Arxana spine

Joe Corneli

November 12, 2025

Contents

This spine collects the Arxana material into fourteen Parts, grouping related modules so the whole is easier to navigate. The grouping is non-destructive: filenames and internal structure are unchanged and can be further refined.

```
    ;; Quickstart
(require 'org)
(require 'ob-tangle)
(require 'cl-lib)

(add-to-list 'org-src-lang-modes '("elisp" . emacs-lisp))
(add-to-list 'org-babel-load-languages '(emacs-lisp . t))
(org-babel-do-load-languages 'org-babel-load-languages org-babel-load-languages)

;; Treat 'elisp' blocks as Emacs Lisp when tangling.
(add-to-list 'org-babel-tangle-lang-exts '("elisp" . "el"))

;; Default: tangle elisp blocks unless :tangle no
(setq org-babel-default-header-args:elisp
      (org-babel-merge-params
       org-babel-default-header-args:elisp
       '((:tangle . "yes"))))

(defun arxana-spine-files (spine-file)
  "Return the list of files included by SPINE-FILE via #+INCLUDE."
  (with-temp-buffer
    (insert-file-contents spine-file)
    (let (files)
      (goto-char (point-min))
      (while (re-search-forward "^#\\+INCLUDE: \"\\([^\"]+\\)\"" nil t)
        (push (expand-file-name (match-string 1)
                                (file-name-directory spine-file))
              files))
      (nreverse files))))

(defun arxana-tangle-spine-concat (spine-file)
  "Tangle Org files listed in SPINE-FILE, concat their tangled .el into
arxana-tangled.el (in order), then load it.

Respects existing :tangle / :tangle no semantics.
Only deletes .el files that were created by this function."
  (interactive "fSpine file: ")
  (let* ((spine-file (expand-file-name spine-file))
         (base-dir  (file-name-directory spine-file))
```

```
     (org-files (arxana-spine-files spine-file))
     ;; Snapshot existing .el files so we know what is new later.
     (before-el (directory-files-recursively base-dir "\\.el$"))
     (tangled-files '()))

;; 1. Tangle each Org file, record resulting .el files.
(dolist (org-file org-files)
  (dolist (f (org-babel-tangle-file org-file))
    (let* ((abs (if (file-name-absolute-p f)
                    f
                  (expand-file-name f (file-name-directory org-file)))))
      (push abs tangled-files))))

;; Preserve order of first occurrence.
(setq tangled-files
      (cl-remove-duplicates (nreverse tangled-files)
                            :test #'string=))

;; 2. Build arxana-tangled.el from all tangled files.
(let ((out (expand-file-name "arxana-tangled.el" base-dir)))
  (with-temp-file out
    (insert ";; Autogenerated from Arxana spine\n\n")
    (dolist (f tangled-files)
      (when (file-exists-p f)
        (insert ";; from " (file-relative-name f base-dir) "\n")
        ;; IMPORTANT: read from the file's own temp buffer,
        ;; not from the output buffer.
        (insert
         (with-temp-buffer
           (insert-file-contents f)
           (buffer-substring-no-properties (point-min) (point-max))))
        (insert "\n\n"))))

  ;; 3. Delete only .el files that are new (created by this run),
  ;; never the combined arxana-tangled.el itself.
  (let* ((after-el (directory-files-recursively base-dir "\\.el$"))
         (new-el (cl-set-difference after-el before-el :test #'string=)))
    (dolist (f new-el)
      (unless (string=
               (file-truename f)
               (file-truename out))
        (delete-file f))))

  ;; 4. Load the unified file.
  (load-file out)))
```

Contents

1 Part I — Reintroduction and Preface

1.1 Logical model

Arxana manages a single logical hypergraph. The core primitives are:

Nema Universal node record with stable id, labels, optional endpoints, and payload.

Article A nema labeled :label/article, holding text and core metadata.

Metadata scholium A nema paired 1–1 with an article, caching backlinks, labels, and indexes.

Event (hyperedge) A nema representing a relation or annotation, with :hx/type and N endpoints.

Plexus A nema that names a working set or profile and carries configuration.

Classic Arxana entities, relations, and scholia embed cleanly:

- Entities → article nemas with derived identifiers.
- Relations → 2-end events (:hx/*) between nemas.
- Scholia → events and/or metadata nemas attached to articles.
- Inclusion / clusion / provenance → multi-end events with appropriate :role values.

1.1.1 Storage substrates

All storage backends implement the same logical model.

1. In-memory storage of one form or another
   - Historically, nemas, articles, events, and plexuses were held in hash tables / HONEY-style networks.
   - This is the reference representation; other substrates hydrate/dehydrate it.
2. Document-based substrates
   - Literate sources (e.g., LaTeX, Org, Markdown, TEI) could be parsed into nemas and events.
   - Arxana can — in principle and, increasingly, in practice — both read **and write** these formats, keeping ids and links stable across round-trips.
3. Historical: Relational substrate
   - Common Lisp provided a bridge to SQL that encode relations in tables.
   - Semantics are those of a structured triple store: subjects, predicates, objects, plus spans and provenance.
   - Suitable for durability, multi-user access, and heavy queries.
4. Possible future: Triple-store substrate
   - Direct RDF/quad representation of the same model — not implemented but it could be if we want it.
   - Again, we could in principle read and write external knowledge graphs (and integrate SPARQL tooling).

5. Current reality: EAV in Clojure (Datascript + XTDB)

   Instead of going the triple route, all storage backends in the current 'beta' generation converge on an open **entity–attribute–value (EAV)** model. Each record—whether an article, event, or plexus—is represented as a set of attribute–value pairs attached to a unique entity id. This design keeps the schema **open**, so new attributes or link types can appear dynamically without migration.

   The canonical implementation uses:

   - **Datascript** for in-memory, client-side operation.
     - Pure Clojure; lightweight and immutable.
     - Supports live queries, undo/redo, and fast incremental updates.
     - Ideal for interactive use inside Emacs or JVM clients.
   - **XTDB** for durable, time-traveling storage.
     - Schemaless documents mirror the same EAV keys verbatim.
     - Transactions are append-only; every version of the graph is queryable.
     - Perfect for provenance, journaling, and long-term archives.

   Together these give Arxana an **open-schema substrate**: a graph database where nemas, events, and plexuses are all entities, and attributes such as ':hx/type', ':article/text', or ':plexus/members' are just first-class keys. Queries can run over the generic EAV view, or use higher-level helpers that expose structured projections (e.g., "articles", "hyperedges", "plexuses").

   > **Design posture:** Start open (EAV); specialize only when scale or semantics demand it. > Structured views—relational tables, triple stores, materialized indexes—can be layered on > demand without altering the underlying data model.

   This approach keeps computational properties stable: query complexity and transaction semantics stay identical whether data are viewed as triples, hash maps, or SQL rows. It also makes cross-substrate replication trivial: each backend stores the same '(entity attribute value)' facts, differing only in query language and persistence model.

1.1.2 Emacs client (Arxana classic)

The Emacs Lisp implementation in this repository provides:

- Interactive creation and editing of articles and scholia.
- Browsing modes over the in-memory graph.
- Operations for inclusion, derivation, and provenance as events.
- Import/export for specific document formats (historically LaTeX/Org).

Internally, it maintains the logical model in memory and (optionally) syncs with external substrates (files, SQL backend).

2 Part II — System Introduction

2.1 Elisp requirements

2.1.1 Packages

I'm not sure we actually need align any more, but cl is of course very helpful.

```
(require 'align)
(require 'cl)
(load "cl-seq")
(load "cl-extra")
```

2.2 Lisp preliminaries

2.2.1 On 'add-or-replace'

It seems that this function is no longer used.

```
(defun add-or-replace (lis elt pred)
  (let ((found nil)
        (n 0)
        (len (length lis)))
    (while (and (not found)
                (< n len))
      (when (funcall pred (nth n lis))
        (setcar (nthcdr n lis) elt)
        (setq found t))
      (setq n (1+ n)))
    (if found
        (1- n)
      (nconc lis (list elt))
      n)))
```

2.2.2 On 'add-to-or-start-list'

This is is used to start store an element on a "possible-list"; unlike 'add-to-list', it works even when list is nil, or even (for now) when possible-list is not a list at all. (Unlike with 'add-to-list', possible-list should not be the name of a list.)

This function is used by 'put-type-labels', 'put-backlinks' and 'label-article'. Note that you have to *save* the result if you want to this function to have a lasting effect.

Like 'add-to-list', this function will only store *one* copy of element.

```
(defun add-to-or-start-list (possible-list element)
  (cond ((null possible-list)
         (list element))
        ((listp possible-list)
         (if (member element possible-list)
             possible-list
           (append possible-list (list element))))
        (t (list possible-list element))))
```

2.2.3 On 'next-single-property-change+predicate'

Like 'next-single-property-change' except instead of automatically judging changes in the text property by the 'eq' predicate, it allows the user to supply a test.

```
(defun next-single-property-change+predicate (pos prop &optional test)
  (let ((starting (get-text-property pos prop))
        (index pos)
```

```
        (cmpfct (or test
                    'eq)))
    (save-excursion
      (while (and (setq index (1+ index))
                  (< index (point-max))
                  (funcall cmpfct
                           starting
                           (get-text-property index prop)))))
    (when (< index (point-max))
      index)))
```

2.2.4 On 'delete-all-dups'

This function differs from 'delete-dups' in being non-destructive and in keeping none of several 'equal' occurrences of an element in list as opposed to one; otherwise, it manifests a similar idea.

```
(defun delete-all-dups (list)
  (let ((tail list)
        ret)
    (while tail
      (cond ((member (car tail) (cdr tail))
             (setq tail (delete (car tail) tail)))
            (t
             (setq ret (cons (car tail) ret)
                   tail (cdr tail)))))
    ret))

(defun set-difference (A B)
  (delete-all-dups (append A B)))

(defun zip (A B)
  (let ((ret (make-hash-table)))
    (while A
      (puthash (car A)
               (car B)
               ret)
      (setq A (cdr A)
            B (cdr B)))
    ret))

(defun flatten (list-structure)
  (apply 'concatenate 'list (list list-structure)))
```

3 Part III — Scholia-Based Documents

3.1 The digital library

3.1.1 On 'article-table'

We begin with an empty library.

```
(defvar article-table
  (make-hash-table :test 'equal))
```

3.1.2 On 'put-article'

This destructively adjusts the values on the hash table (see documentation for 'puthash'). To *intelligently manage* the values in the table takes more work, so we typically 'get-article' before to make various comparions and adjustments to existing values before making changes.

```
(defun put-article (name value)
  (puthash name value article-table))
```

3.1.3 On 'get-article'

Basically just 'gethash', but since the article table is indexed by name and we don't want to pass name around all the time, we add it to the recalled value.

```
(defun get-article (name)
  (let ((content (gethash name article-table)))
    (when content
      (cons name content))))
```

3.1.4 On 'name-of-current-article'

The "current article" is of special importance for display (an article is made "current" when 'display-article' displays it). The name of this article is stored in this variable.

```
(defvar name-of-current-article nil)

(defun current-article ()
  (get-article name-of-current-article))
```

3.1.5 Futon1 synchronization primitives

Arxana can treat Futon1 as the backing graph service. The helpers below provide lightweight HTTP glue that other commands can call when they want to mirror in-memory operations to Futon1's API.

```
(require 'json)
(require 'url)

(defgroup futon4 nil
  "Arxana <-> Futon1 integration."
  :group 'tools)

(defcustom futon4-base-url "http://localhost:8080/api/alpha"
  "Base URL for Futon1 API (ASCII alias for /api/ )."
  :type 'string)

(defcustom futon4-enable-sync t
  "When non-nil, mirror article and relation edits to Futon1."
  :type 'boolean)
```

```
(defcustom futon4-log-requests nil
  "When non-nil, log each Futon1 POST and its JSON payload."
  :type 'boolean)

(defun futon4--post-json (path payload &optional cb)
  "POST PAYLOAD as JSON to Futon1 PATH and run CB on the decoded body."
  (when futon4-enable-sync
    (let* ((json-body (json-encode payload))
           (url-request-method "POST")
           (url-request-extra-headers '(("Content-Type" . "application/json")))
           (url-request-data (encode-coding-string json-body 'utf-8)))
      (when futon4-log-requests
        (message "futon4 POST %s %s" path json-body))
      (condition-case err
          (url-retrieve
           (concat futon4-base-url path)
           (lambda (_status)
            (goto-char (point-min))
            (when (re-search-forward "\n\n" nil t)
               (let* ((json-object-type 'alist)
                      (json-array-type 'list)
                      (json-key-type 'keyword)
                      (body (ignore-errors (json-read))))
                 (when cb (funcall cb body))))
             (let ((buf (current-buffer)))
               (when (buffer-live-p buf)
                 (kill-buffer buf)))))
        (error
         (message "futon4 request to %s failed: %S" path err))))))

(defun futon4--canonical-path (path)
  "Return a canonicalized representation of PATH, or nil."
  (when path
    (convert-standard-filename (expand-file-name path))))

(defun futon4--article-id (path)
  "Build a stable Futon1 article id from PATH or BUFFER name."
  (when path
    (concat "arxana:article:" (format "%s" path))))

(defvar futon4--article-id-cache (make-hash-table :test 'equal)
  "Cache mapping scholium names to Futon1 ids.")

(defun futon4-register-article (name id)
  "Remember that NAME corresponds to Futon1 article ID."
  (when (and name id)
    (puthash name id futon4--article-id-cache)
    id))

(defun futon4-lookup-article-id (name)
```

```
  "Return Futon1 article id cached for NAME, if any."
  (and name (gethash name futon4--article-id-cache)))

(defun futon4--article-id-for (name &optional path)
  "Return or derive the Futon1 id for NAME, optionally using PATH."
  (or (futon4-lookup-article-id name)
      (let ((derived (futon4--article-id (or path name))))
        (when derived
          (futon4-register-article name derived)))))

(defun futon4--link-label (link)
  "Return a readable label string for LINK's metadata."
  (let ((meta (cdr-safe link)))
    (if meta
        (mapconcat (lambda (piece)
                     (cond
                      ((stringp piece) piece)
                      ((symbolp piece) (symbol-name piece))
                      (t (format "%S" piece))))
                   meta
                   " | ")
      "")))

(defun futon4--sync-about-links (source-name about)
  "Mirror ABOUT links for SOURCE-NAME into Futon1."
  (when (and futon4-enable-sync about)
    (let ((src-id (futon4-lookup-article-id source-name)))
      (when src-id
        (dolist (link about)
          (let* ((target (car-safe link))
                 (dst-id (and target
                              (futon4--article-id-for target))))
            (when dst-id
              (futon4-store-nema-simple src-id dst-id
                                        (futon4--link-label link)))))))))

(defun futon4--article-context-for-buffer (buffer)
  "Return plist with :id, :name, and :path for BUFFER."
  (let* ((path (or (futon4--canonical-path (buffer-file-name buffer))
                   (buffer-name buffer)))
         (id (futon4--article-id path))
         (name (buffer-name buffer)))
    (list :id id :name name :path path)))

(defun futon4-ensure-article-entity (id name path &optional spine-p cb)
  "Ensure Futon1 has an article entity with ID, NAME, and PATH.
SPINE-P marks the article as belonging to the current spine when non-nil.
CB is an optional callback that receives the decoded response body."
  (when id
    (let* ((props-alist (delq nil (list (when path
```

```
                                        (cons 'path path))
                                      (when spine-p
                                        (cons 'spine t)))))
          (payload (delq nil (list (cons 'id id)
                                   (cons 'name (if (symbolp name)
                                                   (symbol-name name)
                                                 name))
                                   (cons 'type "arxana/article")
                                   (when props-alist
                                     (cons 'props props-alist))))))
     (futon4--post-json "/entity" payload cb))))

(defun futon4-store-nema-simple (src-id dst-id &optional label cb)
  "Store a simple scholium-style relation between SRC-ID and DST-ID."
  (when (and src-id dst-id)
    (futon4--post-json
     "/relation"
     `((type . "arxana/scholium")
       (src . ,src-id)
       (dst . ,dst-id)
       (props . ((label . ,(or label "")))))
     cb)))
```

3.2 Creating scholia

3.2.1 Links

Each link is associated with precisely one article. Links can have additional link-type information stored about them. The general format of a link is:

```
(<target article's name> &rest <link-types>)
```

The design here maintains partial symmetry between the treatment of article types and link types; the cdr of a link can be processed by the same typedata processing functions as the type data from articles.

For example, the link

```
'(foo (passage 10 15) mistaken)
```

indicates that the region from character position 10 to 15 in the article named "foo" is "mistaken", whereas the link

```
'(foo mistaken)
```

indicates simply that the article *foo* is mistaken.

3.2.2 Format of 'bookkeeping' field

The bookkeeping field has a special format, very similar to the format of metadata articles

one might expect to see something like

```
((owner <owner>)
 (ACL <ACL>)
 ...)
```

Remember that the bookkeeping is *owned*, so we probably shouldn't put anything there that can't be edited by the user, and so in particular, things that are needed for system consistency should go into metadata articles instead.

Do notice that various schemes for access and ownership can be considered. (E.g. maybe anyone on the *access control list* (ACL) can edit the text of the article, but only the owner is allowed to edit the ACL itself.)

It would be worth looking at existing strategies (e.g. as found on PlanetMath) for handling complicated ownership and access arrangements. Actual implementation of useful bookkeeping features will have to come later (for a sort of silly placeholder for the meantime, see Note 6.2.1).

3.2.3 Access functions

A few simple functions to get pieces of (just aliases to help with coding; see Note `2.2-Design_issues.org`). Note 3.2.4 talks about something similar for links.

```
(defalias 'scholium-name 'first)
(defalias 'scholium-text 'second)
(defalias 'scholium-about 'third)
(defalias 'scholium-type 'fourth)
(defalias 'scholium-bookkeeping 'fifth)
```

3.2.4 Link accessors

Links have exactly one linked-to-article together with any number of different link-types.

```
(defalias 'linked-to-article 'first)
(defalias 'link-type 'cdr)
```

3.2.5 Special link accessors

First we need a function that will give us the type-element associated with a given type. Then, we have some important examples.

The function 'link-type-accessor' will produce the (first) specific type-element from link that matches the given type, if one exists.

Note that the format of *version* access should follow the 'link-version' function given here. (Passages don't have versions, so it doesn't make sense to put a version number as an additional element of a "passage" element.)

```
(defun link-type-accessor (link type)
  (car (member-if (lambda (type-elt)
                    (or (eq type-elt type)
                        (and (listp type-elt)
                             (eq (car type-elt) type))))
                  (cdr link))))

(defun link-beginning (link)
  (second (link-type-accessor link 'passage)))

(defun link-end (link)
  (third (link-type-accessor link 'passage)))

(defun link-version (link)
  (second (link-type-accessor link 'version)))
```

3.2.6 Hooks for new and modified scholia

If the scholium is new, run 'new-scholium-hook', otherwise, run individual hooks depending on which pieces of the scholium were modified (if any). We use separate hooks in part because we need to be careful about what sorts of changes we make. If we adjusted the value of some scholium every time the value of a scholium was adjusted, we would expect to encounter infinite loops pretty quickly!

The standard settings for these hooks will be given in the section 3.4; these settings facilitate e.g. the automatic creation of backlinks (Note).

```
(defvar new-scholium-hook nil)
(defvar scholium-modified-text-hook nil)
(defvar scholium-modified-about-hook nil)
(defvar scholium-modified-type-hook nil)
(defvar scholium-modified-book-hook nil)
```

3.2.7 The 'scholium' function

This function is foundational for the system. Lots of other functions will use it, but because it is destructive, users should typically not access it directly. Several interactive interfaces to this function appear in Section . Note that the conditional structure here shouldn't be a 'cond'; we really mean to run several hooks if several parts of the scholium have been modified.

```
(defun scholium (name text &optional about type book)
  (let* ((old-version (get-article name))
         (old-about (and old-version (scholium-about old-version)))
         (maybe-path (and (stringp text)
                          (fboundp 'typedata-includes)
                          (typedata-includes type 'file)
                          (fboundp 'futon4--canonical-path)
                          (futon4--canonical-path text)))
         (article-id (and (fboundp 'futon4--article-id-for)
                          (futon4--article-id-for name maybe-path))))
    (put-article name (list text about type book))
    (when (and article-id
               (not (bufferp text))
               (or (not (fboundp 'typedata-includes))
                   (not (typedata-includes type 'file)))
               (fboundp 'futon4-ensure-article-entity))
      (futon4-ensure-article-entity article-id name maybe-path))
    (let ((meta-override (metadata-override)))
      (when (and (not meta-override)
                 (fboundp 'futon4--sync-about-links)
                 (not (equal old-about about)))
        (futon4--sync-about-links name about))
      (when (not meta-override)
        (if (not old-version)
            (run-hooks 'new-scholium-hook)
          (when (not (equal (scholium-text old-version) text))
            (run-hooks 'scholium-modified-text-hook))
          (when (not (equal (scholium-about old-version) about))
            (run-hooks 'scholium-modified-about-hook))
```

```
        (when (not (equal (scholium-type old-version) type))
          (run-hooks 'scholium-modified-type-hook))
        (when (not (equal (scholium-bookkeeping old-version) book))
          (run-hooks 'scholium-modified-book-hook)))))))
```

3.2.8 On 'metadata-override'

We don't want to run the usual hooks when the article being processed is a metadata article or a "fake" article. Fake articles appear when we create fake references that they don't show up in the scholia display and that aren't permanently attached to anything. See Note 7.10.1.

(At least at one point I had some misgivings about this override, but it seems to get the job done. I don't see why we would want to have weird automatic thing happen after creating a metadata article or a "fake" article, so it may be that the design here is perfectly fine.)

```
(defun metadata-override ()
  (when (or (typedata-includes type 'meta)
            (typedata-includes type 'fake))
    t))
```

3.2.9 Simple scholium derivatives

3.2.10 On 'article-names-from-about-data'

The idea here is to build a list with just the names of the articles that the about data features. Names are uniquified.

```
(defun article-names-from-about-data (about)
  (delete-dups (mapcar (lambda (elt) (car elt)) about)))
```

3.2.11 On 'typedata-includes'

Read typedata in the format used in scholia, and say whether it matches a specific-type. Notice that when typedata and specific-type are both is 'nil', we return a positive match.

```
(defun typedata-includes (typedata specific-type)
  (cond ((eq typedata specific-type) t)
        ((and (listp typedata)
              (member specific-type typedata)) t)
        (t nil)))
```

3.2.12 On 'typedata-includes-element-of-list'

Like 'typedata-includes' (Note 3.2.11), but only requires one type out of those listed to match in order to return 't'.

```
(defun typedata-includes-element-of-list (typedata list-of-types)
  (let (ret)
    (while (and list-of-types
                (not ret))
      (when (typedata-includes typedata (car list-of-types))
        (setq ret t))
      (setq list-of-types (cdr list-of-types)))
    ret))
```

3.2.13 On 'typedata-includes-passage'

A specific test to see if typedata includes type "passage" (noting that the list element that indicates this type should come together with some suffix that tells you which passage is being indicated).

As far as I can tell, this function will always apply to typedata coming from links (but who knows).

```
(defun typedata-includes-passage (typedata)
  (let (ret)
    (when (listp typedata)
      (while (and typedata
                  (not ret))
        (when (eq (car (car typedata)) 'passage)
          (setq ret (car typedata)))))
    ret))
```

3.2.14 Examples

3.2.15 Examples of the 'scholium' function in use

As an example of how the 'scholium' function might be applied, here is a scholium about the article I'm writing now.

```
(scholium "Remark On Scholium Definition"
          "This is an Emacs Lisp function."
          '(((passage "sbdm4cbpp.tex" 49078 49738))))
```

Actually, the article that I'm writing right now won't be in the article list unless we explicitly put it there. So the scholium that we added here is actually attached to a *fictitious* article with the name "sbdm4cbpp.tex". That's OK, but for completeness, I'll add the current buffer as a scholium with the appropriate name:

```
(scholium "sbdm4cbpp.tex"
          (current-buffer))
```

Notice that we've supplied a buffer instead of a string for the text field. We now add another scholium, attached to the previous two, communicating to the user a fact that the system already knows:

```
(scholium "This is a scholium about part of the SBDM document"
          "This scholium is attached to this region."
          '(((article "Remark On Scholium Definition"))
            ((passage "sbdm4cbpp.tex" 31091 31744))))
```

By setting the type data appropriately, we might make this last example display when "Remark On Scholium Definition" is displayed, but not when "sbdm4cbpp.tex" is displayed. It might make sense to mark up the words "this region" with a reference to "sbdm4cbpp.tex". Tricky things like this will be coming in later sections.

3.3 Metadata

3.3.1 On 'metadata-article-name-p'

So, like I said, we're losing some generality in the set of available names by doing things the way we've been doing it.

```
(defun metadata-article-name-p (name)
  (when (and (listp name)
             (eq (car name) 'meta))
    t))

(defun base-name-from-metadata-name (name)
  (second name))
```

3.3.2 On 'metadata-article'

This function retrieves the metadata article associated with the article named name.

```
(defun metadata-article (name)
  (get-article (list 'meta name)))
```

3.3.3 On 'put-metadata-field'

This can be can be used to fill a metadata field with the given value, in the metadata article pertaining to the article named name. If the field has not yet been set, it is created anew with the appropriate value.

```
(defun put-metadata-field (field value name)
  (let* ((metadata (metadata-article name))
         (old-value (assoc field (scholium-text metadata)))
         new-text)
    (if old-value
        (setq new-text (progn (setcdr old-value value)
                              (scholium-text metadata)))
      (setq new-text `((,field . ,value))))
    (scholium (list 'meta name)
              new-text
              nil
              'meta
              'system)))
```

3.3.4 On 'get-metadata-field'

This can be used to retrieve any particular metadata field pertaining to the article named name.

```
(defun get-metadata-field (field name)
  (cdr (assoc field (scholium-text (metadata-article name)))))
```

3.3.5 On 'delete-backlink'

This function is called by 'put-backlinks' to prune away an old backlink.

Also notice that in this function, we know that some metadata and backlinks already exist, so in particular the 'metafield' exists. Hence we don't need the complicated conditions found in the 'add-backlink' function.

