The `jsonparse` package

A handy way to parse, store and access JSON data from files or strings in LaTeX documents

Jasper Habicht *

Version 1.2.3, released on 23 March 2025

---

1 Introduction

Hello guys, I am Jason, the JSON parsing horse. JSON data is my favorite thing to parse! But I found that converting JSON to TeX can be a bit tricky. Therefore, I created this package which I am happy to introduce to you.

The `jsonparse` package provides a handy way to read in JSON data from files or strings in LaTeX documents, parse the data and store it in a user-defined token variable. The package allows accessing the stored data via a JavaScript-flavored syntax.

The package has been tested, but not exhaustively. The author is grateful for reporting any bugs via GitHub at `https://github.com/jasperhabicht/jsonparse/issues`. A site for asking questions about how to use the package and for suggestions for improvement is available at `https://github.com/jasperhabicht/jsonparse/discussions`.

2 Loading the package

To install the package, copy the package file `jsonparse.sty` into the working directory or into the `texmf` directory. After the package has been installed, the `jsonparse` package is loaded by calling `\usepackage{jsonparse}` in the preamble of the document.

The package does not load any dependencies. It can be used with PDFLaTeX, LuaLaTeX or XeLaTeX.

debug

The package can be loaded with the option `debug`. It will then output to the log file every instance of a string, a boolean (true or false) value, a null value, a number as well as the start and end of every object and the start and end of every array that is found while parsing the JSON string or JSON file.

* E-mail: mail@jasperhabicht.de. I am grateful to Joseph Wright, Jonathan P. Spratte and David Carlisle who helped me navigating the peculiarities of TeX and optimizing the code.

This key can be set either as package option or using `\JSONParseSet`. It can also be set locally as option to the commands `\JSONParse` and `\JSONParseFromFile`.

3 General remarks of the parsing procedure

In general, the package will read and store the JSON source and data as string, which means that all characters have category code 12 ("other"), except for spaces and (horizontal) tabs which have category code 10 ("space"). The `\endlinechar` value is set to -1 which means that linefeeds and carriage returns are ignored by TeX. These settings are in line with the JSON specification of handling whitespace. Furthermore, if PDFLaTeX is used, the upper-half of the 8-bit range is set to "active". Additionally, JSON defines a small set of escape sequences and in order to be able to process these, the category code of the backslash is set to 0 ("escape").

During parsing, the package identifies JSON objects, arrays, strings, numbers, boolean values and null values from the JSON data. It stores all these values together with the relevant keys in a property list. Once the parsing process is done, every value can be retrieved from the property list by calling the relevant key. The package ignores whitespace in the JSON data. The JSON data should be an object or an array. In general, the package accepts any valid JSON data. If a key is defined multiple times, the latter definition will silently overwrite the former.

4 Escaping and special treatment of the input

JSON strings cannot contain the two characters `"` and `\`. These two characters need to be escaped with a preceding backslash (`\`). This package therefore redefines locally the TeX control symbols `\"`, `\/`, `\\`, `\b`, `\f`, `\n`, `\r`, `\t` and `\u`. These control symbols are prevented from expanding during parsing. For example, `\"` is first defined as `\exp_not:N \"` and only when typeset, `\"` is expanded to `"`, which ensures that strings are parsed properly.

Similarly, the control symbol `\/` expands eventually to `/` and `\\` to `\c_backslash_str` (i. e. a backslash with category code 12).

The escape sequence `\u` followed by a hex value consisting of four digits eventually expands to `\codepoint_generate:nn` that creates the character represented by the relevant four hex digits with category code 12 ("other"). If two escape sequences `\u` with four hex digits each follow each other and together represent a Unicode surrogate pair, this surrogate pair is converted into the relevant Unicode codepoint.

The JSON escape sequences `\b`, `\f`, `\n`, `\r` and `\t` eventually expand to token variables of which the contents can be set using the relevant `replace` key. See more on setting options below in section 7.

It is possible to insert TeX macros to the JSON source that will eventually be parsed when typesetting. Backslashes of TeX macros need to be escaped by another backslash. The TeX macros `\"` and `\\` must be escaped twice in the JSON source so that they become `\\\"` and `\\\\` respectively.

```
\x{‹token variable name›}{‹key›}
```

Using the control sequence `\x`, it is possible to nest JSON strings into each other. Used inside the `\JSONParse` command, the control sequence takes two arguments delimited by curly braces. The first argument represents the name of the token variable that holds the parsed JSON data where the inserted JSON string should be taken from. The second argument sets the key that should be selected. The following example shows a simple use case:

```
                                \JSONParse{\myJSONdataA}{
                                    { "a" : { "b" : "c" } }
                                }

c                               \JSONParse{\myJSONdataB}{
                                    { "d" : \x{myJSONdataA}{a} }
                                }

                                \JSONParseValue{\myJSONdataB}{d.b}
```

Note that the control sequence `\x` is replaced by the value exactly. Therefore, if the value happens to be a string, the control sequence `\x` should be placed between quotation marks (`"`) in order for the resulting string to be valid JSON. The control sequence `\x` is only available inside the `\JSONParse` command, but not inside the `\JSONParseFromFile` command.

```
escape={all}
escape={none}
escape={number sign}
escape={dollar sign}
escape={percent sign}
escape={ampersand}
escape={circumflex accent}
escape={low line}
escape={tilde}
```

The key `escape` can be used to convert characters that don't require escaping in JSON but in TeX into the relevant TeX escape sequences. Apart from the backslash and curly braces that need to be escaped anyways, these are the number sign, the dollar sign, the percent sign, the ampersand, the circumflex accent, the low line and the tilde. The characters can be selected individually separated by a comma (for example `escape={dollar sign, circumflex accent, low line}`. With `escape={all}`, all escaping sequences are selected, with `escape={none}`, none is selected.

