citeproc-js Documentation

Release 1.1.73

Frank Bennett

Sep 10, 2017

Contents

A JavaScript implementation of the Citation Style Language

Frank Bennett

CHAPTER 1

Table of Contents

Acknowledgements

Frank Bennett

Development of would not have been possible without the patient and careful feedback received from many contributors over the years. Particular thanks are due to the following:

- Nick Bart
- Bruce D'Arcus
- Lennard Fuller
- Fergus Gallagher
- Emiliano Heyns
- Sebastian Karcher
- Simon Kornblith
- Carles Pina
- Robert Richards
- Dan Stillman
- Rintze Zelle

If your name is not in this list, but ought to be, please give a shout!

Please also see the Acknowledgements to Bennett, *Citations Out of the Box* (2013), crediting users who provided invaluable feedback in the early development of the related project "Multilingual Zotero" (now maintained as Juris-M).

Release Announcements

Frank Bennett

v1.1.113

Fix bug in logging code introduced at 1.1.111 Minutes after a Juris-M release incorporating the changes below, a colleague knocked on my door to say that word processor input had suddenly broken. We traced the cause to entries with an invalid language code ("Japanese" rather than "ja"), which should have logged a warning without throwing an error. The cause was a typo in the logging function itself (a missing argument in the signature). Now fixed, and things should work normally.

v1.1.112

Fix Homer-worthy nesting bug A research group (at Heidelberg University) had been getting persistent errors when attempting to generate multilingual bibliographies. The fault was traced to a clear coding error in the code used to output tertiary variants (i.e. the second of two variants to a cite field value). The fault has been fixed, and multilingual bibliographies in all configurations should now render without error.

v1.1.111

Use dump rather than throw if console not available Previously, the `CSL.error()` function would throw a hard (crashing) error on systems that did not have a native `console.log()` function with the expected characteristics (hello Windows). With this fix, the code falls back to the more primitive `dump()` logging function, avoiding a crash. (Note that there was a bug in this fix, repaired at 1.1.113)

v1.1.110

Wrap section field remap in condition For legal item types in CSL-M styles, the `section` field is remapped to `locator`, the allow storage of individual statutory sections as separate items in the database. This behaviour is not desired in standard CSL, but was not properly disabled. With the addition of a conditional, remapping is now properly disabled in vanilla CSL.

v1.1.109

Fix sorting behaviour Sort comparisons in JavaScript engines are (still) producing inconsistent results. This fix tests the effect of separator strings hacked into processor-generated sort keys, choosing the separating character (at-mark `@` or field-separator `|`) that will produce a correct sort. Tests prepared in response to the relevant bug report (from a user with Danish requirements) now pass, and existing sort tests also clear.

v1.1.108

Remove `indexOf()` definition Modern JavaScript engines all have `indexOf()` as a native method on `Array()`, so this workaround is no longer necessary.

Show institution name variants even when abbrev is used Previously, when the short form of an institution name was used, its variants were not displayed, even in the first reference. With this change, variants are displayed. Some further tuning may be in order here, since institution names have differing roles for courts and for "proper" institutional authors, but we will pick up those use cases as they emerge over time.

v1.1.107

Add `default-locale` to `cs-date` Previously, in multi-layout styles, the default-locale form of dates as always used for the `accessed` date variable, and the localized form was used on all other date variables. The use of default locale can now be controlled in the same way as for `cs:label` and `cs:text`.

v1.1.106

Avoid array comprehensions On line of processor code depended on a form of assignment that is apparently not supported in some JavaScript implementations. This has been fixed.

New `default-locale` attribute for `cs:label`, `cs:text` In multi-layout styles, there was no way to force use of the default-locale version of specific terms. With `default-locale="true"` on `cs:label` and `cs:text` this is now possible.

v1.1.105

Extend use of en-dash on locator labels An en-dash was used for hyphen only on a limited subset of labels. It is generally the right thing to do, so its use has been extended.

v1.1.104

Split Institution field

Fix `locator-date` and `locator-extra` bugs The `locator-date` and `locator-extra` variables that depend on content parsed out of the `locator` field were not updating correctly in dynamic environments. This has been fixed.

Fix bugs in new `year-suffix` code from 1.1.100 The fix at 1.1.100 introduced fresh bugs in year-suffix disambiguation. These have been squashed.

v1.1.103

Title-case capitalization following forward slash With `text-case="title"` in an English locale, capitalize a word that follows a forward slash.

Escape `sup` and `sub` tags when capitalizing Properly escape and markup when applying text transforms (fixes a bug unmasked by the change above).

Use title as fallback for `citation-label` When no authors are available for `citation-label`, use a fragment of the title.

Strip font style and weight in multilingual variants When adding multilingual variants in output, suppress italics, oblique, and boldface in supplementary (secondary and tertiary) text.

v1.1.102

Include `citationID` in return from `processCitationCluster()` This is a technical change, with no impact at user level.

While building a small demo of dynamic citation editing is a Web-based WYSIWYG editor, I found that including the value of `citationID` in the return from the `processCitationCluster()` function greatly simplified page updates following a citation edit, so I added that value to the return.

v1.1.101

Delimiter bug with `year-suffix` Certain delimiters were being dropped when rendering an explicit `year-suffix` element (a numeric value rendered as a string). This has been fixed.

Stray `year-suffix` bug

In a bug related to the one above, and apparently triggered by changes in `1.1.100`, an implicit `year-suffix` was rendering on dates with empty variables.

https://forums.zotero.org/discussion/58636/collapse-yearsuffix-in-chicago-manual-of-style/#Item_5

This has been fixed.

v1.1.100

Non-breaking-space joins following initials Retain zero-width non-breaking space (`\uFEFF`) and non-breaking space (`\u00A0`) as the inter-initials join when these are the last character in the `@initialize-with` attribute value.

In the RU (Russian), CS (Czech) and FR (French) locales only, when either of the non-breaking space elements is present in the attribute value, use non-breaking space (`\u00A0`) as the given-to-family join when building initialized names in non-sort order. Otherwise, use an ordinary space for the given-to-family join.

Resolves the issue discussed at:

https://forums.zotero.org/discussion/31693/nonbreaking-space-unavailable-between-initials-and-names

NB: This behaviour was added for the CS locale at later tag `1.1.103`.

Date styling bug The affixes of a `cs:date` node with no variable content could affect the styling of subsequent date nodes. This has been fixed.

Bug in `year-suffix` The `year-suffix` form of disambiguation was misbehaving when used with `collapse="year"` or `collapse="year-suffix"`. This has been fixed.

v1.1.99

Fix pluralism of embedded labels Shortcode labels that differ from the "main" label on a number (as in "p. 123 n. 1 & 2") were not pluralizing properly. This has been fixed.

v1.1.98

Expose `parseNoteFieldHacks()` To allow small extensions to the schema of calling applications (generally not needed in Juris-M, but useful elsewhere), CSL variables can be set in the `note` field of the CSL JSON input

to the processor. The parsing code for doing so is now exposed, so that calling applications (including Juris-M) can make use of it where necessary for their own purposes.

Fix bugs in parsing of names from note field hack syntax When parsing hack syntax out of the note field, single-field names were being returned as two-field names with a value in the `family` field only. This has been fixed.

Pre-title macros in style modules Cites to cases decided by the European Court of Justice require the docket number *before* the case name. Style modules were not capable of generating this cite form, so an additional standard macro was added to the modules for that purpose.

Bibliography entries as strings In an initially unnoticed bug, the processor bibliography function was returning the elements of a bibliography as one-item lists rather than as strings. The bug was unnoticed in many contexts because JavaScript has weak "typing": an array converts to a string when combined with another string; and a one-element list stringifies without braces or comma delimiters. The bug was noticed in a context in Zotero that *required* a real string, and was duly squashed.

Locators with leading space A Zotero user reported that entering a leading space in the locator field in the word processor caused an unwanted page label ("p.") to magically appear in some styles. This has been fixed.

Handle styles with DOS and Mac line endings A Zotero user reported that the latest processor version was crashing hard when styles had DOS or Mac line endings—"normal" line endings are `line-feed` only, DOS line endings are `carriage-return+line-feed`, and Mac line endings are `carriage-return` only. This bug did not affect the distributed Juris-M styles, but it has been fixed, so if you use a non-Unix editor (such as Windows Notepad) to modify a style for some reason, it will continue to work.

Handle styles with signle-quoted XML attribute values In a bug arising from the same set of changes that yielded the line-endings issue, styles (and locales) that used single-quotes on XML attribute values were also crashing the processor. Single-quotes are perfectly valid XML, and this bug has been squashed.

v1.1.90

There are many changes to the infrastructure behind this release, and few changes to functionality apart from bug fixes. This back-room work will allow quicker releases, and lays a solid foundation for the development of legal style modules.

Here are the main items:

Processor code on GitHub Most citation-related programming activity takes place on GitHub, and I finally bit the bullet and moved the citeproc-js code there, for easier deployment and smoother collaboration with developers.

JavaScript engine testing Until recently, the processor was tested only in the Rhino JavaScript engine that runs in Java. Rhino is not used in browsers, and where the processor behaved differently under a browser engine (when sorting citations, for example), the fault was not picked up until a user noticed in the field. The test suite can now run alternative JS engines, and I will always test against the leading four engines (Rhino [Java], Spidermonkey [Firefox], V8 [Google Chrome], and JavaScript Core [Safari]) before releases.

Style and locale parsing To do its thing, the processor must parse the XML of a style file and its associated locale strings for internal use. Although citeproc-js supported several methods of parsing XML (DOM, E4X, and a pre-parsed bespoke JSON format), setup was a non-standard ill-documented headache, with the processor "discovering" an unconnected parser object via JavaScript closure–a procedure that is as clumsy as it sounds. The parsers are now embedded in the processor itself, and it will digest any form of XML that you throw at it. Deployments should be much simpler for it.

Validated test fixtures The test suite that backs up processor development hadn't received much attention in recent years (apart from the addition of many test fixtures). The long-dormant facility for validating the CSL style objects used for testing has been resurrected, and all test CSL now passes validation. This brings greater assurance that what we see in the test framework will replicate in the field.

Locators in legal style modules Modular style code is challenging for locator formatting, in particular, because these are heavily dependent on context, and the context is supplied by the calling style. With revisions to CSL-M, the extended version of CSL used in Juris-M styles, locators can be positioned using "smart conditions" that read the essential features of surrounding context. As a result the burden on legal style development has lightened considerably, and we are now ready to scale the system out to cover additional jurisdictions.

Disambiguation The processor compares the shortest form of citations for ambiguity before adding information to citations. Legal styles that implement a "five-footnote rule" must test the *near* form of cites for this to work. That wasn't happening, but it is now.

Straight-quotes hanging bug When straight quotes were set as the preferred quotation marks in a new CSL locale, it triggered a hanging bug in the processor. This has been fixed.

