8 form or `U+D800`, the invalid lone UTF-16 surrogate discussed in previous point. This string is not valid UTF-8 and should be rejected (see section 2.5 Strings - Raw non-Unicode Characters). In practice however, several parsers leave the string untouched `["EDA080"]` such as cJSON, R rjson and jsonlite, Lua JSON, Lua dkjson and Ruby. Go and Javacript yield `["EFBFBDEFBFBDEFBFBD"]` that is three replacement characters (one per byte). Interestingly, Python 2 converts the sequence into its unicode-escaped form `["\ud800"]`, while Python 3 thows a `UnicodeDecodeError` exception.

- `["\uD800\uD800"]` makes some parsers go nuts. R jsonlite yields `["\U00010000"]`, while Ruby parser yields `["F0908080"]`. I still don't get where this value comes from.

**[Update 2017-11-18]** A RCE vulnerability was found in CouchDB because two JSON parsers handle duplicate key differently. The same JSON object, when parsed in JavaScript, contains `"roles": []`', but when parsed in Erlang it contains `"roles": ["_admin"]`.

## 6. STJSON

STJSON is a Swift 3, 600+ lines JSON parser I wrote to see what it took to consider all pitfalls and pass all tests.

https://github.com/nst/STJSON

STJSON API is very simple:

```
var p = STJSONParser(data: data)

do {
    let o = try p.parse()
    print(o)
} catch let e {
    print(e)
}
```

STJSON can be instantiated with additional parameters:

```
var p = STJSON(data:data,
               maxParserDepth:1024,
               options:[.useUnicodeReplacementCharacter])
```

In fact, there is only one test where STJSON fails: `y_string_utf16.json`. This is because, as in nearly all other parsers, STJSON does not support non UTF-8 encodings, even though it should not be very difficult to add, and I may do so in the future if needed. At least, STJSON does raise appropriate errors when a file starts with a UTF-16 or UTF-32 byte order mark.

## 7. Conclusion

In conclusion, JSON is not a data format you can rely on blindly. I've demonstrated this by showing that the standard definition is spread out over at least seven different documents (section 1), that the latest and most complete document, RFC-8259, is imprecise and contradictory (section 2), and by crafting test files that out of over 30 parsers, no two parsers parsed the same set of documents the same way (section 4).

In the process of inspecting parser results, I also discovered that json_checker.c from json.org did reject valid JSON `[0e1]` (section 4.24), which certainly doesn't help users to know what's right or wrong. In a general way, many parsers authors like to brag about how right is their parsers (including myself), but there's no way to assess their quality since references are debatable and existing test suites are generally poor.

So, I wrote yet another JSON parser (section 6) which will parse or reject JSON document according to my understanding of RFC 8259. Feel free to comment, open issues and pull requests.

This work may be continued by:

- Documenting the behaviour of **more parsers**, especially parsers that run in non-Apple environments.
- Investigating **JSON generation**. I extensively assessed what parsers do or do not parse. (section 4). I briefly assessed the contents that parsers yield when the parsing is successful (section 5). I'm pretty sure that several parsers do generate grammatically invalid JSON or even crash in some circumstances (see Section 4.2.1).
- Investigating differences in the way **JSON mappers** maps JSON contents to model objects.
- **Finding exploits** in existing software stacks (check out my Unicode Hacks presentation)
- Investigating potential interoperability issues in **other serialization formats** such as YAML, BSON or ProtoBuf, which may be a potential successor to JSON. Indeed, Apple already has a Swift implementation https://github.com/apple/swift-protobuf-plugin.

As a final word, I keep on wondering why "fragile" formats such as HTML, CSS and JSON, or "dangerous" languages such as PHP or JavaScript became so immensely popular. This is probably because they are easy to start with by tweaking contents in a text editor, because of too liberal parsers or interpreters, and seemingly simple specifications. But sometimes, simple specifications just mean hidden complexity.

## 8. Appendix

1. Parsing Results http://seriot.ch/json/parsing.html, generated automatically for section 4
2. Tranform Results http://seriot.ch/json/transform.html, created manually for section 6
3. JSONTestSuite https://github.com/nst/JSONTestSuite, contains all tests and code
4. STJSON https://github.com/nst/STJSON, contains my Swift 3 JSON parser

## Acknowledgments

Many thanks to @Reversity, GEndignoux, @ccorsano, @BalestraPatrick and @iPlop.