The `citeproc-js` Citation Processor

Integrator's Manual

version 1.00$_{a113}$

25 August 2011

Author of this manual

- Frank G. Bennett, Jr.

With important feedback from

- Bruce D'Arcus
- Lennard Fuller
- Fergus Gallagher
- Simon Kornblith
- Carles Pina
- Dan Stillman
- Rintze Zelle

Table of Contents

- Introduction
- Setup and System Requirements
  - Getting the `citeproc-js` sources
  - Obtaining the Standard Test Fixtures
  - Obtaining the Locale Files
  - JavaScript interpreters
  - Loading runtime code
- Running the processor
  - Instantiation: `CSL.Engine()`
    - Locally defined system functions
      - `retrieveLocale()`
      - `retrieveItem()`
      - `getAbbreviations()`
    - Configuration commands
      - `setOutputFormat()`
      - `setAbbreviations()`
      - `restoreProcessorState()`
    - Readable Flags
      - `opt.sort_citations`

            - opt.citation_number_sort
    - Loading items to the processor
        - updateItems()
        - updateUncitedItems()
    - Generating bibliographies
        - makeBibliography()
            - Return value
            - Selective output
    - Output of citations
        - processCitationCluster()
        - appendCitationCluster()
        - previewCitationCluster()
    - Handling items with no rendered form
        - No bibliography node in style
        - No item output for bibliography entry
        - No output for citation
- Data Input
    - Item fields
        - Text and numeric variables
        - Names
            - Names with particles
            - Names with an articular
            - "non-Byzantine" names
        - Dates
    - Citation data object
        - Runtime state: Internal objects and arrays
- Dirty Tricks
    - Supplementary fields
    - Input data rescue
        - Names
            - `non-dropping-particle`
            - `dropping-particle`
            - suffix (ordinary)
            - suffix (forced comma)
            - Particles as part of the last name
        - Dates
    - Processor control
        - `author-only`
        - `suppress-author`
        - Automating text insertions
    - Multi-lingual content
        - Selecting multi-lingual variants
        - Data format
            - Title
            - Names
- Date parser
    - Instantiation: CSL.DateParser
    - Parser methods
- Test Suite
    - Running the test suite
    - Fixture layout
- Live Demo

Introduction

This is the site administrator's manual for `citeproc-js`, a JavaScript implementation of the **Citation Style Language (CSL)** used by Zotero, Mendeley and other popular reference tools to format citations in any of the hundreds of styles supplied by the CSL style repository. [1] The processor complies with version 1.0 of the CSL specification, has been written and tested as an independent module, and can be run by any ECMAscript-compliant interpreter. With an appropriate supporting environment, [2] it can be deployed in a browser plugin, as part of a desktop application, or as a formatting backend for a website or web service.

This manual covers the basic operation of the processor, including the command set, the local system code that must be supplied by the integrator, and the expected format of input data. In addition, notes are provided on the test suite, on the infrastructure requirements for running the processor in particular environments, and on extended functionality that is available to address certain special requirements.

Comments and complaints relating to this document and to the processor itself will be gladly received and eventually despatched with. The best channel for providing feedback and getting help is the **project mailing list**.

[1] The repository is currently housed at **zotero.org**. Note that styles in the Zotero styles repository are currently at CSL version 0.8.1. Use the **tools provided by the CSL project** to convert CSL 0.8.1 styles to the version 1.0 syntax supported by this processor.

[2] For further details on required infrastructure, see the sections **Locally defined system functions** and **Data Input** below.

Setup and System Requirements

The processor is written in JavaScript, one of the interesting features of which is the lack of a standard method of I/O. As a result, the processor must be wrapped in other code to get data in and out of it, and every installation is going to be a little different. This manual does not cover the nitty-gritty of setting up the environment for running the processor in a particular environment, but the basic system requirements are described below. If you get stuck and want advice, or if you find something in this manual that is out of date or just wrong, please feel free to drop a line to the **project list**.

Getting the `citeproc-js` sources

The `citeproc-js` sources are hosted on **BitBucket**. To obtain the sources, install the **Mercurial version control system** on a computer within your control (if you're on a Linux distro or a Mac, just do a package install), and run the following command:

```
hg clone http://bitbucket.org/fbennett/citeproc-js/
```

This should get you a copy of the sources, and you should be able to exercise the test framework using the `./test.py` script.

Obtaining the Standard Test Fixtures

To run the test suite, the standard test fixtures must be added to the processor source bundle. To do so, enter the directory `./tests/fixtures`, and issue the following command:

```
hg clone http://bitbucket.org/bdarcus/citeproc-test std
```

Note the explicit target directory "std" following the repository address.

Obtaining the Locale Files

The processor requires a set of standard CSL 1.0 locale files in order to run. These may be installed using the following command (under Linux command line). From the root of the `citeproc-js` source directory:

```
git clone git://github.com/citation-style-language/locales.git locale
```

JavaScript interpreters

An ECMAscript (JavaScript) interpreter is required to run the processor. The processor code itself is written in such a way that it should run on a wide variety of platforms, including Rhino, Spidermonkey and Tracemonkey on the server side, and browsers such as Internet Explorer (version 6 and higher), Firefox, Mozilla, Safari, Google Chrome, and Opera.

To parse the XML files used to define locales and styles, the processor relies on a supplementary module, which must be loaded into the same JavaScript context as the processor itself. The `xmle4x.js` and `xmldom.js` files shipped with the processor source should serve this purpose. The `xmle4x.js` module supports Gecko-based browsers, and other platforms that embed the Rhino, Spidermonkey or Tracemonkey JavaScript interpreters. The `xmldom.js` module supports all other browsers as well.

For an example of working code, the source behind the **processor demo page** may be useful as a reference.

Instructions on running the processor test suite can be found in the section **Running the test suite** at the end of this manual.