Nesting mismatch errors The processor builds citations as deeply nested string sets, with the siblings joined by a delimiter at each level to produce printed output. For performance reasons the nesting is "spoofed" by markers in a list executed from start to finish, and if the markers are incorrectly place, weird things can happen (in theory). The markers were *very slightly* incorrect in two instances that manifested in Zotero/Juris-M, but not in the pre-release test suite. The bugs have been fixed, and the test suite has been fixed to pick these errors up if they every occur again.

Arabic locale The Arabic locale was not loading. At all. Ever. This has been fixed.

Charset sniffing The regular expression used to guess whether the character set of some strings is "romanesque" included some dingbat-type characters. These have been removed.

Safe syntax for global replacements The processor was attempting to perform global replacements with str.replace("old", "new", "g"). This worked in Rhino, but broken in browser JavaScript engines. That code has been replaced with str.split("old").join("new"), which works correctly everywhere.

Safer sorting Internal sort keys included spaces, and spaces sorted differently depending on the JavaScript engine. Spaces have been replaced with "A" in sort keys, which has the effect of forcing the treatment of each element as a separate sort key.

Remove lurking list comprehensions List comprehensions (as in [key, val] = myFun();) were removed from citeproc-js quite some time ago, because they are not valid across all JavaScript engines. Two still remained in a debugging statement. They have been removed.

Demo: Standalone Bibliography

Frank Bennett

This page illustrates a simple deployment that renders a small bibliography. The original of this example was kindly provided by Chris Maloney back when was hosted on BitBucket. I have modified it (very) slightly to illustrate the use of the Zotero API.

My Amazing Bibliography

Style and Data

We're going to need some data. The URL below will pull eight items from the public library "Pitt-Greensburg English Literature Capstone" via the Zotero API.

```
var chosenLibraryItems = "https://api.zotero.org/groups/459003/items?format=csljson&
↪limit=8&itemType=journalArticle";
```

We're also going to need a style. The slug below is the machine-readable name of Chicago Full Note (with bibliography). In this demo we'll use it to fetch the style over the wire, but it could equally well be embedded in our "application," so we just note its name here.

```
var chosenStyleID = "chicago-fullnote-bibliography";
```

On page load, we fetch the citation data, and parse it into a JavaScript object.

```
var xhr = new XMLHttpRequest();
xhr.open('GET', chosenLibraryItems, false);
xhr.send(null);
var citationData = JSON.parse(xhr.responseText);
```

Data rearrangement

The Zotero API delivers citation objects in the CSL JSON format expected by the processor, but the objects need to be reorganized for keyed access. We also need an array of the keys. Here we store the keyed citations as `citations`, and the array of IDs as `itemIDs`.

```
var citations = {};
var itemIDs = [];
for (var i=0,ilen=citationData.items.length;i<ilen;i++) {
  var item = citationData.items[i];
  if (!item.issued) continue;
  if (item.URL) delete item.URL;
  var id = item.id;
  citations[id] = item;
  itemIDs.push(id);
}
```

The `sys` object

The processor needs two hook functions, one to acquire arbitrary locales (`retrieveLocale()`), and another to acquire items by key (`retrieveItem()`). Locales are identified by their RFC-5646 language and region. For this demo we'll pull them from a remote repository, but the files would ordinarily be stored locally.

```
citeprocSys = {
  retrieveLocale: function (lang){
    var xhr = new XMLHttpRequest();
    xhr.open('GET', 'https://raw.githubusercontent.com/Juris-M/citeproc-js-docs/
↪master/locales-' + lang + '.xml', false);
    xhr.send(null);
    return xhr.responseText;
  },
  retrieveItem: function(id){
    return citations[id];
  }
};
```

Processor instantiation

We set up a function to instantiate the processor. In this code, we retrieve the style and locale as serialized XML because it is easy to do. If the objects were in DOM or E4X format, we could deliver that as well: the only constraint

is that both need to be in the same form.

```
function getProcessor(styleID) {
  var xhr = new XMLHttpRequest();
  xhr.open('GET', 'https://raw.githubusercontent.com/citation-style-language/styles/
↪master/' + styleID + '.csl', false);
  xhr.send(null);
  var styleAsText = xhr.responseText;
  var citeproc = new CSL.Engine(citeprocSys, styleAsText);
  return citeproc;
};
```

Putting it all together

To generate a bibliography, we grab the processor, set the item IDs on it, and request bibliography output. The result is a two-element array composed of a memo object and a list of rendered strings. Joining up the strings gives us the HTML for inclusion in the page.

```
function processorOutput() {
  ret = '';
  var citeproc = getProcessor(chosenStyleID);
  citeproc.updateItems(itemIDs);
  var result = citeproc.makeBibliography();
  return result[1].join('\n');
}
```

Demo: Dynamic Citations

Frank Bennett

Dynamic Editing

Style:

This page demonstrates dynamic citation editing in the browser. The underlying code was prepared in response to work by Derek Sifford. Click on the chevrons to open a citation widget, select one or more references, and press "Save" to add them to the document. Use the pulldown list above to transform the document to another style. Click on the "More" button below for information on running the code locally, and on adapting it for use in production.

Running locally

One of the main purposes of this page is to provide a worked example for developers. A good way to explore the way it all works is to run the page locally. Here is how to set that up.

Requirements

- git
- node.js

- npm
- Sphinx

Local setup

Fetch the repo Clone the `citeproc-js` documentation project and enter its top-level directory:

```
git clone https://github.com/Juris-M/citeproc-js-docs.git --recursive
cd citeproc-js-docs
```

Build the docs The following command should work:

```
make html
```

Run a server using `node.js` The built page uses `XMLHttpRequest()` calls, so it must be viewed through a web server. To run a simple server using `node.js`, an incantation like this should do the trick:

```
npm install -g http-server
http-server _build/html
```

Source File Overview

`_static/js/citeproc.js` The CSL processor. Seven years in development, backed up by 1,260 test fixtures and 1,318 unique citation styles, with extended support for multilingual and legal citation.

`_static/js/citeworker.js` A web worker implementing the two API calls on which `citesupport` depends.

`_static/js/citesupport-es6.js` An `es6` class object with DOM logic for dynamic citation editing. With some tweaks, this can be run inside a WYSIWYG editor of your choice.

`_static/css/screen.css` The CSS code for the documentation, including the demo pages.

`_static/data/items` A few sample items for the dynamic editing demo, in CSL JSON format.

`_static/data/locales` The standard CSL locales.

`_static/data/styles` The CSL styles used in the demo. The "JM" styles are from the Juris-M styles repository, and have modular legal style support. The remainder are from the official CSL repository, which feeds the Zotero styles distribution site.

`_static/data/juris` A set of legal style modules resides here. Legal citation support is easily extensible to jurisdictions worldwide via the Juris-M Style Editor (GitHub account required).

Integrator notes

Here are some notes on things relevant to deployment:

- The class should be instantiated as `citesupport`. The event handlers expect the class object to be available in global context under that name.
- If `config.demo` is `true`, the stored object `citationIdToPos` maps citationIDs to the index position of fixed "pegs" in the document that have class `citeme`. In the demo, this map is stored in localStorage, and is used to reconstruct the document state (by reinserting `class:citation` span tags) on page reload.

- If `config.demo` is `false`, the document is assumed to contain `class:citation` span tags, and operations on `citeme` nodes will not be performed. In non-demo mode, `citationIdToPos` carries the index position of citation nodes for good measure, but the mapping is not used for anything.
- The `spoofDocument()` function brings citation data into memory. In the demo, this data is held in localStorage, and `spoofDocument()` performs some sanity checks on data and document. For a production deployment, this is the place for code that initially extracts citation data the document (if, for example, it is stashed in data-attributes on citation nodes).
- The `setCitations()` function is where citation data for individual citations would be saved, at the location marked by NOTE.
- The user-interface functions `buildStyleMenu()` and `citationWidget()` are simple things cast for the demo, and should be replaced with something a bit more functional.
- The `SafeStorage` class should be replaced (or subclassed?) for deployment with a class that provides the same methods. If the citation objects making up `citationByIndex` are stored directly on the `class:citation` span nodes, the getter for that value should harvest the values from the nodes, and store them on `config.citationByIndex`. The setter should set `config.citationByIndex` only, relying on other code to update the node value.
- Probably some other stuff that I've overlooked.

Worker API

The heavy lifting is done by the CSL processor, which runs in a separate thread as a web worker. Only the document-facing interface of the worker is described here: it should not be necessary to tangle with the internals of the worker itself. Its only idiosyncracy is that it assigns note numbers (reflected in the return) in citation sequence—in contrast to word processor context, it assumes that the only footnotes in the document are those generated automatically by a note style. If that is not true in your context, you will want to disable that behavior, and do whatever is necessary on document side to extract real note numbers for delivery to the processor.

The worker is controlled by two methods, `callInitProcessor()` and `callRegisterCitation()`, each with a corresponding message and return event.

`citesupport.callInitProcessor(styleID, localeID)` This method is used on page load, on change of style, and when all citations have been removed from the document. The `styleID` argument is mandatory. If `localeID` is not provided, the processor will be configured with the `en-US` locale.

The `citesupport.callInitProcessor` method implicitly accesses the `config.citationByIndex` array, which must be accessible in page context. If the array is empty, the processor will be initialized without citations. If the array contains citations, the processor will be initialized to that document state, and return an array of arrays as `rebuildData`, for use in reconstructing citations in the document text. Each sub-array contains a citation ID, a note number, and a citation string. For example, if the `styleID` is for a `note` style, and if `config.citationByIndex` yields the citations "Wurzel Gummidge (1990)" and "My Aunt Sally (2001)," the `rebuildData` structure would look like this:

```
[
    [
       "lu7Tu3ki",
       "1",
       "Wurzel Gummidge (1990)"
    ],
    [
       "ko4aNoo9",
       "2",
       "My Aunt Sally (2001)"
```

```
    ]
]
```

`citesupport.callRegisterCitation(citation, preCitations, postCitations)` This method is used to add or to edit citations. All three arguments are mandatory. `citation` is an ordinary citation object as described above. `preCitations` and `postCitations` are arrays of arrays, in which each sub-array is composed of a citation ID and a note number. For example, if a note citation is to be inserted between the "Wurzel Gummidge" and "Aunt Sally" citations in the example above, these would have the following form:

```
preCitations = [
    [
        "lu7Tu3ki",
        "1"
    ]
];

postCitations = [
    [
        "ko4aNoo9",
        "3"
    ]
];
```

Notice the change to the note number: the processor registers note numbers for use in back-references, but maintenance of correct note numbering must be handled in document-side code.

The `citesupport.callRegisterCitation` method returns two values from the processor: `citationByIndex` (described above) and `citations`. The latter is an array of one or more arrays, each composed of a citation position index, a string, and a citation ID. For example, the return value to insert a citation "Calvin (1995); Hobbes (2016)" between the "Wurzel Gummidge" and "My Aunt Sally" citations would look something like this:

```
[
   [
       1,
       "Calvin (1995); Hobbes (2016)",
       "Ith7eg8T"
   ]
]
```

Note that the return value might contain updates for multiple citations.

Demo: Editor

Frank Bennett

Editor

Running locally

Like the previous page, the main purpose here is to provide a worked example for developers. A good way to explore the way it all works is to run the page locally. Here is how to set that up.