The naming of the relevant characters follows their Unicode names. However, `hash` exists as alias for `number sign`, `dollar` as alias for `dollar sign`, `percent` for `percent sign`, `circumflex` for `circumflex accent` and `underscore` for `low line`.

This key can be set using `\JSONParseSet`. It can also be set locally as option to the commands `\JSONParseValue`, `\JSONParseArrayUse` and `\JSONParseArrayMapFunction`.

```
rescan
rescan={‹boolean›}
```

The key `rescan` can be used to activate and deactivate rescanning of the output. This key is active per default. Rescanning converts all tokens to their default category codes and TeX control sequences are expanded before typesetting. Further, during the rescanning process, JSON escape sequences are replaced and characters that don't require escaping in JSON but in TeX are replaced by the relevant TeX escape sequences.

This key can be set using `\JSONParseSet`. It can also be set locally as option to the commands `\JSONParseValue`, `\JSONParseArrayUse` and `\JSONParseArrayMapFunction`.

5 Main user commands

The first part of this section describes the basic commands for parsing JSON data and retrieving values from parsed JSON data. The second part of this section describes the various commands for handling arrays provided by this package.

5.1 Basic parsing commands

\JSONParse[‹options›]{‹token variable›}{‹JSON string›}

The command `\JSONParse` is used to parse a JSON string and globally store the parsed result in a token variable (a property list). The second argument takes the name of the token variable that is created by the command. The third argument takes the JSON string to be parsed.

For example, using `\JSONParse{\myJSONdata}{ { "key" : "value" } }`, the relevant JSON string will be parsed and the result stored in the token variable `\myJSONdata` as property list. In this case, the property list only consists of one entry with the key `key` and the value `value`. The command `\JSONParseValue{\myJSONdata}{key}`, for example, can then be used to extract the relevant value from this property list (see the description below).

The first optional argument can be used to pass options to the command that are then applied locally.

\JSONParseFromFile[‹options›]{‹token variable›}{‹JSON file›}

The command `\JSONParseFromFile` is used to parse a JSON file and store the parsed result in a token variable (a property list). It works the same way as `\JSONParse`, but instead of a JSON string, it takes as third argument the path to the JSON file relative to the working directory.

\JSONParseValue[‹options›]{‹token variable›}{‹key›}

The command `\JSONParseValue` is used to select values from the token variable (property list) that has been created using the commands `\JSONParse` or `\JSONParseFromFile`. The second argument takes the token variable that holds the parsed JSON data. The third argument takes the key to select the relevant entry from the parsed JSON data using JavaScript syntax.

If the JSON string `{ "key" : "value" }` is parsed into the token variable `\myJSONdata`, using `\JSONParseValue{\myJSONdata}{key}` would extract the value associated with the key `key`, which in this case is `value`, and typeset it to the document.

Nested objects and arrays are assigned keys that adhere to JavaScript syntax. For example, if the JSON string `{ "outer_key" : { "inner_key" : "value" } }` is parsed into the token variable `\myJSONdata`, to select the value associated with the key `inner_key`, the command `\JSONParseValue{\myJSONdata}{outer_key.inner_key}` can be used. To give an example for an array, the command `\JSONParseValue{\myJSONdata}{key[0]}` selects the first value of the array associated with the key `key` in the JSON string `{ "key" : [ "one" , "two" ] }`.

The first optional argument can be used to pass options to the command, such as `escape` or `rescan`, that are then applied locally. When the option `rescan` is used, the token list is rescanned before it is typeset (which means that all category codes that may have been changed before are set to the default values). This is the default behavior. If rescanning is not desired, pass the option `rescan=false` to the command.

When a key is associated with an object or array, the whole object or array is output as JSON string. The special key `.` (or the string defined using the key `child sep`) returns the whole JSON object (or the whole JSON array if the JSON data only consists of one array) as string where all characters (except for spaces and tabs) have category code 12 (“other”).

The command `\JSONParseValue` is not expandable and can therefore not be used as argument of certain other arguments where expansion is needed. In such cases, the expandable command `\JSONParseExpandableValue` should be used.

store in={‹token variable›}

The command `\JSONParseValue` accepts the key `store in` that can be used to store the return value in another token variable. If the token variable given as option to the `store in` key has not yet been defined, it will be created by this command.

The token list returned by this command is a string variable where all characters have category code 12 (“other”), except for spaces and (horizontal) tabs that have category code 10 (“space”).