Loading runtime code

The primary source code of the processor is located under `./src`, for ease of maintenance. The files necessary for use in a runtime environment are catenated, in the appropriate sequence, in the `citeproc.js` file, located in the root of the source archive. This file and the test fixtures can be refreshed using the `./test.py -r` command.

To build the processor, the `citeproc.js` source code should be loaded into the JavaScript interpreter context, together with a `sys` object provided by the integrator (see below), and the desired CSL style (as a string).

Running the processor

Instances of the processor are produced using `CSL.Engine()` function. Note that, as detailed below under **Locally defined system functions**, certain local data access functions must be defined separately on an object supplied to the processor as its first argument.

Once instantiated, a processor instance can be configured via a small set of runtime setter methods. Instance methods are also used to load item data into the processor, and to produce output objects suitable for consumption by a word processor plugin, or for use in constructing bibliographies. Details of these and other methods available on processor instances are given below.

Instantiation: `CSL.Engine()`

The `CSL.Engine()` command is invoked as shown in the code illustration below. This command takes up to four arguments, two of them required, and two of them optional:

```
var citeproc = new CSL.Engine(sys,
                              style,
                              lang)
```

Important

See the section **Locally defined system functions** below for guidance on the definition of the functions contained in the `sys` object.

sys
: A JavaScript object containing the functions `retrieveLocale()` and `retrieveItem()`.

style
: The CSL code for a style, as XML in serialized (string) form (not a filename or style name, but the code itself).

lang **(optional)**
: A language tag compliant with RFC 4646. Defaults to `en`. Styles that contain a `default-locale` attribute value on the `style` node will ignore this option unless the `forceLang` argument is set to a non-nil value.

forceLang **(optional)**
: When set to a non-nil value, force the use of the locale set in the `lang` argument, overriding any language set in the `default-locale` attribute on the `style` node.

The version of the processor itself can be obtained from the attribute `processor_version`. The supported CSL version can be obtained from `csl_version`.

Locally defined system functions

While `citeproc-js` does a great deal of the heavy lifting needed for correct formatting of citations and bibliographies, a certain amount of programming is required to prepare the environment for its correct operation.

Two functions must be defined separately and supplied to the processor upon instantiation. These functions are used by the processor to obtain locale and item data from the surrounding environment. The exact definition of each may vary from one system to another; those given below assume the existence of a global `DATA` object in the context of the processor instance, and are provided only for the purpose of illustration.

`retrieveLocale()`

The `retrieveLocale()` function is used internally by the processor to retrieve the serialized XML of a given locale. It takes a single RFC 4646 compliant language tag as argument, composed of a single language tag (`en`) or of a language tag and region subtag (`en-US`). The name of the XML document in the CSL distribution that contains the relevant locale data may be obtained from the `CSL.localeRegistry` array. The sample function below is provided for reference only.

```
sys.retrieveLocale = function(lang){
        var ret = DATA._locales[ CSL.localeRegistry[lang] ];
        return ret;
};
```

`retrieveItem()`

The `retrieveItem()` function is used by the processor to fetch individual items from storage.

```
sys.retrieveItem = function(id){
        return DATA._items[id];
};
```

`getAbbreviations()`

The `getAbbreviations()` command is invoked by the processor at startup, and when the `setAbbreviations()` command is invoked on the instantiated processor. The abbreviation list retrieved by the processor should have the following structure:

```
var ABBREVS = {
   "default": {
      "container-title":{
         "Journal of Irreproducible Results":"J. Irrep. Res."
      },
      "collection-title":{
         "International Rescue Wildlife Series":"I.R. Wildlife Series"
      },
      "authority":{
         "United States Patent and Trademark Office": "USPTO"
               },
      "institution":{
         "Bureau of Gaseous Unformed Stuff":"BoGUS"
      },

        "title": {},
        "publisher": {},
        "publisher-place": {},
        "hereinafter": {}
    };
};
```

If the object above provides the abbreviation store for the system, an appropriate `sys.getAbbreviations()` function might look like this:

```
sys.getAbbreviations = function(name){
   return ABBREVS[name];
};
```

Configuration commands

`setOutputFormat()`

The default output format of the processor is HTML. Output formats for RTF and plain text are defined in the distribution source file `./src/formats.js`. Additional formats can be added if desired. See **the file itself** for details; it's pretty straightforward.

The output format of the processor can be changed to any of the defined formats after instantiation, using the `setOutputFormat()` command:

```
citeproc.setOutputFormat("rtf");
```

This command is specific to the `citeproc-js` processor

`setAbbreviations()`

The processor recognizes abbreviation lists for journal titles, series titles, authorities (such as the Supreme Court of New York), and institution names (such as International Business Machines). A list can be set in the processor using the `setAbbreviations()` command, with the name of the list as sole argument. The named list is fetched and installed by the `sys.getAbbreviations()` command, documented below under **Locally defined system functions**.

```
citeproc.setAbbreviations("default");
```

At runtime, whenever an abbreviation is requested but unavailable, an empty abbreviation entry is opened in the processor `.transform` object. Entries are keyed on the abbreviation category and the long form of the field value. Abbreviation catetories are as follows: `container-title`, `collection-title`, `authority`, `institution`, `title`, `publisher`, `publisher-place`, `hereinafter`.

After any run of the `makeBibliography()` or citation rendering commands, the full set of registered abbreviations (including the empty entries identified at runtime) can be read from the processor. For example, if the processor instance is named `citeproc`, a structure as shown in **Locally defined system functions** → **getAbbreviations()** can be obtained as follows:

```
var ABBREVS = citeproc.transform;
```

The structure thus obtained can then be edited, via the user interface of the calling application, to alter the abbreviations applied at the next run of the processor.

[3] For illustrations of the input syntax for the `makeBibliography()` command, see any test in the **test suite** that uses the CITATION-ITEMS environment described in the **CSL Test Suite manual**.

`restoreProcessorState()`

The `restoreProcessorState()` command can be used to restore the processor state in a single operation, where citation objects, complete with position variables and `sortkeys`, are available. The command takes a single argument, which is an array of such citation objects:

```
citeproc.restoreProcessorState(citations);
```

Uncited items must be restored separately using the `updateUncitedItems()` command.