Requirements

- git
- node.js
- npm
- Sphinx

Local setup

Fetch the repo Clone the `citeproc-js` documentation project and enter its top-level directory:

```
git clone https://github.com/Juris-M/citeproc-js-docs.git --recursive
cd citeproc-js-docs
```

Build the docs The following command should work:

```
make html
```

Run a server using `node.js` The built page uses `XMLHttpRequest()` calls, so it must be viewed through a web server. To run a simple server using `node.js`, an incantation like this should do the trick:

```
npm install -g http-server
http-server _build/html
```

Source File Overview

As this demo is built as a Sphinx page for deployment on ReadTheDocs, the source files are kind of scattered and mixed. The following list should help you find the essentials.

`_static/tinymce/js/tinymce/plugins/citesupport.js` This is the core plugin for citation support. It spins up a web worker (from `classes/citeworker.js`) that runs `citeproc.js` to handle the actual formatting of citations, and manages page updates.

`_static/tinymce/js/tinymce/plugins/citestylemenu.js` This supplies a tinyMCE menu for changing citation styles.

`_static/tinymce/js/tinymce/plugins/citeaddedit.js` This supplies the primitive citation widget used in the demo editor. In production you would obviously want something a *little* more sophisticated. This plugin is the place to implement that.

`_static/css/screen.css` There is some CSS code in here that is relevant to the layout of bibliographies and the decoration of citations and footnotes.

`_templates/layout.html` Check here (particularly at the bottom of the file) for the incantations that bring up tinyMCE with citation support.

`substitutions.txt` The placeholder that tinyMCE installs itself to is supplied by the "editor" substitution element.

`_static/data/items` A few sample items for the dynamic editing demo, in CSL JSON format.

`_static/data/locales` The standard CSL locales.

`_static/data/styles` The CSL styles used in the demo. The "JM" styles are from the Juris-M styles repository, and have modular legal style support. The remainder are from the official CSL repository, which feeds the Zotero styles distribution site.

_static/data/juris A set of legal style modules resides here. Legal citation support is easily extensible to jurisdictions worldwide via the Juris-M Style Editor (GitHub account required).

Worker API

The heavy lifting is done by the CSL processor, which runs in a separate thread as a web worker. Only the document-facing interface of the worker is described here: it should not be necessary to tangle with the internals of the worker itself. Its only idiosyncracy is that it assigns note numbers (reflected in the return) in citation sequence—in contrast to word processor context, it assumes that the only footnotes in the document are those generated automatically by a note style. If that is not true in your context, you will want to disable that behavior, and do whatever is necessary on document side to extract real note numbers for delivery to the processor.

The worker is controlled by two methods, `callInitProcessor()` and `callRegisterCitation()`, each with a corresponding message and return event.

citesupport.callInitProcessor(styleID, localeID) This method is used on page load, on change of style, and when all citations have been removed from the document. The `styleID` argument is mandatory. If `localeID` is not provided, the processor will be configured with the `en-US` locale.

The `citesupport.callInitProcessor` method implicitly accesses the `config.citationByIndex` array, which must be accessible in page context. If the array is empty, the processor will be initialized without citations. If the array contains citations, the processor will be initialized to that document state, and return an array of arrays as `rebuildData`, for use in reconstructing citations in the document text. Each sub-array contains a citation ID, a note number, and a citation string. For example, if the `styleID` is for a `note` style, and if `config.citationByIndex` yields the citations "Wurzel Gummidge (1990)" and "My Aunt Sally (2001)," the `rebuildData` structure would look like this:

```
[
    [
       "lu7Tu3ki",
       "1",
       "Wurzel Gummidge (1990)"
    ],
    [
       "ko4aNoo9",
       "2",
       "My Aunt Sally (2001)"

    ]
]
```

citesupport.callRegisterCitation(citation, preCitations, postCitations) This method is used to add or to edit citations. All three arguments are mandatory. `citation` is an ordinary citation object as described above. `preCitations` and `postCitations` are arrays of arrays, in which each sub-array is composed of a citation ID and a note number. For example, if a note citation is to be inserted between the "Wurzel Gummidge" and "Aunt Sally" citations in the example above, these would have the following form:

```
preCitations = [
    [
        "lu7Tu3ki",
        "1"
    ]
];

postCitations = [
    [
```

```
        "ko4aNoo9",
        "3"
    ]
];
```

Notice the change to the note number: the processor registers note numbers for use in back-references, but maintenance of correct note numbering must be handled in document-side code.

The `citesupport.callRegisterCitation` method returns two values from the processor: `citationByIndex` (described above) and `citations`. The latter is an array of one or more arrays, each composed of a citation position index, a string, and a citation ID. For example, the return value to insert a citation "Calvin (1995); Hobbes (2016)" between the "Wurzel Gummidge" and "My Aunt Sally" citations would look something like this:

```
[
   [
       1,
       "Calvin (1995); Hobbes (2016)",
       "Ith7eg8T"
   ]
]
```

Note that the return value might contain updates for multiple citations.

Getting started

Frank Bennett

Introduction

The Citation Style Language (CSL) is a powerful machine-readable schema for describing citation formats. The processor is a JavaScript implementation of CSL, with extended features to support legal styles and multilingual citations. This chapter explains how to obtain the processor sources and test the installation.

System requirements

The items below are required to download and test the processor. In deployment environments, only JavaScript is required.

git The sources live in GitHub, so you should have installed locally to obtain the sources.

Python The `test.py` script that runs the test suite is written in . Both Python 2.7 and Python 3.5 are supported.

Java The JavaScript interpreter and the XML validator included in the processor sources depend on Java.

Obtaining the sources

Use this incantation at the command line to fetch the sources and enter the project directory:

```
git clone --recursive https://github.com/Juris-M/citeproc-js.git
cd citeproc-js
```

If you forget the `--recursive` option, you can follow up by fetching the submodules with this:

```
git submodule update --init --remote
```

Running the test suite

To confirm that everything is working nicely, run the test suite with the default intepreter:

```
./test.py -r
```

The test run should complete without errors.

Exploring the test suite

Frank Bennett

Introduction

The CSL test suite is an essential backdrop to development. This chapter explains the basic features of the `./test.py` script that runs the suite.

Running individual fixtures

The processor ships with a set of local tests, and includes the official CSL test fixtures as a submodule, in the following locations:

Table 1.1: CSL test fixture locations

Fixture type	Relative path
local tests	`./tests/fixtures/local`
Official CSL test suite	`./tests/fixtures/std/processor-tests/humans`

Fixture filenames have two elements (a group name and a test name), separated by an underscore and with a `.txt` extension. To run a single test, provide the test name as the argument to the `-s` option. For cut-and-paste convenience, various forms of the name are recognized:

```
./test.py -s name_WesternSimple
./test.py -s name_WesternSimple
./test.py -s name_WesternSimple.txt
```

Alternative JavaScript engines

The Rhino JavaScript engine bundled with the processor sources is very solid, but also very slow. For faster processing, install a standalone JS engine compiled for your platform. In addition to Rhino, three engines are supported:

Table 1.2: JavaScript Engine Info

Name	Browser	Configuration Nickname	Default Command
Rhino	(none)	`rhino`	java -client -jar ./rhino/js-1.7R3.jar -opt 8
Spidermonkey	Firefox	`mozjs`	js24
V8	Google Chrome	`v8`	d8
JavascriptCore	Safari	`jsc`	jsc

Here are some numbers to give an idea of the relative performance of the engines. The times are for a complete run of the test suite on a 64-bit Ubuntu laptop that I use for development:

Table 1.3: Completion times for tests

Engine	(seconds)
Rhino	55
Spidermonkey 24	31
V8	22
JavaScriptCore	20

After installing an engine on your system (as a binary package, or by compiling from scratch), check that it will run from the command line, and then edit its `command` entry in the configuration file at `./tests/config/test.cnf` as required:[1]

```
[jsc]
command: jsc

[v8]
command: d8.sh

[mozjs]
command: js24

[rhino]
command: java -client -jar ./rhino/js-1.7R3.jar -opt 8

[validation]
validator: java -client -jar ./jing/jing-20131210.jar -c
schema: ./csl-schemata/csl/csl.rnc
schema-m: ./csl-schemata/csl-m/csl-mlz.rnc
```

Once configured, an alternative engine can be run by giving its nickname to the `./test.py` script:

```
./test.py -e jsc -r
```

Validating test-fixture CSL

The syntax of CSL styles is defined by a RELAX NG schema. In addition to official CSL, the processor supports CSL-M, an extended version of the language with a separate schema. Styles of the two types are distinguished by the `version` attribute on the top-level node:

[1] Note the use of a shell script for `d8`, the standalone version of the Google Chrome V8 engine, which (apparently) must be run from the directory containing its binary.)

Table 1.4: CSL versions

CSL version	attribute value	sample style node
CSL 1.0	1.0	`<style xmlns="http://purl.org/net/xbiblio/csl" version="1.0" class="note"/>`
CSL-M	1.1mlz1	`<style xmlns="http://purl.org/net/xbiblio/csl" version="1.1mlz1" class="note"/>`

The bundled validator can be used to check that fixtures are syntactically correct. To run the validator against all fixtures, use the `-c` option with no argument:

```
./test.py -c
```

To test a single fixture, provide its name:

```
./test.py -c name_WesternSimple.txt
```

In its mode, the script does not explicitly report success (apart from writing a progress dot to the console): it produces chatter only if a test fails.

Running the Processor

Frank Bennett

Introduction

The processor loads as a single `CSL` object, and must be instantiated before use:

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

sys *Required.* A JavaScript object providing (at least) the functions `retrieveLocale()` and `retrieveItem()`.

style *Required.* CSL style as serialized XML (if `xmldom.js` is used) or as JavaScript object (if `xmljson.js` is used).

lang *Optional.* A language tag compliant with RFC 5646. 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.

Required `sys` functions

Two locally defined synchronous functions on the `sys` object must be supplied to acquire runtime inputs.

retrieveLocale

The `retrieveLocale()` function fetches CSL locales needed at runtime. The locale source is available for download from the CSL locales repository. The function takes a single RFC 5646 language tag as its sole argument, and returns a locale object. The return may be a serialized XML string, an E4X object, a DOM document, or a JSON or JavaScript representation of the locale XML. If the requested locale is not available, the function must return a value that tests `false`. The function *must* return a value for the `us` locale.

retrieveItem

The `retrieveItem()` function fetches citation data for an item. The function takes an item ID as its sole argument, and returns a JavaScript object in CSL JSON format.