(I don't suppose we actually need to delete the "backlink" field in the case in which we delete the last backlink.)

```
(defun delete-backlink (linked-to-article link-number)
  (let* ((metaarticle (metadata-article linked-to-article))
```

```
        (metatext (scholium-text metaarticle))
        (metafield (assoc 'backlinks metatext))
        (backlinks (remove (list name link-number)
                           (cdr metafield))))
    (setcdr metafield backlinks)
    (scholium (list 'meta linked-to-article)
              metatext
              nil
              'meta
              'system)))
```

3.3.6 On 'add-backlink'

This function is called by 'put-backlinks' to add a new backlink.

Note that the relevant metadata article may not exist prior to the time this function runs and even if it exists, it may not contain any backlink data. So we have to run the relevant tests to deal with these cases. This accounts for differences between the implementation of 'add-backlink' and 'delete-backlink' (Note 3.3.5).

```
(defun add-backlink (linked-to-article link-number)
  (let* ((metaarticle (metadata-article linked-to-article))
         (metatext (scholium-text metaarticle))
         (metafield (assoc 'backlinks metatext))
         (backlinks (add-to-or-start-list
                     (cdr metafield)
                     (list name link-number))))
    (cond
     (metafield
      (setcdr metafield backlinks))
     (metatext
      (setcdr metatext
              `(((backlinks . ,backlinks)))))
     (t
      (setq metatext
            `((backlinks . ,backlinks)))))
    (scholium (list 'meta linked-to-article)
              metatext
              nil
              'meta
              'system)))
```

3.3.7 On 'put-backlinks'

The basic use for this function is to run it within the scope of the 'scholium' function, to build and store new (or updated) backlink data. Backlinks are added to each of the metadata articles corresponding to articles that this article is about.

The algorithm used here (zap *all* existing backlinks, then place *all* new backlinks) seems likely to be woefully inefficient! If so, this can be improved later.

Notice that we don't take into account name changes with this function (there is no relevant "old-name" variable, at least, not at present).

```
(defun put-backlinks ()
  (let ((old-about (scholium-about old-version))
        (link-number 0))
    (dolist (link old-about)
      (setq link-number (1+ link-number))
      (delete-backlink (car link) link-number)))
  (let ((link-number 0))
    (dolist (link about)
      (setq link-number (1+ link-number))
      (add-backlink (car link) link-number))))

(add-hook 'new-scholium-hook 'put-backlinks)
(add-hook 'scholium-modified-about-hook 'put-backlinks)
```

3.4 Subcollections

3.4.1 On 'modified-type-labels'

An alist of types to look for paired with the corresponding label to apply. Each new or modified article is tested to see if it matches any of the given types; if so, then the corresponding label(s) will be applied. (Note that the "label" label is not treated specially.)

```
(defvar modified-type-labels '((nil . plain)
                               (label . label)
                               (meta . metadata)
                               (list . list)
                               (reference . reference)
                               (section . section)))
```

3.4.2 On 'label-article'

This function is responsible for the mechanics of editing labels, and creates them as needed.

```
(defun label-article (name article-label)
  (scholium article-label
            (add-to-or-start-list (scholium-text
                                   (get-article article-label))
                                  name)
            nil
            'label))
```

3.4.3 On 'label-article-insert-before'

Like 'label-article' (Note 3.4.2), but takes an additional argument before-this to say which entry the new name should be inserted just before.

```
(defun label-article-insert-before (name before-this article-label)
  (let* ((contents (scholium-text (get-article article-label)))
         (before-this-headed (member before-this contents)))
    (when before-this-headed
      (let ((len (- (length contents)
                    (length before-this-headed)
```

```
                1)))
        (if (> len -1)
            (setcdr (nthcdr len contents)
                    (cons name before-this-headed))
          (setq contents (cons name before-this-headed))))
      (scholium article-label
                contents
                nil
                'label))))
```

3.4.4 On 'put-type-labels'

This function runs within the scope of 'scholium'. It might make more sense to be able to add lots of different labels for a new article; the 'cond' function doesn't work that way however.

```
(defun put-type-labels ()
  (unless (typedata-includes type 'label)
    (dolist (type-label modified-type-labels)
      (when (typedata-includes type (car type-label))
        (label-article name (cdr type-label))))))

(add-hook 'new-scholium-hook 'put-type-labels)
(add-hook 'scholium-modified-type-hook 'put-type-labels)
```

3.4.5 On 'generalized-put-article'

Like 'put-article' (Note 3.1.2), but you specify a path as well as a name. The path should specify the path to the namespace wherein the name, value pair is to be stored.

```
(defun generalized-put-article (path name value)
  (puthash name value (generalized-get-article path)))
```

3.4.6 On 'generalized-get-article'

Like 'get-article' but takes a path. The path specifies a list of nested namespaces, and possibly, a final non-namespace item to be found in the last namespace. Optional argument current keeps track of the "current" namespace as the path is digested. If only one item appears in the path and namespace isn't given, the path points to an item in the main article table, and we return the corresponding item.

Be advised that there is a little bit of potentially destructive ambiguity here. But to make up for it, notice the doubly-clever use of 'and' and 'cond' (to deal with the case that we're generalizing).

```
(defun generalized-get-article (path &optional current)
  (let ((first-step (get-article path)))
    (cond
     ((and (not current) first-step)
      first-step)
     ((cdr path)
      (if (typedata-includes (get-article (car path)) 'namespace)
          (generalized-get-article (cdr path) (car path))
        (error "Path element not a namespace")))
     ((and (car path) current)
      (gethash current (car path)))
     ((car path) (get-article (car path))))))
```

3.5 Data access and conversion

3.5.1 Introduction to data access and conversion

The functions in this section give easy access to things we're often interested in later on. The functions here are simple, but it is useful to have names for them anyway (Note `2.2-Design_issues.org`).

```
(defun label-to-list (label)
  (mapcar (lambda (name)
            (format "%s" name))
          (scholium-text (get-article label))))

(defun label-to-propertized-list (label)
  (mapcar (lambda (name)
            (propertize (format "%s" name) 'name name))
          (scholium-text (get-article label))))
```

3.5.2 On 'get-backlinks'

This code produces the backlinks associated with name. See Note for details. Notice that throughout the code I have been applying this idiom

```
(lambda (backlink)
  (get-article (car backlink)))
```

to the element of the returned list in order to come up with the names of the backlinked articles. However, I recently converted the backlink format to be richer than it was before so that we knew *which link* was doing the linking, not just which article. So presumably we should be using this additional data at least some of the time.

```
(defun get-backlinks (name)
  (get-metadata-field 'backlinks name))

(defun get-links (name)
  (cdr (assoc 'links (scholium-text (metadata-article name)))))
```

3.5.3 On 'read-article-name'

We frequently have to read the name of an article from a list. We may be able to do this in several different more intelligent ways than the one we have here! For one thing, we could use 'turn-article-table-into-names' (which see). Another thing to do would be to allow some other source of input, for example, some particular namespace could be specified (this might be best handled with another function, 'read-article-name-from-namespace', say). We could offer some recursion if a namespace is selected.

```
(defun read-article-name ()
  (let* ((completion-ignore-case t)
         (strings (turn-article-table-into-list))
         (string-found (completing-read
                        "Article: "
                        strings))
         (place (- (length strings)
                   (length (member string-found strings)))))
    (nth place (turn-article-table-into-names))))
```

3.5.4 On 'read-article-path'

Like 'read-article-name' but reads a path to an article through namespaces (cf. 'generalized-get-article', Note 3.4.6).

```
(defun read-article-path (&optional namespace path)
  (let* ((completion-ignore-case t)
         (strings (if namespace
                      (turn-namespace-into-list namespace)
                    (turn-article-table-into-list)))
         (string-found (completing-read
                        "Article: "
                        strings))
         (place (- (length strings)
                   (length (member string-found strings))))
         (ret (nth place (if namespace
                             (turn-namespace-into-names namespace)
                           (turn-article-table-into-names)))))
    (setq path (append path (list ret)))
    (if (and (typedata-includes ret 'namespace)
             (y-or-n-p "Read further? "))
        (read-article-path namespace path))
    path))
```

3.5.5 On 'link-about-article-p'

This provides a quick way to tell whether one of the elements of an about list is actually about the article named article-name.

```
(defun link-about-article-p (link article-name)
  (equal (linked-to-article link) article-name))
```

4 Part IV — HONEY Storage Backend

This backend is included here for reference, but its features should be replaced by the new Clojure backend.

4.1 Preliminaries

4.1.1 Required packages

We use the Common Lisp compatibility functions.

```
(require 'cl)
```

4.1.2 On 'filter'

This is a useful utility to filter elements of a list satisfying a condition. Returns the subset of stuff which satisfies the predicate pred.

```
(defun filter (pred stuff)
  (let ((ans nil))
    (dolist (item stuff (reverse ans))
```

```
      (if (funcall pred item)
            (setq ans (cons item ans))
        nil))))

(filter '(lambda (x) (= (% x 2) 1)) '(1 2 3 4 5 6 7))
=> (1 3 5 7)
```

4.1.3 On 'intersection'

Set-theoretic intersection operation. More general than the version coming from the 'cl' package.

```
(defun intersection (&rest arg)
  (cond ((null arg) nil)
        ((null (cdr arg)) (car arg))
        (t (let ((ans nil))
             (dolist (elmt (car arg) ans)
               (let ((remainder (cdr arg)))
                 (while (and remainder
                             (member elmt (car remainder)))
                   (setq remainder (cdr remainder))
                   (when (null remainder)
                     (setq ans (cons elmt ans))))))))))

(intersection '(a b c d e f g h j)
              '(a b h j k)
              '(b d g h j k))
=> (j h b)
```

4.1.4 On 'mapply'

Map and apply rolled into one.

```
(defun mapply (f l)
  (if (member nil l) nil
    (cons (apply f (mapcar 'car l))
          (mapply f (mapcar 'cdr l)))))

(mapply '+ '((1 2) (3 4)))
 => (4 6)
```

4.1.5 On 'sublis'

Substitute objects in a list.

```
(defun sublis (sub lis)
  (cond
   ((null lis) nil)
   ((assoc lis sub) (cadr (assoc lis sub)))
   ((atom lis) lis)
   (t (cons (sublis sub (car lis))
            (sublis sub (cdr lis))))))
```

4.2 Core definitions

4.2.1 HONEY set up

These variables are needed for a coherent set-up.{Explain what they will do.}

```
(defvar plexus-registry '(0 nil))
(defvar current-plexus nil)
```

4.2.2 The 'add-plexus' function

We use this create a new plexus for storage. It defines a counter (beginning at 1), together with several hash tables that allow efficient access to the plexus' contents: an article table, forward links, backward links, forward labels, and backward labels. Additionally, it defines a "ground" and "type" nodes.Explain these things in more detail.{NB. it could be useful to maintain a registry available networks, by analogy with Emacs's 'buffer-list', which I think could be done if we use 'cl-defstruct' below instead of 'list', and set up the constructor suitably (info "(cl) Structures").}

```
(defun add-plexus ()
  "Create a new plexus."
  (let ((newbie (list '*plexus*
                      1                                ; nema counter
                      (make-hash-table :test 'equal)  ; nema table
                      (make-hash-table :test 'equal)  ; forward links
                      (make-hash-table :test 'equal)  ; backward links
                      (make-hash-table :test 'equal)  ; forward labels
                      (make-hash-table :test 'equal)  ; backward labels
                      (car plexus-registry))))
    ;; Define ground and type nodes.
    (puthash 0 '(0 0) (nth 2 newbie))
    (puthash 1 '(0 0) (nth 2 newbie))
    (puthash 0 '((0 . 0) (1 . 0)) (nth 3 newbie))
    (puthash 0 '((0 . 0) (1 . 0)) (nth 4 newbie))
    (puthash 0 '"ground" (nth 5 newbie))
    (puthash '"ground" 0 (nth 6 newbie))
    (puthash 1 '"type" (nth 5 newbie))
    (puthash '"type" 1 (nth 6 newbie))
    ;; Register the new object and return it.
    (setq plexus-registry
          (append
           `(,(+ (car plexus-registry) 1)
             ,newbie)
           (cdr plexus-registry)))
    newbie))
```

4.2.3 The "remove-plexus" function

When we're done with our plexus, we should tidy up after ourselves.

```
(defun remove-plexus (plex)
  "Remove a plexus."
  ;; Wipe out the hash tables
```

```
  (dotimes (i 5)
    (clrhash (nth (+ i 2) plex)))
  ;; Remove the entry from the registry.
  (setq plexus-registry
        (cons
         (car plexus-registry)
         (delete
          (assoc (nth 7 plex)
                 (cdr plexus-registry))
          (cdr plexus-registry)))))

(defun show-plexus-registry ()
  plexus-registry)
```

4.2.4 Network selection

We can work with several networks, only one of which is "current" at any given time.

```
(defun set-current-plexus (plex)
  "Examine a different plexus instead."
  (setq current-plexus plex))

(defmacro with-current-plexus (plex &rest expr)
  (declare (debug (&rest form)))
  (append `(let ((current-plexus ,plex))) expr))

(defun show-current-plexus ()
  "Return the plexus currently being examined."
  current-plexus)
```

4.2.5 On 'next-unique-id'

Increment the identifier that tells us how many nemas are in our network.

```
(defun next-unique-id ()
  "Produce a yet unused unique identifier."
  (1+ (cadr current-plexus)))
```

4.2.6 On 'reset-plexus'

Reset article counter and hash tables. Redefine "ground" and "article-type".

```
(defun reset-plexus ()
  "Reset the database to its initial configuration."
  ; Reset nema counter and hash tables.
  (setcar (cdr current-plexus) 1)
  (dotimes (n 5)
    (clrhash (nth (+ n 2) current-plexus)))
  ;; Define ground and nema-type.
  (puthash 0 '(0 0) (nth 2 current-plexus))
  (puthash 1 '(0 0) (nth 2 current-plexus))
  (puthash 0 '((0 . 0) (1 . 0)) (nth 3 current-plexus))
```

```
  (puthash 0 '((0 . 0) (1 . 0)) (nth 4 current-plexus))
  (puthash 0 '"ground" (nth 5 current-plexus))
  (puthash '"ground" 0 (nth 6 current-plexus))
  (puthash 1 '"type" (nth 5 current-plexus))
  (puthash '"type" 1 (nth 6 current-plexus))
  nil)
```

4.3 Individual Operations

4.3.1 On 'add-nema'

Add record to article table. Add record to list of forward links of source. Add record to list of backward links of sink. Return the id of the new article.{Should we add an alias 'add-triple' for this function, to make it more clear that our middle/frontend is not implementation specific?}

```
(defun add-nema (src txt snk)
  "Enter a new nema to the database."
  (let ((uid (next-unique-id)))
    ;; Add record to nema table.
    (puthash uid
             `(,src ,snk . ,txt)
             (nth 2 current-plexus))
    ;; Add record to list of forward links of source.
    (puthash src
             (cons `(,uid . ,snk)
                   (gethash src (nth 3 current-plexus) nil))
             (nth 3 current-plexus))
    ;; Add record to list of backward links of sink.
    (puthash snk
             (cons
              `(,uid . ,src)
              (gethash snk (nth 4 current-plexus) nil))
             (nth 4 current-plexus))
    ;; Update the counter for long-term storage
    (setcar (cdr current-plexus) uid)
    ;; Return the id of the new nema.
    uid))
```

4.3.2 Retrieving elements of a nema

These functions exist to get the relevant components of a nema, given its uid.

```
(defun get-content (uid)
  "Return the content of the nema."
  (cddr (gethash uid (nth 2 current-plexus))))

(defun get-source (uid)
  "Return the source of the nema."
  (car (gethash uid (nth 2 current-plexus))))

(defun get-sink (uid)
  "Return the sink of the nema."
```

```
  (cadr (gethash uid (nth 2 current-plexus))))

(defun get-triple (uid)
  (list (get-source uid)
        (get-content uid)
        (get-sink uid)))
```

4.3.3 On 'update-content'

old source old sink new content

```
(defun update-content (uid txt)
  "Replace the content of the nema."
  (puthash uid
             (let ((x (gethash uid (nth 2 current-plexus))))
                    `(,(car x)    ; old source
                             ,(cadr x) . ; old sink
                                    ,txt))      ; new content
                (nth 2 current-plexus)))
```

4.3.4 On 'update-source'

Extract current source. Extract current sink. Extract current content. Update the entry in the article table. Remove the entry with the old source in the forward link table. If that is the only entry filed under old-src, remove it from table. Add an entry with the new source in the forward link table. Update the entry in the backward link table.

```
(defun update-source (uid new-src)
  "Replace the source of the nema."
  (let* ((x (gethash uid (nth 2 current-plexus)))
         (old-src (car x))   ; extract current source
         (old-snk (cadr x))  ; extract current sink
         (old-txt (cddr x))) ; extract current content
    ;; Update the entry in the nema table.
    (puthash uid
             `(,new-src ,old-snk . ,old-txt)
             (nth 2 current-plexus))
    ;; Remove the entry with the old source in the
    ;; forward link table.  If that is the only entry
    ;; filed under old-src, remove it from table.
    (let ((y (delete `(,uid . ,old-snk)
                     (gethash old-src
                              (nth 3 current-plexus)
                              nil))))
      (if y
          (puthash old-src y (nth 3 current-plexus))
        (remhash old-src (nth 3 current-plexus))))
    ;; Add an entry with the new source in the
    ;; forward link table.
    (puthash new-src
             (cons `(,uid . ,old-snk)
```

```
                (gethash old-src (nth 3 current-plexus) nil))
            (nth 3 current-plexus))
    ;; Update the entry in the backward link table.
    (puthash old-snk
            (cons `(,uid . ,new-src)
                  (delete `(,uid . ,old-src)
                          (gethash old-src
                                   (nth 4 current-plexus)
                                   nil)))
            (nth 4 current-plexus))))
```

4.3.5 On 'update-sink'

Extract current source. Extract current sink. Extract current content. Update the entry in the article table. Remove the entry with the old sink in the backward link table. If that is the only entry filed under old-src, remove it from table. Add an entry with the new source in the backward link table. Update the entry in the forward link table.

```
(defun update-sink (uid new-snk)
  "Change the sink of the nema."
  (let* ((x (gethash uid (nth 2 current-plexus)))
         (old-src (car x))   ; extract current source
          (old-snk (cadr x))  ; extract current sink
           (old-txt (cddr x))) ; extract current content
    ; Update the entry in the nema table.
    (puthash uid
            `(,old-src ,new-snk . ,old-txt)
            (nth 2 current-plexus))
    ;; Remove the entry with the old sink in the
    ;; backward link table.  If that is the only entry
    ;; filed under old-src, remove it from table.
    (let ((y (delete `(,uid . ,old-src)
                     (gethash old-snk
                              (nth 4 current-plexus)
                              nil))))
      (if y
          (puthash old-snk y (nth 4 current-plexus))
        (remhash old-snk (nth 4 current-plexus))))
    ;; Add an entry with the new source in the
    ;; backward link table.
    (puthash new-snk
            (cons `(,uid . ,old-src)
                  (gethash old-snk
                           (nth 4 current-plexus)
                           nil))
            (nth 4 current-plexus))
    ;; Update the entry in the forward link table.
    (puthash old-src
            (cons `(,uid . ,new-snk)
                  (delete `(,uid . ,old-snk)
                          (gethash old-src
```

```
                                (nth 3 current-plexus)
                                nil)))
            (nth 3 current-plexus))))
```

4.3.6 On 'remove-nema'

Remove forward link created by article. Remove backward link created by article. Remove record from article table.

```
(defun remove-nema (uid)
  "Remove this nema from the database."
  (let ((old-src (car (gethash uid (nth 2 current-plexus))))
        (old-snk (cadr (gethash uid (nth 2 current-plexus)))))
  ;; Remove forward link created by nema.
  (let ((new-fwd (delete `(,uid . ,old-snk)
                         (gethash old-src (nth 3 current-plexus)))))
    (if new-fwd
        (puthash old-src new-fwd (nth 3 current-plexus))
      (remhash old-src (nth 3 current-plexus))))
  ;; Remove backward link created by nema.
  (let ((new-bkw (delete `(,uid . ,old-src)
                         (gethash old-snk (nth 4 current-plexus)))))
    (if new-bkw
        (puthash old-snk new-bkw (nth 4 current-plexus))
      (remhash old-snk (nth 4 current-plexus))))
  ;; Remove record from nema table.
  (remhash uid (nth 2 current-plexus))))
```

4.3.7 Functions for gathering links

Links are stored on triples alongside other elements.

```
(defun get-forward-links (uid)
  "Return all links having given object as source."
  (mapcar 'car (gethash uid (nth 3 current-plexus))))

(defun get-backward-links (uid)
  "Return all links having given object as sink."
  (mapcar 'car (gethash uid (nth 4 current-plexus))))
```

4.3.8 On 'label-nema'

Nemas can be given a unique human-readable label in addition to their numeric uid.

```
(defun label-nema (uid label)
  "Assign the label to the given object."
  (puthash uid label (nth 5 current-plexus))
  (puthash label uid (nth 6 current-plexus)))
```

4.3.9 Label to uid and uid to label lookup

These functions allow the exchange of uid and label.

```
(defun label2uid (label)
  "Return the unique identifier corresponding to a label."
  (gethash label (nth 6 current-plexus) nil))

(defun uid2label (uid)
  "Return the label associated to a unique identifier."
  (gethash uid (nth 5 current-plexus) nil))
```

4.4 Bulk Operations

4.4.1 On 'download-en-masse'

Unpack triplets, obtain labels if they exist. Write data in the network to a list, and return.

```
(defun download-en-masse ()
  "Produce a representation of the database as quintuples."
  (let ((plex nil))
    (maphash (lambda (uid tplt)
               ;; Unpack triplet.
               (let ((src (car tplt))
                     (snk (nth 1 tplt))
                     (txt (nthcdr 2 tplt)))
                 ;; Obtain label if exists.
                 (setq lbl (gethash uid
                                    (nth 5 current-plexus)
                                    nil))
                 ;; Write data to list.
                 (setq plex (cons `(,uid ,lbl ,src ,snk . ,txt)
                                  plex))))
             (nth 2 current-plexus))
    ;; Return list of data.
    (reverse plex)))
```

4.4.2 On 'upload-en-masse'

Unpack quintuplets. Plug into tables. Bump up article counter as needed.

```
(defun upload-en-masse (plex)
  "Load a representation of a database as quintuples into memory."
  (dolist (qplt plex t)
    ; unpack quintuplet
    (let ((uid (car qplt))
          (lbl (nth 1 qplt))
            (src (nth 2 qplt))
              (snk (nth 3 qplt))
                (txt (nthcdr 4 qplt)))
      ; plug into tables
      (puthash uid
```

```
                    `(,src ,snk . ,txt)
                           (nth 2 current-plexus))
      (puthash src
                    (cons `(,uid . ,snk)
                              (gethash src (nth 3 current-plexus) nil))
                           (nth 3 current-plexus))
      (puthash snk
                    (cons
                     `(,uid . ,src)
                     (gethash snk (nth 4 current-plexus) nil))
                           (nth 4 current-plexus))
      (when lbl
          (progn
                (puthash uid lbl (nth 5 current-plexus))
                    (puthash lbl uid (nth 6 current-plexus))))
      ; Bump up nema counter if needed.
      (when (> uid (cadr current-plexus))
             (setcar (cdr current-plexus) uid)))))
```

4.4.3 On 'add-en-masse'

Given several articles, add all of them at once.

```
(defun add-en-masse (plex)
  "Add multiple nemata given as list of quartuplets."
  (mapcar (lambda (qplt)
            (let ((uid (next-unique-id)))
              (add-nema (nth 1 plex)
                        (nthcar 2 plex)
                        (nth 2 plex))
              (label-nema uid (car qplt))))
          plex))
```

4.5 Query

4.5.1 Various lookup functions

These functions allow testing and lookup of various elements of a net.

```
(defun uid-p (uid)
  "Is this a valid uid?"
  (let ((z '()))
    (not (eq z (gethash uid (nth 2 current-plexus) z)))))

(defun uid-list ()
  "List of all valid uid's."
  (maphash (lambda (key val) key)
           (nth 2 current-plexus)))

(defun ground-p (uid)
  "Is this nema the ground?"
  (= uid 0))
```

```
(defun source-p (x y)
  "Is the former nema the sink of the latter?"
  (equal x (get-source y)))

(defun sink-p (x y)
  "Is the former nema the sink of the latter?"
  (equal x (get-sink y)))

(defun links-from (x y)
  "Return all links from nema x to nema y."
  (filter '(lambda (z) (source-p x z))
          (get-backward-links y)))

(defun links-p (x y)
  "Does nema x link to nema y?"
  (when (member x (mapcar
                   'get-source
                   (get-backward-links y)))
    t))

(defun triple-p (x y z)
  "Do the three items form a triplet?"
  (and (source-p y x)
       (sink-p y z)))

(defun plexus-p (x)
  "Is this object a plexus?"
  (let ((ans t))
    (setq ans (and ans
                   (equal (car x) "*plexus*")))
    (setq ans (and ans
                   (integrp (cadr x))))
    (dotimes (n 5)
          (setq ans (and ans (hash-table-p
                                  (nth (+ n 2) x)))))
    ans))
```

4.6 Iteration

4.6.1 Iterating over a plexus

These functions allow users to run loops over a plexus without having to delve into its internal structure.{I forget whether the use of 'apply' here is good form.}

```
(defmacro do-plexus (var res body)
  `((maphash (lambda (,var val) ,body)
             (nth 2 current-plexus))
    ,res))

;; This maps over the keys; func should be
```

```
;; defined appropriately.
(defun map-plexus (func)
  (let ((ans nil))
    (maphash
     (lambda (key val)
       (push (apply func (list key)) ans))
     (nth 2 current-plexus))
    ans))

(defun filter-plexus (pred)
  (let ((ans nil))
    (maphash
     (lambda (key val)
       (when (apply pred (list key))
         (push key ans)))
       (nth 2 current-plexus))
     ans))
```

4.6.2 Filtering convenience functions

Several convenience functions for filtering the plexus can be defined. They give lists of uids, which can be expanded using get-triple.

```
(defun nemas-given-beginning (node)
  "Get triples outbound from the given NODE."
  (filter-plexus
   (lambda (x) (when (equal (get-source x)
                            node)
                 (list node
                       (get-content x)
                       (get-sink x))))))

(defun nemas-given-end (node)
  "Get triples inbound into NODE."
  (filter-plexus
   (lambda (x) (when (equal (get-sink x)
                            node)
                 (list (get-source x)
                       (get-content x)
                       node)))))

(defun nemas-given-middle (edge)
  "Get the triples that run along EDGE."
  (filter-plexus
   (lambda (x) (when (equal (get-content x)
                            edge)
                 (list (get-source x)
                       edge
                       (get-sink x))))))

(defun nemas-given-middle-and-end (edge node)
```

```
  "Get the triples that run along EDGE into NODE."
  (filter-plexus
   (lambda (x) (when (and
                      (equal (get-content x)
                             edge)
                      (equal (get-sink x)
                             node))
                (list (get-source x)
                      edge
                      node)))))

(defun nemas-given-beginning-and-middle (node edge)
  "Get the triples that run from NODE along EDGE."
  (filter-plexus
   (lambda (x) (when (and
                      (equal (get-source x)
                             node)
                      (equal (get-content x)
                             edge))
                (list node
                      edge
                      (get-sink x))))))

(defun nemas-given-beginning-and-end (node1 node2)
  "Get the triples that run from NODE1 to NODE2."
  (filter-plexus
   (lambda (x) (when (and
                      (equal (get-source x)
                             node1)
                      (equal (get-sink x)
                             node2))
                (list node1
                      (get-content x)
                      node2)))))

(defun nemas-exact-match (node1 edge node2)
  "Get the triples that run from NODE1 along EDGE to
NODE2."
  (filter-plexus
   (lambda (x) (when (and
                      (equal (get-source x)
                             node1)
                      (equal (get-content x)
                             edge)
                      (equal (get-sink x)
                             node2))
                (list node1
                      edge
                      node2)))))
```

4.6.3 Additional elementary functions for node access

These functions give access to the various parts of a node.{Note: since 'article-list' is not defined, should these functions be deleted? Or should they be rewritten to access 'current-plexus'?}

```
(defun get-src (n)
  (car (nth 0 (cdr (assoc n (cdr article-list))))))

(defun get-flk (n)
  (cdr (nth 0 (cdr (assoc n (cdr article-list))))))

(defun get-txt (n)
  (nth 1 (cdr (assoc n (cdr article-list)))))

(defun get-snk (n)
  (car (nth 2 (cdr (assoc n (cdr article-list))))))

(defun get-blk (n)
  (cdr (nth 2 (cdr (assoc n (cdr article-list))))))

(defun get-ids nil
  (mapcar (quote car) (cdr article-list)))

(defun get-gnd nil 0)
```

4.6.4 On 'search-cond'

Surround the search within dolist loops on free variables. Wrap no further when finished.{Upgrade this to concatenate the results together. Also maybe allow options to add headers or to only loop over unique tuplets.}{Explain; how does this differ from the function defined at Note 4.6.13?}

```
(defmacro search-cond (vars prop)
  "Find all n-tuplets satisfying a condition"
  (let ((foo '(lambda (vars cmnd)
               (if vars
                   Wrap in a loop.
                   `(dolist (,(car vars) uids)
                      ,(funcall foo (cdr vars) cmnd))
                   cmnd))))
    (funcall foo vars prop)))
```

4.6.5 Overview of the search pipeline

We will implement the search as a pipeline which gradually transforms the query into a series of expressions which produce the sought-after result, then evaluate those expressions.