Readable Flags

The instantiated processor has several readable flags that can be used by the calling application to shape the user interface to the processor. These include the following: [3]

`opt.sort_citations`

True if the style is one that sorts citations in any way.

`opt.citation_number_sort`

True if citations are sorted by citation

[4] Note that these are information variables intended for reading only; changing their value directly will have no effect on the actual behavior of the processor.

Loading items to the processor

`updateItems()`

Before citations or a bibliography can be generated, an ordered list of reference items must ordinarily be loaded into the processor using the `updateItems()` command, as shown below. This command takes a list of item IDs as its sole argument, and will reconcile the internal state of the processor to the provided list of items, making any necessary insertions and deletions, and making any necessary adjustments to internal registers related to disambiguation and so forth.

```
var my_ids = [
    "ID-1",
    "ID-53",
    "ID-27"
]

citeproc.updateItems( my_ids );
```

Hint

The sequence in which items are listed in the argument to `updateItems()` will ordinarily be reflected in the ordering of bibliographies only if the style installed in the processor does not impose its own sort order.

To suppress sorting, give a second argument to the command with a value of `true`.

```
citeproc.updateItems(my_ids, true);
```

Note that only IDs may be used to identify items. The ID is an arbitrary, system-dependent identifier, used by the locally customized `retrieveItem()` method to retrieve actual item data.

`updateUncitedItems()`

The `updateUncitedItems()` command has the same interface as `updateItems()` (including the option to suppress sorting by the style), but the reference items it adds are not subject to deletion when no longer referenced by a cite anywhere in the document.

Generating bibliographies

`makeBibliography()`

The `makeBibliography()` command does what its name implies. If invoked without an argument, it dumps a formatted bibliography containing all items currently registered in the processor:

```
var mybib = citeproc.makeBibliography();
```

Return value

The value returned by this command is a two-element list, composed of a JavaScript array containing certain formatting parameters, and a list of strings representing bibliography entries. It is the responsibility of the calling application to compose the list into a finish string for insertion into the document. The first element —- the array of formatting parameters —- contains the key/value pairs shown below (the values shown are the processor defaults in the HTML output mode):

Important

Matches against the content of name and date variables are not possible, but empty fields can be matched for all variable types. See the `quash` example below for details.

```
[
    {
        maxoffset: 0,
        entryspacing: 0,
        linespacing: 0,
        hangingindent: 0,
        second-field-align: false,
        bibstart: "<div class=\"csl-bib-body\">\n",
        bibend: "</div>",
        bibliography_errors: []
    },
    [
        "<div class=\"csl-entry\">Book A</div>",
        "<div class=\"csl-entry\">Book C</div>"
    ]
]
```

maxoffset
Some citation styles apply a label (either a number or an alphanumeric code) to each bibliography entry, and use this label to cite bibliography items in the main text. In the bibliography, the labels may either be hung in the margin, or they may be set flush to the margin, with the citations indented by a uniform amount to the right. In the latter case, the amount of indentation needed depends on the maximum width of any label. The `maxoffset` value gives the maximum number of characters that appear in any label used

in the bibliography. The client that controls the final rendering of the bibliography string should use this value to calculate and apply a suitable indentation length.

entryspacing
An integer representing the spacing between entries in the bibliography.

linespacing
An integer representing the spacing between the lines within each bibliography entry.

hangingindent
The number of em-spaces to apply in hanging indents within the bibliography.

second-field-align
When the `second-field-align` CSL option is set, this returns either "flush" or "margin". The calling application should align text in bibliography output as described in the CSL specification. Where `second-field-align` is not set, this return value is set to `false`.

bibstart
A string to be appended to the front of the finished bibliography string.

bibend
A string to be appended to the end of the finished bibliography string.

Selective output

The `makeBibliography()` command accepts one optional argument, which is a nested JavaScript object that may contain *one of* the objects `select`, `include` or `exclude`, and optionally an additional `quash` object. Each of these four objects is an array containing one or more objects with `field` and `value` attributes, each with a simple string value (see the examples below). The matching behavior for each of the four object types, with accompanying input examples, is as follows:

`select`
For each item in the bibliography, try every match object in the array against the item, and include the item if, and only if, *all* of the objects match.

```
var myarg = {
   "select" : [
      {
         "field" : "type",
         "value" : "book"
      },
      {  "field" : "categories",
          "value" : "1990s"
      }
   ]
}

var mybib = cp.makeBibliography(myarg);
```

Hint

The target field in the data items registered in the processor may either be a string or an array. In the latter case, an array containing a value identical to the relevant value is treated as a match.

`include`
Try every match object in the array against the item, and include the item if *any* of the objects match.

```
var myarg = {
   "include" : [
      {
         "field" : "type",
            "value" : "book"
        }
    ]
}

var mybib = cp.makeBibliography(myarg);
```

`exclude`
Include the item if *none* of the objects match.

```
var myarg = {
    "exclude" : [
        {
            "field" : "type",
            "value" : "legal_case"
        },
        {
            "field" : "type",
            "value" : "legislation"
        }
    ]
}

var mybib = cp.makeBibliography(myarg);
```

`quash`
Regardless of the result from `select`, `include` or `exclude`, skip the item if *all* of the objects match.