The key `store in` can be used together with the key `rescan` to rescan the return value before storing it in the token variable. This means that the value stored in the token list will have the category codes TeX uses per default. Option settings such as the `escape` option are taken into consideration during the rescan process.

This can, for example, be necessary when numbers stored in the JSON data in scientific format should be formatted using the `siunitx` package. The rescan is needed here, because otherwise the character `e` would have the wrong category code and would hence not be recognized by the formatting parser as exponent marker. Let us assume the key `number` in some JSON source parsed into the token variable `\myJSONnumber` represents the value `-1.1e-1`, then the following could be used to format the output:

-1.1×10^{-1}

```
\JSONParseValue
  [rescan, store in=\mynumber]
  {\myJSONnumber}{number}
\num{\mynumber}
```

The key `store in` can also be set using `\JSONParseSet`. Calling `store in={}` will reset it to its default (empty) value.

`\JSONParseExpandableValue``{`‹token variable›`}{`‹key›`}` ★

Whole objects or arrays can be output as JSON string for further use in other macros using the expandable command `\JSONParseExpandableValue`. The value that is returned by this command is typically a string variable where all characters have category code 12 (“other”), except for spaces and (horizontal) tabs that have category code 10 (“space”). This should be kept in mind if string comparisons should be made. A comparison against a token list with the default category codes used by TeX won’t work, since letters will have category code 11 (“letter”), but it is possible to use `\detokenize` to set the category codes of the token list in such a way that the comparison works.

For example, if the JSON string `{ "key" : "value" }` has been parsed into the token variable `\myJSONdata`, the command `\JSONParseExpandableValue{\myJSONdata}{key}` will have the same meaning as `\detokenize{value}` and expand to a token list with all characters having category code 12 (“other”).

`\JSONParseKeys``[`‹options›`]{`‹token variable›`}{`‹key›`}`

The command `\JSONParseKeys` is used to get all top-level keys of a JSON object as JSON array and return this array as string where all characters (except for spaces and tabs) have category code 12 (“other”). The first argument of the command takes the token variable that holds the parsed JSON data. The second argument takes the key to select the relevant entry from the parsed JSON data using JavaScript syntax.

The command `\JSONParseKeys` accepts as option the key `store in` to get all top-level keys of a JSON object as JSON array and parse this array into a token variable. Note that the return value is stored as property list, not as string. The token variable to store the keys as array is created if it does not exist.

`\JSONParseFilter``{`‹token variable›`}{`‹token variable›`}{`‹key›`}`

The command `\JSONParseFilter` is used to select a part (such as an object or an array) of a JSON object or JSON array and parse this into a token variable (a property list). The first argument denotes the token variable where the value should be stored into. The second argument of the command takes the token variable that holds the parsed JSON data. The third argument takes the key to select the relevant entry from the parsed JSON data using JavaScript syntax.

5.2 Commands for handling arrays

The package offers a variety of commands that can be used to process JSON arrays. Three commands are provided to loop through arrays, `\JSONParseArrayUse`, `\JSONParseArrayMapFunction` and `\JSONParseArrayMapInline` which offer different functionality for different use cases. All three commands are implemented in a unique way and it should not be expected that what works with one of these commands also works with another. The commands differ in various respects, for example:

- With `\JSONParseArrayUse` and `\JSONParseArrayMapInline`, it is possible to store the result in a token list for later use via the option key `store in`, but such is not possible with `\JSONParseArrayMapFunction`.
- It is possible to store non-expandable commands (such as `\emph` or `\textbf`) in a token list using `\JSONParseArrayUse`, but not using `\JSONParseArrayMapInline`.

\JSONParseArrayCount[‹options›]{‹token variable›}{‹key›}

The command `\JSONParseArrayCount` takes as first argument a token variable holding a parsed JSON string or JSON file and as second argument a key to select an array in the JSON data. It returns an integer representing the number of items contained in the selected array.

The command `\JSONParseArrayCount` accepts the use of the key `store in` to store the number of items contained in the selected array in a token variable.

\JSONParseArrayUse[‹options›]{‹token variable›}{‹key›}[‹subkey›]{‹string›}

The command `\JSONParseArrayUse` is used to select all values from an array from a parsed JSON string or JSON file. The second argument takes the token variable that holds the parsed JSON data. The first argument takes the key to select the relevant entry from the parsed JSON data using JavaScript syntax. The third argument is optional and can be used to pass a subkey, i. e. a key that is used to select a value for every item. The last argument takes a string that is inserted between all values when they are typeset.

For example, let us assume the following JSON data structure is parsed into the token variable `\myJSONdata`:

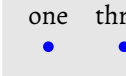
```
{
  "array" : [
    {
      "key_a" : "one" ,
      "key_b" : "two"
    } ,
    {
      "key_a" : "three" ,
      "key_b" : "four"
    }
  ]
}
```

When using `\JSONParseArrayUse{\myJSONdata}{array}[key_a]{, }`, 'one, three' is then typeset to the document.

The first optional argument can be used to pass options to the command, such as `escape` or `rescan`, that are then applied locally.