A search query designates predicates apply to the nodes and the network relationships that apply to them. The network relationships function as wildcards.

The basic model of the data is triplets that point to other triplets. The following query asks for a *funny* link from a *big blue object* to a *small green link* pointing outwards from the big blue object.

```
(((a blue big) (b funny) (c green small)
  ((b src a) (b snk c) (c src a))
```

The first step of processing is to put the quaerenda in some order so that each item links up with at least one previous item:

```
(scheduler
 '((b funny)
   (c green small))
 '((b src a)
   (b snk c)
   (c src a))
 '((a blue big)))
=>
((c (green small) ((b snk c) (c src a)))
 (b (funny) ((b src a)))
 (a blue big))
```

Note that the order is reversed due to technicalities of implementing 'scheduler' — that is to say, a is first and does not link to any other variable, b is next and links to only a, whilst c is last and links to both a and b. At the same time, we have also rearranged things so that the links to previous items to which a given object are listed alongside that object. The next step is to replace the links with the commands which generate a list of such objects:

```
((c (green small) ((b snk c) (c src a)))
 (b (funny) ((b src a)))
 (a blue big))
=>
((c (green small)
    (intersection (list (get-snk b)) (get-forward-links a)))
 (b (funny)
    (intersection (get-backward-links a)))
 (a blue big))
```

This is done using the function 'tplts2cmd', e.g.

```
(tplts2cmd 'c  '((b snk c) (c src a)))
=>
(intersection (list (get-snk b)) (get-forward-links a))
```

Subsequently, we filter over the predicates:

```
((c (filter '(lambda (c) (and (green c) (small c)))
    (intersection (list (get-snk b))
  (get-forward-links))))
 (b (filter '(lambda (b) (and (funny b)))
    (intersection (get-backward-links a)))))
```

This is done with the command 'add-filt':

```
(add-filt 'c
  '(green small)
  '((b snk c) (c src a)))
=>
(c (filter (quote (lambda (c) (and (green c) (small c))))
   (intersection (list (get-snk b))
 (get-forward-links a))))
```

This routine calls up the previously described routine 'tplts2cmd' to take care of the third argument. The last entry, `(a blue big)` gets processed a little differently because we don't as yet have anything to filter over; instead, we generate the initial list by looping over the current network:

```
(a (let ((ans nil))
     (donet 'node
    (when (and (blue (get-content node))
       (big (get-content node)))
      (setq ans (cons node ans))))
     ans))
```

This is done by invoking 'first2cmd':

```
(first2cmd '(blue big))
=>
(let ((ans nil))
  (donet (quote node)
 (when (and (blue (get-content node))
    (big (get-content node)))
 (setq ans (cons node ans))))
 ans)
```

And putting this all together:

```
(query2cmd
 '((c (green small) ((b snk c) (c src a)))
   (b (funny) ((b src a)))
   (a blue big)))
=>
((c (filter (quote (lambda (c) (and (green c) (small c))))
    (intersection (list (get-snk b))
  (get-forward-links a))))
 (b (filter (quote (lambda (b) (and (funny b))))
    (intersection (get-forward-links a))))
 (a (let ((ans nil))
      (donet (quote node)
     (when (and (blue (get-content node))
(big (get-content node)))
       (setq ans (cons node ans))))
      ans)))
```

To carry out these instructions in the correct order and generate a set of variable assignments, we employ the 'matcher' function. Combining this last layer, we have the complete pipeline:

```
(matcher nil
 (query2cmd
  (scheduler
   '((b funny)
     (c green small))
   '((b src a)
     (b snk c)
     (c src a))
   '((a blue big)))))
```

This combination of operations is combined into the 'search' function, which can be called as follows:

```
(search
 '(((a blue big)
    (b funny)
    (c green small))
   ((b src a)
    (b snk c)
    (c src a))))
```

Having described what the functions are supposed to do and how they work together, we now proceed to implement them.

4.6.6 On 'scheduler'

The scheduler function takes a list search query and rearranges it into an order suitable for computing the answer to that query. Specifically, a search query is a pair of lists — the first list consists of lists whose heads are names of variables and whose tails are predicates which the values of the variables should satisfy and the second list consists of triples indicating the relations between the values of the variables. Its arguments are:

- new-nodes, a list of items of the form |(node rest property)|;
- |links|, a list of triplets;
- |sched| is a list whose items consist of triplets of the form |(node (rest property) (rest link))|.

A recursive function to find linked nodes. If done, return answer. New nodes yet to be examined. Element of remaining-nodes currently under consideration. List of links between candidate and old-nodes. List of nodes already scheduled. Loop through new nodes until find one linked to an old node. Look at the next possible node. Find the old nodes linking to the candidate node and record the answer in "ties". Pick out the triplets whose first element is the node under consideration and whose third element is already on the list or vice-versa. Recursively add the rest of the nodes.

```
(defun scheduler (new-nodes links sched)
  (if (null new-nodes)
      sched
    (let ((remaining-nodes new-nodes)
          (candidate nil)
          (ties nil)
          (old-nodes (mapcar 'car sched)))
      (while (null ties)
        (setq candidate (car remaining-nodes))
        (setq remaining-nodes (cdr remaining-nodes))
        (setq ties
              (filter '(lambda (x)
                         (or
                          (and (eq (first x) (car candidate))
                               (member (third x) old-nodes))
                          (and (member (first x) old-nodes)
                               (eq (third x) (car candidate)))))
                      links)))
```

```
      (scheduler (remove candidate new-nodes)
                 links
                 (cons (list (car candidate)
                             (cdr candidate)
                             ties)
                       sched)))))
```

4.6.7 On 'tplts2cmd'

...Explain.

```
(defun tplts2cmd (var tplts)
  (cons 'intersection
        (mapcar
         #'(lambda (tplt)
             (cond ((and (eq (third tplt) var)
                         (eq (second tplt) 'src))
                    `(get-flk ,(first tplt)))
                   ((and (eq (third tplt) var)
                         (eq (second tplt) 'snk))
                    `(get-blk ,(first tplt)))
                   ((and (eq (first tplt) var)
                         (eq (second tplt) 'src))
                    `(list (get-src ,(third tplt))))
                   ((and (eq (first tplt) var)
                         (eq (second tplt) 'snk))
                    `(list (get-snk ,(third tplt))))
                   (t nil)))
         tplts)))
```

4.6.8 On 'add-filt'

...Explain.

```
(defun add-filt (var preds tplts)
  `(,var
    (filter
     #'(lambda (,var)
         ,(cons 'and
                (mapcar
                 #'(lambda (pred)
                     (list pred
                           (list 'get-txt var)))
                 preds)))
     ,(tplts2cmd var tplts))))
```

4.6.9 On 'first2cmd'

...Explain.

```
(defun first2cmd (preds)
  `(let ((ans nil))
```

```
    (dolist (node (get-ids) ans)
      (when
          ,(cons 'and
                 (mapcar
                  #'(lambda (pred)
                      (cons pred '((get-txt node))))
                  preds))
        (setq ans (cons node ans))))))
```

4.6.10 On 'query2cmd'

...Explain.

```
(defun query2cmd (query)
  (let ((backwards (reverse query)))
    (reverse
     (cons
      (list (caar backwards)
            (first2cmd (cdar backwards)))
      (mapcar
       #'(lambda (x)
           (add-filt (first x) (second x) (third x)))
       (cdr backwards))))))
```

4.6.11 On 'matcher'

...Explain.

```
(defun matcher (assgmt reqmts)
  (if (null reqmts) (list assgmt)
    (apply 'append
           (mapcar
            #'(lambda (x)
                (matcher (cons (list (caar reqmts) x)
                               assgmt)
                         (cdr reqmts)))
            (apply 'intersection
                   (eval `(let ,assgmt
                            (mapcar 'eval
                                    (cdar reqmts)))))))))
```

4.6.12 How matcher works

Here are some examples unrelated to what comes up in searching triplets which illustrate how matcher works:

```
(matcher '((x 1)) '((y (list 1 3))
                    (z (list (+ x y) (- y x)))))
=>
(((z 2) (y 1) (x 1))
 ((z 0) (y 1) (x 1))
 ((z 4) (y 3) (x 1))
```

```
 ((z 2) (y 3) (x 1)))

(matcher nil '((x (list 1))
               (y (list 1 3))
               (z (list (+ x y) (- y x)))))
=>
(((z 2) (y 1) (x 1))
 ((z 0) (y 1) (x 1))
 ((z 4) (y 3) (x 1))
 ((z 2) (y 3) (x 1)))
```

4.6.13 On 'search'

…{Explain; how does this differ from the macro defined at Note 4.6.4?}

```
(defun search (query)
  (matcher nil
           (reverse
            (query2cmd
             (scheduler
              (cdar query)
              (cadr query)
              (list (caar query)))))))
```

4.7 Scholium programming

4.7.1 On 'node-fun'

…Explain. Produce a list of commands to produce temporary bindings. Produce a list of commands to reset function values.

```
(defun node-fun (node get-code get-links)
  (let ((code  (funcall get-code node))
        (links (funcall get-links node)))
    (list
     'lambda
     (car code)
     (cons
      'prog1
      (cons
       (append
        '(progn)
        (mapcar #'(lambda (x)
                    `(fset ',(car x)
                           (node-fun ,(cdr x)
                                     ',get-code
                                     ',get-links)))
                links)
        (cdr code))
       (mapcar #'(lambda (x)
                   (if (fboundp (car x))
                       `(fset ',(car x)
```

```
                              ',(symbol-function (car x)))
                        `(fmakunbound ',(car x))))
                links))))))
```

4.7.2 On 'tangle-module'

Recursively replace the chunks to recover executable code.Explain.

```
(defun tangle-module (node get-cont ins-links)
  (insert-chunk
   (funcall get-cont node)
   (mapcar #'(lambda (x)
               (cons (car x)
                     (tangle-module (cdr x)
                                    get-cont
                                    ins-links)))
           (funcall ins-links node))))
```

4.7.3 On 'insert-chunk'

Given a node and an association list of replacement texts, insert the chunks at the appropriate places.

```
(defun insert-chunk (body chunks)
  (cond ((null body) nil)
        ((null chunks) body)
        ((equal (car body) '*insert*)
         (cdr (assoc (cadr body) chunks)))
        (t (cons (insert-chunk (car body) chunks)
                 (insert-chunk (cdr body) chunks)))))
```

4.7.4 Functions for rewriting nemas

Several functions for rewriting nemas.{How does this stuff relate to what's going on in the vicinity of Note 4.3.5?}

```
(defun set-src (n x)
  (if (equal n 0)
      0
    (progn (let ((old-backlink
                  (nth 1 (assoc (get-src n)
                                (cdr article-list)))))
             (setcdr old-backlink
                     (delete n (cdr old-backlink))))
           (let ((new-backlink
                  `(nth 1 (assoc x (cdr article-list)))))
             (setcdr new-backlink (cons n (cdr new-backlink))))
           (setcar (nth 1 (assoc n (cdr article-list))) x))))

(defun set-txt (n x)
  (setcar (cdr (cdr (assoc n (cdr article-list)))) x))

(defun set-snk (n x)
```

```
  (if (equal n 0)
      0
    (progn (let ((old-backlink
                  (nth 3 (assoc (get-snk n)
                                (cdr article-list)))))
             (setcdr old-backlink
                     (delete n (cdr old-backlink))))
           (let ((new-backlink
                  (nth 3 (assoc x (cdr article-list)))))
             (setcdr new-backlink (cons n (cdr new-backlink))))
           (setcar (nth 3 (assoc n (cdr article-list))) x))))

(defun ins-nod (src txt snk)
  (progn (setcdr article-list
                 (cons (list (car article-list)
                             (list src)
                             txt
                             (list snk))
                       (cdr article-list)))
         (let ((backlink
                (nth 3 (assoc snk (cdr article-list)))))
           (setcdr backlink (cons (car article-list)
                                  (cdr backlink))))
         (let ((backlink
                (nth 1 (assoc src (cdr article-list)))))
           (setcdr backlink (cons (car article-list)
                                  (cdr backlink))))
         (- (setcar article-list (+ 1 (car article-list))) 1)))

(defun del-nod (n)
  (if (or (equal n 0)
          (get-blk n)
          (get-flk n))
      nil
    (progn (let ((old-backlink
                  (nth 3 (assoc (get-snk n)
                                (cdr article-list)))))
             (setcdr old-backlink
                     (delete n (cdr old-backlink))))
           (let ((old-backlink
                  (nth 1 (assoc (get-src n)
                                (cdr article-list)))))
             (setcdr old-backlink
                     (delete n (cdr old-backlink))))
           (setcdr article-list
                   (delete (assoc n (cdr article-list))
                           (cdr article-list)))
           t)))
```

4.8 Initialization

4.8.1 Initialize with a new network

For now, we just create one network to import things into. Additional networks can be added later.

```
(set-current-plexus (add-plexus))
```

5 Part V — SQL Storage Backend

This is omitted in the current build, though it could be reintroduced if that was useful.

6 Part VI — Adding and Interacting with Articles

6.1 Database interaction

6.1.1 The 'article' function

You can use this function to create an article with a given name and contents. You can optionally put it in a list by specifying the heading that it is under. (If this is used multiple times with the same heading, that just becomes a cone over the contents.)

```
(defun article-2 (name contents &optional heading)
  (let ((coordinates (add-nema name
                               "has content"
                               contents)))
    (when heading (add-nema coordinates "in" heading))
    coordinates))
```

6.1.2 The 'scholium' function

You can use this function to link annotations to objects. As with the 'article' function, you can optionally categorize the connection under a given heading (cf. Note 6.1.1).

```
(defun scholium-2 (beginning link end &optional heading)
  (let ((coordinates (add-nema beginning
                               link
                               end)))
    (when heading (add-nema coordinates "in" heading))
    coordinates))
```

6.1.3 On 'get-article'

Get the contents of the article named 'name'. We assume that there is only one such article for now.

```
(defun get-article (name)
  (get-sink
   (first
    (nemas-given-beginning-and-middle
     (first
      (nemas-given-beginning-and-end
       name "arxana-merge.tex"))
     "has content"))))
```

6.1.4 On 'get-names'

This function simply gets the names of articles that have names – in other words, every triple built around the "has content" relation.{This seems to work but are both map operations needed?}

```
(defun get-names (&optional heading)
  (mapcar #'get-source
   (mapcar #'get-source (nemas-given-middle "has content"))))
```

6.2 Supplying initial bookkeeping information

6.2.1 On 'sch-book'

Interactive functions that call 'scholium' should always provide bookkeeping information.

The following definition can be extended to add any sort of initial bookkeeping information we might like to maintain. The bookkeeping field will be revised when the article is edited (see Section).

```
(defun sch-book ()
  `((created ,user-login-name ,(current-time-string))))
```

6.3 Adding text from an existing source

6.3.1 On 'make-current-buffer-into-article'

We store the object representing the buffer in the text field. There is no particular reason to set the type (at least, one has yet to appear).

An alternative approach would be to make an article's text reflect the current contents of the buffer (i.e., save the buffer as a string) – but we currently don't do this. However, it would probably be nice to have that option, or even just the option to make a certain selection into an article.

```
(defun make-buffer-into-article (&optional buffer name)
  (let* ((buffer (or buffer
                     (get-buffer (read-buffer
                                  "Buffer: "
                                  (buffer-name
                                   (current-buffer))
                                  t))))
         (name (or name
                   (read-string "Name: "
                                nil
                                nil
                                (buffer-name buffer))))
         (sync-context (and (fboundp 'futon4--article-context-for-buffer)
                            (futon4--article-context-for-buffer buffer))))
    (scholium name buffer nil nil (sch-book))
    (when sync-context
      (let* ((article-path (plist-get sync-context :path))
             (article-id (and (fboundp 'futon4--article-id-for)
                              (futon4--article-id-for name article-path))))
        (when (and article-id (fboundp 'futon4-ensure-article-entity))
          (futon4-ensure-article-entity article-id name article-path))))))

(defun make-current-buffer-into-article (name)
```

```
  (interactive (list (read-string
                      (concat "Name (default "
                              (buffer-name (current-buffer)) "): ")
                      nil
                      nil
                      (buffer-name (current-buffer)))))
  (make-buffer-into-article (current-buffer) name))
```

6.3.2 On 'make-file-into-article'

This function adds a file to the article list. Again, the text field is not filled in with a string representing the text directly, but rather, with the name of the file that holds the text. A token is added to the type field explaining that this is a file.

```
(defun make-file-into-article (path name)
  (interactive
   (let* ((bufn (buffer-file-name))
          (pth (read-file-name
                (if bufn
                    (concat "File (default: " bufn "): ")
                  (concat "File: "))
                nil
                (if bufn
                    bufn
                  (default-directory))))
          (dnme (file-name-nondirectory pth))
          (nme (read-string (concat "Name (default: " dnme "): ")
                            nil
                            nil
                            dnme)))
     (list pth nme)))
  (scholium name path nil '(file) (sch-book))
  (let* ((canonical (and (fboundp 'futon4--canonical-path)
                         (futon4--canonical-path path)))
         (article-id (and (fboundp 'futon4--article-id-for)
                          (futon4--article-id-for name canonical))))
    (when (and article-id (fboundp 'futon4-ensure-article-entity))
      (futon4-ensure-article-entity article-id name canonical))))
```

6.4 Interactively supplying text

6.4.1 Global varibles describing new scholia

The variables 'new-scholium-name' and 'new-scholium-about' are used to build new articles interactively. For now, type is ignored by these functions. Also, we don't have a 'new-scholium-text' field, since we get the text another way. It could be that an approach with a 'new-scholium-text' variable would allow us to unify the treatment here with the one used in previous subsections.

```
(defvar new-scholium-name nil "Name of our new scholium.")
(defvar new-scholium-about nil "What the new scholium is about.")
```

6.4.2 On 'new-scholium-mode'

This mode is invoked by 'make-scholium' in the buffer in which the new scholium's text is to be supplied.

```
(define-minor-mode new-scholium-mode
  "Mode for composing a new scholium.
\\{new-scholium-mode-map}"
  :init-value nil
  :keymap '(("\C-c\C-c" . escape-scholium-creation)))
```

6.4.3 On 'escape-scholium-creation'

Once the new scholium's text has been supplied, this function creates a scholium from that text and other data the user has supplied. It then restores the window configuration that was active before 'make-scholium' ran. It also nullifies 'new-scholium-name' and 'new-scholium-about', since we're done with these things.

```
(defun escape-scholium-creation ()
  (interactive)
  (scholium new-scholium-name
            (buffer-substring-no-properties (point-min) (point-max))
            new-scholium-about
            nil
            (sch-book))
  (kill-buffer (concat "Editing scholium: " new-scholium-name))
  (set-window-configuration sch-win-config)
  (setq new-scholium-name nil
        new-scholium-about nil))
```

6.4.4 On 'make-scholium'

This function is called every time the user makes a scholium with new text (i.e. text that is typed in on the fly). Functions for making scholia about articles, parts of articles, buffers, etc., are given in this document and all use this function.

```
(defun make-scholium ()
  (setq sch-win-config (current-window-configuration))
  ;; we allow this to be set elsewhere
  (unless new-scholium-name
    (setq new-scholium-name (read-string "Scholium name: ")))
  (set-buffer (get-buffer-create "Scholia Display"))
  (other-window -1)
  (split-window-vertically)
  (other-window 1)
  (switch-to-buffer (get-buffer-create (concat "Editing scholium: "
                                               new-scholium-name)))
  (new-scholium-mode))
```

6.4.5 On 'make-scholium-about-current-article'

This function makes a scholium about the article as a whole.

```
(defun make-scholium-about-current-article ()
  (interactive)
  (when name-of-current-article
    (setq new-scholium-about `(((,name-of-current-article))))
    (make-scholium)))
```

6.4.6 On 'make-scholium-about-part-of-current-article'

This function makes a scholium about one specific portion of the article.

This function makes the (somewhat unrealistic seeming) assumption that the current article and the current buffer are the same thing. This situation should be resolved.

However, if no article is current yet, then perhaps we should offer to make a scholium about the current buffer?

```
(defun make-scholium-about-part-of-current-article (beg end)
  (interactive "r")
  (if name-of-current-article
      (progn
        (setq new-scholium-about
              `((,name-of-current-article
                  (passage
                   ,beg
                   ,end))))
        (make-scholium)
        (deactivate-mark)
        (message (concat (format "%s--%s" beg end) " added.")))
    (message "Make some article current first.")))
```

6.4.7 On 'make-scholium-about-current-line'

Here is a little convenience wrapper for working with lists.

```
(defun make-scholium-about-current-line ()
  (interactive)
  (make-scholium-about-part-of-current-article (line-beginning-position)
                                               (line-end-position)))

(defun make-scholium-about-current-line-quickly ()
  (interactive)
  (setq new-scholium-name (buffer-substring-no-properties
                           (line-beginning-position)
                           (line-end-position)))
  (make-scholium-about-part-of-current-article (line-beginning-position)
                                               (line-end-position)))

(defun make-scholium-about-current-line-quickly-and-completely ()
  (interactive)
  (scholium (buffer-substring-no-properties
                           (line-beginning-position)
                           (line-end-position))
            (buffer-substring-no-properties
```

```
                        (line-beginning-position)
                        (line-end-position))
             `((,name-of-current-article
                (passage
                 ,(line-beginning-position)
                 ,(line-end-position))))
             nil
             (sch-book)))

(defun make-scholium-about-current-sentence ()
  (interactive)
  (make-scholium-about-part-of-current-article (line-beginning-position)
                                               (line-end-position)))
```

6.4.8 On 'reading-regions-mode'

This mode is invoked by 'make-scholium-about-several-parts-of-current-article', and adds an editing mode in the buffer containing the current article that enables the user to select regions that the scholium will be about.

```
(define-minor-mode reading-regions-mode
  "Mode for reading in regions.
\\{new-scholium-mode-map}"
  :init-value nil
  :keymap '(("\C-c\C-c" . add-region)
            ("\C-c\C-g" . escape-reading-regions-mode))
  (message "C-c C-c to add regions; C-c C-g to make scholium."))
```

6.4.9 On 'add-region'

This function adds regions to 'new-scholium-about'.

```
(defun add-region (beg end)
  (interactive "r")
  (setq new-scholium-about
        (cons `(,name-of-current-article
                  (passage
                   ,(region-beginning)
                   ,(region-end)))
              new-scholium-about))
  (deactivate-mark)
  (message (concat (format "%s--%s" beg end) " added.")))
```

6.4.10 On 'escape-reading-regions-mode'

When all of the regions desired have been selected, this function calls 'make-scholium' to finish things off.

```
(defun escape-reading-regions-mode ()
  (interactive)
  (reading-regions-mode -1)
  (make-scholium))
```

6.4.11 On 'make-scholium-about-several-parts-of-current-article'

This function makes a scholium that applies to several portions of the article, using the mode and so on featured in this section.

