

Arxana

Joseph Corneli*

Last revised: May 19, 2009

Abstract

A tool for building hackable semantic hypertext platforms.

Contents

1	Introduction	2
2	Using the program	4
3	SQL tables	5
4	Common Lisp-side	7
4.1	Preliminaries	7
4.2	Table definitions	8
4.3	Modifying the database	9
4.4	Queries	13
5	Emacs-side	25
5.1	The interface to Common Lisp	25
5.2	Database interaction	27
5.3	Importing L ^A T _E X documents	30
5.4	Browsing database contents	35
5.5	Exporting L ^A T _E X documents*	40
5.6	Editing database contents*	41
6	Applications	41
6.1	Managing tasks	41
6.2	Other ideas*	44
7	Topics of philosophical interest	45
8	Future plans	46
A	Appendix: Auto-setup	48

*Copyright (C) 2005-2009 Joseph Corneli <holtzermann17@gmail.com>
→ transferred to the public domain.

B Appendix: A simple literate programming system	53
C Appendix: Hypertext platforms	55
D Appendix: Computational Linguistics	56
E Appendix: Resource use	57

1 Introduction

Note 1.1 (*What is “Arxana”?*). *Arxana* is the name of a “next generation” hypertext system that emphasizes annotation. Every object in this system is annotatable. Because of this, I sometimes call Arxana’s core “the scholium system”, but the name “Arxana” better reflects our aim: to explore the mysterious world of links, attachments, correspondences, and side-effects.

Note 1.2 (*The idea*). A scholia-based document model for commons-based peer production will inform the development of our system.¹ In this model, texts are made up of smaller texts until you get to atomic texts; user actions are built in the same way. Multiple users should interact with a shared persistent data-store, through functional annotation, not destructive modification. We should pursue the asynchronous interaction model until we arrive at live, synchronous, settings, where we facilitate real-time computer-mediated interactions between users, and between users and running hackable programs.

Note 1.3 (*The data model*). Three types are fundamental: things, places, and triples. A *thing* is any object that we can store, generally a string. A *place* is a thing that points at the points at just one thing. Places are useful for maintaining distinct copies. A *triple* is a special kind of thing that points at three things. Triples are interpreted as relationships or semantic links; so the triple (A, C, B) indicates that A is related to B by C . Triples can point at anything, including other triples, or components of other triples. (See Note 3.1 for details.)

Note 1.4 (*History*). Thinking about how to improve existing systems for peer-based collaboration in 2004, I designed a simple version of the scholium system that treated textual commentary and markup as scholia.² In 2006, I put together a single-user version of this system that ran exclusively under Emacs.³ The current system is a complete rewrite of that program, bringing in the shared database and various other enhancements for multi-user interaction.

Note 1.5 (*A brisk review of the programming literature*). Many years before I started working on this project, there was something called the Emacs HyperText System.⁴ What we’re doing here updates for modern database methods, uses a more interesting data storage format, and

¹<http://www.metascholar.org/events/2005/freeculture/viewabstract.php?id=19>

²http://wiki.planetmath.org/AsteroidMeta/old_draft_of_scholium_system

³<http://metameso.org/files/sbdm4cbpp.tex>

⁴<http://www.aue.aau.dk/~kock/Publications/HyperBase/>

also considers multiple front-ends to the same database (for example, a web interface).

Contemporary Emacs-based hypertext creation systems include Muse and Emacs Wiki.^{5,6} The browsing side features old standbys, Info and Emacs/w3m. These packages provide ways to author or view what we should now call “traditional” hypertext documents.

Another legacy tool worth mentioning is HyperCard⁷. This system was oriented around the idea of using hypertext to create software, a vision we share, but like just about everyone else working in the field at the time, it used uni-directional links.

Nouveau hypertext is based on semantic triples. The Semantic Web standard provides one specification of the features we can expect from triples.⁸ Triples provide a framework for knowledge representation with more depth and flexibility than the popular “tagging” methodology. For example, suitable collections of triples implement AI-style “frames”. The notion of using triples to organize archival material is generating some interest as Semantic Web ideas spread.^{9,10}

An abstractly similar project to Arxana with some grand goals is being developed by Chris Hanson at MIT under the name “Web-scale Environments for Deduction Systems”.¹¹

Another technically similar project is Freebase, a hand rolled database of open content, organized on frame-based, triple driven, principles. The developer of the Freebase graphd database has more interesting things to say about old and new ways of handling triples.¹²

Note 1.6 (*Fitting in*). Arxana’s theoretical generality, active development status, and superlatively liberal terms of use may make it an attractive option for you to try.

My current development goal is to use this system to create a more flexible multiuser interaction platform than those currently available to web-based collaborative projects (such as PlanetMath¹³). As an intermediate stage, I’m using Arxana to help organize material for a book I’m writing.

Note 1.7 (*What you get*). Arxana comes standard with an Emacs front-end, a Common Lisp middle-end, and an SQL backend. If you want to do some work, any one of these components can be swapped out and replaced with the engine of your choice. I’ve released all of the implementation work on this system into the public domain, and it runs on an entirely free/libre/open source software platform.

Note 1.8 (*Acknowledgements*). Ted Nelson’s “Literary Machines” and Marvin Minsky’s “Society of Mind” are cornerstones in the historical and

⁵<http://mwolson.org/projects/EmacsMuse.html>

⁶<http://mwolson.org/projects/EmacsWiki.html>

⁷<http://en.wikipedia.org/wiki/HyperCard>

⁸<http://www.w3.org/TR/2004/REC-rdf-primer-20040210/>

⁹Cf. recent museum and library conferences

¹⁰Even among academic computer scientists! (Josh Grochow, p.c.)

¹¹<http://publications.csail.mit.edu/abstracts/abstracts07/cph2/cph2.html>

¹²<http://blog.freebase.com/2008/04/09/a-brief-tour-of-graphd/>

¹³<http://planetmath.org>

social contextualization of this work. Alfred Korzybski’s “Science and Sanity” and Gilles Deleuze’s “The Logic of Sense” provided grounding and encouragement. T_EX and GNU Emacs have been useful not just in prototyping this system, but also as exemplary projects in the genre I’m aiming for. McCarthy’s Elephant 2000 was an inspiring thing to look at, and of course Lisp has been a vital ingredient.

I would like to thank (in order of appearance): Steve Corneli, Tom Lenius, Aaron Krowne, Raymond Puzio, Nick Thomas, and Ian Eslick.

2 Using the program

Note 2.1 (*Dependencies*). Our interface is embedded in Emacs. Backend processing is done with Common Lisp. We are currently using the PostgreSQL database. These packages should be available to you through the usual channels. (I’ve been using SBCL, but any Lisp should do; please make sure you are using a contemporary Emacs version.)

We will connect Emacs to Lisp via Slime¹⁴, and Lisp to PostgreSQL via CLSQL.¹⁵ CLSQL also talks directly to the Sphinx search engine, which we use for text-based search.¹⁶ Once all of these things are installed and working together, you should be able to begin to use Arxana.

Setting up all of these packages can be a somewhat time-consuming and confusing task, especially if you haven’t done it before! See Appendix A for help.

Note 2.2 (*Export code and set up the interface*). If you are looking at the source version of this document¹⁷, evaluate the following s-expression in Emacs (type *C-x C-e* with the cursor positioned just after its final parenthesis). This exports the Common Lisp components of the program to suitable files for subsequent use, and prepares the Emacs environment. (See Appendix B.)

```
(save-excursion
  (let ((beg (search-forward "\\begin{verbatim}"))
        (end (progn (search-forward "\\end{verbatim}")
                    (match-beginning 0))))
    (eval-region beg end)
    (lit-process)))
```

Note 2.3 (*To load Common Lisp components at run-time*). Link `arxana.asd` somewhere where Lisp can find it. Then run commands like these in your Lisp; if you like, you can place all of this stuff in your config file to automatically load Arxana when Lisp starts. The final form is only necessary if you plan to use CLSQL’s special syntax on the Lisp command-line.

```
(asdf:operate 'asdf:load-op 'clsql)
(asdf:operate 'asdf:load-op 'arxana)
```

¹⁴<http://common-lisp.net/project/slime/>

¹⁵<http://clsql.b9.com/>

¹⁶<http://www.sphinxsearch.com/>

¹⁷<http://metameso.org/files/arxana.tex>

```
(in-package arxana)
(connect-to-database)
(locally-enable-sql-reader-syntax)
```

Note 2.4 (To connect Emacs to Lisp). Either run `M-x slime RET` to start and connect to Lisp locally, or `M-x slime-connect RET RET` after you have opened a remote connection to your remote server with a command like this: `ssh -L 4005:127.0.0.1:4005 <username>@<host>` and started Lisp and the Swank server on the remote machine. To have Swank start automatically when you start Lisp, put commands like this in your config file.

```
(asdf:operate 'asdf:load-op 'swank)
(setf swank:*use-dedicated-output-stream* nil)
(setf swank:*communication-style* :fd-handler)
(swank:create-server :dont-close t)
```

Note 2.5 (To define database structures). If you haven't yet defined the basic database structures (`tabledefs.lisp`, see Section 3), make sure to load them now!

Note 2.6 (Importing this document into system). You can browse this document inside Arxana: after loading the code, run `M-x autoimport-arxana`.

3 SQL tables

Note 3.1 (Strings, places, and triples). Strings store generic content with simple and familiar semantics. Places exist to store “different copies” of “the same thing”. Between strings and places, we have a system that looks a lot like a Turing tape. Triples provide an overlaid network layer; we'll subsequently develop a special semantics for triples.

Both places and triples are built of references, codes, and supplementary codes. Codes say which table contains the indicated object, and references provide that object's id. The basic codes are 0 for strings, 1 for places, and 2 for triples; we also provide the special codes 3, 4, and 5 for the head, body, and tail of a triple. (Perhaps still more codes should be supplied for head-and-body, etc.)

Supplementary codes are relevant in connection with codes 2 through 5, and say (0) to dereference to the object that appears in that location, or (1) to point at the location itself, regardless of what appears there. (We dereference by default.)

```
CREATE TABLE strings (
  id SERIAL PRIMARY KEY,
  text TEXT NOT NULL UNIQUE
);
```

```
CREATE TABLE places (
  id SERIAL PRIMARY KEY,
  code INT NOT NULL,
  ref INT NOT NULL,
```

```

    sup INT NOT NULL
);

CREATE TABLE triples (
    id SERIAL PRIMARY KEY,
    code1 INT NOT NULL,
    ref1 INT NOT NULL,
    sup1 INT NOT NULL,
    code2 INT NOT NULL,
    ref2 INT NOT NULL,
    sup2 INT NOT NULL,
    code3 INT NOT NULL,
    ref3 INT NOT NULL,
    sup3 INT NOT NULL,
    UNIQUE (code1, ref1, sup1,
            code2, ref2, sup2,
            code3, ref3, sup3)
);

```

Note 3.2 (*Uniqueness of strings and triples*). An attempt to create a duplicate string or triple generates a warning. This saves storage, given possible repetitive use – and avoids confusion. Places allow us to instantiate distinct elements of the classes represented by these strings and triples (or should I say, Platonic forms!?).

Note 3.3 (*Provenance and other details*). We could add much more structure to the database, starting with simple adjustments like adding provenance metadata or versioning into the records for each stored thing. For the time being, I will assume that all metadata will appear in the application or content layer (i.e., as triples).

Note 3.4 (*Models of theories*). I want a basic type of collection, which I call *theories*. I would say “sets”, but I want to convey the idea that an object might be in a collection in each of various different *senses*, maybe even simultaneously; whereas for set membership, it’s simply in or out. (See Note D.2.) The remainder of this note describes several different but “equivalent” ways of working with theories.

We don’t need a SQL table for theories; instead, we represent theory membership using triples: *A is in theory T*. Of course, we could have done it in several different ways.

We could represent $A \in T$ with a pair in a two-column table of elements-[in]-theories. Indeed, for any link that we might use a lot, to save the storage cost associated with all of the “middle” entries, we can create a new table that stores the beginning and end pairs associated via that link.

If we wanted to go to a different extreme, we could opt for a “space-time-like” collection of quadruples, where the first three elements of each quadruple specify a triple and the last element names some theory that this triple is in. In order to talk about the membership of non-triples in theories, we could say something like $(A, \emptyset, \emptyset, T)$. This seems like the utmost in simplicity, but it would require repetitive storage, if, for example, A is also in theory S .

Note 3.5 (*Each place contains one object*). Because I want to assert that each place will contain precisely one thing, it is convenient to describe the object-in-place relationship with a database table. If we were willing to monitor our triples so that triples with the special middle term “ \in_1 ” always had the property that $A \in_1 P$ implies we never see $B \in_1 P$ for $B \neq A$, then we wouldn’t need a separate table for places. However, for ease of use, presumably we would want to keep track of usages of “ \in_1 ” in an indexed theory – which is beginning to sound like a database table all over again.

4 Common Lisp-side

4.1 Preliminaries

System definition

```
(DEFSYSTEM "ARXANA"
  :VERSION "0.0.1"
  :AUTHOR "JOE CORNELI <HOLTZERMANN17@GMAIL.COM>"
  :LICENCE "PUBLIC DOMAIN"
  :COMPONENTS
  ((:FILE "PACKAGES")
   (:FILE "UTILITIES" :DEPENDS-ON ("PACKAGES"))
   (:FILE "DATABASE" :DEPENDS-ON ("UTILITIES"))
   (:FILE "QUERIES" :DEPENDS-ON ("PACKAGES"))))
```

Package definition

```
(DEFPACKAGE :ARXANA
  (:USE #:CL #:CLSQL))
```

Utilities

Note 4.1 (*Useful things*). These definitions are either necessary or useful for working the database and manipulating triple-centric data.

```
(IN-PACKAGE ARXANA)

(DEFUN CONNECT-TO-DATABASE ()
  (CONNECT '("LOCALHOST" "JOE" "JOE" "")
           :DATABASE-TYPE :POSTGRESQL-SOCKET))

(DEFMACRO SELECT-ONE (&REST ARGS)
  '(CAR (SELECT ,@ARGS :FLATP T)))

(DEFMACRO SELECTF (&REST ARGS)
  '(SELECT ,@ARGS :FLATP T))

(DEFUN RESOLVE-AMBIGUITY (STUFF)
  (FIRST STUFF))
```

```
(DEFUN ISOLATE-COMPONENTS (CONTENT I J K)
  (LIST (NTH (1- I) CONTENT)
        (NTH (1- J) CONTENT)
        (NTH (1- K) CONTENT)))
```

```
(DEFUN ISOLATE-BEGINNING (TRIPLE)
  (ISOLATE-COMPONENTS (CDR TRIPLE) 1 2 3))
```

```
(DEFUN ISOLATE-MIDDLE (TRIPLE)
  (ISOLATE-COMPONENTS (CDR TRIPLE) 4 5 6))
```

```
(DEFUN ISOLATE-END (TRIPLE)
  (ISOLATE-COMPONENTS (CDR TRIPLE) 7 8 9))
```

Note 4.2 (*Using a different database*). See Note A.3 for instructions on changes you will want to make if you use a different database.

Note 4.3 (*Resolving ambiguity*). Often it will eventuate that there will be more than one item returned when we were expecting only one item. In order to deal with this sort of ambiguity, it would be great to have either a non-interactive notifier that says that some ambiguity has been dealt with, or an interactive tool that will let the user decide which of the ambiguous options to choose from. For now, we provide the simplest non-interactive tool: just choose the first item from a possibly ambiguous list of items.