The command `\JSONParseArrayUse` accepts as option set in the optional argument the key `store in` which takes a token variable into which the result of the command should be stored. Storing the result of the mapped inline function can be helpful if JSON data should be reformatted for use in another function.

one three



```
\JSONParseArrayUse[store in=\myJSONitems]
  {\myJSONdata}{array}[key_a]{,}

\begin{tikzpicture}
  \foreach \x [count=\i] in \myJSONitems {
    \fill[blue] (\i,0) circle[radius=2pt]
      node[above=5pt, black] {\x};
  }
\end{tikzpicture}
```

`\JSONParseArrayMapFunction``[`‹options›`]{`‹token variable›`}{`‹key›`}[`‹subkey›`]`
`{`‹command›`}`

The command `\JSONParseArrayMapFunction` works in a similar way and takes the same first three arguments as the command `\JSONParseArrayUse`. However, instead of a string that is added between the array items, it takes a command (a token list) as fourth argument. This command can be defined beforehand and will be called for every array item. Inside its definition, the commands `\JSONParseArrayIndex`, `\JSONParseArrayKey` and `\JSONParseArrayValue` can be used which are updated for each item and output the index, the key and the value of the current item respectively. Note that these commands are defined globally to make accessing them as easy as possible.

For example, let us assume the same JSON data structure as defined above parsed into the token variable `\myJSONdata`. Then, the following can be done:

- *one*
- *three*

```
\newcommand{\myJSONitem}{
  \item \emph{\JSONParseArrayValue}
}

\begin{itemize}
  \JSONParseArrayMapFunction{\myJSONdata}
    {array}[key_a]{\myJSONitem}
\end{itemize}
```

It is possible to make use of multiple subkeys by passing them as a comma separated list as third argument to the command. Inside the command that is called for every array item, the different keys and values can be access via commands numbered with uppercase Roman numerals such as `\JSONParseArrayKeyI`, `\JSONParseArrayKeyII`, `\JSONParseArrayKeyIII` etc. and `\JSONParseArrayValueI`, `\JSONParseArrayValueII`, `\JSONParseArrayValueIII` etc.

We can extend the above example in the following way:

- *one:* two
- *three:* four

```
\newcommand{\myJSONitem}{
  \item \emph{\JSONParseArrayValueI :}
    \JSONParseArrayValueII
}

\begin{itemize}
  \JSONParseArrayMapFunction{\myJSONdata}
    {array}[key_a,key_b]{\myJSONitem}
\end{itemize}
```

code before={‹code›}
code after={‹code›}

The `\JSONParseArrayMapFunction` command also accepts the options `code before` and `code after`. These options can be used to place code before and after the output that is generated by the command called for every array item, for example for typesetting tabular contents.

Typesetting the above example in a tabular way can be achieved as follows:

key a	key b
one	two
three	four

```
\newcommand{\myJSONitem}{
  \JSONParseArrayValueI &
  \JSONParseArrayValueII \\
}

\JSONParseArrayMapFunction[
    code before={
      \begin{tabular}{ c c }
        \textbf{key a} &
        \textbf{key b} \\ \hline
    },
    code after={
      \hline \end{tabular}
    }
  ]{\myJSONdata}
  {array}[key_a,key_b]{\myJSONitem}
```

Finally, the first optional argument of the command can be used to pass options to the command, such as `escape` or `rescan`, that are then applied locally.

\JSONParseArrayMapInline[‹options›]{‹token variable›}{‹key›}{‹inline function›}

The command `\JSONParseArrayMapInline` takes as first mandatory argument a token variable holding a parsed JSON string or JSON file and as second mandatory argument a key to select an array in the JSON data. The last argument can contain any code where the index of the current item is represented by `#1`.

Using the above example, the mechanism could be implemented as follows:

- one
- three

```
\begin{itemize}
  \JSONParseArrayMapInline{\myJSONdata}
    {array}{
    \item \JSONParseValue{\myJSONdata}
      {array[#1].key_a}
  }
\end{itemize}
```

Making use of the commands `\JSONParseKeys` and `\JSONParseValue` together with the `store in` option, keys and values can be accessed. Due to the fact that cells create scopes, we need to repeat the part of the code that selects the current key:

key_a: one
key_b: two

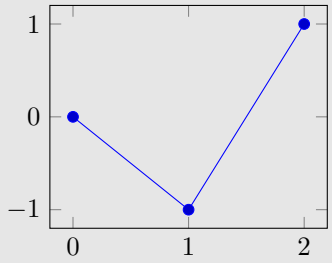
key_a: three
key_b: four

```
\JSONParseArrayMapInline{\myJSONdata}
  {array}{
  \JSONParseKeys[store in=\mykeys]
    {\myJSONdata}{array[#1]}
  \JSONParseValue
    [store in=\mykeya, rescan=false]
    {\mykeys}{[0]}
  \JSONParseValue
    [store in=\mykeyb, rescan=false]
    {\mykeys}{[1]}

  \emph{\mykeya :}
  \JSONParseValue{\myJSONdata}
    {array[#1].\mykeya}\par

  \emph{\mykeyb :}
  \JSONParseValue{\myJSONdata}
    {array[#1].\mykeyb}\par\bigskip
}
```

Note that the underscores in the names of the keys can be printed without changing to math mode in the above example by switching off rescanning via `rescan=false`. This is possible because all JSON data is stored as string where all characters (except for spaces and tabs) have category code 12 (“other”).