(Note, it doesn't seem that this is displayed quite right; I'm getting two copies of the scholium's text in the Scholia Display buffer.)

```
(defun make-scholium-about-several-parts-of-current-article ()
  (interactive)
  (let ((article (get-article name-of-current-article)))
    (if (article-buffered article)
        (switch-to-buffer (get-buffer (scholium-text article)))
      (switch-to-buffer "Main Article Display"))
    (setq new-scholium-about nil)
    (reading-regions-mode 1)))
```

6.4.12 Scholia about the current buffer

6.4.13 On 'call-if-user-adds-current-buffer-to-article-list'

This is used by functions that require the current buffer to be an article; typically they recall themselves after the buffer has been added. It is used by 'make-scholium-about-current-buffer' and 'display-scholia-about-current-buffer'.

```
(defun call-if-user-adds-current-buffer-to-article-list (fct)
  (when (y-or-n-p "Buffer not an article, add to list? ")
        (make-current-buffer-into-article
         (read-string (concat "Name (default: "
                              (buffer-name
                               (current-buffer)) "): ")
                      nil
                      nil
                      (buffer-name (current-buffer))))
        (funcall fct)))
```

6.4.14 On 'make-scholium-about-current-buffer'

This function makes a scholium about the current buffer, requiring that it be an article. (Maybe we should just add the current buffer to the article list transparently, rather than giving the prompt in 'call-if-user-adds-current-buffer-to-article-list'.)

```
(defun make-scholium-about-current-buffer ()
  (interactive)
  (let ((article (get-article (buffer-name (current-buffer)))))
    (if (not article)
        (call-if-user-adds-current-buffer-to-article-list
         'make-scholium-about-current-buffer)
      (setq new-scholium-about
            `(((,(buffer-name (current-buffer))))))
      (make-scholium))))
```

6.4.15 On 'genref'

Instead of using 'gensym' (which causes problems) we do something similar to generate references with unique ids. Note that this might cause some problems when we go to the distributed way of doing things (or even just save and restore articles with references in them after making some intervening edits), since references with the same names might mean different things. But of course, this is true of all scholia with the same names, so I propose not to worry about it too much right now.

It may turn out to be advantageous to use a reference counter that is local to each article.

Actually, the scheme proposed here seems pretty weak; saving files and then reading them back in after Emacs has been shut down could cause problems. It might be much better to have a reference counter in each metadata article, so that the collection of references associated with a given article is always unique. Furthermore, references associated with a given article should possibly be recorded on that article's metadata explicitly *as references*.

```
(defvar *reference-counter* 0)

(defun genref ()
  (setq *reference-counter* (1+ *reference-counter*)))
```

6.4.16 On 'make-reference-in-current-article'

This function is similar to 'make-scholium-about-part-of-current-article' (Note 6.4.6) except that the type is set to "reference". References currently don't have any text, but we could later set things up to let them have a docstring or something like that.

(Are backlinks working properly in an article that has several references associated with it?)

```
(defun make-reference-in-current-article (beg end &optional target)
  (interactive "r")
  (let ((target (or target
                    (read-article-name))))
    (if name-of-current-article
        (when target
          (scholium `(reference ,name-of-current-article ,(genref))
                    nil
                    `((,name-of-current-article
                       (passage
                        ,beg
                        ,end))
                      (,target))
                    'reference)
          (deactivate-mark))
      ;; Maybe the message should instead be an offer to make
      ;; a scholium about the current buffer?
      (message "Make some article current first."))))
```

6.4.17 Speedy reference creation

It would be handy to be able to make a reference to an article and automatically create and follow the reference, all at the same time. (Maybe also subsets of these actions, like create but don't follow.) Probably we could do something similar for generalized references.

```
(defun create-follow-reference ()
  (interactive)
  (let ((name (buffer-substring (point) (mark))))
    (unless (get-article name)
      (save-excursion
        (set-buffer (get-buffer-create name))
        (make-current-buffer-into-article name)))
    (make-reference-in-current-article (min (point) (mark))
                                       (max (point) (mark))
                                       name)
    (display-article name)))
```

6.4.18 Reference component access

These functions give easy access to information specifying the referenced article (the target), the region the reference applies to, and the name of the article the reference lies in. These functions rely on the formulaic nature of the type data of references, namely, a link to the passage wherein the reference is made is given, followed by a link to the referenced article.

Notice that these functions would have to change if we later allow multiple sources (presumably, regions) to be part of the same reference.

```
(defun reference-source-link (reference)
  (first (scholium-about reference)))

(defun reference-from-article (reference)
  (car (reference-source-link reference)))

(defun reference-to-article (reference)
 (car (second (scholium-about reference))))
```

6.4.19 On 'make-new-undirected-article'

Although one could simply create a new buffer and add that buffer to the article list (as in Section), sometimes it may be more intuitive to simply add a new undirected article directly to the article list.

```
(defun make-new-undirected-article ()
  (interactive)
  (setq new-scholium-about nil)
  (make-scholium))
```

6.4.20 Followups

6.4.21 On 'name-of-current-scholium'

The function can be called from anywhere; "current" is defined relative to the position of 'point' in the Scholia Display buffer. (It is natural to assume that there is only one current scholium, given the way the contents of this buffer have been put together.)

Note that *scholia* appear once in the Scholia Display buffer, so there is only one "current scholium" (with one name) when we look at things this way. If we were going to do something like this for the main article buffer, then we'd need to do a bit more. (In fact, various functions to associate marked regions with scholia need just this sort of special touch.)

```
(defun name-of-current-scholium ()
  (interactive)
  (save-excursion
    (set-buffer (get-buffer-create "Scholia Display"))
    (let ((ret (car (scholia-named-at-point))))
      (if ret
          (message (format "%s" ret))
        (message "Position cursor on a scholium in Scholia Display."))
      ret)))

(defun make-scholium-about-current-scholium ()
  (interactive)
  (when (equal (buffer-name (current-buffer)) "Scholia Display")
    (let ((cur (name-of-current-scholium)))
      (when cur
        (progn (setq new-scholium-about `(((,cur))))
               (make-scholium))))))
```

7 Part VII — Rendering and Browsing

7.1 Formatting articles for display

7.1.1 On 'sch-plain-text-hook'

Alternative ways of setting the return value of 'sch-plain-text'.

```
(defvar sch-plain-text-hook nil)
```

7.1.2 On 'sch-plain-text'

This function is called by 'mark-up-scholium' to render scholia that are to be displayed alongside the main article, and also by 'transclude-article' (see Note 9.2.1) and other functions in Section 9.2.

Its goal is to turn arbitrary articles into strings. This will be done in different ways depending on the sort of article in question. (And could be done in other ways depending on other stuff.) Compare Note `2.2-Design_issues.org`.

Here's how it works: different kinds of "objects" are to be distinguished from one another by simple tests – is the 'car' equal to `passage`? Does looking up the object in the article table produce anything? Then act as appropriate, grabbing the text that is specified. Currently it works on input *article names* or input *articles*. (If input is both the name of an article and an article, it will be treated as a name.)

Another thing that might be handy to be able to process is *simple strings*, which aren't article names, or articles (obviously).

Note that links are rendered differently depending on context.

Links can't actually run from arbitrary object to arbitrary object within the current version of the system. That can probably be fixed easily.

Converting the function to render scholium system objects in general is a step towards answering the request in Note 3.1.4 about extending the purview of 'display-article'. At that point, it will take more than just a title as argument, certainly; and it will need a "triage" phase in order to figure out what sort of object it has been applied to; various modifications will have to be made so that it can be applied to various sorts of objects.

```
(defun sch-plain-text (title-or-article)
```

```
;; this overloading of the input & reliance upon `get-article' to
;; sort things out... could probably be revised into something
;; better
(let* ((obj (or (get-article title-or-article)
                title-or-article))
       (text (scholium-text obj))
       (type (scholium-type obj))
       ret)
  ;; This seems like the correct way to override a `cond' form.
  (run-hooks 'sch-plain-text-hook)
  (when (not ret)
    (cond
     ((bufferp text)
      (save-excursion (set-buffer (get-buffer text))
                      (setq ret (buffer-string))))
     ((typedata-includes type 'file)
      (let ((bufs (buffer-list))
            (live nil))
        (while (and bufs (not live))
          (when (equal (buffer-file-name (car bufs))
                       (expand-file-name text))
            (setq live (car bufs)))
          (setq bufs (cdr bufs)))
        (if live
            (progn (set-buffer live)
                   (setq ret (buffer-string)))
          (find-file text)
          (setq ret (buffer-string))
          (kill-buffer (current-buffer)))))
     ;; these quoted things should presumably themselves be rendered
     ;; as links (and we probably don't need the crazy markup for
     ;; things about the whole buffer, ever)
     ((typedata-includes type 'reference)
      (if (equal (reference-to-article obj)
                 name-of-current-article)
          (setq ret
                ;; it might be kind of cool to at least include a
                ;; snippet of the context of this link, say 3 lines
                ;; worth
                (format "\"%s\" links here."
                        (reference-from-article obj)))
        (setq ret
              (format "This is a link from \"%s\" to \"%s\"."
                      (reference-from-article obj)
                      (reference-to-article obj)
                      obj))))
     ((stringp text)
      (setq ret text))
     (t
      (setq ret (format "%S" text)))))
```

```
    ret))
```

7.2 Managing windows and buffers

7.2.1 Dot dot dot

How should scholia in later generations be displayed? In the default mode, without threading, a "dot dot dot" would be useful. (And in the case of scholia that bridge the gap between two articles, the "dot dot dot" probably needs to have a special meaning.) Anyway, note that these dot-dot-dotted things could be found by a depth-first search through the document collection. (Find anything about the current article, then find anything about that, etc., then find the next thing about the current article, etc.)[1]Compare the 2nd figure in Corneli and Krowne (2005) (cf. Footnote
Page **??**).}. % How to get page reference corresponding to a label?

```
(defvar pre-sch-win-config nil "Saved window configuration.")
(defvar sch-win-config nil "Saved window configuration.")

(defvar buffer-associated-with-current-article nil)
```

7.2.2 On 'article-buffered'

Here's a little convenience function that tells you whether the article's text lives in a buffer or not.

```
(defun article-buffered (article)
  (bufferp (scholium-text article)))
```

7.2.3 On 'scholia-overwhelm-display'

This function displays scholium stuff. It is called by 'display-article' (Note 7.5.13) and 'display-scholia-about-current-buffer' (Note 7.5.16). If there is a main article buffer (i.e. the article to be displayed lives in its own buffer), we use that buffer to display the article; otherwise, we use the "Main Article Display" buffer. Scholia that match the appropriate set of conditions in 'mark-things-up' will be rendered to the "Scholia Display" buffer.

```
(defun scholia-overwhelm-display (text)
  (unless pre-sch-win-config
    (setq pre-sch-win-config (current-window-configuration)))
  (delete-other-windows)
  (split-window-horizontally)
  (if rendering-target-buffer
      (pop-to-buffer rendering-target-buffer t)
    (switch-to-buffer (get-buffer-create "Main Article Display") t))
  (erase-buffer)
  (insert text)
  (goto-char (point-min))
  (setq buffer-associated-with-current-article (current-buffer))
  (other-window 1)
  (switch-to-buffer (get-buffer-create "Scholia Display") t)
  (erase-buffer))
```

[1]{

7.2.4 Switching between views

We offer a few convenient functions for switching between the article-plus-scholia browsing display and whatever came before. It would also be nice to offer a function for switching between the article display and the generic listing display.

```
(defun back-to-normal ()
  (interactive)
  (setq sch-win-config (current-window-configuration))
  (set-window-configuration pre-sch-win-config)
  (setq pre-sch-win-config nil))

(defun back-to-other-view ()
  (interactive)
  (setq pre-sch-win-config (current-window-configuration))
  (set-window-configuration sch-win-config)
  (setq sch-win-config nil))
```

7.3 Sorting scholia for markup purposes

7.3.1 On 'first-beginning-about-article'

Return 0 if about is about all of article, i.e., otherwise return the character position of the first region within article that about is actually about. If there is none, return nil. (Actually, let me note that in the usage we have established so far, we would already know that in the case that about is a string, it is about the article here, and in the case that about is a one-layer list, similarly; so some of the tests we do here are as yet unneeded.)

Here we're assuming that if there is *some* link to the article that isn't a passage link, then we treat the link as about the article as a whole. The case in which the link is about the whole article *and* some part of the article might possibly be better treated some other way; but I'll leave that case for subsequent work.

```
(defun first-beginning-about-article (about article)
  (cond
   ;; condition for the scholium to be about the article as a whole.
   ;; The condition is that there is _some_ link to the article that
   ;; is NOT a ``passage''-type link.  In this case, we just return 0.
   ((member-if (lambda (link)
                 (and (equal (car link) article)
                      (not (link-type-accessor link 'passage))))
               about)
    0)
   ;; else, collect the regions of `article' that `about' indicates,
   ;; and sort them.
   (t
    (let* ((marking-links
            (let (marked)
              (mapc (lambda (elt)
                      (when (and
                             (typedata-includes-passage (link-type elt))
                             (equal (linked-to-article elt) article))
                        (setq marked (cons elt marked))))
```

```
                  about)
              marked))
          (earliest-link
           (car (sort
                 marking-links
                 (lambda (link1 link2)
                   (< (link-beginning link1)
                      (link-beginning link2)))))))
     (link-beginning earliest-link)))))
```

7.3.2 On 'sort-scholia-by-beg-position'

This function orders scholia according to the magnitude of the smallest beginning a region of article that the input scholia mark. All links are considered when finding the first marked region.

```
(defun sort-scholia-by-beg-position (scholia article)
  (setq
   scholia
   (sort scholia
         (lambda (scholium1 scholium2)
           (let ((beg1 (first-beginning-about-article
                        (scholium-about scholium1)
                        article))
                 (beg2 (first-beginning-about-article
                        (scholium-about scholium2)
                        article)))
             (and beg1
                  beg2
                  (< beg1 beg2)))))))
```

7.4 Marking things up

7.4.1 On 'new-simple-scholium-face'

Adapted from ttn's `ricette-mode.el`. *Nota bene*: new faces are automatically customizable, so if you don't like the way they look, you can change them.

```
(defmacro new-simple-scholium-face (num color doc)
  (let ((prop (intern (concat "sch-face-" (int-to-string num)))))
    `(progn
       (defvar  ,prop ',prop)
       (defface ,prop
         '((t (:foreground ,(symbol-name color)
               :underline ,(symbol-name color)))) ,doc))))
```

7.4.2 Underlining versus foreground

It is worth considering using the underline attribute instead or in addition to the foreground attribute: underlining would be less obtrusive in documents that already use faces. We could have two variants; one with underlining for the main article, one foreground for the scholia display. We could also give users some immediately-customizable options. (Do the default colors I picked out work well?)

```
(new-simple-scholium-face 1 maroon1 "First scholium face.")
(new-simple-scholium-face 2 aquamarine1 "Second scholium face.")
(new-simple-scholium-face 3 IndianRed1 "Third scholium face.")
(new-simple-scholium-face 4 yellow1 "Fourth scholium face.")
(new-simple-scholium-face 5 firebrick1 "Fifth scholium face.")
(new-simple-scholium-face 6 plum1 "Sixth scholium face.")
```

7.4.3 Reference face

A special face for references. We don't currently have a special face for *visited* references, but this can be added if/when we start keeping track of which references have been visited (see Note). We might also want to record and display information about visited articles in general.

```
(defface sch-reference-face
  '((t (:foreground "red" :underline "red")))
  "Face for references in the scholium system.")
```

7.4.4 Color by number

The 'scholia-count' variable keeps track of how many scholia have been displayed. The 'scholium-face' function selects a face to use when displaying the next scholium according to this count. This simple display mechanism seems sort of lame (hence, it is turned on with the 'use-crazy-font-lock' variable); better things may come later, see, e.g. Note `2.2-Design_issues.org`.

Note that turning off "crazy font lock" makes it so that no text properties are added to the buffer, but of course it would be best if text properties were added and overlays omitted. On the other hand, we need to do some thinking to sort out the use of text properties versus the use of overlays. Cutting and pasting multiple copies of some markup in the same buffer may cause some trouble at commit time. See Section 9.2; also, compare Note .

```
(defvar scholia-count 0 "Number of scholia about the article found.")

(defun scholium-face ()
  (let ((short-count (mod scholia-count 6)))
    (cond ((eq short-count 0)
           'sch-face-1)
          ((eq short-count 1)
           'sch-face-2)
          ((eq short-count 2)
           'sch-face-3)
          ((eq short-count 3)
           'sch-face-4)
          ((eq short-count 4)
           'sch-face-5)
          ((eq short-count 5)
           'sch-face-6))))

(defvar use-crazy-font-lock t)
(defvar main-article-overlays nil)
(defvar scholia-overlays nil)
```

7.4.5 Access to rendering target buffer

The rendering target buffer is either "Main Article Display" if the current article isn't buffered, or whatever buffer the article lives in if it is buffered. (Well, actually, we can always redirect rendering to any buffer of our choice; but the preceding statement accurately describes the default operation.) We use 'get-rendering-target-buffer' as a shorthand when we grab the rendering target buffer, and 'set-buffer-to-rendering-target-buffer' to make that buffer current for editing. (Note that we *could* go about this in a slightly different way, namely set the 'rendering-target-buffer' variable to "Main Article Display" instead of 'nil' when there is nothing to override the default – but what would be the point?)

```
(defun get-rendering-target-buffer ()
  (get-buffer (or rendering-target-buffer
                  (get-buffer "Main Article Display"))))

(defun set-buffer-to-rendering-target-buffer ()
  (set-buffer (get-rendering-target-buffer)))
```

7.4.6 Experiment with many 'scholium' properties

All else equal, it might be advantageous to use independent 'scholia' properties instead of just one 'scholium' property (see Note 8.1.1).

Accordingly, I tried switching over to using one 'scholium' property for each marked-up region. These properties were given somewhat complicated names – namely, the link-ids that are currently stored as elements of the 'scholia' property (Note). However, this doesn't work, as suggested by the following example.

```
(progn (put-text-property (point) (1+ (point)) '(foo 1) t)
       (put-text-property (point) (1+ (point)) '(foo 1) nil)
       (text-properties-at (point)))
```

This is because[2]Thanks Andreas Schwab, help-gnu-emacs, 2005/12/12. }

```
(eq '(foo 1) '(foo 1)) ;=> nil
```

The Emacs text property engine is uniformly 'eq'-based instead of 'equal'-based. (So, if we happened to have an association between link-id's and integers, this would have worked, but that seems like a silly kludge.)

Indeed, this 'eq' preference is pretty much fatal to the utility of non-symbol text properties (unless we managed to hang directly onto the actual link-ids that would be used for the names of the text properties, which seems infeasible; or alternatively used some other sort of weird work-around, as above). A further difficulty is associated with the fact that these various 'scholium' text properties would be indiscriminately mixed in with any other text properties that happened to be stored at point, requiring filtering for any useful en masse processing.

I have some ideas that could possibly improve the prospects for doing away with the 'scholia' property and replacing it with several 'scholium' properties, but it seems like anything realistic would hacking Emacs C. But since it seems that there are no immediate problems associated with using just one 'scholia' property, we plunge ahead that way.

[2]{

7.4.7 On 'add-to-scholia-property-within-region'

For each character in the region between beg and end, this function grabs the 'scholia' property and replaces it with a version that has been modified to include the input value (a link-id; see Note).

The same property is used in the Scholia Display buffer, although that may be a bit of an abuse (cf. Note); something simpler would work for our needs there – but the current way is expedient. Also, it seems to be suggestive – perhaps in the future we'll be able to treat the Scholia Display buffer as a proper scholium-based article itself, presumably by using transclusion and identification, as appropriate (see Section).

```
(defun add-to-scholia-property-within-region (start end value)
  (while (< start end)
    (put-text-property start (1+ start) 'scholia
                       (add-to-or-start-list
                        (get-text-property start 'scholia)
                        value))
    (setq start (1+ start))))
```

7.4.8 On 'current-markup'

This variable will be used to record the regions of the article being displayed which have scholia attached to them and are, consequently, marked up. This information will be stored at markup time by 'mark-up-region' (Note 7.4.10). It is important to have this information on record so that we have something to compare to after editing takes place (see Section 8.3).

Its format is a list of elements of the form

```
((<name> <link number>) <beg> <end>)
```

where 'beg' and 'end' denote the beginning and end of the region marked up via the specified link (but see Note 7.4.9!).

```
(defvar current-markup nil)
```

7.4.9 Investigating 'current-markup' in the context of masks

If the link in question is being masked, then the format of 'current-markup' changes from the form described in Note 7.4.8 to the following:

```
((mask (<name> <link number>)) <beg> <end>)
```

where 'beg' and 'end' denote the beginning and ending of one particular region coming from the mask for the indicated link.

I'm not sure this is really sufficient information. Should we know *which part* of the mask we're dealing with? I.e., use something like a link-id, but for mask components?

7.4.10 On 'mark-up-region'

Like 'add-to-scholia-property-within-region', but also adds to 'current-markup' (see Note 7.4.8).

```
(defun mark-up-region (start end value)
  (add-to-scholia-property-within-region start end value)
  (setq current-markup
        (add-to-or-start-list current-markup
                              (list value start end))))
```

7.4.11 On 'non-printing-types'

This variable will record a list of the types of scholia that we typically don't want to print when displaying an article. The 'derives-from' type is an example of a scholium that we typically don't want to print; see Note 9.3.1.

User should temporarily set 'non-printing-types' to 'nil' before 'mark-things-up' runs whenever they wish to display all scholia.

```
(defvar non-printing-types nil)
```

7.4.12 Masks

In light of the comments in Note , it is possible to think of a mask as a map that takes a given link to to a region, and replaces it with a list of several regions.

The format of a mask, then, is

```
((<name> <link number>) &rest regions)
```

where the regions are pairs (two-element lists), each giving the beginning and end of a region that the link is being redirected to.

This format lacks generality! Certainly a link that is pointing to something other than a region may need to be redirected, for example, a link pointing at a page, when the page gets deleted. For the time being, these interesting cases can be dealt with through *ad hoc* measures. Eventually we'll want to make a study of all of the different redirection cases; but getting the platform working reasonably well takes priority. (See also Note .)

Finally, observe that the format used here is related to the format of the 'current-markup' variable (Note 7.4.8).

For an interesting meditation on masking perceptions, see Lem[3].

7.4.13 On 'compute-usable-about-data'

This function runs within the scope of 'mark-things-up' (Note 7.4.18). Its role is to identify the regions to be marked up, based on the about data expressed by attached scholia, and augmented by any masks associated with this data. In short, this is the mask-applying part of the algorithm described in Note .

The way it works is as follows. We look through all of the links ({although we really don't need to do this anymore, given that we have specifically identified the relevant links by using link-id'ed backlinks in the article being displayed; the change would have to come in at the level of 'mark-things-up' or higher}) – and if the link is relevant, then it will be reflected in the value returned by 'compute-usable-about-data'. However, if the link is masked, it will be the mask that is reflected, instead of the link itself.

Recall that we can't 'assoc' the link across the masks; we need to 'assoc' the link-id instead.

The format of the return value is kind of ugly; we could almost certainly do without the extra 'list' layer.

Note that in the return value, the name of the *scholium* is supplied, not the name of the linked-to article – we'll subsequently know (when applying markup) that all of the links apply to the linked-to article, whereas we won't know where these links are coming from unless we record that specially.

```
(defun compute-usable-about-data ()
  (let (usable-data
        (about (scholium-about scholium))
        (masks (get-metadata-field 'masks
                                   (scholium-name
```

[3]Stanislaw Lem, The Futurological Congress

```
                                    scholium)))
      (link-number 0))
  (dolist (link about)
    (setq link-number (1+ link-number))
    (when (link-about-article-p link name-of-current-article)
      (let* ((link-id (list (scholium-name scholium) link-number))
             (mask (assoc link-id masks)))
        (if mask
            (dolist (reg (cdr mask))
              (setq usable-data
                    (add-to-or-start-list
                     usable-data
                     `((mask ,(car mask)) ,@reg))))
          (setq usable-data
                (add-to-or-start-list
                 usable-data
                 (append (list (list (scholium-name scholium)
                                     link-number))
                         (let ((beg (link-beginning link)))
                           (if beg
                               (list beg (link-end link))
                             (list nil)))))))))))
  usable-data))
```

7.4.14 On 'mark-things-up-customizations'

This may not have to be used.

```
(defvar mark-things-up-customizations nil)
```

7.4.15 On 'scholia-display-pre-update-hook'

This gives us a chance to do various customizations to the environment before the main part of 'mark-things-up' runs.

I imagine that the role that it will play will be similar to the one played by the functions that run right in 'pre-mark-up', namely to zap variables and settings that we don't want to have around anymore.

```
(defvar scholia-display-pre-update-hook nil)
```

7.4.16 On 'pre-mark-up'

This function generally zaps things, to prepare for markup.

```
(defun pre-mark-up ()
  (setq current-markup nil)
  (save-excursion
    (set-buffer-to-rendering-target-buffer)
    (remove-list-of-text-properties (point-min)
                                    (point-max)
                                    '(scholia)))
  (mapcar #'delete-overlay main-article-overlays)
  (setq main-article-overlays nil)
  (setq scholia-count 0))
```

7.4.17 On 'mark-things-up-hook'

For unconditional customizations to 'mark-things-up'. The functions added to this hook run after everything else 'mark-things-up' does is done.

```
(defvar mark-things-up-hook nil)
```

7.4.18 On 'mark-things-up'

This function assembles text, text properties, and overlays: it is the main rendering engine, called by 'display-article' to get scholia and appropriate markup onto the screen. It is important that 'raw-scholia' be defined (and be a list of scholia) for this function to work properly.

The first part of the plot is to delete the old markup; we call 'pre-mark-up' to take care of this.

A list of "raw scholia" is expected to be present in the context in which this function runs; typically this data is provided by 'display-article' (see Note 7.5.13), but it can be supplied by any stand-in (e.g. 'display-scholia-about-current-buffer' of Note 7.5.16).

The raw scholia are first sorted using 'sort-scholia-by-beg-position' (Note 7.3.2) and then translated into a usable form by 'compute-usable-about-data' (Note 7.4.13).

The function 'mark-things-up' can handle different sorts of scholia differently (e.g., references are marked up in the conventional way); see Note .