4.2 Table definitions

Note 4.4 (*Defining tables from within Lisp*). This is Lisp code to define SQL tables described in Section 3.

```
(EXECUTE-COMMAND "CREATE TABLE STRINGS (
  ID SERIAL PRIMARY KEY,
  TEXT TEXT NOT NULL UNIQUE
);")
```

```
(EXECUTE-COMMAND "CREATE TABLE PLACES (
  ID SERIAL PRIMARY KEY,
  CODE INT NOT NULL,
  REF INT NOT NULL,
  SUP INT
);")
```

```
(EXECUTE-COMMAND "CREATE TABLE TRIPLES (
  ID SERIAL PRIMARY KEY,
  CODE1 INT NOT NULL,
  REF1 INT NOT NULL,
  SUP1 INT NOT NULL,
  CODE2 INT NOT NULL,
  REF2 INT NOT NULL,
  SUP2 INT NOT NULL,
  CODE3 INT NOT NULL,
```

```

REF3 INT NOT NULL,
SUP3 INT NOT NULL,
UNIQUE (CODE1, REF1, SUP1,
        CODE2, REF2, SUP2,
        CODE3, REF3, SUP3)
);")

```

Note 4.5 (*Redefining tables*). In case you ever need to redefine these tables, you can run code like this first, to delete the existing copies.

```

(dolist (view (list-views)) (drop-view view))
(execute-command "DROP TABLE strings")
(execute-command "DROP TABLE triples")
(execute-command "DROP TABLE places")

```

4.3 Modifying the database

```

(IN-PACKAGE ARXANA)
(LOCALLY-ENABLE-SQL-READER-SYNTAX)

```

Processing strings

Note 4.6 (*On ‘string-to-id’*). Return the id of ‘text’, if present, otherwise nil.

```

(DEFUN STRING-TO-ID (TEXT)
  (SELECT-ONE [ID] :FROM [STRINGS]
             :WHERE [= [TEXT] TEXT]))

```

Note 4.7 (*On ‘add-string’*). Add the argument ‘text’ to the list of strings. If the string is successfully created, its coordinates are returned. Otherwise, and in particular, if the request was to create a duplicate, nil is returned.

```

(DEFUN ADD-STRING (TEXT)
  "ADD TEXT TO THE STRINGS TABLE."
  (HANDLER-CASE
   (PROGN (INSERT-RECORDS :INTO [STRINGS]
                        :ATTRIBUTES '(TEXT)
                        :VALUES '(,TEXT))
          '(0 ,(STRING-TO-ID TEXT)))
   (SQL-DATABASE-DATA-ERROR ()
    (WARN "\"~A\" ALREADY EXISTS."
          TEXT))))

```

Note 4.8 (*Error handling bug*). The function ‘add-string’ (Note 4.7) exhibits the first of several error handling calls designed to ensure uniqueness (Note 3.2). Experimentally, this works, but I’m observing that when the user tries to add an item that’s already present in the database, the index tied to the associated table increases. This is annoying. I haven’t checked whether this happens on all possible installations of the underlying software.

Parsing general input

Note 4.9 (*On ‘process-standard-input’*). User input to functions like ‘add-triple’ and so on and so forth can be strings, integers (which the function “serializes” as the string versions of themselves), lists of the form (code ref), or lists of the form (code ref sup). This function converts all of these input forms into this last one! It takes an optional argument ‘addstr’ which, if supplied, says to add string data to the database if it wasn’t there already.

```
(DEFUN PROCESS-STANDARD-INPUT (DATA &OPTIONAL ADDSTR)
  (COND
    ((INTEGERP DATA)
     (PROCESS-STANDARD-INPUT (FORMAT NIL "~A" DATA) ADDSTR))
    ((STRINGP DATA)
     (LET ((ID (STRING-TO-ID DATA)))
       (IF ID
           (LIST 0 ID 0)
           (WHEN ADDSTR
              (ADD-STRING DATA))))))
    ((AND (LISTP DATA)
          (EQUAL (LENGTH DATA) 2))
     (APPEND DATA '(0)))
    ((AND (LISTP DATA)
          (EQUAL (LENGTH DATA) 3))
     DATA)
    (T NIL)))
```

Processing triples

Note 4.10 (*On ‘triple-to-id’*). Return the id of the triple (beg mid end), if present, otherwise nil.

```
(DEFUN TRIPLE-TO-ID (BEG MID END)
  (LET ((B (PROCESS-STANDARD-INPUT BEG))
        (M (PROCESS-STANDARD-INPUT MID))
        (E (PROCESS-STANDARD-INPUT END)))
    (SELECT-ONE [ID] :FROM [TRIPLES]
                :WHERE [AND [= [CODE1] (FIRST B)]
                          [= [REF1] (SECOND B)]
                          [= [SUP1] (THIRD B)]
                          [= [CODE2] (FIRST M)]
                          [= [REF2] (SECOND M)]
                          [= [SUP2] (THIRD M)]
                          [= [CODE3] (FIRST E)]
                          [= [REF3] (SECOND E)]
                          [= [SUP3] (THIRD E)]])))
```

Note 4.11 (*On ‘add-triple’*). Elements of triples are parsed by ‘process-standard-input’ (Note 4.9). If the triple is successfully created, its coordinates are returned. Otherwise, and in particular, if the request was to create a duplicate, nil is returned.

```

(DEFUN ADD-TRIPLE (BEG MID END)
  "ADD A TRIPLE COMPRISED OF BEG MID AND END."
  (LET ((B (PROCESS-STANDARD-INPUT BEG T))
        (M (PROCESS-STANDARD-INPUT MID T))
        (E (PROCESS-STANDARD-INPUT END T)))
    (WHEN (AND B M E)
      (HANDLER-CASE
        (PROGN
          (INSERT-RECORDS
            :INTO [TRIPLES] :ATTRIBUTES '(CODE1 CODE2 CODE3
                                          REF1 REF2 REF3
                                          SUP1 SUP2 SUP3)
            :VALUES '(,(FIRST B) ,(FIRST M) ,(FIRST E)
                    ,(SECOND B) ,(SECOND M) ,(SECOND E)
                    ,(THIRD B) ,(THIRD M) ,(THIRD E)))
          '(2 ,(TRIPLE-TO-ID B M E)))
        (SQL-DATABASE-DATA-ERROR ()
          (WARN "\"~A\" ALREADY ENTERED AS [~A ~A ~A].\"
            (LIST BEG MID END) B M E))))))

```

Processing places

Note 4.12 (*On 'get-places'*). A simple function to find the places that are marked with a given symbol. This function is called by 'put-in-place' (Note 4.13) to find the new record it just inserted. Stylistically, it is a preview of Section 4.4.

```

(DEFUN GET-PLACES (SYMBOL)
  "FIND ALL PLACES WHERE SYMBOL APPEARS."
  (LET ((DATA (PROCESS-STANDARD-INPUT SYMBOL)))
    (SELECTF [ID]
      :FROM [PLACES]
      :WHERE [AND [= [CODE] (FIRST DATA)]
              [= [REF] (SECOND DATA)]
              [= [SUP] (THIRD DATA)]])))

```

Note 4.13 (*On 'put-in-place'*). Symbols to be stored are parsed by 'process-standard-input' (Note 4.9). When the symbol is put in a new place, the id number of that place will be the maximum among all ids of with places holding that symbol.

```

(DEFUN PUT-IN-PLACE (SYMBOL &OPTIONAL ID)
  "PUT SYMBOL IN PLACE ID OR A NEW PLACE IF ID IS NOT
  SPECIFIED."
  (LET ((DATA (PROCESS-STANDARD-INPUT SYMBOL T)))
    (IF ID
      (UPDATE-RECORDS [PLACES]
        :AV-PAIRS '((CODE ,(FIRST DATA))
                  (REF ,(SECOND DATA))
                  (SUP ,(THIRD DATA)))
        :WHERE [= [ID] ID])

```

```

(INsert-RECORDS :INTO [PLACES]
  :ATTRIBUTES '(CODE REF SUP)
  :VALUES '(,(FIRST DATA)
            ,(SECOND DATA)
            ,(THIRD DATA)))
(COND
  (ID '(1 ,ID))
  (T '(1 ,(APPLY 'MAX (GET-PLACES SYMBOL))))))

```

Note 4.14 (*Get total image*). We will likely want a function that would list or show every cell where a given symbol appears (both in places and in triples).

Note 4.15 (*Get partial image*). It seems equally important to be able to get all the places that correspond to a given subcollection. (For example, all the places that make up a text document, or all the places that make up a chess board!)

Note 4.16 (*Finding the most recently placed copy*). Although ‘put-in-place’ can, by default, expect the most recently placed copy of the given symbol to be put in the largest numbered place (and, in particular, the largest numbered place occupied by the given symbol), we may need to change things if we start working with places that belong to fixed subcollections (like those described in Note 4.15). Then again, we may not, but it will be something to look at more closely.

Lookup by id number

Note 4.17 (*The data format that’s best for Lisp*). It is a reasonable question to ask whether or not the an id should be considered part of the given piece of data when that data is no longer in the database. For the functions defined here, the id is an input, and so I am not including it in the output again, because it is already known. For functions like ‘triples-given-beginning’ (See Note 4.29), the id is *not* part of the known data, and so it is returned. This convention could change in the future if it proves to be desirable for each data type to always follow one consistent form in Lisp. Our current choice renders the functions defined here mildly incompatible with the functions for finding components of triples from `utilities.lisp` (See Note 4.1).

At least we’ve chose to use a consistent *order* for data returned from the database, namely the default, id, code1, ref1, sup1, code2, ref2, sup2, code3, ref3, sup3.

```

(DEFUN ID-TO-STRING (ID)
  "RETURN THE TEXT CORRESPONDING TO ID, OTHERWISE NIL."
  (SELECT-ONE [TEXT] :FROM [STRINGS]
             :WHERE [= [ID] ID]))

(DEFUN ID-TO-TRIPLE (ID)
  "RETURN THE TRIPLE DESIGNATED BY ID."
  (SELECT-ONE [CODE1] [REF1] [SUP1]
             [CODE2] [REF2] [SUP2]

```

```

[CODE3] [REF3] [SUP3]
:FROM [TRIPLES]
:WHERE [= [ID] ID])

```

```

(DEFUN ID-TO-PLACE (ID)
"Find the information stored in the place with given ID."
  (SELECT-ONE [CODE] [REF] [SUP]
    :FROM [PLACES]
    :WHERE [= [ID] ID]))

```

Note 4.18 (*Succinct idioms for following pointers*). Since we may often want to look at the contents of a place pointed to by the coordinates of a triple, it seems to me that we may want some slightly more succinct versions of the functions above, that save us the trouble of extracting the id of the item we want to investigate.

```

(DEFUN STRING-CONTENTS (COORDS)
"Return the text indicated by COORDS, otherwise NIL."
  (ID-TO-STRING (SECOND COORDS)))

```

```

(DEFUN PLACE-CONTENTS (COORDS)
"Find the place indicated by COORDS."
  (ID-TO-PLACE (SECOND COORDS)))

```

```

(DEFUN TRIPLE-CONTENTS (COORDS)
"Return the triple indicated by COORDS."
  (ID-TO-TRIPLE (SECOND COORDS)))

```

Note 4.19 (*Switchboard*). Even more succinctly, one function that can get the object indicated by any set of coordinates.

```

(DEFUN SWITCHBOARD (COORDS)
  (COND ((EQ (FIRST COORDS) 0)
    (STRING-CONTENTS COORDS))
    ((EQ (FIRST COORDS) 1)
    (PLACE-CONTENTS COORDS))
    ((EQ (FIRST COORDS) 2)
    (TRIPLE-CONTENTS COORDS))))

```

Note 4.20 (*Anti-pasti*). The readability of this code would probably be improved if we used functions like ‘switchboard’ more frequently. In particular, it would be nice to sweep idioms like ‘(2 ,(car triple))’ under the rug.

```

(LOCALLY-DISABLE-SQL-READER-SYNTAX)

```

4.4 Queries

Note 4.21 (*The use of views*). It is easy enough to select those triples which match simple data, e.g., those triples which have the same beginning, middle, or end, or any combination of these. It is a little more complicated to find items that match criteria specified by several different

triples; for example, to *find all the books by Arthur C. Clarke that are also works of fiction*.

Suppose our collection of triples contains a portion as follows:

Profiles of the Future	is a	book
2001: A Space Odyssey	is a	book
Ender's Game	is a	book
Profiles of the Future	has genre	non-fiction
2001: A Space Odyssey	has genre	fiction
Ender's Game	has genre	fiction
Profiles of the Future	has author	Arthur C. Clarke
2001: A Space Odyssey	has author	Arthur C. Clarke
Ender's Game	has author	Orson Scott Card

One way to solve the given problem would be to find those items that *are written by Arthur C. Clarke* (* “has author” and “Arthur C. Clarke”), that *are books* (* “is a” “book”), and that *are classified as fiction* (* “has genre” “fiction”). We are looking for items that match *all* of these conditions.

Our implementation strategy is: collect the items matching each criterion into a view, then join these views. (See the function ‘satisfy-conditions’ 4.32.)

If we end up working with large queries and a lot of data, this use of views may not be an efficient way to go – but we’ll cross that bridge when we come to it.

```
(IN-PACKAGE ARXANA)
(LOCALLY-ENABLE-SQL-READER-SYNTAX)
```

Printing

Note 4.22 (*On ‘print-system-object’*). The function ‘print-system-object’ bears some resemblance to ‘process-standard-input’, but is for printing instead, and therefore has to be recursive (because triples and places can point to other system objects, printing can be a long and drawn out ordeal).

```
(DEFUN PRINT-SYSTEM-OBJECT (DATA &OPTIONAL COMPONENTS)
  (COND
    ;; JUST RETURN STRINGS
    ((STRINGP DATA)
     DATA)
    ;; ASSUME THE SUP CODE IS 0 IF NOT SPECIFIED
    ((AND (LISTP DATA)
          (EQUAL (LENGTH DATA) 2))
     (PRINT-SYSTEM-OBJECT
      (APPEND DATA '(0))))
    ((AND (LISTP DATA)
          (EQUAL (LENGTH DATA) 3))
     (COND ((EQUAL (FIRST DATA) 0)
            (ID-TO-STRING (SECOND DATA)))
           ((EQUAL (FIRST DATA) 1)
```

```

(CONCATENATE 'STRING
              (FORMAT NIL "P~A|" (SECOND DATA))
              (PRINT-SYSTEM-OBJECT
               (ID-TO-PLACE (SECOND DATA)))
              "|")
(EQUAL (FIRST DATA) 2)
(LET ((CONTENT (ID-TO-TRIPLE (SECOND DATA))))
  (IF COMPONENTS
      (LIST
        (PRINT-COMPONENTS CONTENT 1 2 3)
        (PRINT-COMPONENTS CONTENT 4 5 6)
        (PRINT-COMPONENTS CONTENT 7 8 9))
      (CONCATENATE
        'STRING
        (FORMAT NIL "T~A["
                    (SECOND DATA))
        (PRINT-COMPONENTS CONTENT 1 2 3) "."
        (PRINT-COMPONENTS CONTENT 4 5 6) "."
        (PRINT-COMPONENTS CONTENT 7 8 9) "]""))))
((AND (LISTP DATA)
      (EQUAL (LENGTH DATA) 4))
 (CONCATENATE 'STRING
               (FORMAT NIL "P~A|" (FIRST DATA))
               (PRINT-SYSTEM-OBJECT (CDR DATA)) "|"))
((AND (LISTP DATA)
      (EQUAL (LENGTH DATA) 10))
 (LET ((CONTENT (CDR DATA)))
  (IF COMPONENTS
      (LIST
        (PRINT-COMPONENTS CONTENT 1 2 3)
        (PRINT-COMPONENTS CONTENT 4 5 6)
        (PRINT-COMPONENTS CONTENT 7 8 9))
      (CONCATENATE
        'STRING
        (FORMAT NIL "T~A[" (FIRST DATA))
        (PRINT-COMPONENTS CONTENT 1 2 3) "."
        (PRINT-COMPONENTS CONTENT 4 5 6) "."
        (PRINT-COMPONENTS CONTENT 7 8 9) "]""))))
(T NIL))

(DEFUN PRINT-COMPONENTS (CONTENT I J K)
  (PRINT-SYSTEM-OBJECT (ISOLATE-COMPONENTS
                        CONTENT I J K)))

(DEFUN PRINT-BEGINNING (TRIPLE)
  (PRINT-SYSTEM-OBJECT (ISOLATE-BEGINNING TRIPLE)))

(DEFUN PRINT-MIDDLE (TRIPLE)
  (PRINT-SYSTEM-OBJECT (ISOLATE-MIDDLE TRIPLE)))

```

```
(DEFUN PRINT-END (TRIPLE)
  (PRINT-SYSTEM-OBJECT (ISOLATE-END TRIPLE)))
```

Note 4.23 (*Depth*). If we are going to have complicated recursive references, our printer, and anything else that gives the system some semantics, should come with some sort of “layers” switch that can be used to limit the amount of recursion we do in any given computation.