The command `\JSONParseArrayMapInline` accepts as option set in the optional argument the key `store in` which takes a token variable into which the result of the mapped inline function should be stored. Refer to the relevant explanations to command `\JSONParseArrayUse` above for more information. It is important to note that the inline function needs to be fully expandable. For example, it is not possible to use `\JSONParseValue` in the code of the inline function while `\JSONParseExpandableValue` is allowed.

Storing the result of the mapped inline function can be helpful if JSON data should be reformatted for use in a plotting functions. An example for a use case with PGFplots is shown below. In this example, the parsed JSON string `{ "data": [ [0,0], [1,-1], [2,1] ] }` was stored in the token variable `\myJSONplotdata`.




```
\JSONParseArrayMapInline
  [store in={\myJSONplotcoords}, global]
  {\myJSONplotdata}{data}{
    (
    \JSONParseExpandableValue
      {\myJSONplotdata}{data[#1][0]}
    ,
    \JSONParseExpandableValue
      {\myJSONplotdata}{data[#1][1]}
    )
}

\begin{tikzpicture}
  \begin{axis}
    \addplot coordinates
      {\myJSONplotcoords};
  \end{axis}
\end{tikzpicture}
```

6 Externalizing parsed JSON data

Parsing large and complex JSON files can take quite a while. In order to speed up follow-up compilation runs, this package provides a way to store parsed JSON data for future use. Once a file for externalization has been created, the package will try to load the data from this file instead of parsing the JSON data again.

externalize
externalize={‹boolean›}

With the key `externalize` set (or set to true), a file will be created in the working directory that stores the externalization of the parsed JSON data. The file name gets the extension `.jsonparse`. The file name is created automatically and consists of the name of the current file followed by an underscore and the name of the token variable where the JSON data is stored into. If a file with the same name and file extension already exists, an error will be issued.

This key can be set using `\JSONParseSet`. It can also be set locally as option to the commands `\JSONParse` and `\JSONParseFromFile`.

externalize prefix={‹string›}

With the key `externalize prefix`, a prefix can be defined that is added to the file name. Per default this is an empty string.

This key can be set using `\JSONParseSet`. It can also be set locally as option to the commands `\JSONParse` and `\JSONParseFromFile`.

externalize file name={‹token list›}

The key `externalize file name` sets the schema for the file name. The default schema is as follows:

```
\l_jsonparse_externalize_prefix_str \c_sys_jobname_str
  \c_underscore_str \l_jsonparse_current_prop_str
```

The token variable `\l_jsonparse_externalize_prefix_str` contains the prefix that is set using the key `externalize prefix`. `\c_sys_jobname_str` holds the name of the current file (the current job name), `\c_underscore_str` is an underscore and the token variable `\l_jsonparse_current_prop_str` contains the name of the property list where the relevant JSON data is stored into.

This key can be set using `\JSONParseSet`. It can also be set locally as option to the commands `\JSONParse` and `\JSONParseFromFile`.

```
\JSONParsePut{‹token variable›}{‹key›}[‹JSON string›]
```

The command `\JSONParsePut` is used by the externalization procedure to re-read already parsed JSON data to the main file. It just adds a key-value pair to the property list (where the value part is read as string). Hence, it can also be used to append more entries to an already existing property list containing parsed JSON data.

7 Changing parsing and typesetting behavior via option keys

The package provides a set of keys that can be set to change the separators used to select the relevant value in the JSON structure, the output that is generated from the JSON data as well as other things.

```
\JSONParseSet{‹options›}
```

The command `\JSONParseSet` can be used to specify options via key-value pairs (separated by commas). Keys that are presented here as a subkey (i. e. preceded by another key and a slash such as `key/subkey`) can also be set using the syntax `key={subkey}` and multiple subkeys belonging to one key can be combined using commas as separator. Several user commands allow to pass keys directly which are then applied locally.

Not every key takes effect in every situation. Some keys affect the parsing procedure and thus need to be set before parsing. Some keys affect the typeset result and some keys only affect the typeset result when used in combination with specific commands.

7.1 Keys affecting the parsing procedure

Information about the key `externalize` as well as about the related keys `externalize prefix` and `externalize file name` can be found above in section 6.

```
separator/child={‹string›}
separator/array left={‹string›}
separator/array right={‹string›}
```

With the key `separator/child`, the separator for child objects that is used in the key to select a specific value in the JSON data structure can be changed. Per default, the child separator is a dot (`.`).

With the keys `separator/array left` and `separator/array right`, the separators for arrays that are used in the key to select a specific value in the JSON data structure can be changed. Per default, the separators are square brackets (`[` and `]`). Changing these separators to curly braces (`{}`) is not supported due to their grouping function in TeX.