Public methods: data and rendering

The instantiated processor offers a few basic methods for handling input and obtaining rendered output. A few terms will be used with a specific meaning in the descriptions below.

item An item is a single bundle of metadata for a source to be referenced. See the CSL Specification for details on the fields available on an item, and the CSL-JSON chapter of this manual for the format of specific field types. Every item must have an `id` and a `type`.

citation A citation is a set of one or more items, optionally supplemented by locator information, prefixes or suffixes supplied by the user.

registry The processor maintains a stateful registry of details on each item submitted for processing. Registry entries are maintained automatically, and cover matters such as disambiguation parameters, sort sequence, and the first reference in which an item occurs.

citable items Citable items are those meant for inclusion only if used in one or more citations.

uncited items Uncited items are those meant for inclusion regardless of whether they are used in a citation.

updateItems

The `updateItems()` method accepts a single argument when invoked as a public method, and refreshes the registry with a designated set of citable items. Citable items not listed in the argument are removed from the registry:

```
citeproc.updateItems(idList);
```

idList *Required.* An array of item `id` values, which may be number or string:

```
['Item-1', 'Item-2']
```

updateUncitedItems

Like its corollary above, the `updateUncitedItems()` method the registry accepts a single argument, but refreshes the registry with a designated set of uncited items. Uncited items not listed in the argument are removed from the registry.

```
citeproc.updateItems(idList);
```

idList *Required.* A JavaScript array of item `id` values, which may be number or string:

```
['Item-1', 'Item-2']
```

processCitationCluster

Use the `processCitationCluster()` method to generate and maintain citations dynamically in the text of a document. The method takes three arguments:

```
var result = citeproc.processCitationCluster(citation, citationsPre, citationsPost);
```

citation The citation argument is a citation object as described in the CSL-JSON section of this manual.

citationsPre An array citationID/note-number pairs preceding the target citation.

citationsPost A list of citationID/note-number pairs following the target citation (note numbers to reflect the state of the document after the insertion).

The result 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.

Code invoking `processCitationCluster()` on a document from which several citations have already been registered in the processor might look like this:

```
var citation = {
    properties: {
        noteIndex: 3
    },
    citationItems: {
        id: 'Item-X' // A work by Richard Snoakes
    }
}
var citationsPre = [ ["citation-quaTheb4", 1], ["citation-mileiK4k", 2] ];
var citationsPost = [ ["citation-adaNgoh1", 4] ];
var result = citeproc.processCitationCluster(citation, citationsPre, citationsPost);
console.log(JSON.stringify(result[0], null, 2));
{
    "bibchange": true
}
console.log(JSON.stringify(result[1], null, 2));
[
    [ 1,"(Ronald Snoakes 1950)" ], // Existing citation modified by disambiguation
    [ 3,"(Richard Snoakes 1950)" ]
]
```

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

previewCitationCluster

Use `previewCitationCluster()` to generate accurately formatted citations as they would appear at a given location within a document managed using `processCitationCluster()`. The method accepts four arguments, the first three of which are identical to those accepted by `processCitationCluster()`. The fourth argument may be used to control the output mode.

```
var result = citeproc..processCitationCluster(citation, citationsPre, citationsPost,␣
↪format);
```

citation *See ‘‘processCitationCluster()‘‘ above.*

citationsPre *See ‘‘processCitationCluster()‘‘ above.*

citationsPost *See ‘‘processCitationCluster()‘‘ above.*

format The optional format argument may be one of `html`, `text` or `rtf`. If this argument is not provided, the default value set in the instantiated processor is used.

The result is a string representation of the target citation, in the specified format. Changes made to the registry (necessary for correct disambiguation, sorting and other adjustments) are reversed after the citation is generated, leaving the registry in its original state.

makeCitationCluster

Use `makeCitationCluster()` to generate citations without the burden of registry adjustments. The method accepts an array of cite-items as its sole argument:

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

idList An array of cite-items, as specified in the CSL-JSON section of this manual. Note the additional cite-item options noted there.

While `makeCitationCluster()` is faster than its companions, note that it does not perform the citation sort, if any, that might be required by the style, and that it does not perform disambiguation or apply style rules to adjust the cites as appropriate to the context.

makeBibliography

The `makeBibliography()` method returns a single bibliography object based on the current state of the processor registry. It accepts on optional argument.

```
var result = citeproc.makeBibliography(filter);
```

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, with registered items “Item-1” and “Item-2”).

```
[
   {
      maxoffset: 0,
      entryspacing: 0,
      linespacing: 0,
      hangingindent: false,
      second-field-align: false,
      bibstart: "<div class=\"csl-bib-body\">\n",
      bibend: "</div>",
      bibliography_errors: [],
      entry_ids: [\"Item-1\", \"Item-2\"]
   },
   [
      "<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.

Dirty Tricks

Partial suppression of citation content

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

Selective output with `makeBibliography()`

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.

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.

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

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

`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);
```

CSL-JSON

Frank Bennett

Introduction

Citation data is added to the instantiated processor in JSON format, using one of three structures:

Items Each item carries the details of a single unique bibliographic resource.

Items are retrieved by the processor implicitly using its `retrieveItem()` method, based on a unique `id`.

Citations Each citation carries the details of a single in-document cluster of one or more items, with pinpoint details specified by an array of cite-items.

Citations are submitted to the processor using the `processCitationCluster()` or `previewCitationCluster()` methods.

Cite-items Each cite-item carries the pinpoint details of a specific reference to a bibliographic resource described by an item.

Cite-items are either submitted to the processor as part of a citation, or as the primary input to the `makeCitationCluster()` method.

The structure of these objects is described below.

Items

Encoding citation data items in properly formatted CSL-JSON is essential to getting correct results from the CSL Processor. Each citation item is composed of fields of various types. Multiple citation items can be packaged into a container, which allows related citations to be treated as a unit of citations.

Field Types

`id` Field

Required. The *id* field is a simple field containing any string or numeric value. The value of the ID field must uniquely identify the item, as this field is used to retrieve items by their ID value.

```
{
    "id":"unique_string-1219205"
}
```

Type Field

Required. The *type* field is a simple field containing a string value. CSL-JSON constrains the possible for values of the *type* field to a limited set of possible values (*e.g.*, "book" or "article"). The type must be a valid CSL type under the schema of the installed style. See the schemata of CSL and CSL-M for their respective lists of valid types.

```
{
    "id":"unique_string-1219205",
    "type":"book"
}
```

Ordinary Fields

An ordinary field type is a simple field containing a string or numeric value. In ordinary fields, the processor recognizes a limited set of HTML-like tags for visual formatting. (See *HTML-Like Formatting Tags*) below. One common ordinary field is *title* which identifies the title of the citation item. The fields that a citation may have is determined by the item type (see previous). Unrecognized fields will be simply ignored by the CSL processor. For the fields available on each item type, see the listing for CSL provided by Aurimas Vinckevicius, and that for CSL-M provided by yours truly.

```
{
    "id":"unique_string-1219205",
    "type":"book",
    "title":"Book Title",
```

```
    "arbitraryField":"An example arbitrary field with arbitrary data. This field␣
↪will be ignored by the CSL Processor."
}
```

Name Fields

A name field is a complex field that lists persons as authors, contributors, or creators, etc. The field is an array of objects, with each object containing information about one person. Date fields should generally have two properties: "family", and "given". The "family" property represents the familial name that a person inherits. The "given" property represents the name a person has been given.

The CSL-JSON allows some flexibility about how parts of a person's name are encoded. For instance, family name affixes such as "van" and "de las" can be encoded as part the family name in the "family" property. Likewise, suffixes such as titles, generational designations, credentials, and honors can be encoded as part of the given name. Generational designations (like "Jr" or "IV") immediately follow the given name without a comma. All other suffixes should follow next, with each suffix preceeded by a comma.

```
"author": [
    {
        "family": "de las Casas",
        "given": "Bartolomé",
    },
    {
        "family": "King",
        "given": "Rev. Martin Luther Jr., Ph.D.",
    }
]
```

In the previous example, lowercase elements before the family name are treated as "non-dropping" particles, and lowercase elements following the given name as "dropping" particles. However, these special name parts could also be encoded in discrete properties.

```
"author": [
    {
        "family": "Casas",
        "given": "Bartolomé",
        "non-dropping-particle":"de las"
    },
    {
        "family": "King",
        "given": "Martin Luther",
        "suffix":"Jr., Ph.D.",
        "dropping-particle":"Rev."
    }
]
```

Some personal names are represented by a single field (e.g. mononyms such as "Prince" or "Plato"). In such cases, the name can be delivered as a lone family element. Institutional names may be delivered in the same way, but it is preferred to set them instead as a literal element:

```
"author": [
    {
        "family": "Socrates",
    },
    {
```

```
            "literal": "International Business Machines"
        }
]
```

Date Fields

A date field is a complex field that expresses a date or a range of dates. An example date field in CSL is *issued*, which identifies the date an item was issued or published. Date fields can be expressed in two different formats. The first format is an array format (note the double-nesting of the array). To express a date range in this format, the ending date would be set as a second array.

Array Format

```
"archived": {
     "date-parts": [[ 2005, 4, 12 ]]
},
"issued": {
     "date-parts": [[ 2000, 3, 15 ], [2000, 3, 17]]
}
```

The second date format is a raw string. The recommended encoding is a string that represents the date in a numberic year-month-day format. However, the date parser in citeproc-js will correctly interpret a wide variety of sensible date conventions.

Raw Format

```
"archived": {
     "raw": "2005-4-12"
},
"issued": {
     "raw": "2000-3-15/2000-3-17"
}
```

Citations

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 cite-items, as described in the next section.

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.

Cite-Items

Cite-items describe a specific reference to a bibliographic item. The fields that a cite-item may contain depend on its context. In a citation, cite-items listed as part of the `citationItems` array provide only pinpoint, descriptive, and text-suppression fields:

```
{
    id:"item1",
    locator: 123,
    label: page,
    prefix: "See ",
    suffix: " (arguing that X is Y)"
}
```

The `id` field is required. The following additional fields may be included on a cite-item:

- `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 Partial suppression of citation content under Running the Processor :: Dirty Tricks for more details.)
- `prefix`: a string to print before this cite item;
- `suffix`: a string to print after this cite item.

When used in the array argument to `makeCitationCluster()`, cite-items may include additional parameters that affect the styled format of the individual cite-items:

- `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.
- `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.

HTML-Like Formatting Tags

Several tags are recognized in CSL-JSON input. While they are set in an HTML-like syntax for convenience of processing, that mimicry does not imply general support for HTML markup in the processor: tags that do not fit the patterns described below are treated as raw text, and will be escaped and rendered verbatim in output.

Note that tags must be JSON-encoded in the input object:

```
This is &lt;italic&gt; text.
```

italics Set the enclosed text in *italic* style. This tag will "flip-flop," setting the text in roman type if the style applies italic style to the field.

bold Set the enclosed text in **boldface** type. This tag will "flip-flop," setting the text in roman type if the style applies boldface type to the field.

superscript Set the enclosed text in . This tag will "flip-flop," setting the text in roman type if the style applies small-caps to the field.

^{superscript} Set the enclosed text in form.

_{subscript} Set the enclosed text in form.

superscript Suppress case-changes that would otherwise be applied to the enclosed text by the style.

"Cheater Syntax" for Odd Fields

The CSL variables needed to render a particular citation may not be available directly in the data structures of a calling application. For those situations, the processor can recognize supplementary values entered into the `note` field of a CSL item. The facility is available when the processor is run with the `field_hack` option (default is `true`):

```
citeproc.opt.development_extensions.field_hack = true
```

This method of data entry is intended as a temporary workaround to avoid blocking issues in user projects; it is not supported by all CSL processors, and does not form part of the CSL standard.