Note 4.24 (*Printing objects as they appear in Lisp*). With the following functions we provide facilities for printing an object, either from its id or from the expanded form of the data that represents it in Lisp. (This is one good reason to have one standard form for this data; compare Note 4.17. These functions assume that the id *is* part of what’s printed, so if using functions like ‘id-to-triple’ to retrieve data for printing, you’ll have to graft the id back on before printing with these functions.)

```
(DEFUN PRINT-STRING (STRING &OPTIONAL COMPONENTS)
  (PRINT-SYSTEM-OBJECT STRING COMPONENTS))
```

```
(DEFUN PRINT-PLACE (PLACE &OPTIONAL COMPONENTS)
  (PRINT-SYSTEM-OBJECT PLACE COMPONENTS))
```

```
(DEFUN PRINT-TRIPLE (TRIPLE &OPTIONAL COMPONENTS)
  (PRINT-SYSTEM-OBJECT TRIPLE COMPONENTS))
```

```
(DEFUN PRINT-STRING-FROM-ID (ID &OPTIONAL COMPONENTS)
  (PRINT-SYSTEM-OBJECT (LIST 0 ID) COMPONENTS))
```

```
(DEFUN PRINT-PLACE-FROM-ID (ID &OPTIONAL COMPONENTS)
  (PRINT-SYSTEM-OBJECT (LIST 1 ID) COMPONENTS))
```

```
(DEFUN PRINT-TRIPLE-FROM-ID (ID &OPTIONAL COMPONENTS)
  (PRINT-SYSTEM-OBJECT (LIST 2 ID) COMPONENTS))
```

Note 4.25 (*Printing some stuff but not other stuff*). These functions are good for printing lists as come out of the database.

```
(DEFUN PRINT-STRINGS (STRINGS)
  (MAPCAR 'SECOND STRINGS))
```

```
(DEFUN PRINT-PLACES (PLACES &OPTIONAL COMPONENTS)
  (MAPCAR (LAMBDA (ITEM)
    (PRINT-SYSTEM-OBJECT ITEM COMPONENTS))
    PLACES))
```

```
(DEFUN PRINT-TRIPLES (TRIPLES &OPTIONAL COMPONENTS)
  (MAPCAR (LAMBDA (ITEM)
    (PRINT-SYSTEM-OBJECT ITEM COMPONENTS))
    TRIPLES))
```

Note 4.26 (*Printing everything in each table*). These functions collect human-readable versions of everything in each table. Notice that ‘all-strings’ is written differently.

```
(DEFUN ALL-STRINGS ()
  (MAPCAR 'SECOND (SELECT [*] :FROM [STRINGS])))
```

```
(DEFUN ALL-PLACES ()
  (MAPCAR 'PRINT-SYSTEM-OBJECT
    (SELECT [*] :FROM [PLACES])))
```

```
(DEFUN ALL-TRIPLES ()
  (MAPCAR 'PRINT-SYSTEM-OBJECT
    (SELECT [*] :FROM [TRIPLES])))
```

Note 4.27 (*Printing on particular dimensions*). One possible upgrade to the printing functions would be to provide the built-in to “curry” the printout – for example, just print the source nodes from a list of triples. However, it should of course also be possible to do processing like this Lisp after the printout has been made (the point is, it is presumably it is more efficient only to retrieve and format the data we’re actually looking for).

Note 4.28 (*Strings and ids*). Unlike other objects, strings don’t get printed with their ids. We should probably provide an *option* to print with ids (this could be helpful for subsequent work with the strings in question; on the other hand, since strings are being kept unique, we can immediately exchange a string and it’s id, so I’m not sure if it’s necessary to have an explicit “option”).

Functions that establish basic graph structure

Note 4.29 (*Thinking about graph-like data*). Here we have in mind one or more objects (e.g. a particular source and sink) that is associated with potentially any number of triples (e.g. all the possible middles running between these two identified objects). These functions establish various forms of locality or neighborhood within the data.

The results of such queries can be optionally cached in a view, which is useful for further processing (cf. 4.32).

These functions take input in the form of strings and/or coordinates (cf. Note 4.9).

```
(DEFUN TRIPLES-GIVEN-BEGINNING (NODE &OPTIONAL VIEW)
  "GET TRIPLES OUTBOUND FROM THE GIVEN NODE. OPTIONAL
  ARGUMENT VIEW CAUSES THE RESULTS TO BE SELECTED INTO A
  VIEW WITH THAT NAME."
  (LET ((DATA (PROCESS-STANDARD-INPUT NODE))
        (WINDOW (OR VIEW "INTERAL-VIEW")))
    RET)
  (WHEN DATA
    (CREATE-VIEW
      WINDOW
      :AS [SELECT [*]
            :FROM [TRIPLES]
            :WHERE [AND [= [CODE1] (FIRST DATA)]
                    [= [REF1] (SECOND DATA)]
```

```

                                [= [SUP1] (THIRD DATA)]])
  (SETQ RET (SELECT [*] :FROM WINDOW))
  (UNLESS VIEW
    (DROP-VIEW WINDOW))
  RET)))

(DEFUN TRIPLES-GIVEN-END (NODE &OPTIONAL VIEW)
  "GET TRIPLES INBOUND INTO NODE.  OPTIONAL ARGUMENT VIEW
  CAUSES THE RESULTS TO BE SELECTED INTO A VIEW WITH
  THAT NAME."
  (LET ((DATA (PROCESS-STANDARD-INPUT NODE))
        (WINDOW (OR VIEW "INTERAL-VIEW")))
    RET)
  (WHEN DATA
    (CREATE-VIEW
     WINDOW
     :AS [SELECT [*]
          :FROM [TRIPLES]
          :WHERE [AND [= [CODE3] (FIRST DATA)]
                    [= [REF3] (SECOND DATA)]
                    [= [SUP3] (THIRD DATA)]]])
    (SETQ RET (SELECT [*] :FROM WINDOW))
    (UNLESS VIEW
      (DROP-VIEW WINDOW))
    RET)))

(DEFUN TRIPLES-GIVEN-MIDDLE (EDGE &OPTIONAL VIEW)
  "GET THE TRIPLES THAT RUN ALONG EDGE.  OPTIONAL ARGUMENT
  VIEW CAUSES THE RESULTS TO BE SELECTED INTO A VIEW
  WITH THAT NAME."
  (LET ((DATA (PROCESS-STANDARD-INPUT EDGE))
        (WINDOW (OR VIEW "INTERAL-VIEW")))
    RET)
  (WHEN DATA
    (CREATE-VIEW
     WINDOW
     :AS [SELECT [*]
          :FROM [TRIPLES]
          :WHERE [AND [= [CODE2] (FIRST DATA)]
                    [= [REF2] (SECOND DATA)]
                    [= [SUP2] (THIRD DATA)]]])
    (SETQ RET (SELECT [*] :FROM WINDOW))
    (UNLESS VIEW
      (DROP-VIEW WINDOW))
    RET)))

(DEFUN TRIPLES-GIVEN-MIDDLE-AND-END (EDGE NODE &OPTIONAL
  VIEW)
  "GET THE TRIPLES THAT RUN ALONG EDGE INTO NODE.
  OPTIONAL ARGUMENT VIEW CAUSES THE RESULTS TO BE

```

```

SELECTED INTO A VIEW WITH THAT NAME."
(LET ((EDGEDATA (PROCESS-STANDARD-INPUT EDGE))
      (NODEDATA (PROCESS-STANDARD-INPUT NODE))
      (WINDOW (OR VIEW "INTERAL-VIEW")))
      RET)
(WHEN (AND EDGEDATA NODEDATA)
      (CREATE-VIEW
       WINDOW
       :AS [SELECT [*]
           :FROM [TRIPLES]
           :WHERE [AND [= [CODE2] (FIRST EDGEDATA)]
                     [= [REF2] (SECOND EDGEDATA)]
                     [= [SUP2] (THIRD EDGEDATA)]
                     [= [CODE3] (FIRST NODEDATA)]
                     [= [REF3] (SECOND NODEDATA)]
                     [= [SUP3] (THIRD NODEDATA)]]]])
      (SETQ RET (SELECT [*] :FROM WINDOW))
      (UNLESS VIEW
              (DROP-VIEW WINDOW))
      RET)))

(DEFUN TRIPLES-GIVEN-BEGINNING-AND-MIDDLE (NODE EDGE
                                           &OPTIONAL VIEW)
      "GET THE TRIPLES THAT RUN FROM NODE ALONG EDGE.
OPTIONAL ARGUMENT VIEW CAUSES THE RESULTS TO BE SELECTED
INTO A VIEW WITH THAT NAME."
      (LET ((NODEDATA (PROCESS-STANDARD-INPUT NODE))
            (EDGEDATA (PROCESS-STANDARD-INPUT EDGE))
            (WINDOW (OR VIEW "INTERAL-VIEW")))
            RET)
      (WHEN (AND NODEDATA EDGEDATA)
            (CREATE-VIEW
             WINDOW
             :AS [SELECT [*]
                 :FROM [TRIPLES]
                 :WHERE [AND [= [CODE1] (FIRST NODEDATA)]
                           [= [REF1] (SECOND NODEDATA)]
                           [= [SUP1] (THIRD NODEDATA)]
                           [= [CODE2] (FIRST EDGEDATA)]
                           [= [REF2] (SECOND EDGEDATA)]
                           [= [SUP2] (THIRD EDGEDATA)]]]])
            (SETQ RET (SELECT [*] :FROM WINDOW))
            (UNLESS VIEW
                    (DROP-VIEW WINDOW))
            RET)))

(DEFUN TRIPLES-GIVEN-BEGINNING-AND-END (NODE1 NODE2
                                         &OPTIONAL VIEW)
      "GET THE TRIPLES THAT RUN FROM NODE1 TO NODE2.  OPTIONAL
ARGUMENT VIEW CAUSES THE RESULTS TO BE SELECTED

```

```

        INTO A VIEW WITH THAT NAME."
(LET ((NODE1DATA (PROCESS-STANDARD-INPUT NODE1))
      (NODE2DATA (PROCESS-STANDARD-INPUT NODE2))
      (WINDOW (OR VIEW "INTERAL-VIEW")))
      RET)
(WHEN (AND NODE1DATA NODE2DATA)
      (CREATE-VIEW
        WINDOW
        :AS [SELECT [*]
              :FROM [TRIPLES]
              :WHERE [AND [= [CODE1] (FIRST NODE1DATA)]
                          [= [REF1] (SECOND NODE1DATA)]
                          [= [SUP1] (THIRD NODE1DATA)]
                          [= [CODE3] (FIRST NODE2DATA)]
                          [= [REF3] (SECOND NODE2DATA)]
                          [= [SUP3] (THIRD NODE2DATA)]]])
        (SETQ RET (SELECT [*] :FROM WINDOW))
        (UNLESS VIEW
          (DROP-VIEW WINDOW))
        RET)))

;; THIS ONE USE 'SELECT-ONE' INSTEAD OF 'SELECT'

(DEFUN TRIPLE-EXACT-MATCH (NODE1 EDGE NODE2 &OPTIONAL
                          VIEW)
  "GET THE TRIPLES THAT RUN FROM NODE1 ALONG EDGE TO
  NODE2.  OPTIONAL ARGUMENT VIEW CAUSES THE RESULTS TO BE
  SELECTED INTO A VIEW WITH THAT NAME."
  (LET ((NODE1DATA (PROCESS-STANDARD-INPUT NODE1))
        (EDGEDATA (PROCESS-STANDARD-INPUT EDGE))
        (NODE2DATA (PROCESS-STANDARD-INPUT NODE2))
        (WINDOW (OR VIEW "INTERAL-VIEW")))
        RET)
    (WHEN (AND NODE1DATA EDGEDATA NODE2DATA)
      (CREATE-VIEW
        WINDOW
        :AS [SELECT [*]
              :FROM [TRIPLES]
              :WHERE [AND [= [CODE1] (FIRST NODE1DATA)]
                          [= [REF1] (SECOND NODE1DATA)]
                          [= [SUP1] (THIRD NODE1DATA)]
                          [= [CODE2] (FIRST EDGEDATA)]
                          [= [REF2] (SECOND EDGEDATA)]
                          [= [SUP2] (THIRD EDGEDATA)]
                          [= [CODE3] (FIRST NODE2DATA)]
                          [= [REF3] (SECOND NODE2DATA)]
                          [= [SUP3] (THIRD NODE2DATA)]]])
        (SETQ RET (SELECT-ONE [*] :FROM WINDOW))
        (UNLESS VIEW
          (DROP-VIEW WINDOW))
        RET)))

```

```

RET)))

(DEFUN TRIPLE-RELAXED-MATCH (NODE1 EDGE NODE2 &OPTIONAL
  VIEW)
  "GET THE TRIPLES THAT RUN FROM NODE1 ALONG EDGE TO
  NODE2.  OPTIONAL ARGUMENT VIEW CAUSES THE RESULTS TO BE
  SELECTED INTO A VIEW WITH THAT NAME.  THIS VERSION IS
  'RELAXED' IN THAT IT DOESN'T CARE ABOUT SUPPLEMENTARY
  CODES.  "
  (LET ((NODE1DATA (PROCESS-STANDARD-INPUT NODE1))
        (EDGEDATA (PROCESS-STANDARD-INPUT EDGE))
        (NODE2DATA (PROCESS-STANDARD-INPUT NODE2))
        (WINDOW (OR VIEW "INTERNAL-VIEW")))
    RET)
  (WHEN (AND NODE1DATA EDGEDATA NODE2DATA)
    (CREATE-VIEW
     WINDOW
     :AS [SELECT [*]
          :FROM [TRIPLES]
          :WHERE [AND [= [CODE1] (FIRST NODE1DATA)]
                     [= [REF1] (SECOND NODE1DATA)]
                     [= [CODE2] (FIRST EDGEDATA)]
                     [= [REF2] (SECOND EDGEDATA)]
                     [= [CODE3] (FIRST NODE2DATA)]
                     [= [REF3] (SECOND NODE2DATA)]]]])
    (SETQ RET (SELECT-ONE [*] :FROM WINDOW))
    (UNLESS VIEW
      (DROP-VIEW WINDOW))
    RET)))

```

Note 4.30 (*Becoming flexible about a string's status*). One possible upgrade would be to provide versions of these functions that will flexibly accept either a string or a "placed string" as input (since frequently we're interested in content of that sort; see 5.17).