In order to *selectively* display scholia, the 'raw-scholia' variable should be modified before this function runs.

```
(defun mark-things-up ()
  (pre-mark-up)
  (let ((scholia (sort-scholia-by-beg-position
                  raw-scholia
                  name-of-current-article)))
    (dolist (scholium scholia)
      (unless (typedata-includes-element-of-list
               (scholium-type scholium)
               non-printing-types)
        (let ((usable-about-data (compute-usable-about-data))
              (current-position-in-scholia-display (point)))
          (cond
           ((and
             (typedata-includes (scholium-type scholium) 'reference)
             (equal (reference-from-article scholium)
                    name-of-current-article))
            (mark-up-reference))
           (t
            (mark-up-scholium)))))))
  (run-hooks 'mark-things-up-hook))
```

7.4.19 Functions for adding overlays

These function run within the scope of 'mark-up-scholium' to add overlays to the display.

```
(defun add-overlays-in-scholia-display-buffer ()
  (setq scholia-overlays
        (cons
         (make-overlay current-position-in-scholia-display
```

```
                        (point)
                        (get-buffer "Scholia Display")
                        t)
         scholia-overlays))
  (overlay-put (car scholia-overlays)
               'face (scholium-face)))

(defun add-overlays-in-rendering-target-buffer ()
  (setq main-article-overlays
        (cons
         (make-overlay (second elt)
                       (third elt)
                       (get-rendering-target-buffer)
                       t)
         main-article-overlays))
  (overlay-put (car main-article-overlays)
               'face (scholium-face)))
```

7.4.20 On 'mark-up-reference'

We call this function from 'mark-things-up' (Note 7.4.18) to render a scholium if it has reference type and its about data indicates that that the reference originates from (i.e., appears in) the current article. See Note 6.4.18 for a description of the format of the return value of the function 'reference-source-link' used here.

References should perhaps be rendered differently depending on their sub-type (in particular, a different face could be used for references with different sub-types).

(Do we really want to loop through all of the elements of 'usable-about-data'? Mightn't there be something there corresponding to the linked-to article?)

```
(defun mark-up-reference ()
  (save-excursion
    (set-buffer-to-rendering-target-buffer)
    (dolist (elt usable-about-data)
      (mark-up-region (second elt)
                      (third elt)
                      (first elt))
      (when use-crazy-font-lock
        (setq main-article-overlays
              (cons
               (make-overlay (second elt)
                             (third elt)
                             (get-rendering-target-buffer)
                             t)
               main-article-overlays))
        (overlay-put (car main-article-overlays)
                     'face 'sch-reference-face)))))
```

7.4.21 On 'mark-up-scholium'

This inserts scholia and marks them up, together with the marked up regions (when these exist; the function does both whole-article scholia and region-specific scholia).

The ‘scholia-count’ variable is set for purposes of face selection; see Note 7.4.4.

```
(defun mark-up-scholium ()
  ;; this part takes place in the scholia display buffer
  (insert (sch-plain-text (scholium-name scholium)))
  (add-to-scholia-property-within-region
   current-position-in-scholia-display
   (point)
   ;; add a list to make it possible to reuse the scholium property
   (list (scholium-name scholium)))
  (when use-crazy-font-lock
    (add-overlays-in-scholia-display-buffer))
  (insert "\n\n")
  ;; this part is relevant to the buffer containing the main article
  (dolist (elt usable-about-data)
    (when (second elt)
      (save-excursion
        (set-buffer-to-rendering-target-buffer)
        (mark-up-region (second elt)
                        (third elt)
                        (first elt))
        (when use-crazy-font-lock
          (add-overlays-in-rendering-target-buffer)))))
  ;; adjust count once everything else is done here, so same count
  ;; applies in both buffers (useful for coloration purposes)
  (setq scholia-count (1+ scholia-count)))
```

7.5 Display interface

7.5.1 On ‘rendering-target-buffer’

If a given name is associated with a buffered article, then that buffer will be where the article is displayed. If the article to be displayed is in a buffer, this variable will be set that buffer object. Otherwise it will be set to nil.

```
(defvar rendering-target-buffer nil)
```

7.5.2 On ‘scholia-display-post-update-hook’

This provides a way to customize the article (and scholia) being displayed.

```
(defvar scholia-display-post-update-hook nil)

(add-to-list 'non-printing-types 'fake)
```

7.5.3 On ‘scholia-display-extras’

This is used to add some extra stuff to the Scholia Display for purposes of navigation and establishing context. It is called by ‘display-article’ (Note 7.5.13) and ‘display-scholia-about-current-buffer’ (Note 7.5.16).

It might be nice to have this in the main article window, but that could also be confusing, especialy if the main article is associated with some buffer; so I’m taking the route of caution here. Also note that

according to the principle of order 2.2-Design_issues.org this stuff should probably appear much later on in the document, perhaps in Section 7.10.

It might be more appropriate to have some of these extra features display in or above the main article buffer; the info system has a nice un-editable bar for display of various navigational data.

```
(defun scholia-display-extras ()
  ;; this setting is useful for presentations.
  (goto-char (point-min))
  ;; Be careful that this is ignored when the article is saved or
  ;; otherwise processed internally.
  (insert
   "Title: "
   (propertize (format "%s" name-of-current-article) 'face 'italic)
   "\n\n")
  (goto-char (point-max))
  (run-hooks 'scholia-display-extras-hook)
  (goto-char (point-min)))
```

7.5.4 On 'display-style'

As discussed in Section

there are a number of different styles we'd like to offer users to choose between when they go to display an article. For now, the relevant settings for the display style variable are 'plain' (the default, displaying all scholia) and 'contextual' (which causes only those scholia associated with the region being displayed to appear; see Note 2.2-Design_issues.org).

There are a few problems here: if we go with the contextual display, what about scholia that apply to the article as a whole? That's a little tricky. I guess for now we just leave them out?

Set this variable with 'set-display-style', not manually.

```
(defvar display-style 'plain)
```

7.5.5 On 'window-displayed-substring'

This variable will hold the string that is being shown through a given window at a given point in time.

```
(defvar window-displayed-substring nil)
```

7.5.6 On 'set-display-style'

Use this function to control the setting of 'display-style' (Note 7.5.4). Relevant arguments are 'plain', or 'contextual'.

(Actually, anything but 'contextual' will set 'display-style' to its default setting. Eventually, we might want to be able to have contextual display together with some other non-default features, at which point we'll have to adjust this function appropriately.)

```
(defun set-display-style (style)
  (cond ((eq style 'contextual)
         ;; we may need a "double hook" here, so we get the local
         ;; hooks set up in the correct window
         (add-hook 'display-article-hook
                   'initiate-contextual-updating)
         (setq display-style 'contextual))
```

```
        (t
         (remove-hook 'display-article-hook
                      'initiate-contextual-updating)
         (setq display-style 'plain))))
```

7.5.7 On 'initiate-contextual-updating'

This function gets the rendering target buffer set up to do live-updating of scholia.

For now, this function doesn't do anything for labels – since for now, labels typically don't have scholia on their text. However, this can be undone later, if necessary.

In addition, we don't yet have any code for getting rid of contextual updating in an individual buffer OTF.

```
(remove-hook 'window-scroll-functions
             'set-window-displayed-substring t)

(defun initiate-contextual-updating ()
  (unless (typedata-includes (scholium-type article) 'label)
    (save-excursion (set-buffer rendering-target-buffer)
                    (add-hook 'window-scroll-functions
                              'maybe-update-scholia-display nil t))))
```

7.5.8 On 'maybe-update-scholia-display'

In order for this to work optimally, we'd might to keep track of a list of the current scholia that are being displayed. But for now, I suppose (contrary to this function's name) we can just redisplay all the scholia we encounter every time.

```
(defun maybe-update-scholia-display ()
  (set-window-displayed-substring)
  ;; this is just a ridiculous thing to run -- for testing purposes
  ;; only.  Eventually, we'll actually want to analyse the text,
  ;; figure out which scholia are relevant, and update the scholia
  ;; display.
  (save-excursion (set-buffer (get-buffer-create "*scratch*"))
                  (erase-buffer)
                  (insert window-displayed-substring)))
```

7.5.9 On 'set-window-displayed-substring'

This function will be an element of the 'window-scroll-functions' hook when the display mode is set to 'contextual'.

```
(defun set-window-displayed-substring ()
  (setq window-displayed-substring
        (buffer-substring (window-start)
                          (window-end))))
```

7.5.10 Displaying articles

7.5.11 On 'display-article-hook'

We will later do some "interesting" things with this function, so we add a hook. For example, one use for this hook is to maintain a history of articles that have been displayed; see Section 7.9.

```
(defvar display-article-hook nil)
```

7.5.12 On 'raw-scholia-selector'

This function is here both to select raw scholia for display when 'display-article' runs, and to provide for a choice between different ways of selecting raw scholia. It returns a list of scholia.

This function is set up to run within the scope of 'display-article' (Note 7.5.13).

We select slightly different scholia when the article that is being displayed is a label; in particular, we don't want to display scholia that indicate the current article as a 'parent', since these scholia will already be listed in the label's text itself.

If we're trying to display scholia contextually, then this function should probably be doctored with some, to make it so that the initial set of scholia that are displayed are contextually appropriate, i.e., are just those associated with the on-screen portion of the buffer that is being displayed.

```
(defun raw-scholia-selector (&optional what-is-displayed)
  (cond ((eq what-is-displayed 'label)
         (remove-if
          (lambda (scholium)
            (member-if (lambda (link)
                         (member 'parent (cdr link)))
                       (scholium-about scholium)))
          (mapcar (lambda (backlink)
                    (get-article (car backlink)))
                  (get-backlinks name-of-current-article))))
        (t (mapcar (lambda (backlink)
                     (get-article (car backlink)))
                   (get-backlinks name-of-current-article)))))

;; (eq display-style 'contextual)
;;          (save-excursion (set-buffer (get-buffer-create "*scratch*"))
;;                          (erase-buffer)
;;                          (insert window-displayed-substring))
;;          nil
```

7.5.13 On 'display-article'

Display article found via path (which can just be the name of an article in the main article tabel), if said article exists. If the article lives in a buffer, that buffer will be where the article is displayed; otherwise the article is displayed in the "Main Article Display" buffer. Note that this function can also be used to display labels (it calls 'display-label'; but see Note , since some other approach might be valuable sometimes).

I don't want to have all of the children appear as scholia when a label is browsed, at least not by default; but it is kind of neat to know that they can be made to appear; if we don't do the 'remove-if' then the scholia display will be the OTF-assembled compilation mentioned in Note .

It makes more sense to always run 'display-article-hook' after the 'cond', and to put things that are conditional upon being in this specific branch into 'scholia-display-post-update-hook' or, barring that, some additional branch-specific hook. This of course means that the parent will be added to the history list if that is where we browse from; this is intended.

If it turns out to be needed here, we could reuse the trick of running a hook and then making a test before the 'cond', which we're familiar with from e.g. 'sch-plain-text' (Note 7.1.2).

It would be good to make the buffer disposition (left? right?) is consistent when we use this function to display labels; I think we have it sorted out properly for the display of "normal" articles.

We may want to treat namespaces and labels together in the same 'cond' branch (currently namespaces aren't handled specially by this function).

Since we are now reading in paths, these paths have to be parsed. This is the job of 'generalized-get-article' (see Note 3.4.6).

```
(defun display-article (path)
  (interactive (list (read-article-path)))
  (let* ((article (generalized-get-article path))
         (name (scholium-name article)))
    (if (not article)
        (error "No article by that name found")
      (setq name-of-current-article name)
      (cond
       ((typedata-includes (scholium-type article) 'label)
        (display-label name)
        (setq rendering-target-buffer "*Generic List*")
        (switch-to-buffer (get-buffer "*Generic List*"))
        (setq buffer-read-only nil)
        (switch-to-buffer (get-buffer-create "Scholia Display") t)
        (erase-buffer)
        (let ((raw-scholia (raw-scholia-selector 'label)))
          (mark-things-up))
        (scholia-display-extras)
        (switch-to-buffer (get-buffer "*Generic List*"))
        (setq buffer-read-only t)
        (pop-to-buffer "Scholia Display")
        (other-window -1))
       (t
        (if (article-buffered article)
            (setq rendering-target-buffer (scholium-text article))
          (setq rendering-target-buffer nil))
        (run-hooks 'scholia-display-pre-update-hook)
        (scholia-overwhelm-display (sch-plain-text article))
        (let ((raw-scholia (raw-scholia-selector)))
          (mark-things-up))
        (scholia-display-extras)
        (pop-to-buffer (get-rendering-target-buffer))
        (run-hooks 'scholia-display-post-update-hook)))
      (run-hooks 'display-article-hook))))
```

7.5.14 On 'redisplay-article'

This accomplishes a simple task.

It would be nice if we could get the point restored to its original position after this thing runs.

Also, it would be good to have the function run automatically after scholia have been added about the document.

```
(defun redisplay-article ()
  (interactive)
```

```
  (display-article name-of-current-article))
```

7.5.15 Displaying scholia about a given buffer

7.5.16 On 'display-scholia-about-current-buffer'

This function is similar to 'display-article' (7.5.13), but it works directly on the buffer level. (Stylistically this function should probably just be a thin wrapper, but we're running with it for now.)

This function should perhaps check to see whether the current buffer has been *edited* since the last time this function (or similar, through other means) was executed. If the buffer has been edited, the user should probably be prompted, and asked whether to reparse (Section 8.2) before redisplaying.

```
(defun display-scholia-about-current-buffer ()
  (interactive)
  (let ((article (get-article (buffer-name (current-buffer)))))
    (if article
        (progn
          (setq rendering-target-buffer (current-buffer))
          (run-hooks 'scholia-display-pre-update-hook)
          (scholia-overwhelm-display (sch-plain-text article))
          (let ((raw-scholia
                 (mapcar (lambda (backlink)
                           (get-article (car backlink)))
                         (get-backlinks name-of-current-article))))
            (mark-things-up))
          (scholia-display-extras)
          (pop-to-buffer (get-rendering-target-buffer))
          (run-hooks 'scholia-display-post-update-hook)
          ;; weird!
          (run-hooks 'display-article-hook))
      (call-if-user-adds-current-buffer-to-article-list
       'display-scholia-about-current-buffer))))
```

7.5.17 On 'display-label-hook'

This provides a way to customize the behavior of 'display-label'.

```
(defvar display-label-hook nil)
```

7.5.18 On 'currently-displayed-label'

This gives us a handle on the most recently displayed label. This facilitates only the simplest improvement to reverting behavior.

```
(defvar currently-displayed-label nil)
```

7.5.19 On 'display-label'

The function 'display-label' uses the catalog browsing feature of Section 7.8. If we want to do other interesting rendering things with articles that have special types, we can follow the usage of 'display-label' in 'display-article'. It might be good for this function to run its own hook, e.g., for maintaining a special history (see).

```
(defun display-label (name)
  (interactive
   (list
    (let* ((completion-ignore-case t)
           (label-names
            (scholium-text (get-article 'label)))
           (label-strings (mapcar (lambda (name)
                                    (format "%s" name))
                                  label-names))
           (string-found (completing-read
                          "Label: "
                          label-strings))
           (place (- (length label-strings)
                     (length (member string-found label-strings)))))
      (nth place label-names))))
  (article-menu-listing (label-to-propertized-list name))
  (run-hooks 'display-label-hook))

(add-hook 'display-label-hook (lambda ()
                                (setq currently-displayed-label name)))

(defalias 'list-label 'display-label)
```

7.5.20 On 'display-intersection-of-labels'

For displaying everything bearing every one of the input labels.

```
(defun display-intersection-of-labels (&rest labels)
  (let ((intersection (label-to-list (car labels)))
        (ctn (cdr labels)))
    (while ctn
      (setq intersection (intersection intersection
                                       (label-to-list (car ctn))
                                       :test 'equal))
      (setq ctn (cdr ctn)))
    (article-menu-listing (turn-list-into-propertized-list
                           intersection))))
```

7.5.21 On 'display-difference-of-labels'

For purposes of simplicity, this is set up to work with two labels only, for the time being.

```
(defun display-difference-of-labels (label-A label-B)
  (article-menu-listing (turn-list-into-propertized-list
                         (set-difference (label-to-list label-A)
                                         (label-to-list label-B)))))
```

7.5.22 On 'find-marked-regions'

This function is used in the subsequent section (Section
to locate the parts of the buffer that have scholia written about them. Note that this scheme might have been outmoded by stuff in Section 8.2.

```
(defun find-marked-regions ()
  (let (names-and-positions
        (next-change-point (point-min)))
    (while next-change-point
      (let ((next-region (find-next-marked-region)))
        (when next-region
          (setq names-and-positions
                (cons next-region
                      names-and-positions)))))
    names-and-positions))
```

7.5.23 On 'find-next-marked-region'

This function is within the scope of 'find-marked-regions'. Should be able to find all the regions associated with any scholium. Right now, this function is working in a simplified universe in which scholia and regions are mapped to each other in 1-1 way!

```
(defun find-next-marked-region ()
  (let* ((beg (next-single-property-change next-change-point
                                           'scholia))
         (end (when beg
                (next-single-property-change beg
                                             'scholia))))
    (setq next-change-point end)
    (when end
      (list
       (get-text-property beg 'scholia)
       (list (cons beg end))))))
```

7.5.24 Turning overlays off and on

7.5.25 Turning overlays off

It is easy enough to turn overlays off; this is accomplished for the main article buffer and the Scholia Display buffer by 'sch-turn-main-article-overlays-off' and 'sch-turn-scholia-overlays-off', respectively.

```
(defun sch-turn-main-article-overlays-off ()
  (interactive)
  (mapcar #'delete-overlay main-article-overlays)
  (setq main-article-overlays nil))

(defun sch-turn-scholia-overlays-off ()
  (interactive)
  (mapcar #'delete-overlay scholia-overlays)
  (setq scholia-overlays nil))
```

7.5.26 Turning overlays on

Turning overlays on is a bit trickier. In order to be able to turn scholia on, we need to be able to find all the regions that have scholia attached to them. This is accomplished (for the main article buffer only, I think) by 'find-marked-regions'.

```
(defun sch-turn-main-article-overlays-on ()
  (interactive)
  ;; to save from potential overlap weirdness
  (sch-turn-main-article-overlays-off)
  (save-excursion
    (let ((names-and-positions (find-marked-regions)))
        (dolist (info names-and-positions)
    (let* ((name (car info))
           (marked-regions (cadr info))
           (scholium (get-article name))
           (beg (point)))
      (save-excursion
        (set-buffer-to-rendering-target-buffer)
        (dolist (reg marked-regions)
          ;; add "transient" overlay
          (when use-crazy-font-lock
            (setq main-article-overlays
                  (cons
                   (make-overlay (car reg)
                                 (cdr reg)
                                 (get-rendering-target-buffer)
                                 t)
                   main-article-overlays))
            ;; using `scholium-face' here is a bit weird
            (overlay-put (car main-article-overlays)
                         'face (scholium-face)))))))))

(defun sch-turn-scholia-overlays-on ()
  (interactive)
  )
```

7.6 Scholia browsing

7.6.1 On 'move-to-next-region-with-scholium' and 'move-to-previous-region-with-scholium'

This moves the point to the beginning of the next region that has a scholium about it (if there is one).

(I'm noticing a bug when the function is used interactively with the binding selected in Section 11.1 and the cursor is positioned on a right paren; calling the function with M-x in this case doesn't result in the same problem.)

Note these two functions are not quite symmetrical, because we want the cursor to end up at the beginning of the marked region. (I think there will be a problem if we try to go to the beginning of a scholium that is attached at the beginning of the article, but that isn't such a big deal.)

```
(defun move-to-next-region-with-scholium ()
  (interactive)
  (let ((change (next-overlay-change (point))))
    (if (overlays-at change)
        (progn (goto-char change)
               (list change (next-overlay-change change)))
      (if (overlays-at (next-overlay-change change))
          (progn (goto-char (next-overlay-change change))
```

```
                (list (next-overlay-change change)
                      (next-overlay-change (next-overlay-change
                                            change))))
        (message "No subsequent regions with scholia about them.")
        nil))))

(defun move-to-previous-region-with-scholium ()
  (interactive)
  (let ((change (previous-overlay-change (point))))
    (if (overlays-at change)
        (progn (goto-char change)
               (list change (previous-overlay-change change)))
      (if (overlays-at (previous-overlay-change change))
          (progn (goto-char (previous-overlay-change change))
                 (list (previous-overlay-change change)
                       (previous-overlay-change (previous-overlay-change
                                                 change))))
        (message "No previous regions with scholia about them.")
        nil))))

(defun scroll-article-display-to-next-region-for-current-scholium ()
  (interactive)
  (save-excursion
    (set-buffer buffer-associated-with-current-article)
    (let (found
          (curpoint (point)))
      (while (and (not found)
                  (not (eobp))
                  (move-to-next-region-with-scholium))
        (mapc (lambda (overlay)
                (when (equal (overlay-get overlay 'scholia)
                             (name-of-current-scholium))
                  (setq found t)))
              (overlays-at (point)))
        (if found
            (recenter)))
      (when (not found)
        (goto-char curpoint)
        (message "Scholium not about further regions in buffer."))))))

(defun scroll-article-display-to-previous-region-for-current-scholium ()
  (interactive)
  (save-excursion
    (set-buffer buffer-associated-with-current-article)
    (let (found
          (curpoint (point)))
      (while (and (not found)
                  (not (bobp))
                  (move-to-previous-region-with-scholium))
        (mapc (lambda (overlay)
```

```
            (when (equal (overlay-get overlay 'scholia)
                         (name-of-current-scholium))
              (setq found t)))
          (overlays-at (point)))
    (if found
        (recenter)))
  (when (not found)
    (goto-char curpoint)
    (message "Scholium not about previous regions in buffer."))))
```

7.6.2 On 'move-to-first-region-for-scholium'

Move you to the beginning of the region marked up by the scholium named name. Should this be made interactive?) At present, it is only called by 'move-to-first-region-for-current-scholium'.

```
(defun move-to-first-region-for-scholium (name)
  (pop-to-buffer (get-buffer rendering-target-buffer))
  (let ((beg (point-max))
        (about (scholium-about (get-article name))))
    (dolist (link about)
      ;; this should be revised in light of the
      ;; fact that a link can be multiply typed
      (when (and (typedata-includes-passage (link-type elt))
                 (equal (linked-to-article link)
                        name-of-current-article)
                 (< (link-beginning link) beg))
        (setq beg (link-beginning link))))
    (unless (equal beg (point-max))
      (goto-char beg))))
```

7.6.3 On 'move-to-first-region-for-current-scholium'

This uses the function 'move-to-first-region-for-scholium' from section 7.6; the thought behind including the function here is that it establishes a relationship between the Scholia Display buffer and the main article buffer (however it could probably go in section 7.6 equally well).

This should probably be complemented by a function 'move-to-last-region-for-current-scholium'.

Also, it should probably have some intelligent message (not to say "error message") if the scholium applies to the article as a whole.

(Gives some error, complaining about 'elt' being void. Can this run in the Scholia Display buffer as well as the main article buffer?)

```
(defun move-to-first-region-for-current-scholium ()
  (interactive)
  (let ((current (name-of-current-scholium)))
    (move-to-first-region-for-scholium current)))
```

7.7 Local browsing

7.7.1 On 'read-scholia-property-at-point'

Suppose we simply want to make the current scholium into the new current article. That's what the next function is for.

```
(defun read-scholia-property-at-point ()
  (get-text-property (point) 'scholia))
```

7.7.2 On 'scholia-named-at-point'

I think that this should strip out the "mask" tags from the link-ids, but leave the name parts. Since it is only used by interactive functions, this seems fine, and appropriate.

```
(defun scholia-named-at-point ()
  (mapcar (lambda (id)
            (if (eq (car id) 'mask)
                (car (second id))
              (car id)))
          (read-scholia-property-at-point)))
```

7.7.3 On 'follow-scholium'

This causes the current scholium to become the current article.

Eventually we'll want to be able to run this command with a mouse-click.

```
(defun follow-scholium ()
  (interactive)
  (let ((current (name-of-current-scholium)))
    (when current
      (display-article current))))
```

7.7.4 On 'follow-reference'

This reads the 'scholia' property at point and either follows the reference at point (if there is only one) or allows the user to choose between references (if there are several).

We might want to provide an additional function for following links *in general*; basically the strategy for that is, just don't do the 'remove-if' filtering.

```
(defun follow-reference ()
  (interactive)
  (let ((references
         (remove-if (lambda (name)
                      (not (typedata-includes
                            (scholium-type (get-article name))
                            'reference)))
                    (scholia-named-at-point))))
    (cond
     ((equal (length references) 1)
      (let* ((ref (get-article (car references)))
             (to-article (reference-to-article ref)))
        (if (equal to-article name-of-current-article)
            (display-article (reference-from-article ref))
          (display-article to-article)))
      ;; maybe `display-article' should be returning
      ;; some non-`nil' value so that we don't have to do this.
      t)
     (references
```

```
      ;; this sort of disambiguation is really only needed if the
      ;; references have different targets.  Two distinct references
      ;; to the same thing overlaying each other could be treated as
      ;; one for simple following purposes.
      (list-articles references))
     (t
      (message "No reference at point.")
      nil))))

(defun follow-reference-or-scholium ()
  (interactive)
  (unless (follow-reference)
    (follow-scholium)))

(defun display-an-article-that-current-article-is-about ()
  (interactive)
  (let ((abouts (scholium-about
                 (get-article name-of-current-article))))
    (cond
     ((equal (length abouts) 1)
      (display-article (car abouts)))
     (abouts
      (list-articles abouts))
     (t
      (message "Article isn't about any other articles.")))))
```

7.7.5 On 'current-scholium-is-about'

This function is similar to the previous one, but it applies to scholia. Since one presumably knows that the current scholium is about the current article, this is most useful when a scholium is about several different articles, as it allows the user to move "down" to any of them.

```
(defun current-scholium-is-about ()
  (scholium-about (get-article (name-of-current-scholium))))

(defun display-an-article-that-current-scholium-is-about ()
  (interactive)
  (let ((abouts (current-scholium-is-about)))
    (cond
     ((equal (length abouts) 1)
      (display-article (car abouts))
      (message "Note: scholium is only about current article."))
     (abouts
      (list-articles abouts))
     (t
      (message "Article isn't about any other articles.")))))
```

7.8 Catalog browsing

7.8.1 On 'make-generic-menu-mode'

This takes the name of the menu (as a space-separated string) together with a list of bindings to be used in that particular menu mode.