Hint

An empty string given as the field value will match items for which that field is missing or has a nil value.

```
var myarg = {
    "include" : [
        {
            "field" : "categories",
            "value" : "classical"
        }
    ],
    "quash" : [
        {
            "field" : "type",
            "value" : "manuscript"
        },
        {
            "field" : "issued",
            "value" : ""
        }
    ]
}

var mybib = cp.makeBibliography(myarg);
```

Output of citations

The available citation commands are:

- appendCitationCluster()
- processCitationCluster()
- previewCitationCluster()

Citation commands generate strings for insertion into the text of a target document. Citations can be added to a document in one of two ways: as a batch process (BibTeX, for example, works in this way) or interactively (Endnote, Mendeley and Zotero work in this way, through a connection to the user's word processing software). These two modes of operation are supported in `citeproc-js` by two separate commands, respectively `appendCitationCluster()`, and `processCitationCluster()`. A third, simpler command (`makeCitationCluster()`), is not covered by this manual. It is primarily useful as a tool for testing the processor, as it lacks any facility for position evaluation, which is needed in production environments. [4]

The `appendCitationCluster()` and `processCitationCluster()` commands use a similar input format for citation data, which is described below in the **Data Input** → **Citation data object** section below.

`processCitationCluster()`

The `processCitationCluster()` command is used to generate and maintain citations dynamically in the text of a document. It takes three arguments: a citation object, a list of citation ID/note index pairs representing existing citations that precede the target citation, and a similar list of pairs for citations coming after the target. Like the `appendCitationCluster()` command run without a flag, its return value is an array of two elements: a data object, and an array of one or more index/string pairs, one for each citation affected by the citation edit or insertion operation. As shown below, the data object currently has a single boolean value, `bibchange`, which indicates whether the document bibliography is in need of refreshing as a result of the `processCitationCluster()` operation.

```
var citationsPre = [ ["citation-abc",1], ["citation-def",2] ];

var citationsPost = [ ["citation-ghi",4] ];

citeproc.processCitationCluster(citation,citationsPre,citationsPost);

...

[
    {
      "bibchange": true
    },
    [
        [ 1,"(Ronald Snoakes 1950)" ],
        [ 3,"(Richard Snoakes 1950)" ]
    ]
]
```

A worked example showing the result of multiple transactions can be found in the processor test suite.

`appendCitationCluster()`

The `appendCitationCluster()` command takes a single citation object as argument, and an optional flag to indicate whether a full list of bibliography items has already been registered in the processor with the `updateItems()` command. If the flag is true, the command should return an array containing exactly one two-element array, consisting of the current index position as the first element, and a string for insertion into the document as the second. To wit:

```
citeproc.appendCitationCluster(mycitation,true);

[
   [ 5, "(J. Doe 2000)" ]
]
```

If the flag is false, invocations of the command may return multiple elements in the list, when the processor sense that the additional bibliography items added by the citation require changes to other citations to achieve disambiguation. In this case, a typical return value might look like this:

```
citeproc.appendCitationCluster(mycitation);

[
   [ 2, "(Jake Doe 2000)" ],
   [ 5, "(John Doe 2000)" ]
]
```

`previewCitationCluster()`

The `previewCitationCluster()` command takes the same arguments as `processCitationCluster()`, plus a flag to indicate the output mode.

The return value is a string representing the citation as it would be rendered in the specified context. This command will preview citations exactly as they will appear in the document, and will have no effect on processor state: the next edit will return updates as if the preview command had not been run.

```
var citationsPre = [ ["citation-abc",1], ["citation-def",2] ];
var citationsPost = [ ["citation-ghi",4] ];

citeproc.previewCitationCluster(citation,citationsPre,citationsPost,"html");

...

"(Richard Snoakes 1950)"
```

Handling items with no rendered form

The processor might fail to produce meaningful rendered output in three situations:

1. When **makeBibliography()** is run, and the configured style contains no `bibliography` node;
2. When **makeBibliography()** is run, and no variable other than `citation-number` produces output for an individual entry; or
3. When a **citation command** is used, but no element rendered for a particular cite produces any output.

The processor handles these three cases as described below.

No `bibliography` node in style

When the **makeBibliography()** command is run on a style that has no `bibliography` node, the command returns a value of `false`.

No item output for bibliography entry

When the return value of the **makeBibliography()** command contains entries that produce no output other than for the (automatically generated) `citation-number` variable, an error object with ID and position information on the offending entry, and a bitwise error code (always CSL.ERROR_NO_RENDERED_FORM, currently) is pushed to the `bibliography_errors` array in the data segment of the return object:

```
[
   {
      maxoffset: 0,
      entryspacing: 0,
      linespacing: 0,
      hangingindent: 0,
      second-field-align: false,
      bibstart: "<div class=\"csl-bib-body\">\n",
      bibend: "</div>",
      bibliography_errors: [
         {
            index: 2,
            itemID: "ITEM-2",
            error_code: CSL.ERROR_NO_RENDERED_FORM
         }
      ]
   },
   [
      "[1] Snoakes, Big Book (2000)",
      "[2] Doe, Bigger Book (2001)",
      "[3] ",
      "[4] Roe, Her Book (2002)"
   ]
]
```

The calling application may use the information in `bibliography_errors` to prompt the user concerning possible corrective action.

No output for citation

When a citation processing command produces no output for a citation, an error object with ID and position information on the offending cite, and a bitwise error code (always `CSL.ERROR_NO_RENDERED_FORM`, currently) is pushed to the `citation_errors` array in the data segment of the return object.

Note that `previewCitationCluster()` returns only a string value, with no data segment; citation errors are not available with this command.

```
[
   {
     bibchange: true,
     citation_errors: [
        {
           citationID: "citationID_12345",
           index: 4,
           noteIndex: 3,            // for example
           itemID: "itemID_67890",
           citationItem_pos: 0,
           error_code: CSL.ERROR_NO_RENDERED_FORM
        }
     ]
   },
   [
      [ 1,"(Ronald Snoakes 1950)" ],
      [ 4,"[CSL STYLE ERROR: reference with no printed form.]" ],
      [ 5,"(Richard Snoakes 1950)" ]
   ]
]
```

Data Input

Item fields

The locally defined `retrieveItem()` function must return data for the target item as a simple JavaScript array containing recognized CSL fields. [5] The layout of the three field types is described below.