Finding places that satisfy some property

Note 4.31 (*On 'get-places-subject-to-constraint'*). Like 'get-places' (Note 4.12), but this time takes an extra condition of the form (A C B) where one of A, B, and C is 'nil'. We test each of the places in place of this 'nil', to see if a triple matching that criterion exists.

```

(DEFUN GET-PLACES-SUBJECT-TO-CONSTRAINT (SYMBOL CONDITION)
  (LET ((CANDIDATE-PLACES (GET-PLACES SYMBOL))
        ACCEPTED-PLACES)
    (DOLIST (PLACE CANDIDATE-PLACES)
      (LET ((FILLED-CONDITION
             (MAP 'LIST (LAMBDA (ELT) (OR ELT
                                           '(1 ,PLACE))))
            CONDITION)))
        (WHEN (APPLY 'TRIPLE-RELAXED-MATCH

```

```

                                FILLED-CONDITION)
      (SETQ ACCEPTED-PLACES
        (CONS PLACE ACCEPTED-PLACES))))))
ACCEPTED-PLACES))

```

Logic

Note 4.32 (*On ‘satisfy-conditions’*). This function finds the items which match constraints. Constraints take the form (A B C), where precisely one of A, B, or C should be ‘nil’, and any of the others can be either input suitable for ‘process-standard-input’, or ‘t’. The ‘nil’ entry stands for the object we’re interested in. Any ‘t’ entries are wildcards.

The first thing that happens as the function runs is that views are established exhibiting each group of triples satisfying each predicate. The names of these views are then massaged into a large SQL query. (It is important to “typeset” all of this correctly for our SQL ‘query’.) Finally, once that query has been run, we clean up, dropping all of the views we created.

```

(DEFUN SATISFY-CONDITIONS (CONSTRAINTS)
  (LET* ((VIEWS (GENERATE-VIEWS CONSTRAINTS))
        (FORMATTED-LIST-OF-VIEWS (FORMAT-VIEWS
                                   VIEWS))
        (WHERE-CONDITION (GENERATE-WHERE-CONDITION
                           VIEWS
                           CONSTRAINTS)))
    (RET
     ;; LET’S SEE WHAT THE QUERY IS, FIRST OF ALL.
     (QUERY
      (CONCATENATE
       'STRING
       "SELECT V1.ID, V1.CODE1, V1.REF1, V1.SUP1, "
                          "V1.CODE2, V1.REF2, V1.SUP2, "
                          "V1.CODE3, V1.REF3, V1.SUP3 "
       "FROM "
       FORMATTED-LIST-OF-VIEWS
       "WHERE "
       WHERE-CONDITION
       ";"))))
    (MAPC (LAMBDA (NAME) (DROP-VIEW NAME)) VIEWS)
    RET))

```

Note 4.33 (*Subroutines for ‘satisfy-conditions’*). The functions below produce bits and pieces of the SQL query that ‘satisfy-conditions’ submits. The point of the ‘generate-views’ is to create a series of views centered on the term(s) we’re interested in (the ‘nil’ slots in each submitted constraint). With ‘generate-where-condition’, we insist that all of these interesting terms should, in fact, be equal to one another.

```

(DEFUN GENERATE-VIEWS (CONSTRAINTS)
  (LET ((COUNTER 0))

```

```

(MAPCAR
(LAMBDA (CONSTRAINT)
  (SETQ COUNTER (1+ COUNTER))
  (LET ((VIEWNAME (FORMAT NIL "V~A" COUNTER)))
    ;; HERE FOR EACH CONSTRAINT WE MUST SELECT THE
    ;; APPROPRIATE FUNCTION TO GENERATE THE VIEW;
    ;; AT THE VERY END OF THE COND FORM, WE SPIT
    ;; OUT THE VIEWNAME (FOR 'MAPCAR' TO ADD TO
    ;; THE LIST OF VIEWS)
    (COND
      ;; A * ? OR A ? *
      ((OR (AND (EQ (SECOND CONSTRAINT) T)
                (EQ (THIRD CONSTRAINT) NIL))
           (AND (EQ (SECOND CONSTRAINT) NIL)
                (EQ (THIRD CONSTRAINT) T)))
        (TRIPLES-GIVEN-BEGINNING
         (FIRST CONSTRAINT)
         VIEWNAME))
      ;; * B ? OR ? B *
      ((OR (AND (EQ (FIRST CONSTRAINT) T)
                (EQ (THIRD CONSTRAINT) NIL))
           (AND (EQ (FIRST CONSTRAINT) NIL)
                (EQ (THIRD CONSTRAINT) T)))
        (TRIPLES-GIVEN-MIDDLE
         (SECOND CONSTRAINT)
         VIEWNAME))
      ;; * ? C OR ? * C
      ((OR (AND (EQ (FIRST CONSTRAINT) T)
                (EQ (SECOND CONSTRAINT) NIL))
           (AND (EQ (FIRST CONSTRAINT) NIL)
                (EQ (SECOND CONSTRAINT) T)))
        (TRIPLES-GIVEN-END
         (THIRD CONSTRAINT)
         VIEWNAME))
      ;; ? B C
      ((EQ (FIRST CONSTRAINT) NIL)
        (TRIPLES-GIVEN-MIDDLE-AND-END
         (SECOND CONSTRAINT)
         (THIRD CONSTRAINT)
         VIEWNAME))
      ;; A ? C
      ((EQ (SECOND CONSTRAINT) NIL)
        (TRIPLES-GIVEN-BEGINNING-AND-MIDDLE
         (FIRST CONSTRAINT)
         (SECOND CONSTRAINT)
         VIEWNAME))
      ;; A C ?
      ((EQ (THIRD CONSTRAINT) NIL)
        (TRIPLES-GIVEN-BEGINNING-AND-END
         (FIRST CONSTRAINT)

```

```

        (THIRD CONSTRAINT)
        VIEWNAME)))
    VIEWNAME))
  CONSTRAINTS)))

(DEFUN FORMAT-VIEWS (VIEWS)
  (LET ((FORMATTED-LIST-OF-VIEWS ""))
    (MAPC (LAMBDA (VIEW)
      (SETQ FORMATTED-LIST-OF-VIEWS
        (CONCATENATE
          'STRING
          FORMATTED-LIST-OF-VIEWS
          (FORMAT NIL "~A," VIEW))))
      (BUTLAST VIEWS))
    (SETQ FORMATTED-LIST-OF-VIEWS
      (CONCATENATE
        'STRING
        FORMATTED-LIST-OF-VIEWS
        (FORMAT NIL "~A " (CAR (LAST VIEWS)))))
    FORMATTED-LIST-OF-VIEWS))

(DEFUN GENERATE-WHERE-CONDITION (VIEWS CONDITIONS)
  (LET ((WHERE-CONDITION "")
        (C (SELECT-COMPONENT (FIRST CONDITIONS))))
    ;; THERE SHOULD BE ONE LESS "=" CONDITION THAN THERE
    ;; IS # OF THINGS TO COMPARE; UNTIL WE GET TO THE LAST
    ;; VIEW, EVERYTHING IS JOINED TOGETHER BY AN 'AND'.
    ;; -- THIS NEEDS TO CONSIDER (MAP OVER) BOTH 'VIEWS'
    ;; AND 'CONDITIONS'.
    (LOOP
      FOR I FROM 1 UPTO (1- (LENGTH VIEWS))
      DO
        (LET ((COMPI (SELECT-COMPONENT (NTH I CONDITIONS)))
              (VIEWI (NTH I VIEWS)))
          (SETQ
            WHERE-CONDITION
            (CONCATENATE
              'STRING
              WHERE-CONDITION
              (CONCATENATE
                'STRING
                "(v1.CODE" C " = " VIEWI ".CODE" COMPI ") AND "
                "(v1.REF" C " = " VIEWI ".REF" COMPI ") AND "
                "(v1.SUP" C " = " VIEWI ".SUP" COMPI ") AND "))))))
        (LET ((VIEWN (NTH (1- (LENGTH VIEWS)) VIEWS))
              (COMPONENT (SELECT-COMPONENT
                (NTH (LENGTH VIEWS) CONDITIONS))))
          (SETQ
            WHERE-CONDITION
            (CONCATENATE

```

```

'STRING
WHERE-CONDITION
"(V1.CODE" C " = " VIEWN ".CODE" COMPN ") AND "
"(V1.REF" C " = " VIEWN ".REF" COMPN ") AND "
"(V1.SUP" C " = " VIEWN ".SUP" COMPN ")"))
WHERE-CONDITION))

(DEFUN SELECT-COMPONENT (CONDITION)
  (COND ((EQ (FIRST CONDITION) NIL) "1")
        ((EQ (SECOND CONDITION) NIL) "2")
        ((EQ (THIRD CONDITION) NIL) "3")))

(LOCALLY-DISABLE-SQL-READER-SYNTAX)

```

Note 4.34 (*Even more complicated logic*). In order to conveniently manage complex queries, it would be nice if we could store the results of earlier queries into views, so that we can combine several such views for further processing.

5 Emacs-side

5.1 The interface to Common Lisp

Note 5.1 (*On ‘Defun’*). A way to define Elisp functions whose bodies are evaluated by Common Lisp. Trust me, this is a good idea. Besides, it exhibits some fascinating backquote and comma tricks. But be careful: this definition of ‘Defun’ did not work on Emacs version 21.

If we want to be able to feed in a standard arglist to Common Lisp (with optional elements and so forth), we’d have define how these arguments are handled here!

```

(DEFMACRO DEFUN (NAME ARGLIST &REST BODY)
  (DECLARE (INDENT DEFUN))
  ‘(DEFUN ,NAME ,ARGLIST
    (LET* ((OUTBOUND-STRING
            (TRANSLATE-EMACS-SYNTAX-TO-COMMON-SYNTAX
             (FORMAT "%S"
                    (APPEND
                     (LIST
                      (APPEND (LIST 'LAMBDA ',ARGLIST)
                               ',BODY))
                     (MAPCAR
                      (LAMBDA (ARG) ‘’,ARG)
                      (LIST
                       ,@(REMOVE-IF
                          (LAMBDA (TESTELT)
                            (EQ TESTELT
                              '&OPTIONAL))
                          ARGLIST)))))))
      (RETURNED-STRING
       (SECOND

```

```

;; WE NOW SPECIFY THE RIGHT PACKAGE!
(SLIME-EVAL
 (LIST 'SWANK:EVAL-AND-GRAB-OUTPUT
       OUTBOUND-STRING
       :ARXANA)))
(PROCESS-SLIME-OUTPUT RETURNED-STRING)))

```

Note 5.2 (*On ‘process-slime-output’*). This should downcase all constituent symbols, but for expediency I’m just downcasing ‘NIL’ at the moment. Will come back for more testing and downcasing shortly. (I suspect the general case is just about as easy as what happens here.)

```

(DEFUN PROCESS-SLIME-OUTPUT (STR)
 (CONDITION-CASE NIL
  (LET ((READ-VALUE (READ STR)))
    (IF (SYMBOLP READ-VALUE)
        (READ (DOWNCASE STR))
        (NSUBST NIL 'NIL READ-VALUE))
    (ERROR STR)))

(DEFUN TRANSLATE-EMACS-SYNTAX-TO-COMMON-SYNTAX (STR)
 (WITH-TEMP-BUFFER
  (INSERT STR)
  (DOLIST (SWAP '("("(\\' " "'))
               ("(\\", " ","))))
    (GOTO-CHAR (POINT-MIN))
    (WHILE (SEARCH-FORWARD (FIRST SWAP) NIL T)
      (GOTO-CHAR (MATCH-BEGINNING 0))
      (FORWARD-SEXP)
      (DELETE-CHAR -1)
      (GOTO-CHAR (MATCH-BEGINNING 0))
      (DELETE-REGION (MATCH-BEGINNING 0)
                     (MATCH-END 0))
      (INSERT (SECOND SWAP))))
 (BUFFER-SUBSTRING-NO-PROPERTIES (POINT-MIN)
                                  (POINT-MAX))))

```

Note 5.3 (*Interactive ‘Defun’*). Note, an improved version of this macro would allow me to specify that some Defuns are interactive and some are not. This could be done by examining the submitted body, and adjusting the defun if its car is an ‘interactive’ form. Most of the Defuns will be things that people will want to use interactively, so making this change would probably be a good idea. What I’m doing in the mean time is just writing 2 functions each time I need to make an interactive function that accesses Common Lisp data!

Note 5.4 (*Common Lisp evaluation of code chunks*). Another potentially beneficial and simple approach is to write a form like ‘progn’ that evaluates its contents on Common Lisp. This saves us from having to rewrite all of the ‘defun’ facilities into ‘Defun’ (e.g. interactivity). But... the problem with *this* is that Common Lisp doesn’t know the names of all the variables that are defined in Emacs! I’m not sure how to get all of

the values of these variable substituted *first*, before the call to Common Lisp is made.

Note 5.5 (*Debugging ‘Defun’*). In order to make debugging go easier, it might be nice to have an option to make the code that is supposed to be evaluated by Defun actually *print* on the REPL instead of being processed through an invisible back-end. There could be a couple of different ways to do that, one would be to simulate just what a user might do, the other would be a happy medium between that and what we’re doing now: just put our computery auto-generated code on the REPL and evaluate it. (To some extent, I think the **slime-events** buffer captures this information, but it is not particularly easy to read.)

Note 5.6 (*Interactive Common Lisp?*). Suppose we set up some kind of interactive environment in Common Lisp; how would we go about passing this environment along to a user interacting via Emacs? (Note that SLIME’s presentation of the debugging loop is one good example.)

5.2 Database interaction

Note 5.7 (*The ‘article’ function*). You can use this function to create an article with a given name and contents. If you like you can put it in a theory and/or place.

```
(DEFUN ARTICLE (NAME CONTENTS &OPTIONAL THEORY PLACE)
  (LET ((COORDINATES (ADD-TRIPLE NAME
                                "HAS CONTENT"
                                CONTENTS)))
    (WHEN THEORY (ADD-TRIPLE COORDINATES "IN" THEORY))
    (WHEN PLACE (IF (NUMBERP PLACE)
                    (PUT-IN-PLACE COORDINATES PLACE)
                    (PUT-IN-PLACE COORDINATES)))
    COORDINATES))
```

Note 5.8 (*The ‘scholium’ function*). You can use this function to link annotations to objects. As with the ‘article’ function, you can optionally categorize the connection in a theory or instantiate it in a place (cf. Note 5.7).

```
(DEFUN SCHOLIUM (BEGINNING LINK END &OPTIONAL THEORY PLACE)
  (LET ((COORDINATES (ADD-TRIPLE BEGINNING
                                LINK
                                END)))
    (WHEN THEORY (ADD-TRIPLE COORDINATES "IN" THEORY))
    (WHEN PLACE (IF (NUMBERP PLACE)
                    (PUT-IN-PLACE COORDINATES PLACE)
                    (PUT-IN-PLACE COORDINATES)))
    COORDINATES))
```

Note 5.9 (*Uses of coordinates*). Note that, if desired, you can feed input of the form *'(code) (ref)'* into ‘article’ and ‘scholium’. It’s convenient to do further any processing of the object we’ve created, while we still have ahold of the coordinates returned by ‘add-triple’ (cf. Note 5.25 for an example).