Two forms of "cheater syntax" are recognized in the CSL `note` field. When `field_hack` is enabled, the processor will recognize both, and the two forms may be mixed within the field.

Tip: Variable Names

In the braced-entry and line-entry descriptions below, a recognized `<variable_name>` is one which consists of:

- Uppercase roman characters *only* (i.e. `[A-Z]+`); or
- Lowercase roman characters, hyphens, or underscores *only* (i.e. `[-_a-z]+`).

Braced-entry

Awkward to type, ugly to read and fragile, the braced-entry syntax has been recognized by the processor for several years, and a significant proportion of entries in circulation rely on it. The general markup pattern for a single variable looks like this:

```
{:&lt;variable_name&gt;:&lt;value&gt;}
```

A recognized `<variable_name>` must appear be between the colons (see *Tip: Variable Names* above). Arbitrary text, including spaces but excluding newlines, is permitted after the second colon (leading and trailing space will be trimmed from the variable value). There is no means of escaping a closing brace.

Line-entry

Full support for the line-entry format debuted with `citeproc-js` version `1.1.135`, in belated response to repeated suggestions on the Zotero forums. The general markup for inline syntax looks like this:

```
<variable_name>:<value>
```

The line must begin with a recognized `<variable_name>` punctuated with a colon (see *Tip: Variable Names* above). Arbitrary text, including spaces, is permitted to the end of the line (leading and trailing space will be trimmed from the variable value).

Handling of entries

Conforming braced-entries and line-entries must occur at the top of the `note` field. The set of entries may begin on the first *or* the second line of the field: parsing stops when the parser encounters a non-empty line *after the first* that does not fit the descriptions above. Parsed entries are removed from the `note` field before onward processing.

Multiple entries are recognized. In the case of name variables (see below), entries are cumulative; for other variables, the last entry encountered wins (but see the cautionary notes on the override of existing item data, below).

Value formats

Variables that can be set with the "cheater syntax" are of four types: *date*, *name*, *type*, and *ordinary*

Date variables Date variables should be entered in ISO year-month-day syntax:

```
original-date: 2001-12-31
```

The processor's internal date parsing function, applied to the date content, recognizes date ranges as well:

```
original-date: 2001-12-15/2001-12-31
```

> **Caution: Overrides (date variables)**
>
> By default, date variables entered in "cheater syntax" will override an existing value in the item. This is permitted because the processor's internal parser may offer better recognition of date forms (such as ranges) than that of the calling application. Date value override may be disabled by setting the relevant toggle to `false` after processor instantiation:
>
> ```
> citeproc.opt.development_extensions.allow_field_hack_date_override = false
> ```

Name variables Name variables come in two flavors: single-field names and two-field names. Personal names are ordinarily of the two-field flavor. Fields are separated by *double* field-separator characters (`||`):

```
editor: Thompson || Hunter S.
```

Single-field names may represent institutions, and these may be composed of several sub-units separated by a *single* field-separator character (`|`):

```
author: Prince
author: National Weather Service|Office of International Affairs
```

> **Caution: Overrides (name variables)**
>
> Unlike the other variable types, entries for name variables are cumulative; however, name entries in "cheater syntax" will not override an existing creator of the same CSL category.

Item type The item type may also be set with either syntax:

```
type: dataset
```

Note, however, that when a type not recognized by the target style is set in this way, the item will be processed as an untyped item (sometimes described as a generic type such as "Document" in calling applications).

> **Caution: Overrides (item type)**
>
> An item type value set in "cheater syntax" always overrides the existing item type.

Ordinary variables Entries other than those above are assumed to be ordinary variables. these are set literally on the item, with no pre-processing.

> **Caution: Overrides (ordinary variables)**
>
> Ordinary variables set in "cheater syntax" will not override an existing value for the same variable.

Extended `sys` functions

Frank Bennett

Introduction

CSL-M: extensions to CSL

Frank Bennett

This supplement is a companion to the CSL 1.0.1 Specification. It is aimed at style authors, and documents differences between official CSL and the CSL-m schema recognised by Multilingual Zotero.

The changes are of two types:

- **Modifications** alter the behaviour or validation rules of existing CSL elements or attributes. Changes of this kind can cause official CSL repository to fail validation under the CSL-m schema: such styles will run, but may produce unexpected results. The potential effects of this category of changes are indicated where relevant.
- **Extensions** add entirely new elements or attributes to the schema. Styles making use of extended syntax will fail validation under the official CSL schema, and can be used only with Multilingual Zotero.

Table of Contents

        * `supplement` *(extension)*
        * `volume-title` *(extension)*
    - *Attributes*
        * `default-locale-sort` *(extension)*
        * `form="imperial"` *(modification)*
        * `form="short"` *(modification)*
        * `is-parallel` *(extension)*
        * `label-form` *(extension)*
        * `leading-noise-words` *(extension)*
        * `match="nand"` *(extension)*
        * `name-as-reverse-order` *(extension)*
        * `name-as-sort-order` *(extension)*
        * `name-never-short` *(extension)*
        * `oops` *(extension)*
        * `prefix` *and* `suffix` *(modification)*
        * `skip-words` *(extension)*
        * `subgroup-delimiter`, `subgroup-delimiter-precedes-last`, `and` *(extension)*
        * `suppress-min` *(extension)*
            · `suppress-min` *with a value of* `0`
        * `suppress-max` *(extension)*
        * `text-case` *(extension)*
        * `year-range-format` *(extension)*
    - *Conditions*
        * `context` *(extension)*
        * `genre` *(extension)*
        * `has-day` *(extension)*
        * `has-to-month-or-season` *(extension)*
        * `has-year-only` *(extension)*
        * `jurisdiction` *(extension)*
            · *Jurisdiction constraint list*
            · *Controlled list*
        * `locale` *(extension)*
        * `page` *(extension)*
        * `subjurisdictions` *(extension)*
    - *Terms*

        * *unpublished (extension)*
    - *Locator Terms*

Item Types

classic extension

Use the `classic` type for sources commonly cited within a field. Cites of this type do not appear in the bibliography, and can be completely reformatted to a compact, style-specific form using the **Classic** abbreviation list in the **Abbreviation Filter**.

When a short form is supplied for a **Classic** item, the `title` variable is suppressed, and the short form entirely replaces the `author` variable. When a volume number or other details are included, these can be rendered on either side of the composite abbreviated form, but not of course within it.

```
<choose>
  <if type="classic">
    <group delimiter=" ">
      <text variable="volume"/>
      <group delimiter=", ">
        <names variable="author"/>
        <text variable="title"/>
      </group>
  </if>
</choose>
```

gazette extension

Use the `gazette` type for instruments published through an official gazette. Typical use cases would be cites to amending acts, to the initial version of legislation, or to orders and other instruments that are not available from other official sources. For consolidated acts or codified statutes or regulations, use the `legislation` (MLZ **Statute**) or `regulation` (MLZ **Regulation**) types instead.

```
<choose>
  <if type="gazette">
     <text macro="gazette-mac"/>
  </if>
</choose>
```

The format of gazette citations may vary among jurisdictions. Test the `jurisdiction` variable (see below) to discriminate between citation forms.

hearing extension

The `hearing` type is primarily intended for transcripts of official hearings by government committees and the like (other documents produced by a committee should be cast as the `report` type instead). The body conducting a hearing is set in the `authority` variable (**Legis. Body** in MLZ).

```
<choose>
  <if type="hearing">
     <text macro="hearing-mac"/>
  </if>
</choose>
```

regulation extension

Use the regulation type for administrative orders at all levels of government.

```
<choose>
  <if type="regulation">
     <text macro="hearing-mac"/>
  </if>
</choose>
```

video extension

The video type is appropriate for video works that are not disseminated through an access-restricted distribution channel. For example, content hosted on YouTube should be set to video, while a DVD release of "The Wizard of Oz" should be set to motion_picture.

```
<choose>
  <if type="regulation">
     <text macro="hearing-mac"/>
  </if>
</choose>
```

Elements

cs:conditions extension

Condition statements in official CSL take a single "match" attribute, which determines how the tests will be combined. The match attribute value (all, any, none) applies to all tests within the statement: grouping of tests with separate match values is not possible. To simplify the coding of complex styles, CSL-m introduces an optional alternative syntax for condition statements.

The alternative syntax may be applied to cs:if or cs:else-if elements (the "parent node" in this description). The parent node must have no attributes, and a single cs:conditions node as its first child element. The cs:conditions node must have one or more cs:condition children. The cs:condition children each define a conditional statement with attributes specified in the CSL 1.0.1 schema and in this Supplement. The cs:condition statements are joined according to a mandatory "match" attribute on cs:conditions.

Note that CSL-m adds a "nand" match value (true if at least one of the tests or condition statements to which it applies returns false), in addition to "all", "any", and "none".

```
<choose>
  <if>
```

```
    <conditions match="any">
      <condition type="chapter"/>
      <condition variable="container-title collection-title" match="nand"/>
    </conditions>
    <text macro="some-chapter-mac"/>
  </if>
</choose>
```

cs:institution extension

Institutional names are fundamentally different in structure from personal names. CSL provides quite robust support for the presentation and sorting of personal names, but in CSL 1.0.1, institutional names have just one fixed form, and are otherwise treated the same as personal names in a list of creators.

Some publishing environments require greater flexibility. Institution names can consist of multiple subunits. Individuals may be credited together with the institution to which they belong. Unaffiliated personal authors may be cited together with an institution or with individuals affiliated with it. Some examples:

1. Research & Pub. Policy Dep't, Nat'l Urban League
2. United Nations - ECLAC
3. ECLAC (Economic Commission for Latin America and the Carribean)
4. Canadian Conservation Institute (CCI)
5. Nolan J. Malone and others, U.S. Bureau of the Census
6. World Trade Organization and World Health Organization
7. Smith with Jones, Bureau of Sloth, Ministry of Fear
8. Doe et al. with Roe et al., Ministry of Fear & Noakes, Ministry of Destruction

Examples 3 and 4 render both the full form and the acronym of a single institution name, with arbitrary ordering of the two name parts. Example 1 begins with the smallest subunit in a list of related institutions, and example 2 does the opposite. Examples 1 and 2 are pure organizations, while example 5 is a mix of personal and institutional names. Examples 1, 2, 3 and 4 would be entered as literal strings currently, which has obvious drawbacks. Example 5 would require that the authorship information be spread across two variables, although all parties listed are equally authors of the resource. Example 6 can be produced in CSL 0.8, but examples 7 and 8 cannot.

The MLZ extensions to CSL 1.0.1 provide a cs:institution element, which can be used to produce any of the above forms, without interfering with the formatting of ordinary personal names. The extension is always enabled in , but the application calling (i.e. Zotero) must specially flag institutional names for it to take effect. MLZ provides this flag, while the official Zotero client does not. For this reason, this extension only works with the multilingual client at present.