This should provide a docstring for the mode it creates.

```
(defmacro make-generic-menu-mode
  (mode bindings)
  (let* ((modedash (downcase (replace-regexp-in-string " " "-" mode)))
         (modesymbol (intern (concat modedash "-mode")))
         (mapname (intern (concat modedash "-map"))))
    `(progn
       (defvar ,mapname)
       (setq ,mapname (make-keymap))
       (suppress-keymap ,mapname t)
       (dolist (binding ,bindings)
         (define-key ,mapname (car binding) (cdr binding)))
       (defun ,modesymbol ()
         (kill-all-local-variables)
         (use-local-map ,mapname)
         (setq major-mode (quote ,modesymbol))
         (setq mode-name ,mode)
         (setq truncate-lines t)
         (setq buffer-read-only t)))))
```

7.8.2 On 'generic-menu-noselect'

The basic idea of this is that we have some objects and some functions to map across the objects to extract the information from the objects (accessors). The functions must be set up to produce strings as their output. The functions correspond to columns in the display; individual objects correspond to rows. This is the same idea no matter what the source of the objects is. (In particular, it might be a good idea for a later version of this function to accept either a hash table or a list as the source of the objects; see Note .) Note that columns of the display are assumed to be as wide as their widest item!

```
(defun generic-menu-noselect (objects accessors)
  (let (cols)
    (dolist (get-this accessors)
      ;; if we built this front-to-back rather than back-to-front,
      ;; that would be better
      (setq cols (cons
                  (cons (car get-this)
                        (mapcar (cdr get-this) objects))
                  cols)))
    ;; find the width of the columns.
    (let ((lens (mapcar (lambda (col)
                          (let ((len 0))
                            (dolist (str col)
                              (let ((candidate (length str)))
                                (when (> candidate len)
                                  (setq len candidate))))
```

```
                        len))
                    cols)))
    (with-current-buffer (get-buffer-create "*Generic List*")
      (setq buffer-read-only nil)
      (erase-buffer)
      (while cols
        (goto-char (point-min))
        (goto-char (line-end-position))
        (dolist (str (car cols))
          (insert str " ")
          ;; fill with spaces to make up for lost space
          (insert-char 32 (- (car lens) (length str)))
          (unless (equal (forward-line) 0)
            (newline))
          (goto-char (line-end-position)))
        (setq cols (cdr cols))
        (setq lens (cdr lens)))
      (goto-char (point-min))
      (current-buffer)))))
```

7.8.3 Improved sorting of article listing

Sorting could be done using autocompletion on the name of the column to sort on, too, which would be kind of nice. Or we could make a command to sort on the current column, or reorder columns according to the values on the current line or just about anything you might like.

But I don't think sorting is going to work at all until we have a consistent way of identifying the fields to sort; in the case of strings with spaces in them, 'sort-fields' won't work. Since we compute the width of each of the columns in 'generic-menu-noselect', if we were to store this info (perhaps as a text property attached to each column heading), we could adroitly divide the text up to find the strings we're trying to sort. See Note .

```
(defun Generic-menu-sort (col)
  (interactive "P")
  (save-excursion
    (sort-fields (or col 1) (progn (goto-line 2)
                                   (point))
                 (point-max))))
```

7.8.4 On 'standard-article-menu-accessors'

Note that it would be easy to provide more metadata – just revise this variable with additional fields as desired. See Note 7.8.2 for a description of how accessors work.

```
(defvar standard-article-menu-accessors
  '(("Name" . identity)
    ("C"    . (lambda (elt) " "))))
```

7.8.5 On 'article-menu-listing-hook'

This is here to customize the behavior of the article menu listing. Currently it is used to offset the activities invoked by 'currently-displayed-label-hook'; whereas we want 'currently-displayed-label' to be

defined when the listing is used to display a label, we'd rather it be 'nil' when something other than a label has been displayed, since anything else could be misleading.

```
(defvar article-menu-listing-hook nil)
```

7.8.6 On 'article-menu-listing'

The optional input subset is a list of article names to pump into the generic menu; it defaults to the list of "plain" articles as recorded on the corresponding label. The optional input accessors specifies the functions to use to extract information from the articles named by subset; it is in the format of, and defaults to, 'standard-article-menu-accessors'. (Note that function plays a similar role to 'Buffer-menu-revert' from buff-menu.el.)

```
(defun article-menu-listing (&optional subset accessors)
  (interactive)
  (pop-to-buffer
   (generic-menu-noselect
    ;; maybe this should always handle propertizing itself?
    (or subset
        (label-to-propertized-list 'plain))
    (or accessors
        standard-article-menu-accessors)))
  ;; note, this runs every time, even if the current article
  ;; isn't on the list.
  (article-menu-point-out-current-article)
  (article-menu-mode)
  (run-hooks 'article-menu-listing-hook))

(add-hook 'article-menu-listing-hook (lambda ()
                                       (setq currently-displayed-label
                                             nil)))
```

7.8.7 On 'turn-article-table-into-list'

I hate to actually use this function the way it is used in 'display-article'… probably we should just select from the *plain* articles there, and write a separate function to display other articles. In short, there really shouldn't be any need to use this function, except maybe for debugging purposes or *explicit* listing of all the articles (whenever that really needs to be done).

```
(defun turn-article-table-into-list ()
  (let ((names (list t)))
    (maphash (lambda (name value)
               ;; It might be nice to have %S here, but
               ;; I don't know if it would be _useful_
               (nconc names (list (format "%s" name))))
             article-table)
    (cdr names)))
```

7.8.8 On 'turn-article-table-into-names'

This is a variant of 'turn-article-table-into-list' that produces the actual names of the articles; probably it should supercede the other, since we could get the print names by mapping over the output of this function, obviously.

```
(defun turn-article-table-into-names ()
  (let ((names (list t)))
    (maphash (lambda (name value)
               (nconc names (list name)))
             article-table)
    (cdr names)))
```

7.8.9 On 'turn-article-table-into-propertized-list'

This combines the best of both 'turn-article-table-into-list' and 'turn-article-table-into-names'. Compare 'label-to-propertized-list'.

```
(defun turn-article-table-into-propertized-list ()
  (let ((names (list t)))
    (maphash (lambda (name value)
               (nconc names
                      (list
                       (propertize (format "%s" name) 'name name))))
             article-table)
    (cdr names)))
```

7.8.10 On 'turn-list-into-propertized-list'

Turns an arbitrary list into a propertized list.

```
(defun turn-list-into-propertized-list (lis)
  (let (names)
    (mapc (lambda (name)
            (setq names (cons
                         (propertize (format "%s" name) 'name name)
                         names)))
          lis)
    names))
```

7.8.11 Special-purpose listings

Here are a few functions to list special collections of articles. The function 'article-menu-list-labels' is perhaps particularly noteworthy; browsing labels seems to be a powerful way of organizing and retrieving information, see Note .

```
(defun article-menu-list-plain-articles ()
  (interactive)
  (article-menu-listing))

(defun article-menu-list-all-articles ()
  (interactive)
  (article-menu-listing (turn-article-table-into-propertized-list)))

(defun article-menu-list-metadata-articles ()
  (interactive)
  (article-menu-listing (label-to-propertized-list 'metadata)))
```

```
(defun article-menu-list-labels ()
  (interactive)
  (article-menu-listing (label-to-propertized-list 'label)))

(defun list-articles (lis)
  (interactive)
  (article-menu-listing (turn-list-into-propertized-list lis)))
```

7.8.12 On 'article-menu-display-article'

The point is to grab the name of the article on the currrent line of the listing and display it. This needs to be checked a bit in the multicolumn case (which itself needs to be explored).

```
(defun article-menu-display-article ()
  (interactive)
  (when (> (line-number-at-pos) 1)
    (save-excursion
      (goto-char (line-beginning-position))
      (search-forward-regexp "[. >] .")
      (setq name-of-current-article
            (get-text-property (point) 'name))
      (article-menu-point-out-current-article)
      (display-article name-of-current-article))))
```

7.8.13 On 'article-menu-mark-article'

Use this to mark the article mentioned on this line.

```
(defun article-menu-mark-article ()
  (interactive)
  (setq buffer-read-only nil)
  (when (> (line-number-at-pos) 1)
    (goto-char (line-beginning-position))
    (delete-char 1)
    (insert ">")
    (when (< (line-number-at-pos)
             (progn (save-excursion (goto-char (point-max))
                                    (line-number-at-pos))))
      (forward-line 1)))
  (setq buffer-read-only t))
```

7.8.14 On 'article-menu-unmark-article'

Use this to remove any mark on the article mentioned on this line.

```
(defun article-menu-unmark-article ()
  (interactive)
  (setq buffer-read-only nil)
  (when (and (> (line-number-at-pos) 1)
             (not (save-excursion (goto-char (line-beginning-position))
                                  (looking-at "\\."))))
    (goto-char (line-beginning-position))
```

```
    (delete-char 1)
    (insert " ")
    (when (< (line-number-at-pos)
             (progn (save-excursion (goto-char (point-max))
                                    (line-number-at-pos))))
      (forward-line 1)))
  (setq buffer-read-only t))

(defun article-menu-unmark-all-articles ()
  (interactive)
  (setq buffer-read-only nil)
  (save-excursion (goto-line 2)
                  (goto-char (line-beginning-position))
                  (while (re-search-forward "^>" nil t)
                    (replace-match " ")))
  (setq buffer-read-only t))
```

7.8.15 On 'article-menu-point-out-current-article'

This is a non-interactive function that puts a dot in front of the current article when the article listing is generated.

```
(defun article-menu-point-out-current-article ()
  (goto-char (point-min))
  (setq buffer-read-only nil)
  (save-excursion (when (search-forward-regexp "^\\." nil t)
                    (replace-match " ")))
  (when (and
         name-of-current-article
         (search-forward-regexp
          ;; maybe this `(format "%s" name-of-current-article)'
          ;; stuff should be stored as some kind of function,
          ;; like `print-name-of-article' or something like that
          (concat "^[. ] " (regexp-quote
                            (format "%s" name-of-current-article)))
          nil t))
    (replace-match (concat ". " (substring (match-string 0) 2))))
  (goto-char (line-beginning-position))
  (setq buffer-read-only t))
```

7.8.16 Features of the article menu

The only "actionable" feature of the current listing is display. See also Note .

```
(make-generic-menu-mode "Article Menu"
                        '(("g" . article-menu-listing)
                          ("m" . article-menu-mark-article)
                          ("u" . article-menu-unmark-article)
                          ("U" . article-menu-unmark-all-articles)
                          ("q" . quit-window)
                          ("\C-m" . article-menu-display-article)))
```

7.8.17 On 'article-menu-list-articles-matching-regexp'

This makes a listing showing all of the articles that match regexp.

{(It would be nice if we could leave out meta, reference, code, and label types of articles in the default version of this.)}

```
(defun article-menu-list-articles-matching-regexp (regexp)
  (interactive "MRegexp: ")
  (let ((matches (mapcar
                  (lambda (name)
                    (propertize (format "%s" name) 'name name))
                  (remove-if
                   (lambda (elt)
                     (not
                      (with-temp-buffer
                        ;; rough, not bothering with
                        ;; `sch-plain-text'
                        (let ((article (get-article elt)))
                          (insert
                           (format "%s"
                                   (scholium-name (get-article elt)))
                           "\n"
                           (format "%s"
                                   (scholium-text article)))
                          (goto-char (point-min))
                          (search-forward-regexp regexp
                                                 nil t)))))
                   (turn-article-table-into-names)))))
    (if matches
        (article-menu-listing matches)
      (message "No hits."))))
```

7.8.18 On 'article-menu-list-articles-matching-predicate'

This is a way to pick out all of the articles in the library that match a given predicate.

```
(defun article-menu-list-articles-matching-predicate (pred)
  (let ((matches (mapcar
                  (lambda (name)
                    (propertize (format "%s" name) 'name name))
                  (remove-if (lambda (elt) (not (funcall pred elt)))
                             (turn-article-table-into-names)))))
    (if matches
        (article-menu-listing matches)
      (message "No hits."))))
```

7.8.19 On 'display-article-listing'

This just puts the cursor in the article listing.

```
(defun display-article-listing ()
  (interactive)
  (pop-to-buffer "*Generic List*"))
```

7.9 Temporal browsing

7.9.1 Thematic histories

It might be useful to keep a record of several different sorts of histories, e.g. of editing events or "catalog pages" (Note). These alternate histories wouldn't have much to do with the temporal browser per se. Emacs does this in some cases (e.g. recording the input history for various interactive functions separately).

If we're going to do this, probably the code in this section should be written in a somewhat more generic way.

```
(defvar sb-history nil)

(add-hook 'display-article-hook
          (lambda ()
            (sb-add-article-to-history name-of-current-article)))
```

7.9.2 Complexities could be handled by 'display-article'

Of course, additional sorts of access instructions (like those alluded to in Note) could be handled by 'display-article'. *This section* does not depend on article names being stored as strings.

```
(defun sb-add-article-to-history (article)
  (setq sb-history (nconc sb-history (list article))))

(defun sb-back ()
  (interactive)
  (let ((current (car (last sb-history)))
        (n 0)
        found)
    (while (not found)
      (if (equal (nth n sb-history) current)
          (progn (setq found t)
                 (if (> n 0)
                     (display-article (nth (1- n) sb-history))
                   (message "Already at beginning of history.")))
        (setq n (1+ n))))))

(defun sb-forward ()
  (interactive)
  (let* ((current (car (last sb-history)))
         (max (1- (length sb-history)))
         (n (1- max))
         found)
    (while (not found)
      (if (equal (nth n sb-history) current)
          (progn (setq found t)
                 (if (< n (1- max))
                     (display-article (nth (1+ n) sb-history))
                   (message "Already at end of future")))
        (setq n (1- n))))))
```

7.9.3 On 'sb-previous'

This gives the most recently browsed article besides the current one.

```
(defun sb-previous ()
  (car (last sb-history 2)))
```

7.10 Linear browsing

7.10.1 On 'add-visible-back-temporal-link'

This makes a link to the previous-browsed page appear in the Scholia Display. It runs within the scope of 'scholia-display-extras' (Note 7.5.3).

It like we should potentially be able to change the strategy here, so as to add more information to the text properties and not actually create any corresponding article. I think this would require some adjustments to the way text properties are handled.

```
(defun add-visible-back-temporal-link ()
  ;; this should be marked up to become a fake link (one that isn't
  ;; displayed in the main buffer)
  (when (sb-previous)
    (insert "\n\n"
            "Back (temporal): ")
    (let ((beg (point)))
      (insert (propertize
               (format "%s" (sb-previous)) 'face 'sch-reference-face))
      (scholium 'reference-to-previous-article
                nil
                `((,name-of-current-article
                    (passage
                     ,beg
                     ,(point)))
                  (,(sb-previous)))
                '(reference fake))
      (add-to-scholia-property-within-region
       beg
       (point)
       '(reference-to-previous-article 1 1)))))
```

7.10.2 On 'add-visible-parent-and-sibling-links'

This adds links to the parent and nearest siblings in the Scholia Display buffer. It runs within the scope of 'scholia-display-extras' (Note 7.5.3).

```
(defun add-visible-parent-and-sibling-links ()
  ;; identify the link to the parent, if it exists.
  ;; (this assumes that there is only one parent)
  (let ((link-to-parent (car (member-if (lambda (link)
                                          (member 'parent (cdr link)))
                                        (scholium-about article)))))
    (when link-to-parent
      (let* ((parent (get-article (first link-to-parent)))
```

```
     (parent-data
      (scholium-text parent))
     (this-name-headed (member name-of-current-article
                               parent-data))
     (next (cadr this-name-headed))
     (prev (car (last (butlast parent-data
                               (length this-name-headed))))))
(when parent
  (insert "\n\n"
          "Parent: ")
  (let ((beg (point)))
    (insert (propertize (format "%s" (scholium-name parent))
                        'face 'sch-reference-face))
    (scholium 'reference-to-parent
              nil
              `((,name-of-current-article
                  (passage
                   ,beg
                   ,(point)))
                (,(scholium-name parent)))
              '(reference fake))
    (add-to-scholia-property-within-region
     beg
     (point)
     '(reference-to-parent 1 1))))
(when prev
  (insert "\n\n"
          "Back (in parent): ")
  (let ((beg (point)))
    (insert (propertize (format "%s" prev) 'face 'sch-reference-face))
    (scholium 'reference-to-previous-article-in-parent
              nil
              `((,name-of-current-article
                  (passage
                   ,beg
                   ,(point)))
                (,prev))
              '(reference fake))
    (add-to-scholia-property-within-region
     beg
     (point)
     '(reference-to-previous-article-in-parent 1 1))))
(when next
  (insert "\n\n"
          "Forward (in parent): ")
  (let ((beg (point)))
    (insert (propertize (format "%s" next)
                        'face
                        'sch-reference-face))
    (scholium 'reference-to-next-article-in-parent
```

```
                    nil
                    `((,name-of-current-article
                        (passage
                         ,beg
                         ,(point)))
                      (,next))
                    '(reference fake))
              (add-to-scholia-property-within-region
               beg
               (point)
               '(reference-to-next-article-in-parent 1 1))))))))

(add-hook 'scholia-display-extras-hook
          'add-visible-back-temporal-link)
(add-hook 'scholia-display-extras-hook
          'add-visible-parent-and-sibling-links)
```

7.10.3 On 'forward-in-parent'

A quick command to get the label identified as this object's parent, and move to the next mentioned after this one in this label. (It would also be possible to follow the next reference in a page, i.e., use a page like a label; but this function doesn't do that.)

It would be nice to have this function select the "next cousin" (or other suitable relative) if we are out of siblings.

We'll also want a quick command to hop *to* an article's parent.

```
(defun forward-in-parent ()
  (interactive))
```

7.10.4 On 'backward-in-parent'

Like 'forward-in-parent' (Note 7.10.3) but moves backward instead.

```
(defun backward-in-parent ()
  (interactive))
```

8 Part VIII — Editing and Maintenance

8.1 Initiating edits

8.1.1 Inserting text in the middle of a marked up region

When someone inserts text into the middle of a marked up region, does the new text take on the markup properties of the surrounding text, or does it not, or can we make it an option, to possibly be exercised in different ways in different places? I assume we will want different behavior at different times. (Cf. Note). The reference to consider is the "Sticky Properties" node from the Elisp manual[4](info "(elisp)Sticky Properties")}.

There they tell us that

> By default, a text property is rear-sticky but not front-sticky; thus, the default is to inherit all the properties of the preceding character, and nothing from the following character.

[4]{

```
          --
        ------
       ------------------
      ------
      1234567890
```

Figure 1: Markup simulated with indicator functions

Furthermore,

> If a character's 'rear-nonsticky' property is 't', then none of its properties are rear-sticky. If the 'rear-nonsticky' property is a list, properties are rear-sticky *unless* their names are in the list.

Thus, we have nothing to worry about in terms of front-stickiness, but we must maintain an appropriate rear-nonsticky list throughout the regions in which we want some of the properties to be non-sticky.

Something like this (enhanced, of course) will work:

```
(progn (put-text-property (line-beginning-position) (line-end-position)
       'face 'italic)
       (put-text-property (line-beginning-position) (line-end-position)
       'rear-nonsticky '(face)))
```

8.2 Finding revised 'about' data by parsing text properties

8.2.1 On 'scholia-property-at-changepoints'

This returns a list of pairs; points where the 'scholia' property changes paired with the value of the 'scholia' property at those points.

```
(defun scholia-property-at-changepoints ()
  (let ((next-change-point (point-min))
        change-points)
    (while next-change-point
      (setq change-points (cons (list next-change-point
                                      (get-text-property
                                       next-change-point
                                       'scholia))
                                change-points))
      (setq next-change-point (next-single-property-change+predicate
                               next-change-point
                               'scholia
                               'equal)))
    (when (not (null (get-text-property
                      (caar change-points)
                      'scholia)))
      (setq change-points (cons (point-max) change-points)))
    (reverse change-points)))
```

8.2.2 On 'detect-scholia-structure'

This function says which passages of the current article have been marked up with which scholia. Typically, this function will be run after editing, to recover the modified markup information and propagate it to attached scholia (see Section 8.3).

The way the function works is as follows. At each of the change points, we find the the annotations that have either just appeared or just disappeared, by using the 'set-difference' function to compare the value of the 'scholia' property at the *upcoming* change point with the value at the *current* one. We then sort the elements of this difference into "opening" and "closing" sets based on whether they are present at the *upcoming* change point, or the *current* one. In order to ensure that there is an adequate basis for comparison in all cases, exogenous points at "$-\infty$" and "∞" are introduced, with no scholia attached to them.

Every "opening" element corresponds to some as-yet-unseen marked-up region; when encountering such an item, we add a term to the return value that looks like "(VALUE BEG)" where VALUE and BEG are, respectively

- the value of the 'scholia' property – which will be the name of a scholium together with a unique link identifier, as described in Note [[id:the-scholia-property]];
- and the point where this link starts to attach to the current article.

Similarly, every "closing" element corresponds to the end of some already-seen marked-up region. When encountering such elements, we modify the corresponding previously added term in the return value to read "(VALUE BEG END)", END here being the point where the link in question stops attaching to the current document.

Note that since 'scholia-property-at-changepoints' gives us the values in increasing order, new "opening" items are encountered and added to the list in increasing order as well. This property is important for later.

```
(defun detect-scholia-structure ()
  (let ((begs-and-ends
         (cons '(-infinity nil)
               (append
                (scholia-property-at-changepoints)
                (list '(infinity nil)))))
        marked-regions)
    (while (> (length begs-and-ends) 1)
      (let ((difference (set-difference (second
                                         (first begs-and-ends))
                                        (second
                                         (second begs-and-ends))))
            opening
            closing)
        (when difference
          (dolist (elt difference)
            (if (member elt (second (second begs-and-ends)))
                (setq opening (cons elt opening))
              (setq closing (cons elt closing))))
          (dolist (elt opening)
            (setq marked-regions
                  (add-to-or-start-list
                   marked-regions
```

```
                `(,elt ,(first (second begs-and-ends))))))
          (dolist (elt closing)
            (let ((found (member-if (lambda (pass)
                                      (equal (first pass) elt))
                                    marked-regions)))
              (setcdr (cdr (car found))
                      (list (first (second begs-and-ends)))))))
        (setq begs-and-ends (cdr begs-and-ends))))
    marked-regions))
```

8.3 Committing edits

8.3.1 On 'store-link-masks'

This function runs within the scope of 'commit-edits', after 'adjusted-markup' has been found by parsing (Note 8.3.4). Indeed, so far, this is the only significant default action of 'commit-edits' (cf. Note).

The function compares the elements of 'current-markup' (Note 7.4.8) to the elements of 'adjusted-markup'. When changes are present in the markup data, 'store-link-masks' adds or changes masks.

More precisely, for each element of 'current-markup', we select from 'adjusted-markup' those elements with the same link-id (or maybe mask-id). These are the 'matching-regions' corresponding to that markup element. Unless there is only one matching region and this matching region happen to point to extend over exactly the range specified by the original link, we execute an operation on masks.

This operation is to swap in a replacement mask, if the markup element was coming from a mask already, or to create a brand new mask otherwise. If we're doing a replacement, we look for the *one* mask that corresponds to the link-id borne by the current markup element. We then substitute *all* of the matching regions, for whatever was previously stored in this mask.

Otherwise, we just create a new mask with the appropriate link-id, containing the relevant regions.

```
(defun store-link-masks ()
  (dolist (link-ext current-markup)
    (let* ((matching-regions (remove-if
                              (lambda (markup-item)
                                (not (equal (car markup-item)
                                            (car link-ext))))
                              adjusted-markup))
           (num-matching-regions (length matching-regions))
           (name-of-linking-article (if (eq (first
                                             (car link-ext)) 'mask)
                                        (first (second
                                                (first link-ext)))
                                        (first (first link-ext)))))
      (unless (and (eq num-matching-regions 1)
                   (equal (cdr (first matching-regions))
                          (cdr link-ext)))
        (if (eq (caar link-ext) 'mask)
            ;; swap in a different mask
            (let* ((masks
                    (get-metadata-field 'masks
                                        name-of-linking-article))
                   (current-mask-headed
                    (member-if (lambda (item)
```

```
                        (equal (second (car link-ext))
                               (car item)))
                      masks)))
    (setcdr (car current-mask-headed)
            (mapcar (lambda (ided-reg)
                      (cdr ided-reg))
                    matching-regions))
    (put-metadata-field 'masks
                        masks
                        name-of-linking-article))
;; create a new mask
(let ((masks (or (get-metadata-field
                  'masks
                  name-of-linking-article)
                 (list 'masks))))
  (put-metadata-field
   'masks
   (setcdr
    masks
    (add-to-or-start-list
     (cdr masks)
     `(,(car link-ext)
       ,@(mapcar (lambda (ided-reg) (cdr ided-reg))
                 matching-regions))))
   name-of-linking-article)))))))
```

8.3.2 On 'commit-edits-hook'

This is the single point of customization for the commit function, to be used in future sections. It runs within the scope of 'commit-edits', after 'adjusted-markup' has been found by parsing. Thus, functions that are added to this hook can take advantage of this knowledge about the current state of markup.

```
(defvar commit-edits-hook nil)
```

8.3.3 On 'after-committing-edits-hook'

This facilitates additional actions after the main committing routines are finished.

```
(defvar after-committing-edits-hook nil)
```

8.3.4 On 'commit-edits'

Anything run by the 'commit-edits-hook' that changes the text of the article being committed should store the desired value on 'adjusted-text'. It is going to take a bit of thinking about how to make several functions (particularly functions having to do with derivatives) run properly, in serial. (But I'm pretty sure it can be done.)

Note that when committing we currently make sure that exactly one newline is attached to the end of the page when the text is saved. This has a beneficial effect on exporting the scholium system (see 12.2.11).