Note 5.10 (*Finding all the places in a theory?*). There's an easy way to tell what type a given target has based on inspection of the targeting link. We just need to make sure that everything that needs to be "in" the theory is, in fact, in the theory, then narrow that selection according to type.

Note 5.11 (*On 'get-article'*). Get the contents of the article named 'name'. Optional argument 'theory' lets us find and use the place in the given theory that holds the name, instead of the name itself.

We do not yet deal well with the ambiguous case in which there are several places that correspond to the given name the given theory.

Note also that out of the listing returned by 'triples-given-beginning-and-middle', we should pick the (hopefully just) ONE that corresponds to the given theory. This means we need to pick over the list of triples returned here, and test each one to see if it is in our theory. As to WHY there might be more than one "has content" for a place that we know to be in our theory... I'm not sure. I guess we can go with the assumption that there is just one, for now.

```
(DEFUN GET-ARTICLE (NAME &OPTIONAL THEORY)
  (LET* ((PLACE-PSEUDONYMS
         (IF THEORY
             (GET-PLACES-SUBJECT-TO-CONSTRAINT
              NAME '(NIL "IN" ,THEORY))
             (GET-PLACES NAME)))
        (GOES-BY (COND
                  ((EQ (LENGTH PLACE-PSEUDONYMS) 1)
                   '(1 ,(CAR PLACE-PSEUDONYMS)))
                  ((TRIPLE-EXACT-MATCH
                    NAME "IN" THEORY)
                   NAME)
                  ((NOT THEORY) NAME)
                  (T NIL))))
    (WHEN GOES-BY
      ;; IT MIGHT BE NICE TO ALSO RETURN 'GOES-BY'
      ;; SO WE CAN ACCESS THE APPROPRIATE PLACE AGAIN.
      (THIRD (PRINT-TRIPLE
              (RESOLVE-AMBIGUITY
               (TRIPLES-GIVEN-BEGINNING-AND-MIDDLE
                GOES-BY "HAS CONTENT"))
              T))))))
```

Note 5.12 (*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.

```
(DEFUN GET-NAMES (&OPTIONAL THEORY)
  (LET ((CONDITIONS (LIST (LIST NIL "HAS CONTENT" T))))
    (WHEN THEORY
      (SETQ CONDITIONS
        (APPEND CONDITIONS
          (LIST (LIST NIL "IN" THEORY))))))
```

```

(MAPCAR
 (LAMBDA (PLACE-OR-STRING)
  (COND ((EQ (FIRST PLACE-OR-STRING) 1) ; PLACE CASE
        (PRINT-SYSTEM-OBJECT
         (ID-TO-PLACE (SECOND PLACE-OR-STRING))))
        ; STRING CASE
        ((EQ (FIRST PLACE-OR-STRING) 0)
         (PRINT-SYSTEM-OBJECT PLACE-OR-STRING))))
 (MAPCAR
  (LAMBDA (TRIPLE)
   (ISOLATE-COMPONENTS (CDR TRIPLE) 1 2 3))
  (SATISFY-CONDITIONS CONDITIONS))))

```

Note 5.13 (*Contrasting cases*). Consider the difference between

```

(? "has author" "Arthur C. Clarke")
(? "has genre" "fiction")

```

and

```

(name "has content" *)
(name "in" "theory")

```

where, in the latter case, we know *who* we're talking about, and we just want to limit the list of items generated by the "*" by the second condition. This should help illustrate the difference between 'get-names' (which is making a general query) and 'get-article' (which already knows the name of a specific article), and the logic that they use.

Note 5.14 (*Placing items from Emacs*). We periodically need to place items from within Emacs. The function 'place-item' is a wrapper for 'put-in-place' that makes this possible (it also provides the user with an extra option, namely to put the place itself into a theory).

Notice that when the symbol is placed in some pre-existing place (which can only happen when 'id' is not nil), that place may already be in some other theory. We will ignore this case for now (since it seems that putting objects into *new* places will be the preferred action), but later we will have to look at what to do in this other case.

```

(DEFUN PLACE-ITEM (SYMBOL &OPTIONAL ID THEORY)
 (LET ((COORDINATES (PUT-IN-PLACE SYMBOL ID)))
  (WHEN THEORY (ADD-TRIPLE COORDINATES "IN" THEORY))
  COORDINATES))

```

Note 5.15 (*Automatic classifications*). It will presumably make sense to offer increasingly "automatic" classifications for new objects. At this point, we've set things up so that the user can optionally supply the name of *one* theory that their new object is a part of.

It may make more sense to allow an '&rest theories' argument, and add the triple to all of the specified theories. This would require modifying 'Defun' to accommodate the '&rest' idiom; see Note 5.1.

Note 5.16 (*Postconditions and provenance*). After adding something to the database, we may want to do something extra; perhaps generating

provenance information, perhaps checking or enforcing database consistency, or perhaps running a hook that causes some update in the frontend (cf. Note 3.3). Provisions of this sort will come later, as will short-hand convenience functions for making particularly common complex entries.

5.3 Importing L^AT_EX documents

Note 5.17 (*Importing sketch*). The code in this section imports a document as a collection of (sub-)sections and notes. It gathers the sections, sub-sections, and notes recursively and records their content in a tree whose nodes are places (Note 3.5) and whose links express the “component-of” relation described in Note 5.19.

This representation lets us see the geometric, hierarchical, structure of the document we’ve imported. It exemplifies a general principle, that geometric data should be represented by relationships between places, not direct relationships between strings. This is because “the same” string often appears in “different” places in any given document (e.g. a paper’s many sub-sections titled “Introduction” will not all have the same content).

What goes into the places is in some sense arbitrary. The key is that whatever is *in* or *attached* to these places must tell us everything we need to know about the part of the document associated with that place (e.g. in the case of a note, its title and contents). That’s over and above the *structural* links which say how the places relate to one another. Finally, all of these places and structural links will be added to a theory that represents the document as a whole.

A natural convention we’ll use will be to put the name of any document component that’s associated with a given place into that place, and add all other information as annotations.

Note 5.18 (*Ordered versus unordered data*). The code in this section is an example of one way to work with ordered data (i.e. L^AT_EX documents are not just hierarchical, but the elements at each level of the hierarchy are also ordered).

Since *many* artifacts are hierachical (e.g. Lisp code), we should try to be compatible with *native* methods for working with order (in the case of Lisp, feed the code into a Lisp processor and use CDR and CAR, etc.).

We *can* use triples such as (**rank 1 Fred**) and (**rank 2 Barney**) to talk about order. There may be some SQL techniques that would help. (FYI, order can be handled very explicitly in Elephant!)

In order to account for *different* orderings, we need one more piece of data – some explicit treatment of where the order *is*; in other words, theories. (This table illustrates the fact that a theory is not so different from “an additional triple”; indeed, the only reason to make them different is to have the extra convenience of having their elements be numbered.)

rank	1	Fred	Friday
rank	2	Barney	Friday
rank	1	Barney	Saturday
rank	2	Fred	Saturday

Note 5.19 (*The order of order*). The triples (rank 1 Fred) and (rank 2 Barney) mentioned in Note 5.18 are easy enough to read and understand; it might be more “natural” for us to say (Fred rank 1) and (Barney rank 2); see Note 7.4. In this section, we’re concerned with talking about the ordered parts of a document, and (A n B) seems like an intuitive way to say “Document A’s nth component is Article B”.

Note 5.20 (*It’s not overdoing it, right?*). When importing *this* document, we see links like the following. I hope that’s not “overdoing it”. (Take a look at Note 5.11 and Note 5.12 to see how we go about getting information out of the database.) We could get rid of one link if theories were database objects (cf. Note 8.1).

```
"T557[P135|Web interface|.in.arxana.tex]"
"T558[Future plans.9.P135|Web interface|]"
"T559[T558[Future plans.9.P135|Web interface|].in.arxana.tex]"
```

Note 5.21 (*Importing in general*). We will eventually have a collection of parsers to get various kinds of documents into the system in various different ways (Note 8.10). For now, this section gives a simple way to get some sorts of L^AT_EX documents into the system, namely documents structured along the same lines as the document you’re reading now!

An interesting approach to parsing *math* documents has been undertaken in the L^AT_EXML project.¹⁸ Eventually it would be nice to get that level of detail here, too! Emacspeak is another example of a L^AT_EX parser that deals with large-scale textual structures as well as smaller bits and pieces.¹⁹

It would probably be useful to put together some parsers for HTML and wiki code soon.

Note 5.22 (*On ‘import-buffer’*). This function imports L^AT_EX documents, taking care of the non-recursive aspects of this operation. It imports frontmatter (everything up to the first `\begin{section}`), but assumes “backmatter” is trivial, and does not import it. The imported material is classified as a “document” with the same name as the imported buffer.

```
(DEFUN IMPORT-BUFFER (&OPTIONAL BUFFERNAME)
  (SAVE-EXCURSION
    (SET-BUFFER (GET-BUFFER (OR BUFFERNAME
                              (CURRENT-BUFFER))))
    (GOTO-CHAR (POINT-MIN))
    (SEARCH-FORWARD-REGEXP "\\begin{DOCUMENT}")
    (SEARCH-FORWARD-REGEXP "\\begin{SECTION}")
    (GOTO-CHAR (MATCH-BEGINNING 0))
    ;; OTHER LINKS WILL BE MADE IN THE "THEORY OF THIS
    ;; DOCUMENT", BUT HERE WE MAKE A BROADER ASSERTION.
    (SCHOLIUM BUFFERNAME "IS A" "DOCUMENT")
    (SCHOLIUM BUFFERNAME
              "HAS FRONTMATTER"
```

¹⁸<http://dlmf.nist.gov/LaTeXML/>

¹⁹<http://www.cs.cornell.edu/home/raman/aster/aster-thesis.ps>

```

(BUFFER-SUBSTRING-NO-PROPERTIES
 (POINT-MIN)
 (POINT))
BUFFERNAME)
;;; THESE SHOULD MAYBE BE SCHOLIA ATTACHED TO
;;; ROOT-COORDS (BELOW), BUT FOR SOME REASON THAT
;;; WASN'T WORKING SO WELL -- INVESTIGATE LATER --
;;; MAYBE IT JUST WASN'T GOOD TO RUN AFTER RUNNING
;;; 'IMPORT-WITHIN'.
(LET* ((ROOT-COORDS (PLACE-ITEM BUFFERNAME NIL
                               BUFFERNAME))
       (LEVELS
        '("SECTION" "SUBSECTION" "SUBSUBSECTION"))
       (CURRENT-PARENT BUFFERNAME)
       (LEVEL-END NIL)
       (SECTIONS (IMPORT-WITHIN LEVELS))
       (INDEX 0))
 (WHILE SECTIONS
  (LET ((COORDS (CAR SECTIONS)))
    (SETQ INDEX (1+ INDEX))
    (SCHOLIUM ROOT-COORDS
              INDEX
              COORDS
              BUFFERNAME))
    (SETQ SECTIONS (CDR SECTIONS))))))

```

Note 5.23 (*On 'import-within'*). Recurse through levels of sectioning to import \LaTeX code.

It would be good if we could do something about sections that contain neither subsections nor notes (for example, a preface), or, more generally, about text that is not contained in any environment (possibly that appears before any section). We'll save things like this for another editing round!

For the moment, we've decided to build the document hierarchy with links that are blind to whether the k th component of a section is a note or a subsection. Children that are notes are attached in the subroutine 'import-notes' and those that are sections are attached in 'import-within'. Users can find out what type of object they are looking at based on whether or not it "has content".

Incidentally, when looking for the end of an importing level, 'nil' is an OK result – if this is the *last* section at this level *and* there is no subsequent section at a higher level.

```

(DEFUN IMPORT-WITHIN (LEVELS)
 (LET ((THIS-LEVEL (CAR LEVELS))
       (NEXT-LEVEL (CAR (CDR LEVELS))) ANSWER)
 (WHILE (RE-SEARCH-FORWARD
        (CONCAT
         "^\\\\" THIS-LEVEL "\\([\^\\N]*\\)"
         "\\( +\\\\" LABEL{\\})?"
         "\\([\^\\N]*\\)?")
        LEVEL-END T)

```

```

(LET* ((NAME (MATCH-STRING-NO-PROPERTIES 1))
      (AT (PLACE-ITEM NAME NIL BUFFERNAME))
      (LEVEL-END
        (OR (SAVE-EXCURSION
              (SEARCH-FORWARD-REGEXP
                (CONCAT "^\\\\" THIS-LEVEL "{.*}")
                LEVEL-END T))
              LEVEL-END))
      (NOTES-END
        (IF NEXT-LEVEL
            (OR (PROGN (POINT)
                       (SAVE-EXCURSION
                         (SEARCH-FORWARD-REGEXP
                           (CONCAT "^\\\\"
                                NEXT-LEVEL "{.*}")
                           LEVEL-END T)))
                  LEVEL-END)
              LEVEL-END))
      (INDEX (LET ((CURRENT-PARENT AT))
                (IMPORT-NOTES NOTES-END)))
      (SUBSECTIONS (LET ((CURRENT-PARENT AT))
                      (IMPORT-WITHIN (CDR LEVELS))))))
(WHILE SUBSECTIONS
  (LET ((COORDS (CAR SUBSECTIONS)))
    (SETQ INDEX (1+ INDEX))
    (SCHOLIUM AT
              INDEX
              COORDS
              BUFFERNAME)
    (SETQ SUBSECTIONS (CDR SUBSECTIONS)))
  (SETQ ANSWER (CONS AT ANSWER)))
(REVERSE ANSWER)))

```

Note 5.24 (*On ‘import-notes’*). We’re going to make the daring assumption that the “textual” portions of incoming L^AT_EX documents are contained in “Notes”. That assumption is true, at least, for the current document. The function returns the count of the number of notes imported, so that ‘import-within’ knows where to start counting this section’s non-note children.

Would this same function work to import all notes from a buffer without examining its sectioning structure? Not quite, but close! (Could be a fun exercise to fix this.)

```

(DEFUN IMPORT-NOTES (END)
  (LET ((INDEX 0))
    (WHILE (RE-SEARCH-FORWARD (CONCAT "\\begin{NOTATE}"
                                   "{\\([^\n]*\\)}"
                                   "\\( +\\label{\\}?"
                                   "\\([^\n]*\\)}?")
                              END T)
      (LET* ((NAME

```

```

(MATCH-STRING-NO-PROPERTIES 1))
(TAG (MATCH-STRING-NO-PROPERTIES 3))
(BEG
  (PROGN (NEXT-LINE 1)
    (LINE-BEGINNING-POSITION)))
(END
  (PROGN (SEARCH-FORWARD-REGEXP
    "\\END{NOTATE}")
    (MATCH-BEGINNING 0)))
(COORDS (PLACE-ITEM NAME NIL BUFFERNAME)))
(SETQ INDEX (1+ INDEX))
(SCHOLIUM CURRENT-PARENT
  INDEX
  COORDS
  BUFFERNAME)
;; NOT IN THE THEORY
(SCHOLIUM COORDS
  "HAS CONTENT"
  (BUFFER-SUBSTRING-NO-PROPERTIES
    BEG END))
(IMPORT-CODE-CONTINUATIONS COORDS)))
INDEX))

```

Note 5.25 (*On ‘import-code-continuations’*). This runs within the scope of ‘import-notes’, to turn the series of Lisp chunks or other code snippets that follow a given note into a scholium attached to that note. Each separate snippet becomes its own annotation.