Text and numeric variables

Text and numeric variables are not distinguished in the data layer; both should be presented as simple strings.

```
{  "title" : "My Anonymous Life",
   "volume" : "10"
}
```

Names

When present in the item data, CSL name variables must be delivered as a list of JavaScript arrays, with one array for each name represented by the variable. Simple personal names are composed of `family` and `given` elements, containing respectively the family and given name of the individual.

```
{ "author" : [
    { "family" : "Doe", "given" : "Jonathan" },
    { "family" : "Roe", "given" : "Jane" }
  ],
  "editor" : [
    { "family" : "Saunders",
      "given" : "John Bertrand de Cusance Morant" }
  ]
}
```

Institutional and other names that should always be presented literally (such as "The Artist Formerly Known as Prince", "Banksy", or "Ramses IV") should be delivered as a single `literal` element in the name array:

```
{ "author" : [
    { "literal" : "Society for Putting Things on Top of Other Things" }
  ]
}
```

Names with particles

Name particles, such as the "von" in "Werner von Braun", can be delivered separately from the family and given name, as `dropping-particle` and `non-dropping-particle` elements.

```
{ "author" : [
    { "family" : "Humboldt",
      "given" : "Alexander",
      "dropping-particle" : "von"
    },
    { "family" : "Gogh",
      "given" : "Vincent",
      "non-dropping-particle" : "van"
    },
    { "family" : "Stephens",
      "given" : "James",
      "suffix" : "Jr."
    },
    { "family" : "van der Vlist",
      "given" : "Eric"
    }
  ]
}
```

Names with an articular

Name suffixes such as the "Jr." in "Frank Bennett, Jr." and the "III" in "Horatio Ramses III" can be delivered as a `suffix` element.

```
{ "author" : [
    { "family" : "Bennett",
      "given" : "Frank G.",
      "suffix" : "Jr.",
      "comma-suffix": "true"
    },
    { "family" : "Ramses",
      "given" : "Horatio",
      "suffix" : "III"
    }
  ]
}
```

Hint

A simplified format for delivering particles and name suffixes to the processor is described below in the section **Dirty Tricks** → **Input data rescue** → **Names**.

Note the use of the `comma-suffix` field in the example above. This hint must be included for suffixes that are preceded by a comma, which render differently from "ordinary" suffixes in the ordinary long form.

"non-Byzantine" names

Names not written in the Latin or Cyrillic scripts [6] are always displayed with the family name first. No special hint is needed in the input data; the processor is sensitive to the character set used in the name elements, and will handle such names appropriately.

```
{ "author" : [
    { "family" : "村上",
      "given" : "春樹"
    }
  ]
}
```

Sometimes it might be desired to handle a Latin or Cyrillic transliteration as if it were a fixed (non-Byzantine) name. This behavior can be prompted by including a `static-ordering` element in the name array. The actual value of the element is irrelevant, so long as it returns true when tested by the JavaScript interpreter.

Hint

When the romanized transliteration is selected from a multi-lingual name field, the `static-ordering` flag is not required. See the section **Dirty Tricks** → **Multi-lingual content** below for further details.

```
{ "author" : [
    { "family" : "Murakami",
      "given" : "Haruki",
      "static-ordering" : 1
    }
  ]
}
```

Dates

Date fields are JavaScript objects, within which the "date-parts" element is a nested JavaScript array containing a start date and optional end date, each of which consists of a year, an optional month and an optional day, in that order if present.

```
{  "issued" : {
      "date-parts" : [
         [ "2000", "1", "15" ]
      ]
   }
}
```

Hint

A simplified format for providing date input is described below in the section **Dirty Tricks** → **Input data rescue** → **Dates**.

Date elements may be expressed either as numeric strings or as numbers.

```
{  "issued" : {
      "date-parts" : [
         [ 1895, 11 ]
      ]
   }
}
```

The `year` element may be negative, but never zero.

```
{  "issued" : {
      "date-parts" : [
         [ -200 ]
      ]
   }
}
```

A `season` element may also be included. If present, string or number values between `1` and `4` will be interpreted to correspond to Spring, Summer, Fall, and Winter, respectively.

```
{   "issued" : {
        "date-parts" : [
            [ 1950 ]
        ],
        "season" : "1"
    }
}
```

Other string values are permitted in the `season` element, but note that these will appear in the output as literal strings, without localization:

```
{   "issued" : {
        "date-parts" : [
            [ 1975 ]
        ],
        "season" : "Trinity"
    }
}
```

For approximate dates, a `circa` element should be included, with a non-nil value:

```
{   "issued" : {
        "date-parts" : [
            [ -225 ]
        ],
        "circa" : 1
    }
}
```

To input a date range, add an array representing the end date, with corresponding elements:

```
{   "issued" : {
        "date-parts" : [
            [ 2000, 11 ],
            [ 2000, 12 ]
        ]
    }
}
```

To specify an open-ended range, pass nil values for the end elements:

```
{   "issued" : {
        "date-parts" : [
            [ 2008, 11 ],
            [ 0, 0 ]
        ]
    }
}
```

A literal string may be passed through as a `literal` element:

```
{   "issued" : {
       "literal" : "13th century"
    }
}
```

Citation data object

A minimal citation data object, used as input by both the `processCitationCluster()` and `appendCitationCluster()` command, has the following form:

```
{
   "citationItems": [
      {
         "id": "ITEM-1"
      }
   ],
   "properties": {
      "noteIndex": 1
   }
}
```

The `citationItems` array is a list of one or more citation item objects, each containing an `id` used to retrieve the bibliographic details of the target resource. A citation item object may contain one or more additional optional values:

- `locator`: a string identifying a page number or other pinpoint location or range within the resource;
- `label`: a label type, indicating whether the locator is to a page, a chapter, or other subdivision of the target resource. Valid labels are defined in the **CSL specification**.
- `suppress-author`: if true, author names will not be included in the citation output for this cite;
- `author-only`: if true, only the author name will be included in the citation output for this cite -- this optional parameter provides a means for certain demanding styles that require the processor output to be divided between the main text and a footnote. (See the section **Processor control**, in the **Dirty Tricks** section below for more details.)
- `prefix`: a string to print before this cite item;
- `suffix`: a string to print after this cite item.