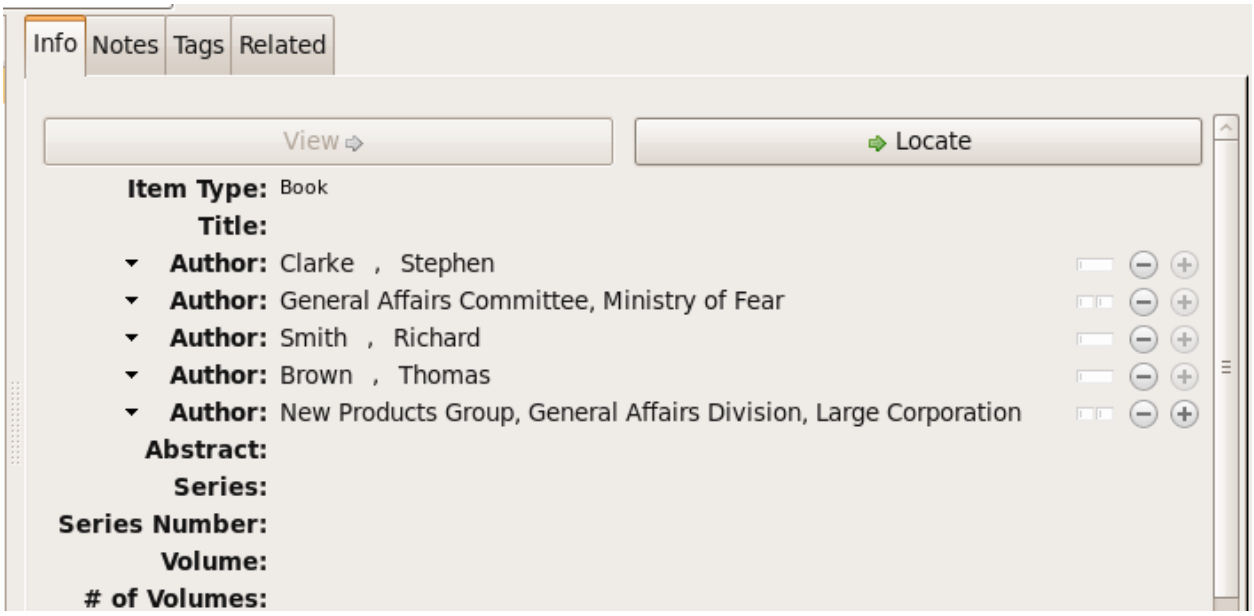
Entry conventions

In multilingual Zotero, names entered in two-field mode are personal, and those entered in single-field mode are treated as organizations. Names should be entered in the order in which they should appear in citations, with one (extremely rare) exception: when an unaffiliated author is included in a list of names that includes one or more institutions, the name of the unaffiliated author(s) should come *after* that of the last institution in the list.

Subunits of an organizational name should be separated with a field separator character `|`.

Affiliated authors

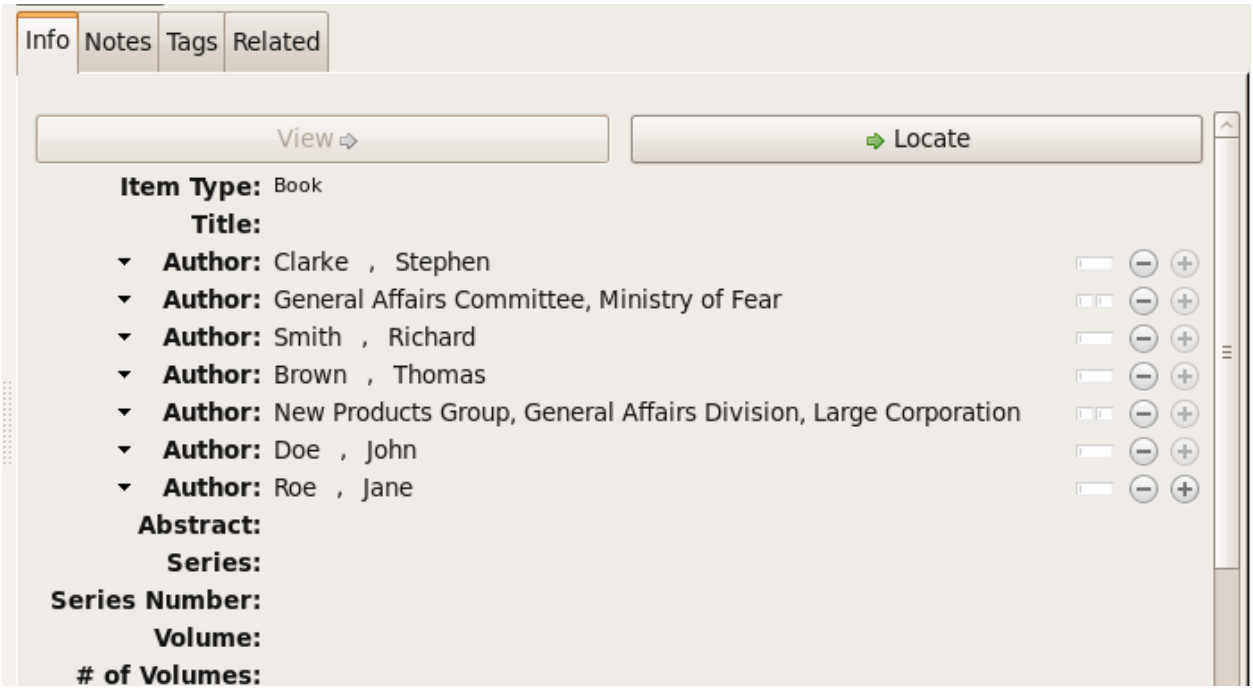
Single or multiple personal Names that are co-authors with an organization would be entered above the relevant organization name.



In a very simple style, the sample above might be rendered as: *Clarke, Ministry of Fear and Smith & Brown, Large Corporation*.

Unaffiliated authors

Authors with no affiliation would be listed after any organizational names:



In a very simple style, the sample above might be rendered as: *Doe & Roe with Clarke, Ministry of Fear and Smith & Brown, Large Corporation* (note the reverse ordering in this case, with the names at the end placed at the front of the rendered list of names).

The structure of mixed personal and organizational names can thus be expressed in the current Zotero UI. We now turn to the extended CSL syntax used to control the appearance of such names.

CSL Extension

Element: `cs:institution`

A `cs:institution` element can be placed immediately after the `cs:name` element to control the formatting of organization names.

Attributes: `delimiter` and `and`

The value of the `delimiter` attribute on `cs:institution` is used in the following locations:

- between organization names;
- between the subunits of an organization;
- between affiliated authors and their institution.

The `and` attribute on `cs:institution`, if any, is used for the final join between two or more author/organization units.

A simple use of `cs:institution` might read as follows:

```
<names variable="author">
  <name and="symbol" initialize-with=". "/>
  <institution and="text" delimiter=", ">
</names>
```

With this CSL, all of the delimiters in the following string would be drawn from attributes on `cs:institution`: *R. Smith, Small Committee, Large Corporation, G. Brown, Busy Group, Active Laboratory, and S. Noakes, Powerful Ministry*.

Attributes: `use-first`, `substitute-use-first` and `use-last`

To control the omission of names from the middle of the list of organizational subunits, either of `use-first` or `substitute-use-first` may be used to pick names from the front of the list. The `use-last` attribute picks names from the end. The `substitute-use-first` attribute includes the leading (smallest) subunit if and only if no personal names are associated with the organization.

The following CSL code would format both example 1 and example 5 from the list of samples at the top of this section:

```
<names variable="author" delimiter=", ">
    <name
      and="symbol"
      delimiter-precedes-last="never"
      et-al-min="3"
      et-al-use-first="1"/>
    <et-al term="and others"/>
    <institution
```

```
      delimiter=", "
      substitute-use-first="1"
      use-last="1"/>
</names>
```

Attribute: `reverse-order`

By convention, organizational names are rendered in "big endian" order, from the smallest to the largest organizational unit. To provide for cases such as example 2 in the list of samples, a `reverse-order` attribute can be applied on `cs:institution`:

```
<names variable="author" delimiter=", ">
    <name/>
    <institution
      delimiter=" - "
      use-first="1"
      use-last="1"
      reverse-order="true"/>
</names>
```

Attribute: `institution-parts`

The components of organization names are normally rendered in their long form only. When the Zotero Abbreviations Gadget is used with Zotero, abbreviated forms for these names may be available to the processor.

To use the short form, or combinations of the long and short form, an `institution-parts` attribute is available on `cs:institution`. The attribute accepts values of `long`, `short`, `short-long` and `long-short`. This attribute would be used to produce examples 3 and 4 in the list of samples, with values of `short-long` and `long-short` respectively. A value of `short` behaves in the same way as `form="short"` in other contexts in CSL, using the short form if it is available, and falling back to the long form otherwise.

Element: `cs:institution-part`

One or more cs:institution-part elements can be used to control the formatting of long and short forms of organization names. Like `cs:name-part`, these elements are unordered, and affect only the formatting of the target name element, specified (as on `cs:name-part`) with a required `name` attribute.

Attribute: `if-short`

In example 3, the parentheses should be included only if a short form of the institution name is available. The `if-short` attribute, available on `cs:institution-part` only when applied to the long form of an organization name, makes the formatting in the element conditional on the availability of a short form of the name. The following CSL would render example 3 in the list of samples:

```
<names variable="author">
    <name/>
    <institution institution-parts="short-long">
        <institution-part name="long" if-short="true" prefix=" (" suffix=")"/>
    </institution>
</names>
```

Element: cs:with

In rendered output, unaffiliated personal names are joined to a following organizational name using an implicit localizable term `with`. Styling of this term is permitted through an optional `cs:with` element, placed immediately above `cs:institution`:

```
<names variable="author">
    <name/>
    <with font-style="italic" prefix=" " suffix=" "/>
    <institution institution-parts="short-long">
        <institution-part name="long" if-short="true" prefix=" (" suffix=")"/>
    </institution>
</names>
```

Simple style example

The simple style used in the illustrated examples in the *Entry conventions* section above would look like this in CSL:

```
<names variable="author">
    <name form="short" and="symbol" delimiter=", "/>
    <institution use-last="1" and="text" delimiter=", "/>
</names>
```

cs:layout extension

In publishing outside of the English language domain, citation of foreign material in the style of the originating language is the norm. For example, a Japanese publication might include the following references in a single work:

- D. H. McQueen, "Patents and Swedish University Spin-off Companies: Patent Ownership and Economic Health", *Patent World*, March 1996, pp.22–27.
- No.465, 7 (2000)

To meet such requirements, the MLZ extensions to CSL permit multiple `cs:layout` elements within `cs:citation` and `cs:bibliography`. Each `cs:layout` element but the last must include a `locale` attribute specifying one or more recognized CSL locales, and the final element must not carry a `locale` attribute. The locale applied to an item is determined by matching it against the locale set in the `language` variable of the item (this value is passed by Zotero). An example:

```
<citation>
  <layout locale="en es de">
      <text macro="layout-citation-roman"/>
  </layout>
  <layout locale="ru">
      <text macro="layout-citation-cyrillic"/>
  </layout>
  <layout>
      <text macro="layout-citation-ja"/>
  </layout>
</citation>
```

In the example above, an item with `en`, `es` or `de` (or `de-AT`) set in the `language` variable will be render by the `layout-citation-roman` macro, with locale terms set to the appropriate language.