The “conditional regexps” used here only work with Emacs version 23 or higher.

I’m noticing a problem with the way the ‘looking-at’ form behaves. It matches the expression in question, but then the match-end is reported as one character less than it supposed to be. Maybe ‘looking-at’ is just not as good as ‘re-search-forward’? But it’s what seems easiest to use.

```

(DEFUN IMPORT-CODE-CONTINUATIONS (COORDS)
  (LET ((POSSIBLE-ENVIRONMENTS
    "\\(1?:LISP\\|IDEA\\|COMMON\\)"))
    (WHILE (LOOKING-AT
      (CONCAT "\\N*?\\END{"
        POSSIBLE-ENVIRONMENTS
        "}"))
      (LET* ((BEG (MATCH-END 0))
        (ENVIRONMENT (MATCH-STRING 1))
        (END (PROGN (SEARCH-FORWARD-REGEXP
          (CONCAT "\\END{"
            ENVIRONMENT
            "}"))
          (MATCH-BEGINNING 0)))
        (CONTENT (BUFFER-SUBSTRING-NO-PROPERTIES
          BEG
          END))))

```

```
(SCHOLIUM (SCHOLIUM COORDS
           "HAS ATTACHMENT"
           CONTENT)
          "HAS TYPE"
          ENVIRONMENT)))))
```

Note 5.26 (*On ‘autoimport-arxana’*). This just calls ‘import-buffer’, and imports this document into the system.

```
(DEFUN AUTOIMPORT-ARXANA ()
  (INTERACTIVE)
  (IMPORT-BUFFER "ARXANA.TEX"))
```

Note 5.27 (*Importing textual links*). Of course, it would be good to import the links that users make between articles, since then we can quickly navigate from an article to the various articles that cite that article, as well as follow the usual forward-directional links. Indeed, we should be able to browse each article within a “neighborhood” of other related articles. (We’ll need to import labels as well, of course.)

5.4 Browsing database contents

Note 5.28 (*Browsing sketch*). This section facilitates browsing of documents represented with structures like those created in Section 5.3, and sets the ground for browsing other sorts of contents (e.g. collections of tasks, as in Section 6.1).

In order to facilitate general browsing, it is not enough to simply use ‘get-article’ (Note 5.11) and ‘get-names’ (Note 5.12), although these functions provide our defaults. We must provide the means to find and display different things differently – for example, a section’s table of contents will typically be displayed differently from its actual contents.

Indeed, the ability to display and select elements of document sections (Note 5.32) is basically the core browsing deliverable. In the process we develop a re-usable article selector (Note 5.30; cf. Note 6.9). This in turn relies on a flexible function for displaying different kinds of articles (Note 5.29).

Note 5.29 (*On ‘display-article’*). This function takes in the name of the article to display. Furthermore, it takes optional arguments ‘retriever’ and ‘formatter’, which tell it how to look up and/or format the information for display, respectively.

Thus, either we make some statement up front (choosing our ‘formatter’ based on what we already know about the article), or we decide what to display after making some investigation of information attached to the article, some of which may be retrieved and displayed (this requires that we specify a suitable ‘retriever’ and a complementary ‘formatter’).

For example, the major mode in which to display the article’s contents could be stored as a scholium attached to the article; or we might maintain some information about “areas” of the database that would tell us up front what which mode is associated with the current area. (The default is to simply insert the data with no markup whatsoever.)

Observe that this works when no theory argument is given, because in that case ‘get-article’ looks for *all* place pseudonyms. (But of course that won’t work well when we have multiple theories containing things with the same names, so we should get used to using the theory argument.)

(The business about requiring the data to be a sequence before engaging in further formatting is, of course, just a matter of expediency for making things work with the current dataset.)

```
(DEFUN DISPLAY-ARTICLE
  (NAME &OPTIONAL THEORY RETRIEVER FORMATTER)
  (INTERACTIVE "MNAME: ")
  (LET* ((DATA (IF RETRIEVER
                  (FUNCALL RETRIEVER NAME THEORY)
                  (GET-ARTICLE NAME THEORY))))
    (WHEN (AND DATA (SEQUENCEP DATA))
      (SAVE-EXCURSION
        (IF FORMATTER
          (FUNCALL FORMATTER DATA THEORY)
          (POP-TO-BUFFER (GET-BUFFER-CREATE
                        "*ARXANA DISPLAY*"))
          (DELETE-REGION (POINT-MIN) (POINT-MAX))
          (INSERT "NAME: " NAME "\N\N")
          (INSERT DATA)
          (GOTO-CHAR (POINT-MIN)))))))
```

Note 5.30 (*An interactive article selector*). The function ‘get-names’ (Note 5.12) and similar functions can give us a collection of articles. The next few functions provide an interactive functionality for moving through this collection to find the article we want to look at.

We define a “display style” that the article selector uses to determine how to display various articles. These display styles are specified by text properties attached to each option the selector provides. Similarly, when we’re working within a given theory, the relevant theory is also specified as a text property.

At selection time, these text properties are checked to determine which information to pass along to ‘display-article’.

```
(DEFVAR DISPLAY-STYLE '(NIL . (NIL NIL)))

(DEFUN THING-NAME-AT-POINT ()
  (BUFFER-SUBSTRING-NO-PROPERTIES
   (LINE-BEGINNING-POSITION)
   (LINE-END-POSITION)))

(DEFUN GET-DISPLAY-TYPE ()
  (GET-TEXT-PROPERTY (LINE-BEGINNING-POSITION)
                     'ARXANA-DISPLAY-TYPE))

(DEFUN GET-RELEVANT-THEORY ()
  (GET-TEXT-PROPERTY (LINE-BEGINNING-POSITION)
                     'ARXANA-RELEVANT-THEORY))
```

```

(DEFUN ARXANA-LIST-SELECT ()
  (INTERACTIVE)
  (APPLY 'DISPLAY-ARTICLE
    (THING-NAME-AT-POINT)
    (GET-RELEVANT-THEORY)
    (CDR (ASSOC (GET-DISPLAY-TYPE)
      DISPLAY-STYLE))))

(DEFINE-DERIVED-MODE ARXANA-LIST-MODE FUNDAMENTAL-MODE
  "ARXANA-LIST" "ARXANA LIST MODE.

\\{ARXANA-LIST-MODE-MAP}")

(DEFINE-KEY ARXANA-LIST-MODE-MAP (KBD "RET")
  'ARXANA-LIST-SELECT)

```

Note 5.31 (*On 'pick-a-name'*). Here 'generate' is the name of a function to call to generate a list of items to display, and 'format' is a function to put these items (including any mark-up) into the buffer from which individual items can then be selected.

One simple way to get a list of names to display would be to reuse a list that we had already produced (this would save querying the database each time). We could, in fact, store a history list of lists of names that had been displayed previously (cf. Note 5.39).

We'll eventually want versions of 'generate' that provide various useful views into the data, e.g., listing all of the elements of a given section (Note 5.32).

Finding all the elements that match a given search term, whether that's just normal text search or some kind of structured search would be worthwhile too. Upgrading the display to e.g. color-code listed elements according to their type would be another nice feature to add.

```

(DEFUN PICK-A-NAME (&OPTIONAL GENERATE FORMAT THEORY)
  (INTERACTIVE)
  (LET ((ITEMS (IF GENERATE
    (FUNCALL GENERATE)
    (GET-NAMES THEORY))))
    (WHEN ITEMS
      (SET-BUFFER (GET-BUFFER-CREATE "*ARXANA ARTICLES*"))
      (TOGGLE-READ-ONLY -1)
      (DELETE-REGION (POINT-MIN)
        (POINT-MAX))
      (IF FORMAT
        (FUNCALL FORMAT ITEMS)
        (MAPC (LAMBDA (ITEM) (INSERT ITEM "\n")) ITEMS))
      (TOGGLE-READ-ONLY T)
      (ARXANA-LIST-MODE)
      (GOTO-CHAR (POINT-MIN))
      (POP-TO-BUFFER (GET-BUFFER "*ARXANA ARTICLES*")))))

```

Note 5.32 (*On ‘display-section’*). When browsing a document, if you select a section, you should display a list of that section’s constituent elements, be they notes or subsections. The question comes up: when you go to display something, how do you know whether you’re looking at the name of a section, or the name of an article?

When you get the section’s contents out of the database (Note 5.33)

```
(DEFUN DISPLAY-SECTION (NAME THEORY)
  (INTERACTIVE (LIST (READ-STRING
                     (CONCAT
                      "NAME (DEFAULT "
                      (BUFFER-NAME) "): ")
                      NIL NIL (BUFFER-NAME))))
  ;; SHOULD THIS POP TO THE ARTICLES WINDOW?
  (PICK-A-NAME '(LAMBDA ()
                (GET-SECTION-CONTENTS
                 ,NAME ,THEORY))
               '(LAMBDA (ITEMS)
                (FORMAT-SECTION-CONTENTS
                 ITEMS ,THEORY))))

(ADD-TO-LIST 'DISPLAY-STYLE
             '(SECTION . (DISPLAY-SECTION
                          NIL)))
```

Note 5.33 (*On ‘get-section-contents’*). Sent by ‘display-section’ (Note 5.32) to ‘pick-a-name’ as a generator for the table of contents of the section with the given name in the given theory.

This function first finds the triples that begin with the (placed) name of the section, then checks to see which of these are in the theory of the document we’re examining (in other words, which of these links represent structural information about that document). It also looks at the items found at the end of these links to see if they are sections or notes (“noteness” is determined by them having content). The links are then sorted by their middles (which show the order in which these components have in the section we’re examining). After this ordering information has been used for sorting, it is deleted, and we’re left with just a list of names in the appropriate order together with an indication of their noteness.

```
(DEFUN GET-SECTION-CONTENTS (NAME THEORY)
  (LET (CONTENTS)
    (DOLIST (TRIPLE (TRIPLES-GIVEN-BEGINNING
                    '(1 ,(RESOLVE-AMBIGUITY
                        (GET-PLACES NAME))))))
      (WHEN (TRIPLE-EXACT-MATCH
            '(2 ,(CAR TRIPLE)) "IN" THEORY)
        (LET* ((NUMBER (PRINT-MIDDLE TRIPLE))
              (SITE (ISOLATE-END TRIPLE))
              (NOTENESS
               (WHEN (TRIPLES-GIVEN-BEGINNING-AND-MIDDLE
                     SITE "HAS CONTENT"))
```

```

      T)))
  (SETQ CONTENTS
    (CONS (LIST NUMBER
              (PRINT-SYSTEM-OBJECT
               (PLACE-CONTENTS SITE))
              NOTENESS)
           CONTENTS))))
  (MAPCAR 'CDR
    (SORT CONTENTS
      (LAMBDA (COMPONENT1 COMPONENT2)
        (< (PARSE-INTEGER (CAR COMPONENT1))
           (PARSE-INTEGER (CAR COMPONENT2)))))))

```

Note 5.34 (*On 'format-section-contents'*). A formatter for document contents, used by 'display-document' (Note 5.35) as input for 'pick-a-name' (Note 5.31).

Instead of just printing the items one by one, like the default formatter in 'pick-a-name' does, this version adds appropriate text properties, which we determine based the second component of of 'items' to format.

```

(DEFUN FORMAT-SECTION-CONTENTS (ITEMS THEORY)
  ;; JUST REPLICATING THE DEFAULT AND BUILDING ON THAT.
  (MAPC (LAMBDA (ITEM)
        (INSERT (CAR ITEM))
        (LET* ((BEG (LINE-BEGINNING-POSITION))
              (END (1+ BEG)))
          (UNLESS (SECOND ITEM)
            (PUT-TEXT-PROPERTY BEG END
                               'ARXANA-DISPLAY-TYPE
                               'SECTION))
          (PUT-TEXT-PROPERTY BEG END
                             'ARXANA-RELEVANT-THEORY
                             THEORY))
        (INSERT "\n"))
    ITEMS))

```

Note 5.35 (*On 'display-document'*). When browsing a document, you should first display its top-level table of contents. (Most typically, a list of all of that document's major sections.) In order to do this, we must find the triples that are begin at the node representing this document *and* that are in the theory of this document. This boils down to treating the document's root as if it was a section and using the function 'display-section' (Note 5.32).

```

(DEFUN DISPLAY-DOCUMENT (NAME)
  (INTERACTIVE (LIST (READ-STRING
                     (CONCAT
                      "NAME (DEFAULT "
                      (BUFFER-NAME) "): ")
                     NIL NIL (BUFFER-NAME))))
  (DISPLAY-SECTION NAME NAME))

```

Note 5.36 (*Work with ‘theory’ argument*). We should make sure that if we know the theory we’re working with (e.g. the name of the document we’re browsing) that this information gets communicated in the background of the user interaction with the article selector.

Note 5.37 (*Selecting from a hierarchical display*). A fancier “article selector” would be able to display several sections with nice indenting to show their hierarchical order.

Note 5.38 (*Browser history tricks*). I want to put together (or put back together) something similar to the multihistoried browser that I had going in the previous version of Arxana and my Emacs/Lynx-based web browser, Nero²⁰. The basic features are: (1) forward, back, and up inside the structure of a given document; (2) switch between tabs. More advanced features might include: (3) forward and back globally across all tabs; (4) explicit understanding of paths that loop.

These sorts of features are independent of the exact details of what’s printed to the screen each time something is displayed. So, for instance, you could flip between section manifests a la Note 5.32, or between hierarchical displays a la Note 5.37, or some combination; the key thing is just to keep track in some sensible way of whatever’s been displayed!

Note 5.39 (*Local storage for browsing purposes*). Right now, in order to browse the contents of the database, you need to query the database every time. It might be handy to offer the option to cache names of things locally, and only sync with the database from time to time. Indeed, the same principle could apply in various places; however, it may also be somewhat complicated to set up. Using two systems for storage, one local and one permanent, is certainly more heavy-duty than just using one permanent storage system and the local temporary display. However, one thing in favor of local storage systems is that that’s what I used in the the previous prototype of Arxana – so some code already exists for local storage! (Caching the list of *names* we just made a selection from would be one simple expedient, see Note 5.31.)

Note 5.40 (*Hang onto absolute references*). Since ‘get-article’ (Note 5.11) translates strings into their “place pseudonyms”, we may want to hang onto those pseudonyms, because they are, in fact, the absolute references to the objects we end up working with. In particular, they should probably go into the text-property background of the article selector, so it will know right away what to select!

5.5 Exporting L^AT_EX documents*

Note 5.41 (*Roundtripping*). The easiest test is: can we import a document into the system and then export it again, and find it unchanged?