Changing the separators can be useful if keys in the JSON structure already use these characters. These settings take place already during parsing.

These keys can be set using `\JSONParseSet`. They can also be set locally as option to the commands `\JSONParse` and `\JSONParseFromFile`. When set using `\JSONParseSet`, these keys only take effect when set before parsing.

```
zero-based
zero-based={⟨boolean⟩}
```

If the key `zero-based` is set (or explicitly set to `true`), the index of array items starts with zero. If set to false, the indexing starts with one instead. Per default, the package uses zero-based indexing to match JavaScript notation. This setting affects indexing already during parsing.

This key can be set using `\JSONParseSet`. It can also be set locally as option to the commands `\JSONParse` and `\JSONParseFromFile`. When set using `\JSONParseSet`, this key only takes effect when set before parsing.

```
check num
check num={⟨boolean⟩}
```

If set to `false`, the key `check num` omits an internal check of numerical expressions against the JSON specification for numbers. Turning off this feature can increase the parsing speed if many numbers are to be parsed. Checks are carried out per default.

This key can be set using `\JSONParseSet`. It can also be set locally as option to the commands `\JSONParse` and `\JSONParseFromFile`.

7.2 Keys affecting the typesetting

Some keys that change the typesetting behavior are explained in other parts of this documentation.

- Information about the keys `escape` and `rescan` can be found above in section 4.
- Information about the key `store in` can be found above in the context of the description to the command `\JSONParseValue` as well as in the description to `\JSONParseKeys`, `\JSONParseArrayCount`, `\JSONParseArrayUse` and `\JSONParseArrayMapInline`.
- Information about the keys `code before` and `code after` can be found above in the description to the command `\JSONParseArrayMapFunction` above.

```
keyword/true={⟨string⟩}
keyword/false={⟨string⟩}
keyword/null={⟨string⟩}
```

With the keys `keyword/true`, `keyword/false` and `keyword/null`, the string that is typeset for true, false and null values can be changed. The default strings that are typeset are `true`, `false` and `null` respectively. Only strings can be used as replacement. These replacements take place already during parsing.

These keys can be set using `\JSONParseSet`. They can also be set locally as option to the commands `\JSONParse` and `\JSONParseFromFile`. When set using `\JSONParseSet`, these keys only take effect when set before parsing.

```
replace/backspace={⟨string⟩}
replace/formfeed={⟨string⟩}
replace/linefeed={⟨string⟩}
replace/carriage return={⟨string⟩}
replace/horizontal tab={⟨string⟩}
```

These keys can be used to set the replacement text for the JSON escape sequences `\b` (backspace), `\f` (formfeed), `\n` (linefeed), `\r` (carriage return) and `\t` (horizontal tab). The default replacement string is a space in each case. Only strings can be used as replacement. These replacements take place only during typesetting.

These keys can be set using `\JSONParseSet`. They can also be set locally as option to the commands `\JSONParseValue`, `\JSONParseArrayUse` and `\JSONParseArrayMapFunction`.

```
global
global={⟨boolean⟩}
```

The key `global` can be used together with the key `store in` to globally set the value of the relevant token list. Detailed information about the key `store in` can be above.

8 Deprecated commands

The following commands displayed in red boxes on the left have been deprecated and the relevant replacement displayed in green boxes on the right should be used. To simplify the representation of the code and clarify how arguments are supposed to be used, numbers are used to identify the arguments.

```
\JSONParseSetValue
  {⟨1⟩}{⟨2⟩}{⟨3⟩}
```

```
\JSONParseValue[store in={⟨1⟩},
  rescan=false]{⟨2⟩}{⟨3⟩}
```

```
\JSONParseSetRescanValue
  {⟨1⟩}{⟨2⟩}{⟨3⟩}
```

```
\JSONParseValue[store in={⟨1⟩},
  rescan]{⟨2⟩}{⟨3⟩}
```

```
\JSONParseSetKeys
  {⟨1⟩}{⟨2⟩}{⟨3⟩}
```

```
\JSONParseKeys
  [store in={⟨1⟩}]{⟨2⟩}{⟨3⟩}
```

```
\JSONParseSetArrayCount
  {⟨1⟩}{⟨2⟩}{⟨3⟩}
```

```
\JSONParseArrayCount
  [store in={⟨1⟩}]{⟨2⟩}{⟨3⟩}
```

```
\JSONParseArrayValues
  [⟨1⟩]{⟨2⟩}{⟨3⟩}[⟨4⟩]{⟨5⟩}
```

```
\JSONParseArrayUse
  [⟨1⟩]{⟨2⟩}{⟨3⟩}[⟨4⟩]{⟨5⟩}
```

```
\JSONParseArrayValuesMap
  [⟨1⟩]{⟨2⟩}{⟨3⟩}[⟨4⟩]{⟨5⟩}
  [⟨6⟩][⟨7⟩]
```

```
\JSONParseArrayMapFunction
  [⟨1⟩, code before={⟨6⟩},
    code after={⟨7⟩}]
  {⟨2⟩}{⟨3⟩}[⟨4⟩]{\⟨5⟩}
```

The command `\JSONParseArrayMapFunction` takes as last argument a command denoting the relevant mapping function including the preceding backslash, while the deprecated command `\JSONParseArrayValuesMap` required the name of this function without preceding backslash.