We may at some point want one or more commit functions that will store more different kinds of edits from more different places. For example, we'll want to be able to commit edits that take place in the Scholia Display buffer. Similarly for edits to 'about' data, and so forth (see Note).

```
(defun commit-edits ()
  (interactive)
  (let* ((adjusted-markup (copy-tree (detect-scholia-structure)))
         (old-contents (get-article name-of-current-article))
         (adjusted-text (buffer-substring-no-properties (point-min)
                                                        (point-max)))
         (old-text (scholium-text old-contents)))
    (store-link-masks)
    (run-hooks 'commit-edits-hook)
    ;; here is where the criterion for checking that some change has
    ;; actually been made would be inserted
    (when (not (equal adjusted-text old-text))
      (scholium name-of-current-article
                (with-temp-buffer (insert adjusted-text)
                                  (goto-char (point-max))
                                  (insert "\n")
                                  (delete-blank-lines)
                                  (buffer-substring-no-properties
                                   (point-min)
                                   (point-max)))
                ;; of course, for more advanced versions of
                ;; this code, these won't be static
                (scholium-about old-contents)
                (scholium-type old-contents)
                ;; this should be reformatted to store the old
                ;; version!
                (scholium-bookkeeping old-contents)))
    (run-hooks 'after-committing-edits-hook)))
```

8.3.5 After committing, redisplay article

Hopefully this won't cause any problems. The one concerning point I can think of might come up when committing fully rendered and identified compilations (Section 9.6), but we haven't worked out the major details of that yet, so there's no reason to worry about this one minor one.

```
(add-hook 'after-committing-edits-hook 'redisplay-article)
```

8.4 Editing en masse

8.4.1 On 'label-marked-articles'

Apply the specified label to each marked article. New labels can be applied no problem.

```
(defun label-marked-articles (label)
  (interactive (list
                (intern (let ((completion-ignore-case t))
                          (completing-read
                           "Label: "
                           (label-to-list 'label))))))
  (let (article-names)
    (save-excursion
      (goto-line 2)
```

```
      (goto-char (line-beginning-position))
      (while (re-search-forward "^> ." nil t)
        (setq article-names
              (append article-names
                      (list
                       (get-text-property (point) 'name))))))
    (dolist (name article-names)
      (label-article name label))))
```

8.5 Deletion

8.5.1 Set current article to 'nil'

If the current article is deleted, then the 'name-of-current-article' variable should probably be set to nil. (Either that, or something from the temporal browser should be used to set the variable to the last-browsed page.)

```
(defvar delete-article-hook nil)

(defun delete-article (name)
  (interactive (list (read-article-name)))
  (let ((article (get-article name)))
    (if article
        (progn
          (remhash name article-table)
          (when (equal name name-of-current-article)
            (setq name-of-current-article nil))
          (run-hooks 'delete-article-hook))
      (error "No article by that name found"))))
```

8.5.2 On 'remove-appearances-in-history'

This function will run inside the scope of 'delete-article'.

```
(defun remove-appearances-in-history ()
  (setq sb-history (delete name sb-history)))

(add-hook 'delete-article-hook 'remove-appearances-in-history)
```

8.5.3 On 'delete-scholium-associated-with-current-marked-region'

Provides a quick way to delete the scholium attached to region containing point. If more than one scholium is so attached, we bring up a menu to select items for deletion.

This function assumes that the 'scholia' property is simply comprised of a list of the names of attached scholia; I'm not so sure that this is an accurate assumption (if not, then we should be able to easily extract such a list from the actual 'scholia' property).

This is just one of several functions that would benefit from the addition of progressive markup modifications.

```
(defun delete-scholium-associated-with-current-marked-region ()
  (interactive)
  (let ((linked-scholia (scholia-named-at-point)))
```

```
    (cond
     ((eq (length linked-scholia) 0)
      (message "No scholium attached here."))
     ((eq (length linked-scholia) 1)
      (delete-article (car linked-scholia)))
     (t
      (list-articles linked-scholia)
      (message
       "Inspect articles and select items for deletion with \"d\".")))))
```

8.6 Editing labels

8.6.1 On 'article-menu-insert-new-article'

This function lets you add new articles to a recently browsed label.

For now, we assume that we really are adding a *new* article (and not some existing one); and that this new article will take the quotidian form offered by 'make-scholium' (Note 6.4.4). These assumptions may be relaxed later.

The function one of 'label-article' and 'label-article-insert-before' to add the label to the new article (Note 3.4.2, Note 3.4.3).

(When creating this new article, I hope that enough ancillary metadata , type and so on, is added to allow it to be recognized as part of the relevant hierarchy later.)

```
(defun article-menu-insert-new-article ()
  (interactive)
  (if currently-displayed-label
      (let ((new-entry (read-string "New entry: "))
            (old-entry (save-excursion
                         (goto-char (line-beginning-position))
                         (search-forward-regexp "[. >] ." nil t)
                         (setq name-of-current-article
                               (get-text-property (point) 'name)))))
        (setq new-scholium-name new-entry)
        (let ((line (line-number-at-pos)))
          (if old-entry
              (label-article-insert-before
               new-entry old-entry currently-displayed-label)
            (label-article new-entry currently-displayed-label))
          (display-label currently-displayed-label)
          (goto-line line))
        (make-scholium))
    (message "Listing doesn't represent a label.")))

(define-key article-menu-map "i" 'article-menu-insert-new-article)
```

9 Part IX — Derivative Composition

9.1 Introduction to clusions

9.1.1 Format of text with derivatives

The format of the text field for articles with derivatives is as follows:

```
(twd <string | clusion> ...)
```

Each clusion is a link with the corresponding instruction consed onto the front of it (cf. Note 3.2.1). The three instructions are 'transclude', 'identify', and 'include'.

For example,

```
(twd "This Note can be used as part of an example.\n"
     (transclude "Format of text with derivatives"))
```

or,

```
(twd "Part of this Note can be used as part of an example.\n"
     (transclude "Format of text with derivatives"
                 (passage 1 25)))
```

There is a small break in transparency here; presumably this won't be a problem (Note `2.2-Design_issues.org`).

9.2 Transclusion

9.2.1 On 'transclude-article'

First we check to see whether the article contains transclusions already, and if not, we put the text field into the format used for text with transclusions (see Note 9.1.1).

Next, we divide the text of the article in two; everything prior to point, and everything after point. (We have to divide the *rendered* text, fairly clearly; since there might be more than one different rendering, we hope that the different possible divisions are consistent, or at least that we know how to do all of the divisions.)

Of course, we also need to be able to specify the objects that are to be transcluded!

```
;; totally not working yet!
(defun transclude-article (article)
  (interactive (list (read-article-name)))
  (let ((current (get-article name-of-current-article))
        newtext)
    ;; it really doesn't make sense to use something like this here --
    ;; we only deal with the internal format at commit time.
    (unless (and (listp current)
                 (eq (car current) 'twd))
            (setq newtext `(twd
                            ,(scholium-text current))))
    (let ((beg (point)))
      (insert (sch-plain-text (get-article article)))
      ;; this condition isn't really strong enough
      (when name-of-current-article
        (scholium
         `(derives-from ,name-of-current-article ,article ,(genref))
         nil
         `((passage (,name-of-current-article
                     ,beg
                     ,(point))))
         'derives-from
         'system)))))
```

9.3 Inclusion

9.3.1 On 'include-article'

It is simple enough to find an article and insert its text at point (compare 'insert-buffer').

%% We then need to add 'derives-from' and 'derives-to' properties (note %% similarity to backlinking; indeed, we reuse 'put-backlinks', relying %% on the general way in which it was written; see Note %% 3.3.7).

```
(defun include-article (article)
  (interactive (list (read-article-name)))
  (let ((beg (point)))
    (insert (sch-plain-text (get-article article)))
    ;; this condition isn't really strong enough
    (when name-of-current-article
      (scholium
       `(derives-from ,name-of-current-article ,article ,(genref))
       nil
       `((passage (,name-of-current-article
                   ,beg
                   ,(point))))
       'derives-from
       'system))))
```

9.3.2 By default, don't print 'derives-from' scholia

Typically we don't need to see 'derives-from' scholia, although it could be handy to write a special display mode in which these scholia are printed (and probably special semantics for how they are displayed).

```
(add-to-list 'non-printing-types 'derives-from)
```

9.4 Identification

9.4.1 On 'insert-identification-at-point'

This is similar to Note 9.3.1. For now, I'm just going to write it so that you can do identification with other articles; later, more general objects will be allowed (Note).

We don't actually need to create any scholia, but we can use the scholia property. It should be exactly the same when we render articles that already had identifications recorded internally.

```
(defun insert-identification-at-point (article-name)
  (interactive (list (read-article-name)))
  (let ((beg (point)))
    (insert (sch-plain-text article-name))
    (add-to-scholia-property-within-region
     beg
     (point)
     ;; there aren't any links, so there's no proper linkid
     `(identifies-with ,article-name))))
```

9.4.2 On 'propagate-changes-to-identification-source'

This runs within the context of 'store-updates-from-identification-images' (when it runs). Its purpose is to copy changes from the recipient article in an identification to the source article.

```
(defun propagate-changes-to-identification-source ()
  (when (not (equal (buffer-substring-no-properties
                     (second (car identifications))
                     (third (car identifications)))
                    (sch-plain-text
                      (second (caar identifications)))))
    (let ((source (get-article (second (caar identifications)))))
      ;; compare `commit-edits'.  When we actually do
      ;; document versions, this should update the version
      ;; number and any other metadata about how this
      ;; version was created that we get interested in.
      (scholium (second (caar identifications))
                ;; this is bad, since it is storing
                ;; the rendered text, whereas we should of course
                ;; be storing the formatted-for-internal-storage
                ;; text.  But I suppose this is OK for testing
                ;; purposes (wherein we'll only have one level
                ;; of identification).
                (buffer-substring-no-properties
                 (second (car identifications))
                 (third (car identifications)))
                (scholium-about old-contents)
                (scholium-type old-contents)
                (scholium-bookkeeping old-contents)))))
```

9.4.3 On 'store-updates-from-identification-images'

This function will run within the scope of 'commit-edits' via the 'commit-edits-hook' (see Note 8.3.2). We must first identify the identifications (as it were), which we do by examining the contents of the 'adjusted-markup' list. Then, once these are found, we form a suitable internal representation.

Everything *between* identifications should get stored as plain text, whereas the identifications themselves should be stored as tags.

In addition to adjusting the internal representation of the article with the identification commands, we should adjust the text of the source articles, if necessary. (For simplicity's sake, we could just store updated text unconditionally for now.)

Concerning the use of 'buffer-substring-no-properties': zapping the properties is questionable, if we're planning to commit these various derivative properties serially. Will have to think about this later. I think that since this thing is being used to build internal representations, zapping text properties might actually be OK. But I'm not sure.

For clarification on the use of 'adjusted-text', see comments on 'commit-edits'.

Note that the only relevant option besides "less" is "equal" in the 'if' here.

```
(defun store-updates-from-identification-images ()
  (let ((identifications (remove-if
                          (lambda (elt)
                            (not (eq (caar elt) 'identifies-with)))
```

```
                          adjusted-markup))
        (pt 1)
        formatted-contents)
    (while identifications
      (if (< pt (second (car identifications)))
          (setq formatted-contents
                (append
                 formatted-contents
                 (list (buffer-substring-no-properties
                        pt (second (car identifications)))
                       `(ident ,(second (caar identifications)))))
                pt (third (car identifications)))
        (setq formatted-contents
              (append
               formatted-contents
               (list `(ident ,(second (caar identifications)))))
              pt (third (car identifications))))
;      (propagate-changes-to-identification-source)
      (setq identifications (cdr identifications)))
    (when formatted-contents
      (when (< pt (point-max))
        (setq formatted-contents
              (append
               formatted-contents
               (list (buffer-substring-no-properties
                      pt (point-max))))))
      (setq adjusted-text (list 'twd formatted-contents)))))

(add-hook 'commit-edits-hook 'store-updates-from-identification-images)
```

9.5 Rendering articles containing derivative portions

9.5.1 Cluded parts

These variables will be modified by 'unwind-derivatives' to contain a list of the transcluded sections of a rendered document. Specifically, the items on the list are of the form

```
(<object> <beg> <end>)
```

where 'object' is a link to the thing being transcluded, and 'beg' and 'end' represent the beginning and end positions of the image of the object under this particular clusion, in the rendered version of the cluding article.

This information will subsequently be used as part of the markup routine, where it will allow us to add appropriate text properties indicating cluded regions.

(It may be advantageous to assume that we can always use a *cached* version of the article we're cluding. This could be a help when the article itself is derivative.)

```
(defvar transcluded-parts nil)
(defvar identified-parts nil)
(defvar included-parts nil)
```

9.5.2 On 'unwind-derivatives'

This will run within the scope of 'sch-plain-text', and make recursive calls to 'sch-plain-text' as needed, to turn cluded texts into strings suitable for display. Specifically, a simple test has been added to 'sch-plain-text-hook' that causes 'unwind-derivatives' to run whenever rendering an article whose text field is a list that begins with the token 'twd' (see Note 9.5.4).

The boundaries of the cluded regions are recorded for subsequent use, as described in Note 9.5.1.

The contents of the " **Unwinding Derivatives**" buffer needs to be cleared out before this function first runs for it to return the correct string. (Um, is this all going to work out properly if we enter into the fully recursive run of 'unwind-derivatives'?) That is accomplished by modifications to 'scholia-display-pre-update-hook'; see Note 9.5.3.

The need for (extensive) recursion in this function could be done away with if we cached a *rendered* version of every article that used clusions somewhere where it could be found by other articles that clude from it.

However, if we are going to have things like references from cluded documents appear in the assembled document, or (in general) if we want to have access to the scholia attached to the cluded regions (Note), we'll either have to pre-render the derivative components, or come up with some scheme for mapping the positions of markup associated with these things into suitable positions in the assembled document.

Note that at present markup will be added by 'add-inclusion-and-transclusion-markup' (Note 9.5.5).

```
(defun unwind-derivatives ()
  (set-buffer (get-buffer-create " *Unwinding Derivatives*"))
  (dolist (elt (cdr text))
    (if (stringp elt)
        (insert elt)
      (let (object)
        (cond
         ((eq (car elt) 'transclude)
          (setq object (cdr elt))
          (setq transcluded-parts (cons `(,object ,(point))
                                        transcluded-parts)))
         ((eq (car elt) 'identify)
          (setq object (cdr elt))
          (setq identified-parts (cons `(,object ,(point))
                                       identified-parts)))
         (t
          (setq object elt)))
        (insert (sch-plain-text object))
        (cond
         ((eq (car elt) 'transclude)
          (setcdr (cdar transcluded-parts) `(,(point))))
         ((eq (car elt) 'ident)
          (setcdr (cdar identified-parts) `(,(point)))))))))
  (setq ret (buffer-string)))
```

9.5.3 Preparation for rendering text with derivative components

We need to zap 'transcluded-parts', 'identified-parts' and the contents of the buffer in which derivatives are to be unwound before 'unwinding-derivatives' runs.

```
(defun prep-for-rendering-text-with-derivative-components ()
```

```
  (setq transcluded-parts nil
        identified-parts nil
        included-parts nil)
  (save-excursion
    (set-buffer (get-buffer-create
                 " *Unwinding Derivatives*"))
    (erase-buffer)))

(add-hook 'scholia-display-pre-update-hook
          'prep-for-rendering-text-with-derivative-components)
```

9.5.4 Adding 'unwind-derivatives' to the rendering pathway

We make a call to 'unwind-derivatives' in 'sch-plain-text' when the appropriate criterion is satisifed (see Note 7.1.1). The relevant criterion is that we are rendering an object (i.e., something that gives us an article) whose text field is a list that begins with the token 'twd'.

(The complex wording here has to do with the matter that we sometimes render objects that aren't articles, e.g. a link to a certain passage. We will have to make sure that the criterion used here is correct for those cases as well as the easy case of rendering a whole article. See Note 7.1.2 for details.)

```
(add-hook 'sch-plain-text-hook
          '(lambda ()
             (when (and (listp text) (eq (car text) 'twd))
               (unwind-derivatives))))
```

9.5.5 On 'add-inclusion-and-transclusion-markup'

For identifications, this should add the same sort of faked-up markup that we added with 'insert-identification-at-point' (Note 9.4.1).

It might be kind of cute to put some "corners" in, showing where the included text begins and ends (see Note `2.2-Design_issues.org`).

These modifications to 'mark-things-up' appear to be desired unconditionally (unlike e.g. the modifications to 'sch-plain-text' we just saw in Note 9.5.4). So we just put 'add-inclusion-and-transclusion-markup' directly on the 'mark-things-up-hook' (cf. Note 7.4.17).

```
(defun add-inclusion-and-transclusion-markup ()
  (save-excursion
    (set-buffer-to-rendering-target-buffer)
    (dolist (elt identified-parts)
      (add-to-scholia-property-within-region
       (second elt)
       (third elt)
       `(identifies-with ,(first elt))))
    ;; this may end up having to be considerably more complicated
    (dolist (elt transcluded-parts)
      (add-to-scholia-property-within-region
       (second elt)
       (third elt)
       `(transclusion-of ,(first elt))))))

(add-hook 'mark-things-up-hook 'add-inclusion-and-transclusion-markup)
```

9.6 Quick compilations and other listing tricks

9.6.1 On 'find-names-in-listing'

This can be used to find the names of listed articles.

```
(defun find-names-in-listing (just-marked)
  (let ((names (list t)))
    (save-excursion
      (set-buffer (get-buffer-create "*Generic List*"))
      (goto-char (point-min))
      (while (and (not (eobp))
                  (if just-marked
                      (search-forward-regexp "^>" nil t)
                    t))
        (let* ((next-change (next-single-property-change
                             (point) 'name))
               (prop (when next-change
                       (get-text-property next-change 'name))))
          (if (not next-change)
              (goto-char (point-max))
            (goto-char next-change)
            (when prop
              (nconc names (list prop)))))))
    (cdr names)))
```

9.6.2 Making a compilation from a listing of articles

In Note , we talk about making all the articles that match a given criterion into a new article. A quick way to get this functionality to the user is to turn an article listing into a compilation.

The version here is very preliminary; eventually we'll be putting in identifications or something like that to make the various pieces of the compilation "hot".

It be nice to have various display options, either features that screen certain kinds of content in or out at render time (e.g. printing of attached code snippets could be optionally be made automatic), or that add actionable features to the display (e.g. to enable the user to expand and collapse cluded articles, or to select from a cluded label).

We present a couple of variants here.

```
(defun listing-to-compilation (just-marked-items)
  (interactive "P")
  (let ((names (find-names-in-listing just-marked-items)))
    (pop-to-buffer (get-buffer-create "*Compilation*"))
    (erase-buffer)
    (dolist (name names)
      (let ((article (get-article name)))
        (insert (upcase
                 (propertize (format "%s" (scholium-name article))
                             'face 'italic))
                "\n"
                (format "%s" (scholium-text article))
                "\n")))
    (goto-char (point-min))))
```

```
(defun listing-to-compilation-1 (just-marked-items)
  (interactive "P")
  (let ((names (find-names-in-listing just-marked-items)))
    (pop-to-buffer (get-buffer-create "*Compilation*"))
    (erase-buffer)
    (dolist (name names)
      (let ((article (get-article name)))
        (insert  "\*** "
                 (propertize (format "%s" (scholium-name article))
                             'face 'italic)
                 "


"
                 "\n"
                 (format "%s" (scholium-text article))
                 "\
"
                 "\n\n")
        (dolist (scholium (mapcar (lambda (backlink)
                                    (get-article (car backlink)))
                                  (get-backlinks name)))
          (when (typedata-includes (scholium-type scholium) 'code)
            (insert "\\b" "egin{lisp}\n"
                    (scholium-text scholium)
                    "\\e" "nd{lisp}\n\n")))))
    (goto-char (point-min))))
```

9.6.3 On 'listing-to-label'

It can be handy to turn an arbitrary listing into a more permanent label.

```
(defun listing-to-label (label-name)
  (interactive "MLabel: ")
  (let ((names (find-names-in-listing)))
    (scholium label-name
              (cdr names)
              nil
              'label
              (sch-book))))
```

10 Part X — Persistence and Collaboration

10.1 Saving and restoring

10.1.1 Autosave

We might consider making some environment variable that would cause things to be saved to disk automatically.

```
(defun save-all-scholia (filename)
```

```
  (interactive (list
                (read-file-name "Filename: ")))
  (save-window-excursion
    (gather-scholia)
    (write-file filename)
    (kill-buffer (current-buffer))))
```

10.1.2 On 'gather-scholia'

It might be a good idea for this function to take an optional predicate or label, and gather only scholia that match that predicate. We could easily cause the gathering function to run only within a certain context as well.

This function doesn't really have anything to do with saving, logically speaking, so perhaps it shouldn't go in this section.

```
(defun gather-scholia ()
  (interactive)
  (set-buffer (get-buffer-create "*Scholia*"))
  (delete-region (point-min) (point-max))
  (maphash (lambda (name val)
             (write-scholium (cons name val)))
           article-table)
  (display-buffer "*Scholia*"))
```

10.1.3 Improving the design of 'write-scholium'

One could probably make these things print a bit nicer, e.g. to make everything fit within 80 columns, but this seems to be good enough for the time being.

```
(defun write-scholium (article)
  (let ((nl "\n          "))
    (insert
     (concat "(scholium " (maybe-quoted-format (scholium-name article))
             nl (maybe-quoted-format (scholium-text article))
             nl (maybe-quoted-format (scholium-about article))
             nl (format "'%S" (scholium-type article))
             nl (format "'%S" (scholium-bookkeeping article))
             ")\n\n"))))

(defun maybe-quoted-format (obj)
  (if (and (not (null obj))
           (or (atom obj) (consp obj)))
      (format "'%S" obj)
    (format "%S" obj)))
```

10.1.4 Design of 'read-scholia-file'

This reads and evaluates all of the scholia that have been written out into the file stored at filepath.

```
(defun read-scholia-file (filepath)
  (interactive "fFile: ")
  (find-file-literally filepath)
```

```
  (read-scholia-buffer)
  (kill-buffer (current-buffer)))

(defun read-scholia-buffer ()
  (while (condition-case nil
             (eval (read (current-buffer)))
           (error nil))))
```

10.1.5 Apparently excessive generality of 'read-scholia-file'

This system would actually read any elisp file (er, I'm not sure it would deal well with comments). There are built-in functions that accomplish the same thing ('load-file' and 'eval-buffer' come to mind). The idea here was that we might want to do something somewhat more complicated than simply evaluating the code found in the file. That may still happen.

```
(defvar search-directory-for-scholia t)

(add-hook 'find-file-hook 'search-directory-for-scholia)

(defun search-directory-for-scholia ()
  (list-directory default-directory t)
  (let ((sch-file
         (replace-regexp-in-string "\\..*" ".sch" (buffer-file-name))))
    (when (search-forward sch-file nil t)
      (read-scholia-file sch-file)
      (display-scholia-about-current-buffer)))
  (kill-buffer "*Directory*"))
```

11 Part XI — Bindings and Environment

11.1 Bindings and environment variables

11.1.1 Room to improve on bindings

This is just an attempt to make something usable, but it is probably nowhere near close to optimal. One thing that would be nice would be to have some prefix for browsing relative to all scholia versus some other prefix for browsing relative to only the current scholium. Thus, you could cycle between regions for the current scholium or regions for all scholia (or go to the first region or whatever) depending on what prefix you specified – but the final keystroke ("f" for forward or whatever) would be the same. Perhaps we could substitute in the first register (as opposed to the second) for changing the general style of browsing (e.g. from scholia to temporal or whatever), but keep second and third pretty much the same. It would be good to spell out the analogies in a somewhat detailed table.

```
(mapc
 (lambda (elt) (global-set-key (eval (car elt)) (eval (cdr elt))))
 '(([?\C-\;] . (make-keymap))
   ([?\C-\;?m] . (make-sparse-keymap))
   ([?\C-\;?d] . (make-sparse-keymap))
   ([?\C-\;?o] . (make-sparse-keymap))
   ([?\C-\;?s] . (make-sparse-keymap))
```

```
((kbd "C-; c") . 'commit-edits)
((kbd "C-; g") . 'gather-scholia)
((kbd "C-; f") . 'follow-reference-or-scholium)
((kbd "C-; n") . 'name-of-current-scholium)
((kbd "C-; r") . 'save-all-scholia)

((kbd "C-; v n") . 'back-to-normal)
((kbd "C-; v o") . 'back-to-other-view)

((kbd "C-; b b") . 'sb-back)
((kbd "C-; b f") . 'sb-forward)

((kbd "C-; l a") . 'article-menu-list-all-articles)
((kbd "C-; l l") . 'article-menu-list-labels)
((kbd "C-; l m") . 'article-menu-list-articles-matching-regexp)
((kbd "C-; l d") . 'article-menu-list-metadata-articles)
((kbd "C-; l p") . 'article-menu-list-plain-articles)

((kbd "C-; m a") . 'make-scholium-about-current-article)
((kbd "C-; m b") . 'make-scholium-about-current-buffer)
((kbd "C-; m l") . 'make-scholium-about-current-line)
((kbd "C-; m n") . 'make-new-undirected-article)
((kbd "C-; m i") . 'make-current-buffer-into-article)
((kbd "C-; m p") . 'make-scholium-about-part-of-current-article)
((kbd "C-; m P") .
 'make-scholium-about-several-parts-of-current-article)
((kbd "C-; m s") . 'make-scholium-about-current-scholium)

((kbd "C-; d a") . 'display-article)
((kbd "C-; d b") . 'display-scholia-about-current-buffer)
((kbd "C-; d l") . 'display-article-listing)
((kbd "C-; d p") .
 'display-an-article-that-current-article-is-about)
((kbd "C-; d c") .
 'display-an-article-that-current-scholium-is-about)
((kbd "C-; d r") . 'redisplay-article)

((kbd "C-; o y") . 'sch-turn-main-article-overlays-on)
((kbd "C-; o n") . 'sch-turn-main-article-overlays-off)

((kbd "C-; s n") .
 'scroll-article-display-to-next-region-for-current-scholium)
((kbd "C-; s p") .
 'scroll-article-display-to-previous-region-for-current-scholium)
((kbd "C-; s b") . 'move-to-previous-region-with-scholium)
((kbd "C-; s f") . 'move-to-next-region-with-scholium)
((quote [S-tab]) . 'move-to-next-region-with-scholium)
((kbd "C-; s a") . 'move-to-first-region-for-current-scholium)))
```

12 Part XII — Applications and Experiments

12.1 Use the scholium system to maintain a library of projects

12.1.1 Library of major works

We have an overall structure for the digital library as a whole worked out already (Section 3.1), but not yet a specific list of major works. It is convenient to have an index of these (just like, later, it will be convenient to have many different indices of different kinds of documents with different features). So, we create a virtual library to index these works in. (Once this is set up properly, of course the imported version of the scholium system itself will be included.)