Note 5.42 (*Data format*). We should be able to *stably* import and export a document, as well as export any modifications to the document that were generated within Arxana. This means that the exporting functions will have to read the data format that the importing functions use,

²⁰<http://metameso.org/~joe/nero.el>

and that any functions that edit document contents (or structure) will also have to use the same format. Furthermore, *browsing* functions will have to be somewhat aware of this format. So, this is a good time to ask – did we use a good format?

5.6 Editing database contents*

Note 5.43 (*Roundtripping with changes*). Here, we should import a document into the system and then make some simple changes, and after exporting, check with diff to make sure the changes are correct.

Note 5.44 (*Re-importing*). One nice feature would be a function to “re-import” a document that has changed outside of the system, and make changes in the system’s version wherever changes appeared in the source version.

Note 5.45 (*Editing document structure*). The way we have things set up currently, it is one thing to make a change to a document’s textual components, and another to change its structure. Both types of changes must, of course, be supported.

6 Applications

6.1 Managing tasks

Note 6.1 (*What are tasks?*). Each task tends to have a *name*, a *description*, a collection of *prerequisite tasks*, a description of other *material dependencies*, a *status*, some *justification of that status*, a *creation date*, and an *estimated time of completion*. There might actually be several “estimated times of completion”, since the estimate would tend to improve over time. To really understand a task, one should keep track of revisions like this.

Note 6.2 (*On ‘store-task-data’*). Here, we’re just filling in a frame. Since “filling in a frame” seems like the sort of operation that might happen over and over again in different contexts, to save space, it would probably be nice to have a macro (or similar) that would do a more general version of what this function does.

```
(DEFUN STORE-TASK-DATA
  (NAME DESCRIPTION PREREQS MATERIALS STATUS
    JUSTIFICATION SUBMITTED ETA)
  (ADD-TRIPLE NAME "IS A" "TASK")
  (ADD-TRIPLE NAME "DESCRIPTION" DESCRIPTION)
  (ADD-TRIPLE NAME "PREREQS" PREREQS)
  (ADD-TRIPLE NAME "MATERIALS" MATERIALS)
  (ADD-TRIPLE NAME "STATUS" STATUS)
  (ADD-TRIPLE NAME "STATUS JUSTIFICATION" JUSTIFICATION)
  (ADD-TRIPLE NAME "DATE SUBMITTED" SUBMITTED)
  (ADD-TRIPLE NAME "ESTIMATED TIME OF COMPLETION" ETA))
```

Note 6.3 (*On ‘generate-task-data’*). This is a simple function to create a new task matching the description above.

```

(DEFUN GENERATE-TASK-DATA ()
  (INTERACTIVE)
  (LET ((NAME (READ-STRING "NAME: "))
        (DESCRIPTION (READ-STRING "DESCRIPTION: "))
        (PREREQS (READ-STRING
                  "TASK(S) THIS TASK DEPENDS ON: "))
        (MATERIALS (READ-STRING "MATERIAL DEPENDENCIES: "))
        (STATUS (COMPLETING-READ
                 "STATUS (TABLED, IN PROGRESS, COMPLETED):
                 " '("TABLED" "IN PROGRESS" "COMPLETED"")))
        (JUSTIFICATION (READ-STRING "WHY THIS STATUS? "))
        (SUBMITTED
         (READ-STRING
          (CONCAT "DATE SUBMITTED (DEFAULT "
                  (SUBSTRING (CURRENT-TIME-STRING) 0 10)
                  "): "))
         NIL NIL (SUBSTRING (CURRENT-TIME-STRING) 0 10)))
        (ETA
         (READ-STRING "ESTIMATED DATE OF COMPLETION:")))
    (STORE-TASK-DATA NAME DESCRIPTION PREREQS MATERIALS
                     STATUS
                     JUSTIFICATION SUBMITTED ETA)))

```

Note 6.4 (*Possible enhancements to ‘generate-task-data’*). In order to make this function very nice, it would be good to allow “completing read” over known tasks when filling in the prerequisites. Indeed, it might be especially nice to offer a type of completing read that is similar in some sense to the tab-completion you get when completing a file name, i.e., quickly completing certain sub-strings of the final string (in this case, these substrings would correspond to task areas we are progressively zooming down into).

As for the task description, rather than forcing the user to type the description into the minibuffer, it might be nice to pop up a separate buffer instead (a la the Emacs/w3m textarea). If we had a list of all the known tasks, we could offer completing-read over the names of existing tasks to generate the list of ‘prereqs’. It might be nice to systematize date data, so we could more easily e.g. sort and display task info “by date”. (Perhaps we should be working with predefined database types for dates and so on; but see Note 8.11.)

Also, before storing the task, it might be nice to offer the user the chance to review the data they entered.

Note 6.5 (*On ‘get-filler’*). Just a wrapper for ‘triples-given-beginning-and-middle’. (Maybe add ‘theory’ as an option here.)

```

(DEFUN GET-FILLER (FRAME SLOT)
  (THIRD (FIRST
          (PRINT-TRIPLES
           (TRIPLES-GIVEN-BEGINNING-AND-MIDDLE FRAME
           SLOT))))))

```

Note 6.6 (*On ‘get-task’*). ses ‘get-filler’ (Note 6.5) to assemble the elements of a task’s frame.

```

(DEFUN GET-TASK (NAME)
  (WHEN (TRIPLE-EXACT-MATCH NAME "IS A" "TASK")
    (LIST (GET-FILLER NAME "DESCRIPTION")
          (GET-FILLER NAME "PREREQS")
          (GET-FILLER NAME "MATERIALS")
          (GET-FILLER NAME "STATUS")
          (GET-FILLER NAME "STATUS JUSTIFICATION")
          (GET-FILLER NAME "DATE SUBMITTED")
          (GET-FILLER NAME
            "ESTIMATED TIME OF COMPLETION"))))

```

Note 6.7 (*On ‘review-task’*). This is a function to review a task by name.

```

(DEFUN REVIEW-TASK (NAME)
  (INTERACTIVE "MNAME: ")
  (LET ((TASK-DATA (GET-TASK NAME)))
    (IF TASK-DATA
      (DISPLAY-TASK TASK-DATA)
      (MESSAGE "NO DATA."))))

```

```

(DEFUN DISPLAY-TASK (DATA)
  (SAVE-EXCURSION
    (POP-TO-BUFFER (GET-BUFFER-CREATE
      "*ARXANA DISPLAY*"))
    (DELETE-REGION (POINT-MIN) (POINT-MAX))
    (INSERT "NAME: " NAME "\N\n")
    (INSERT "DESCRIPTION: " (FIRST DATA) "\N\n")
    (INSERT "TASKS THIS TASK DEPENDS ON: "
      (SECOND DATA) "\N\n")
    (INSERT "MATERIAL DEPENDENCIES: "
      (THIRD DATA) "\N\n")
    (INSERT "STATUS: " (FOURTH DATA) "\N\n")
    (INSERT "WHY THIS STATUS?: " (FIFTH DATA) "\N\n")
    (INSERT "DATE SUBMITTED:" (SIXTH DATA) "\N\n")
    (INSERT "ESTIMATED TIME OF COMPLETION: "
      (SEVENTH DATA) "\N\n")
    (GOTO-CHAR (POINT-MIN))
    (FILL-INDIVIDUAL-PARAGRAPHS (POINT-MIN) (POINT-MAX))))

```

Note 6.8 (*Possible enhancements to ‘review-task’*). Breaking this down into a function to select the task and another function to display the task would be nice. Maybe we should have a generic function for selecting any object “by name”, and then special-purpose functions for displaying objects with different properties.

Using text properties, we could set up a “field-editing mode” that would enable you to select a particular field and edit it independently of the others. Another more complex editing mode would *know* which fields the user had edited, and would store all edits back to the database properly. See Section 5.6 for more on editing.

Note 6.9 (*Browsing tasks*). The function ‘pick-a-name’ (Note 5.31) takes two functions, one that finds the names to choose from, and the other that says how to present these names. We can therefore build ‘pick-a-task’ on top of ‘pick-a-name’.

```
(DEFUN GET-TASKS ()
  (MAPCAR #'FIRST
    (PRINT-TRIPLES
      (TRIPLES-GIVEN-MIDDLE-AND-END "IS A" "TASK")
      T)))

(DEFUN PICK-A-TASK ()
  (INTERACTIVE)
  (PICK-A-NAME
    'GET-TASKS
    (LAMBDA (ITEMS)
      (MAPC (LAMBDA (ITEM)
        (LET ((POS (LINE-BEGINNING-POSITION)))
          (INSERT ITEM)
          (PUT-TEXT-PROPERTY POS (1+ POS)
            'ARXANA-DISPLAY-TYPE
            'TASK)
          (INSERT "\n")))) ITEMS))))

(ADD-TO-LIST 'DISPLAY-STYLE
  '(TASK . (GET-TASK DISPLAY-TASK)))
```

Note 6.10 (*Working with theories*). Presumably, like other related functions, ‘get-tasks’ should take a theory argument.

Note 6.11 (*Check display style*). Check if this works, and make style consistent between this usage and earlier usage.

6.2 Other ideas*

Note 6.12 (*A browser within a browser*). All the stuff we’re doing with triples can be superimposed over the existing web and existing web interfaces, by, for example, writing a web browser as a web app, and in this “browser within a browser” offer the ability to annotate and rewrite other people’s web pages, and share the mods with other subscribers to the service. (Previous websites have tried limited versions of what’s possible.)

Note 6.13 (*Improvements to the PlanetMath backend*). From one point of view, the SQL tables are the main thing in Noosphere. We could say that getting the things out of SQL and storing new things there is what Noosphere mainly does. Following this line of thought, anything that adjusts these tables will do just as well, e.g., it shouldn’t be terribly hard to develop an email-based front-end. But rather than making Arxana work with the Noosphere relational table system, it is probably advantageous to translate the data from these tables into the scholium system.

Note 6.14 (*A new communication platform*). One of the premier applications I have in mind is a new way to handle communications in an

online-forum. I have previously called this “subchanneling”, but really, joining channels is just as important.

Note 6.15 (*Some tutorials*). It would be interesting to write a tutorial for Common Lisp or just about any other topic with this system. For example, some little “worksheets” or “gymnasia” that will help solidify user knowledge in topics on which questions keep appearing.

7 Topics of philosophical interest

Note 7.1 (*Research and development*). In Note 1.2, I mentioned a model that could apply in many contexts; it is an essentially metaphysical conception. I’m pretty sure that the data model of Note 1.3 provides a general-enough framework to represent anything we might find “out there”. However, even if this is the case, questions as to *efficient* means of working with such data still abound (cf. Note 3.4, Note 4.21).

I propose that along with *development* of Arxana as a useful system for *doing* “commons-based peer production” should come a *research* programme for understanding in much greater detail what “commons-based peer production” *is*. Eventually we may want to change the name of the subject of study to reflect still more general ideas of resource use.

While the “frontend” of this research project is anthropological, the “backend” is much closer to artificial intelligence. On this level, the project is about understanding *effective* means for solving human problems. Often this will involve decomposing events and processes into constituent elements, making increasingly detailed treatments along the lines described in Note 1.1.

Note 7.2 (*The relationship between text and commentary*). Text under revision might be marked up by a copyeditor: in cases like these, the interpretation is clear. However, what about marginalia with looser interpretations? These seem to become part of the copy of the text they are attached to. What about steering processes applied to a given course of action? How about the relationship of thoughts or words to perception and action? How can we lower the barrier between conception and action, while still maintaining some purchase on wisdom?

You see, a lot of issues in life have to do with overlays, multi-tracking, interchange between different systems; and in these terms, a lot of philosophy reduces to “media awareness” which extends into more and more immediate contexts (Note 1.2).

Note 7.3 (*Heuristic flow*). Continuing the notion above: one does not need a fully-developed “theory” of work in order to do work – instead, one wants some straightforward heuristics that will enable the desired work to get done. So, even supposing the work is “theory building”, it can progress without becoming overwhelmed in abstractions – because theories and heuristics are different things.

Note 7.4 (*Limits of simple languages*). Triples are frequently “subject, verb, object” statements, although with the annotation features, we can modify any part of any such statement; for example, we can apply an adverb to a given verb.

“Tags”, of course, already provide “subject, predicate” relationships. It will be interesting to examine the degree to which human languages can be mapped down into these sorts of simple languages. What features are needed to make such languages *useful*? (Lisp’s ‘car’ and ‘cdr’ seem related to the idea of making predicates useful.)

How are triples and predicates “enough”? What, if anything, do they lack? The difference between triples and predicates illustrates the issue. How should we characterize Arxana’s additions to Lisp?

Note 7.5 (*Higher dimensions*). Why stop with three components? Why not have (A, B, C, D, T) represent a semantic relationship between all of A , B , C , and D (in theory T , of course)? Actually, there is no reason to stop apart from the fact that I want to explore simple languages (Note 7.4). In real life, things are not as simple, and we should be ready to deal with the complexities! (Cf., for example, Note 8.3).

8 Future plans

Note 8.1 (*Theories as database objects*). It seems like it will be very useful to make theories into database objects; I expect we’ll do that soon. We can start to explore attaching places to specific theories (like, the theory of a particular chess board); cf. Note 5.14.

Note 8.2 (*Search engine/elements*). One of the features that came very easy in the Emacs-only prototype was textual search. With the strings stored in a database, Sphinx seems to be the most suitable search engine to use. It is tempting to try to make our own inverted index using triples, so that text-based search can be even more directly integrated with semantic search. (Perhaps Sphinx makes it direct enough, though.) More to the point, it is important for this project that the scholia-based document model be transparently extended down to the level of words and characters. It is helpful to think about text as hypertext, and a document (or a word in the inverted index) as a theory.

Note 8.3 (*Pointing at database elements and other things*). We will want to be able to point at other tables and at other sorts of objects and make use of their contents. The plan is that our triples will provide a sort of guide or backbone superimposed over a much larger data system.

Note 8.4 (*Feature-chase*). There are lots of different features that could be explored, for example: multi-dimensional history lists; a full theory of “clusions”, MS Word-like colorful annotations, etc. Many of these features are already prototyped.²¹

Note 8.5 (*Deleting and changing things*). How will we deal with unlinking, disassociating, forgetting, entropy, and the like? Changes (perhaps modeled by an insertion following a deletion) also present a variety of interesting challenges.

Note 8.6 (*Tutorial*). Right now the system is simple enough to be pretty much self-explanatory, but if it becomes much more complicated, it might

²¹See footnote 3.

be helpful to put together a simple guide to some likely-to-be-interesting features.

Note 8.7 (*Computing possible paths and connections*). If we can find all the *direct* paths from one node to another using ‘triples-given-beginning-and-end’, can we inject some algorithms for finding longer, indirect paths into the system, and find ways to make them useful?

Similarly, we can satisfy local conditions (Note 4.32), but we’ll want to deal with increasingly “non-local” conditions (even just using the logical operator “or”, instead of “and”, for example).

Note 8.8 (*Monster Mountain*). In Summer 2007, we checked out the Monster Mountain MUD server²², which would enable several users to interact with one LISP, instead of just one database. This would have a number of advantages, particularly for exploring “scholiumific programming”, but also towards fulfilling the user-to-user interaction objective stated in Note 1.2. I plan to explore this after the primary goal of multi-user interaction with the database has been solidly completed.

Note 8.9 (*Web interface*). A finished web interface may take a considerable amount of work (if the complexity of an interesting Emacs interface is any indication), but the basics shouldn’t be hard to put together soon.