To ensure backward compatibility, the deprecated commands are still supported, but their use is not recommended. The commands `\JSONParseSetRescanValue` and `\JSONParseSetKeys` will locally set the relevant token variable.

9 L3 commands

The following commands are provided for defining user functions by package authors. For the conditional functions described above, apart from the variant that provides a true and a false branch, the and variants that only provide an argument for the true or for the false branch respectively are defined as well which is indicated by the letters *TF* printed in italics.

```
\jsonparse_parse:n {‹JSON string›}
\jsonparse_parse:o {‹JSON string›}
\jsonparse_parse:e {‹JSON string›}
```

The command `\jsonparse_parse:n` takes as argument a JSON string and populates the token variable (property list) `\g_jsonparse_entries_prop` with key-value pairs representing all elements of the JSON data structure represented by this string. This command does not escape the input in any way.

```
\jsonparse_parse_to_prop:Nn ‹token variable› {‹JSON string›}
\jsonparse_parse_to_prop:No ‹token variable› {‹JSON string›}
\jsonparse_parse_to_prop:Ne ‹token variable› {‹JSON string›}
```

The command `\jsonparse_parse_to_prop:Nn` creates the token variable given as the first arguments as property list and, after having called `\jsonparse_parse:n` using the second argument, globally sets this newly created property list equal to `\g_jsonparse_entries_prop`. If escaping is activated, this command will pre-process the input according to the selected escaping mode before forwarding it to `\jsonparse_parse:n`. See more on escaping above in section 4.

```
\jsonparse_parse_to_prop_local:Nn ‹token variable› {‹JSON string›}
\jsonparse_parse_to_prop_local:No ‹token variable› {‹JSON string›}
\jsonparse_parse_to_prop_local:Ne ‹token variable› {‹JSON string›}
```

The command `\jsonparse_parse_to_prop_local:Nn` works in the very same way as the command `\jsonparse_parse_to_prop:Nn`, but the property list is set locally.

```
\jsonparse_parse_keys:NN ‹token variable› ‹string variable›
```

The command `\jsonparse_parse_keys:NN` processes the token variable given as the first arguments as property list and selects all top-level keys which are then stored in the string variable as JSON array. The pseudo key `.` (or the string defined using the key `child sep`) to select the complete JSON data is ignored. If the JSON data is an array, the indices (wrapped into the separators defined by `separator/array left` and `separator/array right`) of the items are used as keys.

```
\jsonparse_rescan:n {‹JSON value›}
\jsonparse_rescan:e {‹JSON value›}
```

The command `\jsonparse_rescan:n` rescans the JSON value given in the argument. Rescanning converts all tokens to their default category codes and TeX control sequences are expanded. Further, during the rescanning process, JSON escape sequences are replaced and characters that don't require escaping in JSON but in TeX are replaced by the relevant TeX escape sequences.

```
\jsonparse_set_rescan:Nn ‹token variable› {‹JSON value›}
\jsonparse_set_rescan:Ne ‹token variable› {‹JSON value›}
```

The command `\jsonparse_set_rescan:Nn` rescans the JSON value given in the second argument and stores the result in the token variable specified in the second argument.

```
\jsonparse_gset_rescan:Nn ‹token variable› {‹JSON value›}
\jsonparse_gset_rescan:Ne ‹token variable› {‹JSON value›}
```

The command `\jsonparse_set_rescan:Nn` rescans the JSON value given in the second argument and stores the result globally in the token variable specified in the second argument.

```
\jsonparse_put_right_rescan:Nn ‹token variable› {‹JSON value›}
\jsonparse_put_right_rescan:Ne ‹token variable› {‹JSON value›}
```

The command `\jsonparse_put_right_rescan:Nn` rescans the JSON value given in the second argument and adds the result to the end of the token variable specified in the second argument.

```
\jsonparse_gput_right_rescan:Nn ‹token variable› {‹JSON value›}
\jsonparse_gput_right_rescan:Ne ‹token variable› {‹JSON value›}
```

The command `\jsonparse_gput_right_rescan:Nn` rescans the JSON value given in the second argument and adds the result globally to the end of the token variable specified in the second argument.

```
\jsonparse_filter:Nn ‹token variable› {‹key›}
```

The command `\jsonparse_filter:Nn` processes the token variable given as the first arguments as property list and filters it according to the key given as second argument. Filtering means that for every entry in the property list, the key of this entry is compared against the key given to the command. If the key in the property list starts with the given key, the matching part is removed from the key in the property list. If the keys do not match, the entry is completely removed from the property list. If the second argument matches the pseudo key `.` (or the string defined using the key `child sep`) exactly, the complete property list except for this key is returned.

```
\jsonparse_array_count:NN ‹token variable› ‹integer variable›
```