However right now this isn't working, so not exporting that code!

```
(scholium 'major-articles nil nil 'label 'system)
```

12.2 Use the scholium system to do literate programming

12.2.1 Importing LaTeX docs

Note that importing a LaTeX\ document (this one, in particular) wouldn't be so different from importing a wiki (see Section 12.3).

One of the issues is how we're going to represent cross references (see Note). In compiled LaTeX, they typically appear as a number, whereas in source they appear as a tag. In the scholium system, reference markup should presumably be used, and I also suppose that we may as well use the name of the article being referred to directly. So, for example, this article would be linked to by text that read "see 'Importing LaTeX docs'." This is similar to the way references appear in Texinfo.

Outside references (footnotes) could be rendered in a different color (Note), and made browsable within the scholium system (Note).

On the implementation: 'end' might be followed by a lisp expression that should be attached to the note, but this should be fun for a trial run.

The format of the scholia representing the notes is kind of weird. We shouldn't have junk in the text field of the article. It would probably make more sense for items of type "note" to be rendered specially (if one wished) and maybe to store the tag as a part of a formatted text field, or, more likely, part of the *type* data.

I think I'd be relatively comfortable adding identification properties to the regions of the larger document, to get them to inherit from the individual pieces. Later we could parse the section structure.

It would, generally speaking, be a good idea if lower levels in the hierarchy were about their parents in such a way that we could easily move "up".

```
(add-hook 'scholia-display-post-update-hook 'text-mode)

(add-to-list 'modified-type-labels '(note . note) t)
(add-to-list 'modified-type-labels '(section . section) t)
(add-to-list 'modified-type-labels '(subsection . subsection) t)
(add-to-list 'modified-type-labels '(subsubsection . subsubsection) t)
```

12.2.2 On 'map-label'

This gives you a way to apply a function to every article that bears a given label.

```
(defun map-label (function label)
  (mapc function (scholium-text (get-article label))))
```

12.2.3 On 'swap-out-latex-references'

Does the opposite of 'swap-in-latex-references' (Note 12.2.13): that is, this function makes `\ref` tags in the text look like hyperlinks.

```
(defun swap-out-latex-references ()
  (let ((tags-alist (scholium-text (get-article
                                    'Tag-to-Name-converter))))
    (map-label (lambda (name)
                 (with-temp-buffer
                   (let ((article (get-article name)))
                     (insert (scholium-text article))
                     (goto-char (point-min))
                     (while (re-search-forward
                             ;; I don't think we use "xrefs" any more.
                             "\\\\\\(x\\)?ref{\\([^}]+\\)}" nil t)
                       (let ((target (cdr
                                      (assoc (match-string 2)
                                             tags-alist))))
                         (when target
                           (replace-match target t t)
                           (let ((name-of-current-article name))
                             (make-reference-in-current-article
                              (match-beginning 0)
                              (+ (match-beginning 0) (length target))
                              target)))))
                     ;; we don't want these references to be fake
                     (scholium name
                               (buffer-substring-no-properties
                                (point-min)
                                (point-max))
                               (scholium-about article)
                               '(note)))))
                 'note)))
```

12.2.4 On 'import-scholium-system'

For now, this is just 'import-sections', but it would be good if we could do something about sections that contain neither subsections nor notes, for example, the preface. Other nice features (e.g. creating "function type" scholia for functions, as in Note) should be added too, eventually. Sometimes work arounds can be applied for the time being.

{Hint: I've found that for now one has to '(clrhash article-table)' before running this function for things to work properly. I intend to address this issue soon.}

(We might want to make this article use some kind of identification routine; alternatively, make it work with a "large structure" map of the document's contents.)

```
(defun import-scholium-system ()
  (interactive)
  (import-sections)
;  (label-article 'section 'major-articles)
  (swap-out-latex-references))
```

12.2.5 On 'import-sections'

For each section, import any notes and any subsections.

I'm going to try to get this to import front- and back-matter (before the first section and after the last note, respectively). This will make it easier to typeset everything with one command.

```
(defun import-sections ()
  (save-excursion
    (set-buffer "sbdm4cbpp.tex")
    (goto-char (point-min))
    (search-forward-regexp "^\\\\section{Prelude}")
    (goto-char (match-beginning 0))
    (scholium "Scholium system frontmatter"
              (buffer-substring-no-properties (point-min) (point))
              nil
              '(note))
    (while (re-search-forward
            (concat
             "^\\\\section\\*?{\\([^}\n]*\\)}"
             "\\( +\\\\label{\\)?"
             "\\([^}\n]*\\)?")
            nil t)
      (let* ((name (match-string-no-properties 1))
             (tag (match-string-no-properties 3))
             (section-end (save-excursion
                            (search-forward-regexp
                             "^\\\\section{.*" nil t)))
             (notes-end (or (save-excursion
                              (search-forward-regexp
                               "^\\\\subsection{.*"
                               section-end t))
                            section-end))
             (notes (let ((current-parent name))
                      (import-notes)))
             (subsections (let ((current-parent name))
                            (import-subsections))))
        (when (not (equal tag ""))
          (scholium 'Tag-to-Name-converter
                    (add-to-or-start-list
                     (scholium-text (get-article
                                     'Tag-to-Name-converter))
                     `(,tag . ,name))))
        (scholium name
                  `(,@notes
                    ,@subsections)
                  '((section parent))
                  '(section label))))
    (goto-char (point-max))
    (search-backward-regexp "^\\\\clearpage")
    (scholium "Scholium system backmatter"
              (buffer-substring-no-properties (point) (point-max))
```

```
            nil
            '(note))))
```

12.2.6 On 'import-subsections'

For each subsection, import any notes and any subsubsections. notes are imported if they appear before the first subsubsection, if there is one, or before the end of the subsection, otherwise, or before the end of the section, if this is the last subsection

```
(defun import-subsections ()
  (let (subsections)
    (while (re-search-forward
            (concat
             "^\\\\subsection{\\([^}\n]*\\)}"
             "\\( +\\\\label{\\)?"
             "\\([^}\n]*\\)?")
            section-end t)
      (let* ((name (match-string-no-properties 1))
             (tag (match-string-no-properties 3))
             (subsection-end (or (save-excursion
                                   (search-forward-regexp
                                    "^\\\\subsection{.*"
                                    section-end t))
                                 section-end))
             (notes-end (or (save-excursion
                              (search-forward-regexp
                               "^\\\\subsubsection{.*"
                               subsection-end t))
                            subsection-end
                            section-end))
             (notes (let ((current-parent name))
                      (import-notes)))
             (subsubsections (let ((current-parent name))
                               (import-subsubsections))))
        (when (not (equal tag ""))
          (scholium 'Tag-to-Name-converter
                    (add-to-or-start-list
                     (scholium-text (get-article
                                     'Tag-to-Name-converter))
                     `(,tag . ,name))))
        (scholium name
                  `(,@notes
                    ,@subsubsections)
                  `((,current-parent parent))
                  '(subsection label))
        (setq subsections
              (append subsections (list name)))))
    subsections))
```

12.2.7 On 'import-subsubsections'

For each subsubsection, import any notes that appear before the end of the subsubsection (if this is the last subsection, then before the end of the subsection, etc.).

```
(defun import-subsubsections ()
  (let (subsubsections)
    (while (re-search-forward
            (concat
             "^\\\\subsubsection{\\([^}\n]*\\)}"
             "\\( +\\\\label{\\)?"
             "\\([^}\n]*\\)?")
            subsection-end t)
      (let* ((name (match-string-no-properties 1))
             (tag (match-string-no-properties 3))
             (notes-end (or (save-excursion
                              (search-forward-regexp
                               "^\\\\subsubsection{.*"
                               subsection-end t))
                            subsection-end
                            section-end))
             (notes (let ((current-parent name))
                      (import-notes))))
        (when (not (equal tag ""))
          (scholium 'Tag-to-Name-converter
                    (add-to-or-start-list
                     (scholium-text (get-article
                                     'Tag-to-Name-converter))
                     `(,tag . ,name))))
        (scholium name
                  notes
                  `((,current-parent parent))
                  `(subsubsection label))
        (setq subsubsections
              (append subsubsections (list name)))))
    subsubsections))
```

12.2.8 On 'import-notes'

This imports the notes at the current level in the section hierarchy.

```
(defun import-notes ()
  (let (notes)
    (while
        (re-search-forward (concat "\\\\begin{notate}"
                                   "{\\([^}\n]*\\)}"
                                   "\\( +\\\\label{\\)?"
                                   "\\([^}\n]*\\)?")
                           notes-end t)
      (let* ((name (match-string-no-properties 1))
             (tag (match-string-no-properties 3))
```

```
            (beg (progn (next-line 1)
                        (line-beginning-position)))
            ;; no need to bound the search for the end, because we
            ;; assume that every "notate" environment is actually
            ;; closed
            (end (progn (search-forward-regexp
                         "\\\\end{notate}")
                        (match-beginning 0))))
       (when (not (equal tag ""))
         (scholium 'Tag-to-Name-converter
                   (add-to-or-start-list
                    (scholium-text (get-article
                                    'Tag-to-Name-converter))
                    `(,tag . ,name))))
       (scholium name
                 (buffer-substring-no-properties beg end)
                 `((,current-parent parent))
                 '(note))
       (setq notes
             (append notes (list name)))
       (import-code-continuations)))
   notes))
```

12.2.9 On 'import-code-continuations'

This runs within the scope of 'import-notes', to turn any Lisp chunks that follow a given Note into scholia attached to that note. (This won't import the one stand-alone "verbatim" environment used in the code; we could change this function, or include the environment inside of a Note. Or we could just put the literate programming code into another file. Minor issue. Also, figure environments would be left out, except for a small trick, namely including them within surrounding Notes.) Notice that previous versions of these articles shouldn't exist at import time, or Lisp sections will be doubled.

The code continuations and the code descriptions wind up being mutually about one another, which is kind of cute.

```
(defun import-code-continuations ()
  ;; ugly formatting = latex overrides!
  (while (looking-at (concat "\n\n\\\\b" "egin{lisp}"))
    (scholium `(code-continuation ,name)
              (let ((old-text (scholium-text
                               (get-article
                                `(code-continuation ,name))))
                    (new-text (buffer-substring-no-properties
                               (progn (next-line 3)
                                      (line-beginning-position))
                               (progn (search-forward-regexp
                                       (concat "\\\\e" "nd{lisp}"))
                                      (match-beginning 0)))))
                (if old-text
                    (concat old-text "\n" new-text)
                  new-text))
              `(((article ,name)))
```

```
            'code)
    ;; this should add an appropriate link to the article
    ;; that this is a code-continuation of.
    (let ((article-to-edit (get-article name)))
      (scholium (scholium-name article-to-edit)
                (scholium-text article-to-edit)
                (add-to-or-start-list
                 (scholium-about article-to-edit)
                 `(code-continuation (code-continuation ,name)))
                (scholium-bookkeeping article-to-edit)))))
```

12.2.10 Identification of notes is slow

I tried identifying each note with its source region by running (add-to-scholia-property-within-region beg end '(identifies-with ,name)) towards the end of 'import-notes', but it slowed everything down.

We may be able to get something similar by other means anyway (Note).

```
(defun browse-scholium-system ()
  (interactive)
  (import-scholium-system)
  (display-article 'section)
  (message "You've successfully imported the system!"))
```

12.2.11 On 'export-scholium-system'

We use a similar sort of "recursive" style to that used in the import. First, loop through the sections. These must of course be exported in their own limited fashion. Then, for each, examine the contents. The items contained in each section are either subsections or notes. If the item really is a subsection, then do something similar to what we did with sections; otherwise, it is a note and we simply export it. The items at the penultimate level are either subsubsections or notes; and finally, in the end, the items are all notes.

Note that getting the "Tag-to-Name-converter" each time it is used in the child functions is really sort of excessive, since it could be found once in the parent. However, things seem to run fast enough.

```
(defun export-scholium-system ()
  (set-buffer (get-buffer-create "*Export*"))
  (erase-buffer)
  (insert (scholium-text (get-article "Scholium system frontmatter")))
  (dolist (sect (scholium-text (get-article 'section)))
    (export-section sect)
    (let ((contents (scholium-text (get-article sect))))
      (dolist (item contents)
        (let ((current-item (get-article item)))
          (if (typedata-includes (scholium-type current-item)
                                 'subsection)
              (export-subsection (scholium-name current-item))
            (export-note current-item))))))
  (insert (scholium-text (get-article "Scholium system backmatter"))))
```

12.2.12 On 'export-note'

This is for exporting notes. When the note has a code continuation it also exports that code.

To really work, this is going to have to replace references with `\refs` and `\refs`. Hopefully we're storing the relevant information somewhere easily accessible at import time? We should at very least be able to do the opposite of what is done with 'swap-out-latex-references'.

We essentially need to render the article to know what the relevant references are (and where they are). This suggests to me that if we don't want to disrupt the user's experience of the rendered articles appearing in the standard display, we should be able to "beam" a rendering to some other buffer and then pick it up from there. Also, if some references have been generated that don't correspond to items with tags, we'll have to accomodate that.

```
(defun export-note (note)
  (set-buffer (get-buffer "*Export*"))
  (let* ((name (scholium-name note))
         (tag (car (rassoc name (scholium-text
                                 (get-article
                                  'Tag-to-Name-converter))))))
    (insert "\*** " name "

"
            (if tag
                (concat " \\label{" tag "}\n")
              "\n")
            (swap-in-latex-references note)
            "\
\n\n")
    (dolist (scholium (mapcar (lambda (backlink)
                                (get-article (car backlink)))
                              (get-backlinks name)))
      ;; we were missing one function from a list of two in my last
      ;; test of exporting, should look into this.
      (when (typedata-includes (scholium-type scholium) 'code)
        (insert "\\b" "egin{lisp}\n"
                (scholium-text scholium)
                "\\e" "nd{lisp}\n\n")))))
```

12.2.13 On 'swap-in-latex-references'

Maybe it would be easier if each of the references had the name of the tag stored locally? This doesn't seem like it would actually help. The point I guess is that we don't want to detect the scholia structure all at once, but rather, zip through and make changes to each item we encounter, in order. I.e., the endpoints of the regions that we'll be modifying are always in transition.

'move-to-next-region-with-scholium' will have to be set up to return 'nil' if there is no next scholium. It would also be nice if this returned the boundaries of the region to which the scholium is attached, when there is a scholium, since we're going to want to do a replacement of the text in that region. (If it proves more reasonable to use a different function for this purpose, that would be fine, but this is roughly what it should do.) Note that 'move-to-next-region-with-scholium' currently works in terms of overlays, which isn't really right.

After we find the scholia at the start and end of the marked region, we could check a complicated condition to see if there is a reference at the beginning and another copy of the same reference at the end. But for the time being we aren't going to work with this general case of overlapping references, and we'll just assume one reference per marked region and get on with things.

```
(defun swap-in-latex-references (note)
  (save-excursion
    (let ((tags-alist (scholium-text (get-article
                                      'Tag-to-Name-converter))))
      (display-article (scholium-name note))
      (set-buffer (get-buffer "Main Article Display"))
      (let ((next-region (move-to-next-region-with-scholium)))
        (while next-region
          (let ((scholium-property-at-start
                 (get-text-property (first next-region) 'scholia))
                (scholium-property-at-end
                 (get-text-property (second next-region) 'scholia)))
            (let* ((scholium-id (first scholium-property-at-start))
                   (scholium-name (first scholium-id))
                   (possible-reference (get-article scholium-name)))
              (when (typedata-includes
                     (scholium-type possible-reference)
                     'reference)
                (let ((scholium-tag
                       (car (rassoc (reference-to-article
                                     possible-reference)
                                    tags-alist))))
                  (if scholium-tag
                      (progn (delete-region (first next-region)
                                            (second next-region))
                             (goto-char (first next-region))
                             (insert "\[[id:" scholium-tag "]]"))
                    (let ((new-tag (replace-regexp-in-string
                                    " " "_"
                                    (buffer-substring-no-properties
                                     (first next-region)
                                     (second next-region)))))
                      (delete-region (first next-region)
                                     (second next-region))
                      (goto-char (first next-region))
                      (insert "\[[id:" new-tag "]]"))))))
              (setq next-region (move-to-next-region-with-scholium)))))
      (buffer-substring-no-properties (point-min)
                                      (point-max)))))

(defun export-section (section-name)
  (set-buffer (get-buffer "*Export*"))
  (let* ((tag (car (rassoc section-name (scholium-text
                                         (get-article
                                          'Tag-to-Name-converter))))))
    (insert "\\section{" section-name "}"
            (if tag
                (concat " \\label{" tag "}\n\n")
              "\n\n"))))
```

```
(defun export-subsection (subsection-name)
  (set-buffer (get-buffer "*Export*"))
  (let* ((tag (car (rassoc subsection-name (scholium-text
                                            (get-article
                                             'Tag-to-Name-converter))))))
    (insert "\\subsection{" subsection-name "}"
            (if tag
                (concat " \\label{" tag "}\n\n")
              "\n\n")))
  (let ((contents (scholium-text current-item)))
    (dolist (item contents)
      (let ((current-item (get-article item)))
        (if (typedata-includes (scholium-type current-item)
                               'subsubsection)
            (export-subsubsection (scholium-name current-item))
          (export-note current-item))))))

(defun export-subsubsection (subsubsection-name)
  (set-buffer (get-buffer "*Export*"))
  (let* ((tag (car (rassoc subsubsection-name (scholium-text
                                               (get-article
                                                'Tag-to-Name-converter))))))
    (insert "\\subsubsection{" subsubsection-name "}"
            (if tag
                (concat " \\label{" tag "}\n\n")
              "\n\n")))
  (let ((contents (scholium-text current-item)))

    (dolist (item contents)
      (let ((current-item (get-article item)))
        (export-note current-item)))))
```

12.3 Simulating a wiki with the scholium system

12.3.1 On 'mark-up-wiki-with-reference-scholia'

Find things in double brackets, possibly with an intervening "|", making sure to find the minimal double bracket pair. What *should* happen (though the current draft is a bit different) is: a reference scholium should be created for each reference, and stuck into the article list in some appropriate place (presumably the namespace associated with the article in question, more specifically, on some specific reference subspace of that space). *Then* when someone goes to render the document in question, we grab the references out of their storage facility and mark up the document with text properties. (The current implementation is really just a regexp check…)

```
(defun mark-up-wiki-with-reference-scholia ()
  (save-excursion
    (goto-char (point-min))
    (while (re-search-forward
            ;; we could probably get all types of reference in
            ;; one regexp.  I'm not sure why the old
            ;; version for grabbing external references was deleted,
```

```
        ;; but it is featured in the Sept 4 printout.
        "\\[\\[\\([^]|\n]+\\)\\(|\\)?\\([^]]+?\\)?]]" nil t)
    (replace-match (propertize (or (match-string 3)
                                   (match-string 1))
                               'scholia
                               "Reference")))))
```

12.4 Using the scholium system for HDM things

12.4.1 Types for the APM-Ξ

We keep track of the sections, entries, and stubs that belong to APM-Ξ as special types.

```
(add-to-list 'modified-type-labels '(apmxi-section . apmxi-section) t)
(add-to-list 'modified-type-labels '(apmxi-entry . apmxi-entry) t)
(add-to-list 'modified-type-labels '(apmxi-stub . apmxi-stub) t)
```

12.4.2 On 'make-apmxi-sections-into-articles'

This function makes every section in the APM-Ξ into its own article. The list of these articles is recorded on the article 'apmxi-section'.

```
(defun make-apmxi-sections-into-articles ()
  (save-window-excursion
    (find-file "../p2/Xi.tex")
    (goto-char (point-min))
    (while (re-search-forward "^\\\\section{{ \\(.+\\)}} " nil t)
      (let ((article-name (match-string-no-properties 1))
            (beg (point))
            (end (save-excursion (search-forward "\\section" nil t)
                                 (match-beginning 0))))
        (when end
          (scholium article-name
                    (buffer-substring-no-properties beg end)
                    nil
                    'apmxi-section))))))
```

12.4.3 On 'chunk-out-apmxi-definitions'

Apparently some of the terms in APM-Ξ were defined in multiple sections. At least, *stubs* appear in multiple places. This should presumably create multiple definitions when multiple definitions are given, or get rid of stub-like definitions, or something.

Note that the way this is set up, even sections that contain only stubs can also contain references to the real articles, when they exist – this means that the real articles may appear in several listings, which I think is as it should be.

```
(defun chunk-out-apmxi-definitions ()
  (map-label (lambda (name)
               (with-temp-buffer
                 ;; this ensures that sexps are defined properly
                 (latex-mode)
                 (let ((topic (get-article name))
```

```
                contents)
              (insert (scholium-text topic))
              (goto-char (point-min))
              (while (re-search-forward "^(" nil t)
                (let* ((beg (match-beginning 0))
                       (title-beg (match-end 0))
                       (title-end (progn (goto-char beg)
                                         (forward-sexp)
                                         (1- (point))))
                       (stub-p (looking-at " *$"))
                       (title (buffer-substring-no-properties
                               title-beg
                               title-end))
                       (end (save-excursion
                              (search-forward-regexp "^$")))
                       (possible-previous (get-article title)))
                  (when (or (not possible-previous)
                            (typedata-includes (scholium-type
                                                possible-previous)
                                               'apmxi-stub))
                    (scholium title
                              (buffer-substring-no-properties
                               beg
                               end)
                              `((,name parent))
                              (if stub-p
                                  '(apmxi-stub apmxi-entry)
                                'apmxi-entry)))
                  (setq contents (cons title contents))))
              (scholium name
                        contents
                        nil
                        '(apmxi-section label)))))
      'apmxi-section))
```

12.4.4 On 'import-apmxi'

This function puts into play all of the other functions developed above, to fully process the APM-Ξ and get it nicely situated in the scholium system.

```
(defun import-apmxi ()
  (make-apmxi-sections-into-articles)
  (chunk-out-apmxi-definitions))
```

12.4.5 Selecting all APM-Ξ stubs

If you want to look at a list of all stubs in the APM-Ξ, run this code:

```
(article-menu-list-articles-matching-predicate
 (lambda (name)
   (when (typedata-includes (scholium-type (get-article name))
```

```
                              'apmxi-stub)
    t)))
```

(See Note 7.8.18.)

12.4.6 Screening out APM-Ξ stubs

If you want to screen *out* the stubs in the APM-Ξ, run

```
(display-difference-of-labels 'apmxi-entry 'apmxi-stub)
```

(see Note 7.5.21). Of course, eventually we're going to want to do some more processing to make the stubs into useful things.

12.4.7 Facilitate collaboration between HDM authors

13 Part XIII — Reflections and Philosophy

14 Part XIV — Conclusions and Appendices

14.1 Appendix: Overview of specifications

14.1.1 Spec for article

An article is an element of some namespace, by default, the main article-table (a hash table). It takes the form of a quintuplet,

```
(name text about type bookkeeping)
```

where, when stored in the hash table, name is used as the key. (See Note .)

14.1.2 Spec for link

Each link takes the form

```
(<target article's name> &rest <link-types>)
```

The design here maintains partial symmetry between the treatment of article types and link types; the cdr of a link can be processed by the same typedata processing functions as the type data from articles. (See Note 3.2.1.)

14.1.3 Spec for metadata article

An article A has metadata article $\tilde{A}$ whose name is

```
(meta <name of A>)
```

The text of a metadata article takes the form of a list of lists, with each sub-list headed by a tag naming the contents of that sub-list. The about data is 'nil', the type is 'meta', and the bookkeeping data is 'system'. (See Note .)

14.1.4 Spec for backlink

An article's backlinks are stored on the 'backlink' field of its metadata article. An individual backlink takes the form

```
(<name> <link number> [<version number>])
```

where "name" is the name of the article being backlinked, and link number indicates the particular link to which this backlink corresponds. The optional version number is used if the backlink corresponds to a specific version of the linking article other than the most current one. (See Note .)

14.1.5 Spec for link-id

The purpose of a link-id is to specify a link when marking up a piece of text with a text property corresponding to a scholium, namely the link through with that region was indicated (see Note ; the 'scholia' text property contains a list of link-ids.).

The format of a link-id is

```
(<name> <link number>)
```

Notice that link-id's also come up in the context of masks and other places. They are a sort of idiom used in several places in the work (as you can easily see by looking around at the specs shown in this appendix). One thing to notice is that the "usable about data" of Note 7.4.13 *et seq.* leads up to markup fairly directly, and, in particular, that it computes markup by applying masks to links as needed. So, it is no wonder that there is some continuity between these various parts of the system; and, in particular, the fact that link-id's are a part of that continuity makes sense.

14.1.6 Spec for mask

An article's masks are stored in the 'masks' field of its metadata article. An individual mask takes the form

```
((<name> <link number>) &rest regions)
```

The regions are two-element lists, each giving the beginning and end of a region that the link is being redirected to. See Note and Note 7.4.12, and for further discussion of the lack of generality in the current mask format, see Note .

14.1.7 Spec for 'usable-about-data'

Something called "usable-about-data" is produced by the function 'compute-usable-about-data' (see Note 7.4.13). This stuff is used when marking things up (in particular, by 'mark-up-scholium' and 'mark-up-reference'). The reason for documenting it here is that its format is confusing and easy to forget. I think that it would be best if we could find another way to get this data to where it needs to be.

The format of the return value of 'compute-usable-about-data' is a list of elements of the form

```
((<name> <link number>) <beg end | nil>)
```

where name is the name of a scholium, 'beg' and 'end', if specified, give the beginning and end of a marked region (yes, only one region per link, as mentioned in Note `3.2-Creating_scholia.org`), whereas 'nil' is specified if the link indicated by the 'car' of this form points at the whole of the article being rendered.

(I am not a huge fan of this format, but it does seem to work; however, maybe the name should be changed to "usable about data for markup" or something like that, to make it clear why we are focusing on data describing regions.)

(Also, perhaps we should note in this spec that instead of the name, sometimes "(mask name)" is used instead. The development here seems a bit confusing: if we use (mask name), is this a proper link-id? Something to think about, I guess.)

14.2 Appendix: Annotated non-bibliography

14.2.1 Outside references

For now, references are contained in footnotes. To view a compilation of the Notes that contain footnotes, evaluate this form:

```
(progn (article-menu-list-articles-matching-regexp "\\footnote")
  (listing-to-compilation nil))
```

Eventually we may have a nicer way of presenting this data. Also, there are a number of references made in the text for which there are no footnotes, and the footnotes that do exist don't necessarily take a proper or in any way standard form. This can be fixed up eventually.