In the `properties` portion of a citation, the `noteIndex` value indicates the footnote number in which the citation is located within the document. Citations within the main text of the document have a `noteIndex` of zero.

The processor will add a number of data items to a citation during processing. Values added at the top level of the citation structure include:

- `citationID`: A unique ID assigned to the citation, for internal use by the processor. This ID may be assigned by the calling application, but it must uniquely identify the citation, and it must not be changed during processing or during an editing session.
- `sortedItems`: This is an array of citation objects and accompanying bibliographic data objects, sorted as required by the configured style. Calling applications should not need to access the data in this array directly.

Values added to individual citation item objects may include:

- `sortkeys`: an array of sort keys used by the processor to produce the sorted list in `sortedItems`. Calling applications should not need to touch this array directly.
- `position`: an integer flag that indicates whether the cite item should be rendered as a first reference, an

immediately-following reference (i.e. *ibid*), an immediately-following reference with locator information, or a subsequent reference.

- `first-reference-note-number`: the number of the `noteIndex` of the first reference to this resource in the document.
- `near-note`: a boolean flag indicating whether another reference to this resource can be found within a specific number of notes, counting back from the current position. What is "near" in this sense is style-dependent.
- `unsorted`: a boolean flag indicating whether sorting imposed by the style should be suspended for this citation. When true, cites are rendered in the order in which they are presented in `citationItems`.

Runtime state: Internal objects and arrays

Citations are registered and accessed by the processor internally in arrays and JavaScript objects. Data that may be of use to calling applications can be accessed at the following locations (note that the content of these variables should not be modified directly; the processor will update them automatically when citation data is processed or updated via the API):

```
citeproc.registry.citationreg.citationById      // (object, returns object)

citeproc.registry.citationreg.citationsByItemId // (object, returns array)

citeproc.registry.citationreg.citationByIndex   // (array of objects)

citeproc.registry.getSortedRegistryItems()      // (returns an array of
                                                // registry objects, with
                                                // ``uncited`` set ``true``
                                                // as appropriate, and
                                                // item data on key ``ref``)
```

[5] For information on valid CSL variable names, please refer to the CSL specification, available via http://citationstyles.org/.

[6] The Latin and Cyrillic scripts are referred to here collectively as "Byzantine scripts", after the confluence of cultures in the first millenium that spanned both.

Dirty Tricks

This section presents features of the `citeproc-js` processor that are not properly speaking a part of the CSL specification. The functionality described here may or may not be found in other CSL 1.0 compliant processors, when they arrive on the scene.

Supplementary fields

Where the calling application provides a user interface for adding and editing bibliographic items, a limited set of fields is typically provided for each if the item types recognized by the application. Fields that map to valid CSL variables needed for a particular type of reference may not be available.

If the calling application provides a mapping of the `note` variable to all types, the processor can parse missing fields out of this variable, for use in rendering citations. This facility is intended only for testing purposes. It provides a means of illustrating citation use cases, with a view to requesting an adjustment to the field lists or the user interface of the calling application. It should not be relied upon as a permanent workaround in production data; and it should *never* be used to add variables that are not in the CSL specification.

The syntax for adding supplementary fields via the `note` variable is as follows:

```
{:authority:Superior Court of California}{:section:A}
```

Supplementary variables are read by the processor as flat strings, so names and date parsing will not work with them.

Input data rescue

Names

Systems that use a simple two-field entry format can encode `non-dropping-particle`, `dropping-particle` and `suffix` name sub-elements by writing them appropriately in the `family` or `given` name fields and setting a `parse-names` flag on the name object. The processor will then attempt to parse out the elements and convert them to the explicit form (as documented under **Data input** → **Names** above) before rendering. With the `parse-names` flag, sub-elements are recognized as follows.

`non-dropping-particle`

A string at the beginning of the `family` field consisting of spaces and lowercase roman or Cyrillic characters will be treated as a `non-dropping-particle`.

```
{ "author" : [
    { "family" : "van Gogh",
      "given" : "Vincent",
      "parse-names" : "true"
    }
  ]
}
```

`dropping-particle`

A string at the end of the `given` name field consisting of spaces and lowercase roman or Cyrillic characters will be treated as a `dropping-particle`.

```
{ "author" : [
    { "family" : "Humboldt",
      "given" : "Alexander von",
      "parse-names" : "true"
    }
  ]
}
```

`suffix` (ordinary)

Content following a comma in the `given` name field will be parse out as a name `suffix`.

```
{ "author" : [
    { "family" : "King",
      "given" : "Martin Luther, Jr.",
      "parse-names" : "true"
    },
    { "family" : "Gates",
      "given" : "William Henry, III",
      "parse-names" : "true"
    }
  ]
}
```

`suffix` (forced comma)

Modern typographical convention does not place a comma between suffixes such as "Jr." and the last name, when rendering the name in normal order: "John Doe Jr." If an individual prefers that the traditional comma be used in rendering their name, the comma can be force by placing a exclamation mark after the comma:

```
{ "author" : [
    { "family" : "Bennett",
      "given" : "Frank G.,! Jr.",
      "parse-names" : "true"
    }
  ]
}
```

Particles as part of the last name

The particles preceding some names should be treated as part of the last name, depending on the cultural heritage and personal preferences of the individual. To suppress parsing and treat such particles as part of the `family` name field, enclose the `family` name field content in double-quotes:

```
{ "author" : [
    { "family" : "\"van der Vlist\"",
      "given" : "Eric",
      "parse-names" : "true"
    }
  ]
}
```

Dates

The `citeproc-js` processor contains its own internal parsing code for raw date strings. Clients may take advantage of the processor's internal parser by supplying date strings as a single `raw` element:

```
{  "issued" : {
      "raw" : "25 Dec 2004"
   }
}
```

Note that the parsing of raw date strings is not part of the CSL 1.0 standard. Clients that need to interoperate with other CSL processors should be capable of preparing input in the form described above under **Data Input** → **Dates**.