The command `\jsonparse_array_count:NN` processes the token variable given as the first arguments as property list and, assuming that it is an array, counts its items and stores the result in the integer variable. If the token variable does not expand to a key that represents an array item, that is if the key does not start with the character defined by `separator/array left`, the command will return an error. The command `\JSONParseArrayCount` serves as a wrapper of this command.

```
\jsonparse_if_num:nTF {‹string›} {‹true code›} {‹false code›}                ★
\jsonparse_if_num:VTF {‹string›} {‹true code›} {‹false code›}
\jsonparse_if_num_p:n {‹string›}
\jsonparse_if_num_p:V {‹string›}
```

The expandable conditional function `\jsonparse_if_num:nTF` checks whether a string is a valid JSON number according the relevant specification. It executes the true code if the string is a valid JSON number and the false code if not. The variants that only provide an argument for the true or false case work accordingly. The command `\jsonparse_if_num_p:n` returns a boolean true or false (i. e. `\c_true_bool` or `\c_false_bool`).

```
\jsonparse_unicode_if_high_surrogate:nTF {‹codepoint›}                       ★
  {‹true code›} {‹false code›}
\jsonparse_unicode_if_high_surrogate:eTF {‹codepoint›}
  {‹true code›} {‹false code›}
\jsonparse_unicode_if_high_surrogate_p:n {‹codepoint›}
\jsonparse_unicode_if_high_surrogate_p:e {‹codepoint›}
```

The expandable conditional function `\jsonparse_unicode_if_high_surrogate:nTF` checks whether the codepoint entered as argument (an integer that can be hexadecimal if preceded by `"`) is in the range of `"D800` and `"DBFF` which means that it is the first part of a surrogate pair (a high surrogate). The conditional function executes the true or false code depending on the evaluation. The variants that only provide an argument for the true or false case work accordingly. The command `\jsonparse_unicode_if_high_surrogate_p:n` returns a boolean true or false (i. e. `\c_true_bool` or `\c_false_bool`).

\jsonparse_unicode_if_low_surrogate:n*TF* {‹codepoint›} ★
  {‹true code›} {‹false code›}
\jsonparse_unicode_if_low_surrogate:e*TF* {‹codepoint›}
  {‹true code›} {‹false code›}
\jsonparse_unicode_if_low_surrogate_p:n {‹codepoint›}
\jsonparse_unicode_if_low_surrogate_p:e {‹codepoint›}

The expandable conditional function `\jsonparse_unicode_if_low_surrogate:nTF` checks whether the codepoint entered as argument (an integer that can be hexadecimal if preceded by `"`) is in the range of `"DC00` and `"DFFF` which means that it is the last part of a surrogate pair (a low surrogate). The conditional function executes the true or false code depending on the evaluation. The variants that only provide an argument for the true or false case work accordingly. The command `\jsonparse_unicode_if_low_surrogate_p:n` returns a boolean true or false (i. e. `\c_true_bool` or `\c_false_bool`).

\jsonparse_unicode_convert_surrogate_pair:nn {‹codepoint›} {‹codepoint›} ★
\jsonparse_unicode_convert_surrogate_pair:ee {‹codepoint›} {‹codepoint›}

The expandable command `\jsonparse_unicode_convert_surrogate_pair:nn` converts a surrogate pair to the relevant Unicode codepoint. The returned value is an integer. It takes as first argument the codepoint of the low surrogate and as second argument the codepoint of the high surrogate. It does not check whether the codepoints actually belong to the relevant ranges of codepoints for high and low surrogates.

10 Changes

v0.5.0 (2024/04/09) Changed from string token variables to token lists to support Unicode.

v0.5.6 (2024/04/11) Bug fixes, escaping of special chars added.

v0.5.7 (2024/04/14) Bug fixes, key-value option setting added.

v0.7.0 (2024/04/18) Renaming and rearranging of keys, escaping of special JSON escape sequences added.

v0.7.1 (2024/04/20) Access to top-level keys of object added.

v0.8.0 (2024/04/24) Internal rewrite, escaping procedures changed.

v0.8.2 (2024/04/26) Bug fixes, externalizing parsed data.

v0.8.3 (2024/04/28) Escaping of characters with special meaning in TeX.

v0.9.0 (2024/08/27) Adaption to updated verbatim tokenization.

v0.9.1 (2024/09/21) Added functions to test for valid JSON numbers.

v0.9.3 (2024/10/24) Fixed a bug that prevented tabs in source from being parsed properly.

v0.9.6 (2024/10/31) Allowing for multiple return values when mapping over arrays.

v0.9.8 (2024/11/19) Bug fixes; adding possibility to store value in token list.

v0.9.12 (2025/01/17) Bug fixes; adding commands to access items in arrays.

v1.0.1 (2025/01/21) Fixes in documentation. Added user command for filtering.

v1.0.2 (2025/01/23) Support for Unicode surrogate pairs.

v1.1.0 (2025/01/30) Unified names of user functions; renaming key for keywords.

v1.1.1 (2025/02/03) Added option to store result of mapped inline function.

v1.1.2 (2025/02/08) Added option to store result of array function.

v1.2.1 (2025/02/24) Unified functions, added option to store result globally.

v1.2.3 (2025/03/23) Enable nesting of mapped inline function.