Variables

authority modification

In CSL-m, `authority` is handled as an institutional creator, not as an ordinary variable. It is rendered with a `cs:names` element.

```
<macro name="std-authority-child">
  <names variable="authority">
    <name suppress-min="0"/>
    <institution institution-parts="short" use-first="1"/>
  </names>
</macro>
```

The use of `suppress-min="0"` in this example is documented below under *suppress-min (extension)*.

available-date extension

The CSL-m `available-date` variable is appropriate for the date on which a `treaty` was made available for signing.

```
<group delimiter=" ">
  <text value="opened for signature" font-style="italic"/>
  <date date-parts="year-month-day" form="text" variable="available-date"/>
</group>
```

dummy extension

The `dummy` name variable is always empty. Use it to force *all* name variables called through a `cs:names` node to render through `cs:substitute`, and so suppress whichever is chosen for rendering to be suppressed through the remainder of the current cite.

```
<names variable="dummy">
  <name/>
  <label/>
  <substitute>
    <names variable="author"/>
    <names variable="editor"/>
  </substitute>
</names>
```

hereinafter extension

The `hereinafter` variable is a backreference form specific to a particular item and style. In MLZ, it can be set only through the Abbreviation Filter. The role of the variable in a given cite (i.e. whether it provides an alternative title, an acronym or a more complete formatted citation) depends on the the style and context.

```
<choose>
  <if match="all" type="bill gazette legislation" variable="hereinafter">
    <text variable="hereinafter"/>
  </if>
  <else>
    <text variable="title" form="short"/>
  </else>
</choose>
```

The Abbreviation Filter will offer an entry in the `hereinafter` list for every item cited in the document. It is not necessary to use `form="short"` on the `cs:text` node that renders the variable.

`locator` modification

In CSL-m, `locator` is a numeric variable. Validation requires that it be rendered with `cs:number`.

```
<number variable="locator"/>
```

The `locator` variable will render exactly once in a citation; subsequent attempts will render nothing. As with variables suppressed via `cs:substitute`, the test for `variable="locator"` will continue to return `true` after the variable is suppressed from further output.

`locator-date` and `locator-extra` extension

The variable "locator-date" is parsed out from the user-supplied locator, using the following syntax:

```
123|2010-12-01
```

In this example, "123" is the value of the `locator` variable (a page or other pinpoint string), the `|` character marks the end of the pinpoint, and the ten-character string immediately following is a full date. If supplied, dates must be given as shown above, zero-padded, in year-month-day order, and with no space between the date and the `|` character. Non-conforming strings following the `|` marker will be treated as a `locator-extra` variable.

The `locator-extra` variable consists of a string that is not a date, following the `locator-date` string (if any) as described above. If supplied without a `locator-date`, the `locator-extra` string must be preceded by a `|` field separator character. This variable can be used for version descriptions associated with some looseleaf services.

These extensions are useful with looseleaf services, because the dates of the content in these services varies depending on the page cited and the time at which the resource was referenced. These extensions permit a single item in the calling application's database to represent the volume on the library shelf, the page date being optionally supplied by the user when citing into a document.

`page` and `page-first` modification

The `page` and `page-first` variables are numeric in CSL-m. The validator requires that they be rendered with `cs:number`.

```
<number variable="page"/>
```

publication-date extension

CSL-m adds a `publication-date` variable to the language schema. It is available on the `gazette`, `legal_case`, `legislation`, `patent` and `regulation` types, and provides the date on which the instrument was published in the given reporter. On the `patent` type, it represents the date on which the patent was published for opposition (applicable only in certain jurisdictions).

```
<date date-parts="year-month-day" form="text" variable="publication-date"/>
```

publication-number extension

The `publication-number` variable is available on the `patent` type, and provides the number assigned to a patent published for opposition. It is a numeric variable, and validation requires that it be rendered with `cs:number`.

```
<number variable="publication-number"/>
```

supplement extension

The `supplement` variable and its associated locale term are useful for secondary sources that are regularly updated between fresh editions. Such fine-grained updates are found in secondary legal publications. Although a supplement may be identified by number or by name, `supplement` is a numeric variable, and validation requires that it be rendered with `cs:number`.

```
<choose>
  <if is-numeric="supplement">
    <group delimiter=" ">
      <label variable="supplement"/>
      <number variable="supplement"/>
    </group>
  </if>
  <else>
    <number variable="supplement"/>
  </else>
</choose>
```

volume-title extension

The `volume-title` variable is available on items of the `book` and `chapter` type. Use it to identify the name of a volume within a larger work known by an umbrella title (which may in turn be a part of a publisher's series, described by `collection-title`).

```
<text variable="volume-title"/>
```

Attributes

default-locale-sort extension

Use the default-locale-sort attribute on the cs:style node to specify the language collation to govern sorting behaviour. The sort locale has no effect on the language of standard terms and labels.

```
<style xmlns="http://purl.org/net/xbiblio/csl"
       class="note" version="1.1mlz1"
       default-locale="en-US"
       default-locale-sort="zh-TW">
```

If this attribute is *not* set, the sort locale is aligned with the default locale of the style or processor instance.

form="imperial" modification

CSL-m allows conversion of ordinary Gregorian dates to the Japanese form often used in legal citations and other government records. The conversion is only valid for dates after 1873. Prior dates on the traditional lunar calendar are not supported, and should be written as literal strings in the input.

```
<date variable="issued" form="imperial" date-parts="year-month-day"/>
```

form="short" modification

In CSL 1.0.1, rendering the title variable with the attribute form="short" produces the same result with any item type: if the title-short variable has a value, that it used; otherwise the title variable is rendered as a fallback.

In CSL-m, on the legal_case type only, the form="short" attribute does not attempt to render title-short, but instead renders the title variable, transformed by the Abbreviation Filter if an entry for it exists in the list there. This permits the application of style-specific abbreviation rules, as required by law-specific style guides such as *The Bluebook: A Uniform System of Citation*.

```
<text variable="title" form="short"/>
```

is-parallel extension

Set on a cs:group node, the is-parallel attribute includes or suppresses the content of the group node depending on whether it is rendered in a trailing parallel cite. Four values are recognised on the attribute:

false Renders when the cite is not one of a series of parallel cites.

true Renders when the cite is one of several cites in a parallel series.

master Renders when the cite is one of several in a parallel series, and is the first cite in the series.

servant Renders when the cite is one of several in a parallel series, and is *not* the first in the series.

```
<group delimiter=" ">
  <text font-style="italic" value="supra"/>
  <text value="note"/>
```

```
  <text variable="first-reference-note-number"/>
</group>
<group delimiter=" " is-parallel="false">
  <text value="at"/>
  <number variable="page-first"/>
</group>
<group delimiter=" " is-parallel="true">
  <number variable="volume"/>
  <text variable="container-title"/>
  <number variable="page-first"/>
</group>
```

label-form extension

The label-form attribute can be used on cs:number and on cs:text with the macro attribute. It accepts the same arguments as the form attribute for localised terms: long, verb, short, verb-short and symbol. Its effect is to override the form attribute applicable to terms called via the parent cs:text or cs:number node. This can be useful where macros are copied across styles that require different label forms.

```
<text macro="locator-mac" label-form="symbol"/>
```

leading-noise-words extension

When set on a cs:style-options node in a locale file or in a style, the leading-noise-words attribute takes a comma-delimited list of words as its argument.

When a list is set, the same attribute on a cs:text node with variable="title" takes an argument of ``demote or drop. With the demote attribute, noise words at the start of the field are rendered after the remainder of the title field, delimited by a comma. With the drop attribute the leading noise words are simply removed.

```
<style-options leading-noise-words="a,an,the"/>
```

and

```
<text variable="title" leading-noise-words="demote"/>
```

match="nand" extension

With the nand argument to the match attribute, a node test is true if at least one of the tests it invokes is false.

```
<choose>
  <if variable="volume issue" match="nand">
    <text macro="volume-issue-mac"/>
  </if>
</choose>
```

name-as-reverse-order extension

This is the counterpart of name-as-sort-order, available on cs:style-options only. Its operation is identical to its counterpart, but has the opposite effect of forcing reverse-order rendering of all names bearing the specified language tags. The name-as-sort-order attribute is described immediately below.

name-as-sort-order extension

A name-as-sort-order attribute is available on the cs:style-options locale element, taking a list of space-delimited country codes as its argument. Country codes in the argument should be limited to a single element (i.e. the language only, without a country or other specifier): other elements will be ignored.

When the first element of the field language (either explictly set, or defaulting to the item language field value, if any) matches one of the specified locales, the name is forced to "sort order", regardless of its character set.

```
<style-options name-as-sort-order="kr ja zh"/>
```

The example above will force names tagged as Korean, Japanese, and Chinese to sort order (i.e. family name first). This attribute does not interfere with short-form rendering or abbreviation (see the next heading below for that setting).

name-never-short extension

The name-never-short attribute on the cs:style-options locale element takes a list of space-delimited country codes as its argument. Country codes in the argument should be limited to a single element (i.e. the language only, without a country or other specifier): other elements will be ignored.

When the first element of the field language (either explictly set, or defaulting to the item language field value, if any) matches one of the specified locales, the effect of form="short" is suppressed when rendering the name concerned.

```
<style-options name-never-short="hu kr ja my vi zh"/>
```

The example above suppresses form="short" on names tagged as Hungarian, Korean, Japanese, Myanmar, Vietnamese, or Chinese.

oops extension

This attribute is deprecated. Use is-parallel instead.

prefix and suffix modification

Ordinary affixes in CSL-m are subject to a restriction: a prefix attribute may not begin with a space, and a suffix attribute may not end with a space. Affixes on cs:label within a cs:names element, and affixes within the scope of a cs:date element are not subject to this constraint.

The purpose of this requirement is to assure that styles are incapable of rendering cites with stray punctuation or multiple spaces. Where spaces are required between elements, they should be applied using a `delimiter` attribute value on a `cs:group` element.

```
<group delimiter=", ">
  <number variable="volume"/>
  <text variable="container-title"/>
</group>
```

skip-words extension

The processor carries a list of prepositions and other terms that will not be capitalised when rendering a field with `text-case="title"`. Within a locale, the `skip-words` attribute on `cs:style-options` can be used to replace this list of terms with another. The attribute value should be a comma-delimited list of words or phrases.

```
<style-options skip-list="a,an,the,or,and,over,under"/>
```

subgroup-delimiter, subgroup-delimiter-precedes-last, and extension

The `subgroup-delimiter` attribute is a field-parsing hack coded into the `citeproc-js` processor, enabled when the processor is run in CSL-m mode. In a group containing *only* `cs:text` elements rendering the `publisher` and `publisher-place` variables, the processor will split the content of both fields on a semicolon. If the length of the resulting list objects is equal, each `publisher`/`publisher-place` pair will be joined with the `delimiter` string set on the enclosing `cs:group` element. The composed pairs are then joined using the `subgroup-delimiter` value.