Processor control

In ordinary operation, the processor generates citation strings suitable for a given position in the document. To support some use cases, the processor is capable of delivering special-purpose fragments of a citation.

`author-only`

When the `makeCitationCluster()` command (not documented here) is invoked with a non-nil `author-only` element, everything but the author name in a cite is suppressed. The name is returned without decorative markup (italics, superscript, and so forth).

```
var my_ids = {
  ["ID-1", {"author-only": 1}]
}
```

You might think that printing the author of a cited work, without printing the cite itself, is a useless thing to do. And if that were the end of the story, you would be right ...

`suppress-author`

To suppress the rendering of names in a cite, include a `suppress-author` element with a non-nil value in the supplementary data:

```
var my_ids = [
    ["ID-1", { "locator": "21", "suppress-author": 1 }]
]
```

This option is useful on its own. It can also be used in combination with the `author-only` element, as described below.

Automating text insertions

Calls to the `makeCitationCluster()` command with the `author-only` and to `processCitationCluster()` or `appendCitationCluster()` with the `suppress-author` control elements can be used to produce cites that divide their content into two parts. This permits the support of styles such as the Chinese national standard style GB7714-87, which requires formatting like the following:

> **The Discovery of Wetness**
>
> While it has long been known that rocks are dry [1] and that air is moist [2] it has been suggested by Source [3] that water is wet.
>
> **Bibliography**
>
> [1] John Noakes, *The Dryness of Rocks* (1952).
>
> [2] Richard Snoakes, *The Moistness of Air* (1967).
>
> [3] Jane Roe, *The Wetness of Water* (2000).

In an author-date style, the same passage should be rendered more or less as follows:

> **The Discovery of Wetness**
>
> While it has long been known that rocks are dry (Noakes 1952) and that air is moist (Snoakes 1967) it has been suggested by Roe (2000) that water is wet.
>
> **Bibliography**
>
> John Noakes, *The Dryness of Rocks* (1952).
>
> Richard Snoakes, *The Moistness of Air* (1967).

Jane Roe, *The Wetness of Water* (2000).

In both of the example passages above, the cites to Noakes and Snoakes can be obtained with ordinary calls to citation processing commands. The cite to Roe must be obtained in two parts: the first with a call controlled by the `author-only` element; and the second with a call controlled by the `suppress-author` element, *in that order*:

```
var my_ids = {
  ["ID-3", {"author-only": 1}]
}

var result = citeproc.makeCitationCluster( my_ids );
```

... and then ...

```
var citation, result;

citation = {
  "citationItems": ["ID-3", {"suppress-author": 1}],
  "properties": { "noteIndex": 5 }
}

[data, result] = citeproc.processCitationCluster( citation );
```

In the first call, the processor will automatically suppress decorations (superscripting). Also in the first call, if a numeric style is used, the processor will provide a localized label in lieu of the author name, and include the numeric source identifier, free of decorations. In the second call, if a numeric style is used, the processor will suppress output, since the numeric identifier was included in the return to the first call.

Detailed illustrations of the interaction of these two control elements are in the processor test fixtures in the "discretionary" category:

- AuthorOnly
- CitationNumberAuthorOnlyThenSuppressAuthor
- CitationNumberSuppressAuthor
- SuppressAuthorSolo

Multi-lingual content

The version of `citeproc-js` described by this manual incorporates a mechanism for supporting cross-lingual and mixed-language citation styles, such as 我妻栄 [Wagatsuma Sakae], 債権各論 [Obligations in Detail] (1969). The scheme described below should be considered experimental for the present. The code is intended for deployment in the Zotero reference manager; when it is eventually accepted for deployment (possibly with further modifications), the implementation can be considered stable.

Selecting multi-lingual variants

For multi-lingual operation, a style may be set to request alternative versions and translations of the `title` field, and of the author and other name fields. There are two methods of setting multilingual parameters: via `default-locale` (intended primarily for testing) and via API methods (intended for production use).

When set via `default-locale`, extensions consist of an extension tag, followed by a language setting that conforms to RFC 5646 (typically constructed from components listed in the IANA Language Subtag Registry). When set via an API method, the argument to the appropriate method should be a list of RFC 5646 language tags.

Recognized extension tags for use with `default-locale` [and corresponding API methods] are as follows:

-x-pri- **[setLangTagsForCslTransliteration()]**
: Sets a preferred language for transliterations.

-x-sec- **[setLangTagsForCslTranslation()]**
: Sets an optional secondary translation for the title field. If this tag is present, a translation in the target language will (if available) be placed in square braces immediately after the title text.

-x-sort- **[setLangTagsForCslSort()]**
: Sets the preferred language or transliteration to be used for both the title field and for names in sort keys.

An example of `default-locale` configuration:

```
<style
    xmlns="http://purl.org/net/xbiblio/csl"
    class="in-text"
    version="1.0"
    default-locale="en-US-x-pri-ja-Hrkt">
```

Multiple tags may be specified, and tags are cumulative. For readability in test fixtures, individual tags may be separated by newlines within the attribute. The following will attempt to render titles in either Pinyin transliteration (for Chinese titles) or Hepburn romanization (for Japanese titles), sorting by the transliteration.

```
<style
    xmlns="http://purl.org/net/xbiblio/csl"
    class="in-text"
    version="1.0"
    default-locale="en-US
        -x-pri-zh-Latn-pinyin
        -x-pri-ja-Latn-hepburn
        -x-sort-zh-Latn-pinyin
        -x-sort-ja-Latn-hepburn">
```

An example of API configuration:

```
citeproc.setLangTagsForCslSort(["zh-alalc97", "ja-alalc97"]);
```

The processor offers three boolean API methods that are not available via `default-locale`:

setOriginalCreatorNameFormsOption()
: With argument `true`, instructs the processor to append the personal names in their original form in parentheses after transliterations. Default is `false`.

setOriginalCreatorNameFormatOption()
: With argument `true`, instructs the processor to use the name-part ordering conventions appropriate to the original form of personal names when transliterating. When this option is in effect, names represented in a non-Byzantine script in their original form will not have their given name part truncated to initials, regardless of the setting provided by the CSL style. Default is `false`.

setAutoVietnameseNamesOption()
: With argument `true`, the processor will attempt to identify Vietnamese names, and format them correctly, with the family name always in first position. Default is `false`, because there is a small possibility of false positives when this code is enabled.