The `subgroup-delimiter-precedes-last` attribute controls the use of the delimiter between the last and the penultimate pair in the same manner as `delimiter-precedes-last` on a `cs:name` element. The `and` attribute with an argument of `text` or `symbol` may be used on the `cs:group` element to join the final item with the specified term.

```
<group delimiter=" " subgroup-delimiter=", "
       subgroup-delimiter-precedes-last="always">
  <text variable="publisher"/>
  <text variable="publisher-place"/>
</group>
```

or

```
<group delimiter=" " subgroup-delimiter=", "
       subgroup-delimiter-precedes-last="never"
       and="symbol">
  <text variable="publisher"/>
  <text variable="publisher-place"/>
</group>
```

suppress-min extension

In the MLZ extended schema, names can be suppressed in two ways. First, using `suppress-min` and `suppress-max` with values of `1` or above, names rendered via a `cs:name` element can be suppressed entirely

when the number of individual names is at or below a minimum, or at or above a maximum.

Second, with a value of `0`, `suppress-min` can be used on a `cs:name` *or* `cs:institution` element to suppress *only* names of that type. See the description of `suppress-min` below for an example of how that works and why it might sometimes be useful.

An example of `suppress-min` with a value of `4`:

```
<locale xml:lang="en">
  <terms>
    <term name="and others"></term>
  </terms>
</locale>
<macro name="first-position-author">
  <names variable="author">
    <name et-al-min="2" et-al-use-first="1"
          suppress-min="4"
          name-as-sort-order="first"/>
    <et-al term="and others"/>
  </names>
</macro>
<macro name="second-position-author">
  <names variable="author">
    <name et-al-min="4" et-al-use-first="1" delimiter=", "/>
  </names>
</macro>
<citation>
  <layout>
    <group delimiter=" / ">
      <group delimiter=" ">
        <text macro="first-position-author"/>
        <text variable="title"/>
      </group>
      <text macro="second-position-author"/>
    </group>
  </layout>
</citation>
```

In the above example, an item with two authors will render as follows:

> Stamou, A.I. Title of the Article / A.I. Stamou, I. Katsiris

An item with four authors, however, will render as follows:

> Title of the Article / A.I. Stamou et al.

`suppress-min` with a value of `0`

When set to zero, the `suppress-min` attribute is specific to the `cs:name` or `cs:institution` node only (for clarity, the attribute with this value should always be set directly on the affected node, rather than relying on inheritance). The effect of the setting is to suppress all institution or all personal names, leaving a list of the remaining names in place. This can be useful where personal and institutional authors must be listed in separate places in a citation—one example of such formatting being Rule 21.7.3 of the Bluebook 18th ed. (applicable to U.N. reports) which provides the following guidance and example:

> If a personal author is given along with the institutional author, the author [sic] should be included in a parenthetical at the end of the citation.

> U.N. Econ. & Soc. Council [ECOSOC], Sub-Comm. on Prevention of Discrimination & Prot. of Minorities, Working Group on Minorities, *Working Paper: Universal and Regional Mechanisms for Minority Protection*, ¶ 17, U.N. Doc. E/CN.4/Sub.2/AC.5/1999/WP.6 (May 5, 1999) (*prepared by* Vladimir Kartashkin).

suppress-max extension

```
<macro name="authors">
  <group delimiter=" ">
    <names variable="author">
      <name name-as-sort-order="all"
            et-al-min="11" et-al-use-first="3"
            and="text"/>
    </names>
    <group delimiter=" " prefix="(" suffix=")">
      <names variable="author">
        <name suppress-max="10" form="count"/>
      </names>
      <text value="co-authors"/>
    </group>
  </group>
</macro>
<citation>
  <layout>
    <text macro="authors"/>
  </layout>
</citation>
```

In this example, an item with four authors would render as follows:

> Doe, J, Roe, J, Noakes, R, and Snoakes, H

An item with eleven authors, on the other hand, would render like this:

> Doe, J, Roe, J, Noakes, R, et al. (11 co-authors)

text-case extension

CSL-m adds a `normal` argument to the possible values of `text-case`. This is mainly useful when rendering a `cs:number` element in the scope of an element that applies a `text-case` transform, to prevent text content in the rendered variable from being affected by the transform.

(The need for this attribute value is open to question.)

```
<number variable="number" text-case="normal"/>
```

year-range-format extension

CSL-m offers a `year-range-format` attribute on `cs:style`, as a complement to `page-range-format`. It takes the same arguments: `expanded`, `minimal`, `minimal-two`, and `chicago`. When used, this attribute specifies a collapsing format for year ranges separate from that applied to page ranges.

```
<style xmlns="http://purl.org/net/xbiblio/csl" class="note" default-locale="en-GB"
       page-range-format="chicago"  year-range-format="expanded"
       version="1.1mlz1">
```

Conditions

context extension

The `context` test attribute takes exactly one of the arguments `bibliography` or `citation`. It does exactly what its name and value suggest, returning true when the condition is executed in the relevant context. This test is useful where complex logic is needed to compose a particular element, which changes only slightly in the other context.

```
<choose>
  <if context="citation">
    <text variable="title" font-variant="small-caps"/>
  </if>
  <else>
    <text variable="title" font-variant="small-caps" form="short"/>
  </else>
</choose>
```

genre extension

The `genre` test attribute takes one of five arguments: `email`, `instant-message`, `podcast`, `radio-broadcast`, or `television-broadcast`. These correspond to strings automatically inserted into the `genre` variable by MLZ (where no value is set manually by the user) on the **Email**, **Instant Message**, **Podcast**, **Radio Broadcast** and **Television Broadcast** types respectively.

This is obviously a hack, and mimicks the effect of having five separate types for these items in CSL, as opposed to two (i.e. *personal_communication* and *broadcast*).

```
<choose>
  <if genre="podcast">
    <text macro="podcast-mac"/>
  </if>
</choose>
```

has-day extension

The `has-day` condition attribute tests whether the date variable given as argument has a value that includes a day.

```
<choose>
  <if has-day="issued">
    <date variable="issued">
      <date-part name="month" form="text"/>
      <date-part name="day" form="numeric" prefix=" "/>
    </date>
  </if>
</choose>
```

has-to-month-or-season extension

The `has-to-month-or-season` condition attribute tests whether the date variable given as argument has a month or season (and no day).

```
<choose>
  <if has-to-month-or-season="issued">
    <date variable="issued">
      <date-part name="month"/>
    </date>
  </if>
</choose>
```

has-year-only extension

The `has-year-only` condition attribute test whether the date variable given as argument has only a year (and no day, month or season).

```
<choose>
  <if has-year-only="issued">
    <date variable="issued"
          date-parts="year" form="text"
          prefix="[" suffix="]"/>
  </if>
</choose>
```

jurisdiction extension

When citing primary legal resources, the form of citation is often fixed, for ease of reference, by the issuing jurisdiction— "jurisdiction" referring in this case to international rule-making bodies as well as national governments. CSL 1.0.1 provides a `jurisdiction` variable, but it cannot be used because Zotero does not currently have a corresponding field.

The particular requirement for this variable is that it be tested in a `cs:if` and `cs:else-if` condition, so that citations can be varied according to the issuing jurisdiction. Testing of field content is contrary to the design of CSL, so the approach of the MLZ extended CSL schema is strictly circumscribed to address this particular need, without opening a door to uncontrolled general testing of field content.

The solution is in two parts, described below.

Jurisdiction constraint list

The CSL schema has been extended in accordance with the proposed URN:LEX standard for a uniform resource namespace for sources of law. This standard provides a concept of "jurisdiction" that suits the requirements of legal citation, including both national jurisdictions and international rule-making bodies. Following URN:LEX, the schema has been extended with an explicit list of the national jurisdictions of the world, plus selected rule-making international organizations designated by their permanent domain name. The former are drawn from ISO 3166 Alpha-2. The latter

do not yet have official sanction, as URN:LEX is still at the proposal stage, but the list in the schema extension is conservative, including only a few of the most stable (and widely cited) organizations.

Controlled list

The list of acceptable jurisdictions codes is coupled with an extension of the `cs:if` and `cs:else-if` elements, providing a `jurisdiction` test attribute. In styles, the value set on the attribute *must* be present in the list of acceptable jurisdiction values. A style that uses other values is invalid.

When the `jurisdiction` test attribute is used, its value is compared with the value of the `jurisdiction` variable on the item being processed. If the values match, the test returns true, otherwise false. Matching is done at the granularity of the argument provided in the test.

```
<choose>
  <if jurisdiction="us">
    <text macro="us-mac"/>
  </if>
</choose>
```

The test above will be true for items with a `jurisdiction` value of `jp` or `de`, and false for values of `us`, `us;federal;oh` or `us;ny`.

locale extension

In CSL-m, alternative `cs:layout` nodes can be set by giving each a separate `locale` attribute. See the description of *cs:layout (extension)* above for further details.

page extension

In CSL-m, the `page` variable can be tested in the same way as `locator`. The test evaluates the label value, if any, set at the start of the field using a recognized locator abbreviation. See *Locator Terms* below for a list of recognised abbreviations and their corresponding CSL-m label values.

```
<choose>
  <if page="page" match="none">
    <label form="symbol" variable="page"/>
  </if>
</choose>
```

The `cs:label` element in this example will render the localised term for the label set in the `page` field (e.g. "`para.3`" will render in English as "¶ 3").

subjurisdictions extension

The CSL-m `jurisdiction` variable contains a juridiction specifier divided into subfields, with a colon as the subfield delimiter.

The `subjurisdictions` test attribute takes an integer as argument. It returns true if the `jurisdiction` field on the item contains a value with at least the specified number of subjurisdictions.

```
<choose>
  <if subjurisdictions="2">
    <text variable="jurisdiction" form="short"/>
  </if>
</choose>
```

The example above will render the value of the `jurisdiction` field, "abbreviated" according the any mapping set for the style in the Abbreviation Filter. In the example, a value of `us` or `us;ca` will return false, while a value of `us;federal;ny` will return true.

Terms

unpublished 

The `unpublished` localised term is available for use by CSL-m styles.

```
<text term="unpublished"/>
```

Locator Terms

Label	Abbreviation	Label	Abbreviation
article	art.	paragraph	para.
book	bk.	subparagraph	subpara.
chapter	ch.	part	pt.
subchapter	subch.	rule	r.
column	col.	section	sec.
figure	fig.	subsection	subsec.
folio	fol.	sub-verbo	sv.
line	l.	schedule	sch.
note	n.	title	tit.
issue	no.	verse	vrs.
opus	op.	volume	vol.
page	p. or pp.		

In addition to the extended terms (`article`, `rule`, and `title`), CSL-m styles automatically set the special alternative terms `Chapter` and `Section` when rendering the corresponding terms in the context of the first item in a parallel reference.