Data format

Multi-lingual operation depends upon the presence of alternative representations of field content embedded in the item data. When alternative field content is not availaable, the "real" field content is used as a fallback. As a result, configuration of language and script selection parameters will have no effect when only a single language is available (as will normally be the case for an ordinary Zotero data store).

Title

For titles and other ordinary string fields, alternative representations are placed in a separate `multi` segment on the item, keyed to the field name and the language tag (note the use of the `_keys` element on the `multi` object):

```
{ "title" : "民法",
  "multi": {
    "_keys": {
      "title": {
        "ja-alalc97": "Minpō",
        "en":"Civil Code"
      }
        }
  }
}
```

Names

For names, alternative representations are set on a `multi` segment of the name object itself (note the use of the `_key` element on the `multi` object):

```
{ "author" : [
    { "family" : "穂積",
      "given" : "陳重",
      "multi": {
        "_key": {
          "ja-alalc97": {
            "family" : "Hozumi",
            "given" : "Nobushige"
          }
        }
      }
    },
    { "family" : "中川",
      "given" : "善之助"
      "multi": {
        "_key": {
          "ja-alalc97": {
            "family" : "Nakagawa",
            "given" : "Zennosuke"
          }
        }
      }
    }
  ]
}
```

Hint

As described above, fixed ordering is used for **non-Byzantine names**. When such names are transliterated, the `static-ordering` element is set on them, to preserve their original formatting behavior.

Date parser

A parser that converts human-readable dates to a structured form is available as a self-contained module, under the name `CSL.DateParser`.

Instantiation: `CSL.DateParser`

When used as a standalone module, the parser must be instantiated in the usual way before use:

```
var parser = new CSL.DateParser;
```

Parser methods

The following methods are available on the parser object, to control parsing behavior and to parse string input.

`parser.parse(str)`

Parse the string `str` and return a date object. Within the date object, the parsed date may be represented either as a set of key/value pairs (see `returnAsKeys()`, below), or as a nested array under a `date-parts` key (see `returnAsArray()`, below).

`parser.returnAsArray()`

Set the date value on the date object returned by the `parse()` method as a nested array under a `date-parts` key. For example, the date range 31 January 2000 to 28 February 2001 would look like this in array format:

```
{
  "date-parts": [
    [2000, 1, 31],
    [2001, 2, 28]
  ]
}
```

`parser.returnAsKeys()` [default]

Set the date value on the date object returned by the `parse()` method as a set of name/value pairs. For example, the date range 31 January 2000 to 28 February 2001 would look like this in keys format:

```
{
  year: 2000,
  month: 1,
  day: 31,
  year_end: 2001,
  month_end: 2,
  day_end: 28
}
```

`parser.setOrderMonthDay()` [default]

When parsing human-readable numeric dates, assume mm/dd/yyyy ordering.

`parser.setOrderDayMonth()`

When parsing human-readable numeric dates, assume dd/mm/yyyy ordering.

parser.addMonths(str)
: Extend the parser to recognize a set of 12 additional space-delimited human-readable text months. The parser so extended will recognize months by their first three characters, unless additional characters are required to distinguish between different months with similar names. To extend also by seasons, add four additional season names to the space-delimited list of names (for a list of 16 names).

parser.resetMonths()
: Reset month recognition to the default of `jan feb mar apr may jun jul aug sep oct nov dec spr sum fal win`.

Test Suite

`Citeproc-js` ships with a large bundle of test data and a set of scripts that can be used to confirm that the system performs correctly after installation. The tests begin as individual human-friendly fixtures written in a special format, shown in the sample file immediately below. Tests are prepared for use by grinding them into a machine-friendly form (JSON), and by preparing an appropriate JavaScript execution wrapper for each. These operations are performed automatically by the top-level test runner script that ships with the sources.

Important

Note that the standard CSL test fixtures are not distributed with the processor, and must be added to the source tree separately.

Running the test suite

Tests are controlled by the `./test.py` script in the root directory of the archive. To run all standard tests in the suite using the `rhino` interpreted shipped with the processor, use the following command:

```
./test.py -s
```

Options and arguments can be used to select an alternative JavaScript interpreter, or to change or limit the set of tests run. The script options are as follows:

--help:
: List the script options with a brief description of each and exit

--tracemonkey
: Use the tracemonkey JS engine, rather than the Rhino default.

--cranky
: validate style code for testing against the CSL schema using the `jing` XML tool.

--grind
: Force grinding of human-readable test code into machine- readable form.

--standard
: Run standard tests.

--release
: Bundle processor, apply license to files, and test with bundled code.

--processor
: Run processor tests (cannot be used with the `-c`, `-g` or `-s` opts, takes only test name as single argument).

--verbose
: Display test names during processing.

The `--tracemonkey` option requires the `jslibs` JavaScript development environment. The sources for `jslibs` can be obtained from Google Code. After installation, adjust the path to the `jshost` utility in `./tests/config/test.cnf`.

Fixture layout

For infomation on the layout of the test fixtures, see the CSL Test Suite manual.

Live Demo

When accessed using a JavaScript-enabled browser with E4X support (such as **Firefox**), the `./demo/demo.html` file in the source archive (or **online**) will invoke the processor to render a few citations. The JavaScript files accompanying the page in the `./demo` directory show the basic steps required to load and run the processor, whether in the browser or server-side.