Note 8.10 (*Parsing input*). Complicated objects specified in long-hand (e.g. triples pointing to triples) can be read by a relatively simple parser – which we’ll have to write! The simplest goal for the parser would be to be able to distinguish between a triple and a string – presumably that much isn’t hard. And of course, building complexes of triples that represent statements from natural language is a good long-term goal. (Right now, our granularity level is set much higher.)

Note 8.11 (*Choice of database*). I expect Elephant²³ may become our preferred database at some point in the future; we are currently awaiting changes to Elephant that make nested queries possible and efficient. Some core queries related to managing a database of semantic links with the current Elephant were constructed by Ian Eslick, Elephant’s maintainer.²⁴

On the other hand, it might be reasonable to use an Emacs database and redo the whole thing to work in Emacs (again), e.g. for single-user applications or users who want to work offline a lot of the time.

Note 8.12 (*Different kinds of theories*). Theories or variants thereof are of course already popular in other knowledge representation contexts.^{25,26} We’ll want to adopt some useful techniques for knowledge management as soon as the core systems are ready.

Various notions of a mathematical theory exist.²⁷ It would be nice to be able to assign specific logic to theories in Arxana, following the “little theories” design of e.g. IMPS.²⁸

²²<http://code.google.com/p/mmtn/>

²³<http://common-lisp.net/project/elephant/>

²⁴<http://planetx.cc.vt.edu/~jcorneli/arxana/variant-4.lisp>

²⁵<http://www.cyc.com/cycdoc/vocab/mt-expansion-vocab.html>

²⁶http://www.stanford.edu/~kdevlin/HHL_SituationTheory.pdf

²⁷<http://planetmath.org/encyclopedia/Theory.html>

²⁸<http://imps.mcmaster.ca/manual/node13.html>

A Appendix: Auto-setup

Note A.1 (*Setting up auto-setup*). This section provides code for satisfying dependencies and setting up the program. This code assumes that you are using a Debian/APT-based system (but things are not so different using say, Fedora or Fink; writing a multi-package-manager-friendly installer shouldn't be hard). Of course, feel free to set things up differently if you have something else in mind!

```
(DEFALIAS 'SET-UP 'SHELL-COMMAND)

(DEFUN ALTERNATIVE-SET-UP (STRING)
  (SAVE-EXCURSION
   (POP-TO-BUFFER (GET-BUFFER-CREATE "*ARXANA HELP*"))
   (GOTO-CHAR (POINT-MAX))
   (INSERT STRING "\n")))

(DEFUN SET-UP-ARXANA-ENVIRONMENT ()
  (INTERACTIVE)
  (IF (Y-OR-N-P
       "RUN COMMANDS (Y) (OR JUST SHOW INSTRUCTIONS)? ")
      (FSET 'SET-UP 'SHELL-COMMAND)
      (FSET 'SET-UP 'ALTERNATIVE-SET-UP))
  (WHEN (Y-OR-N-P "INSTALL DEPENDENCIES? ")
        (SET-UP "MKDIR ~/ARXANA")
        (SET-UP "CD ARXANA"))

  (WHEN (Y-OR-N-P "DOWNLOAD LATEST ARXANA? ")
        (SET-UP "WGET HTTP://METAMESO.ORG/FILES/ARXANA.TEX"))

  (UNLESS (Y-OR-N-P "IS YOUR EMACS GOOD ENOUGH?... ")
          (SET-UP
           (CONCAT "CVS -z3 -d"
                   ":PSEVER:ANONYMOUS@CVS.SAVANNAH.GNU.ORG:"
                   "/SOURCES/EMACS CO EMACS"))
          (SET-UP "MV EMACS ~")
          (SET-UP "CD ~/EMACS")
          (SET-UP "./CONFIGURE && MAKE BOOTSTRAP")
          (SET-UP "CD ~/ARXANA"))

  (DEFVAR PAC-MAN NIL)

  (COND ((Y-OR-N-P
          "DO YOU USE AN APT-BASED PACKAGE MANAGER? ")
         (SETQ PAC-MAN "APT-GET"))
        (T (MESSAGE
            "OK, GET LISP AND SQL ON YOUR OWN, THEN!"))))

  (WHEN PAC-MAN
    (WHEN (Y-OR-N-P "INSTALL COMMON LISP? ")
          (SET-UP (CONCAT PAC-MAN " INSTALL SBCL")))))
```

```

(WHEN (Y-OR-N-P "INSTALL POSTGRESQL? ")
  (SET-UP (CONCAT PAC-MAN " INSTALL POSTGRESQL")))
(WHEN (Y-OR-N-P "HELP SETTING UP POSTGRESQL? ")
  (SAVE-EXCURSION
    (POP-TO-BUFFER (GET-BUFFER-CREATE "*ARXANA HELP*"))
    (INSERT "AS SUPERUSER (ROOT),
EDIT /ETC/POSTGRESQL/7.4/MAIN/PG_HBA.CONF
MAKE SURE IT SAYS THIS:
HOST ALL ALL 127.0.0.1 255.255.255.255 TRUST
THEN EDIT /ETC/POSTGRESQL/7.4/MAIN/POSTGRESQL.CONF
AND MAKE IT SAY
TCPIP_SOCKET = TRUE
THEN RESTART:
/ETC/INIT.D/POSTGRESQL-7.4 RESTART
SU POSTGRES
CREATEUSER USERNAME
EXIT
AS USERNAME, RUN
CREATEDB -U USERNAME\N")))))))

(WHEN (Y-OR-N-P "INSTALL SLIME...? ")
  (SET-UP (CONCAT "CVS -D :PSEVER:ANONYMOUS"
                  ":ANONYMOUS@COMMON-LISP.NET:"
                  "/PROJECT/SLIME/CVSROOT CO SLIME")))
  (SET-UP
    (CONCAT "ECHO \";; ADDED TO ~/.EMACS FOR ARXANA:\N\N"
            "(ADD-TO-LIST 'LOAD-PATH \"~/SLIME/\")\N"
            "(SETQ INFERIOR-LISP-PROGRAM \"/USR/BIN/SBCL\")\N"
            "(REQUIRE 'SLIME)\N"
            "(SLIME-SETUP '(SLIME-REPL))\N\N"
            "| CAT - ~/.EMACS > ~/UPDATED.EMACS &&"
            "MV ~/UPDATED.EMACS ~/.EMACS"))))

(WHEN (Y-OR-N-P "SET UP COMMON LISP ENVIRONMENT? ")
  (SET-UP "MKDIR ~/.SBCL")
  (SET-UP "MKDIR ~/.SBCL/SITE")
  (SET-UP "MKDIR ~/.SBCL/SYSTEMS")
  (SET-UP "CD ~/.SBCL/SITE")
  (SET-UP (CONCAT "WGET HTTP://FILES.B9.COM/"
                  "CLS QL/CLS QL-LATEST.TAR.GZ")))
  (SET-UP "TAR -ZXF CLS QL-4.0.3.TAR.GZ")
  (SET-UP (CONCAT "WGET HTTP://FILES.B9.COM/"
                  "UFFI/UFFI-LATEST.TAR.GZ")))
  (SET-UP "TAR -ZXF UFFI-1.6.0.TAR.GZ")
  (SET-UP (CONCAT "WGET HTTP://FILES.B9.COM/"
                  "MD5/MD5-1.8.5.TAR.GZ")))
  (SET-UP "TAR -ZXF MD5-1.8.5.TAR.GZ")
  (SET-UP "CD ~/.SBCL/SYSTEMS")
  (SET-UP "LN -S ../SITE/MD5-1.8.5/MD5.ASD .")

```

```

(SET-UP "LN -S ../SITE/UFFI-1.6.0/UFFI.ASD .")
(SET-UP "LN -S ../SITE/CLSQL-4.0.3/CLSQL.ASD .")
(SET-UP "LN -S ../SITE/CLSQL-4.0.3/CLSQL-UFFI.ASD .")
(SET-UP (CONCAT "LN -S ../SITE/CLSQL-4.0.3/"
                "CLSQL-POSTGRESQL-SOCKET.ASD ."))
(SET-UP "LN -S ~/ARXANA/ARXANA.ASD ."))

(WHEN (Y-OR-N-P "MODIFY ~/.SBCLRC SO CL ALWAYS STARTS ARXANA? ")
  (SET-UP
    (CONCAT "ECHO \";; ADDED TO ~/.SBCLRC FOR ARXANA:\N\n"
            " (REQUIRE 'ASDF)\N\n"
            "(ASDF:OPERATE 'ASDF:LOAD-OP 'SWANK)\N"
            "(SETF SWANK:*USE-DEDICATED-OUTPUT-STREAM* NIL)\N"
            "(SETF SWANK:*COMMUNICATION-STYLE* :FD-HANDLER)\N"
            "(SWANK:CREATE-SERVER :PORT 4006 :DONT-CLOSE T)\N\n"
            "(ASDF:OPERATE 'ASDF:LOAD-OP 'CLSQL)\N"
            "(ASDF:OPERATE 'ASDF:LOAD-OP 'ARXANA)\N"
            "(IN-PACKAGE ARXANA)\N"
            "(CONNECT-TO-DATABASE)\N"
            "(LOCALLY-ENABLE-SQL-READER-SYNTAX)\N\n"
            "| CAT ~/.SBCLRC - > ~/UPDATED.SBCLRC &&"
            "MV ~/UPDATED.SBCLRC ~/.SBCLRC"))))

(WHEN (Y-OR-N-P "INSTALL MONSTER MOUNTAIN? ")
  (SET-UP "CD ~/.SBCL/SYSTEMS")
  (SET-UP (CONCAT
            "DARCS GET HTTP://COMMON-LISP.NET/PROJECT/"
            "BORDEAUX-THREADS/DARCS/BORDEAUX-THREADS/"))
  (SET-UP (CONCAT
            "SVN CHECKOUT SVN://COMMON-LISP.NET/PROJECT/"
            "USOCKET/SVN/USOCKET/TRUNK USOCKET-SVN"))
  ;; I'VE HAD PROBLEMS WITH THIS APPROACH TO SETTING CCLAN
  ;; MIRROR...
  (SET-UP
    (CONCAT
      "WGET \"HTTP://WW.TELENT.NET/CCLAN-CHOOSE-MIRROR"
      "?M=HTTP%3A%2F%2FTHINGAMY.COM%2FCCLAN%2F\""))
  (SET-UP (CONCAT "WGET HTTP://WW.TELENT.NET/CCLAN/"
                  "SPLIT-SEQUENCE.TAR.GZ"))
  (SET-UP "TAR -ZXF SPLIT-SEQUENCE.TAR.GZ")
  (SET-UP
    (CONCAT "SVN CHECKOUT HTTP://MMTN.GOOGLECODE.COM/"
            "SVN/TRUNK/ MMTN-READ-ONLY"))
  (SET-UP
    "LN -S ~/BORDEAUX-THREADS/BORDEAUX-THREADS.ASD .")
  (SET-UP "LN -S ~/USOCKET-SVN/USOCKET.ASD .")
  (SET-UP "LN -S ~/SPLIT-SEQUENCE/SPLIT-SEQUENCE.ASD .")
  (SET-UP "LN -S ~/MMTN/SRC/MMTN.ASD ."))))

```

Note A.2 (*Postgresql on Fedora*). There are some slightly different

instructions for installing postgresql on Fedora; the above will be changed to include them, but for now, check them out on the web.²⁹

Note A.3 (*Using MySQL and CLISP instead*). Since my OS X box seems to have a variety of confusing PostgreSQL systems already installed (which I'm not sure how to configure), and CLISP is easy to install with fink, I thought I'd try a different set up for simplicity and variety.

In order to make it work, I enabled root user on Mac OS X per instructions on web, and installed and configured mysql; used a slight modification of the strings table described previously; download and installed cffi³⁰; changed the definition of 'connect-to-database' in Arxana's utilities.lisp; doctored up my /.clisprc.lisp; and changed how I started Lisp. Details below.

```
;; on the shell prompt
sudo apt-get install mysql
sudo mysqld_safe --user=mysql &
sudo daemonicon enable mysql
sudo mysqladmin -u root password root
mysql --user=root --password=root -D test
create database joe; grant all on joe.* to joe@localhost
identified by 'joe'

;; in tabledefs.lisp
(execute-command "CREATE TABLE strings (
  id SERIAL PRIMARY KEY, text TEXT, UNIQUE INDEX
  (text(255))
);")

;; in ~/asdf-registry/ or whatever you've designated as
;; your asdf:*central-registry*
ln -s ~/cffi_0.10.4/cffi-uffi-compat.asd .
ln -s ~/cffi_0.10.4/cffi.asd .

;; In utilities.lisp
(defun connect-to-database ()
  (connect ("localhost" "joe" "joe" "joe")
           :database-type :mysql))

;; In ~/.clisprc.lisp
(asdf:operate 'asdf:load-op 'clsql)
(push "/sw/lib/mysql/"
      CLSQL-SYS:*FOREIGN-LIBRARY-SEARCH-PATHS*)

;; From SLIME prompt, and not in ~/.clisprc.lisp
(in-package #:arxana)
(connect-to-database)
(locally-enable-sql-reader-syntax)
```

²⁹http://www.flmnh.ufl.edu/linux/install_postgresql.htm

³⁰http://common-lisp.net/project/cffi/releases/cffi_latest.tar.gz

Note A.4 (*Installing Sphinx*). Here are some tips on how to install and configure Sphinx.

```
;; Fedora/Postgresql flavor
yum install postgresql-devel
./configure --without-mysql
    --with-pgsql
    --with-pgsql-libs=/usr/lib/pgsql/
    --with-pgsql-includes=/usr/include/pgsql

;; Fink/MySQL flavor
./configure --with-mysql
    --with-mysql-includes=/sw/include/mysql
    --with-mysql-libs=/sw/lib/mysql
```

Note A.5 (*Getting Sphinx set up*). Here are some instructions I've used to get Sphinx set up.

Note A.6 (*Create a sphinx.conf*). I want a very minimal sphinx.conf, this seems to work. (We should probably set this up so that it gets written to a file when the Arxana is set up.)

```
## Copy this to /usr/local/etc/sphinx.conf when you want
## to use it.

source strings
{
    type            = mysql
    sql_host        = localhost
    sql_user        = joe
    sql_pass        = joe
    sql_db          = joe
    sql_query       = SELECT id, text FROM strings
}

## index definition

index strings
{
    source          = strings
    path            = /Users/planetmath/sphinx/search-testing
    morphology      = none
}

## indexer settings

indexer
{
    mem_limit       = 32M
}

## searchd settings
```

```

searchd
{
  listen          = 3312
  listen          = localhost:3307:mysql41
  log             = /Users/planetmath/sphinx/searchd.log
  query_log      = /Users/planetmath/sphinx/searchd_query.log
  read_timeout   = 5
  max_children   = 30
  pid_file       = /Users/planetmath/sphinx/searchd.pid
  max_matches    = 1000
}

```

Note A.7 (*Working from the command line*). Then you can run commands like these.

```

/usr/local/bin/indexer strings
/usr/local/bin/search "but, then"

```

```

% mysql -h 127.0.0.1 -P 3307
mysql> SELECT * FROM strings WHERE MATCH('but, then');

```

Note A.8 (*Integrating this with Lisp*). Since we can talk to Sphinx via Mysql protocol, it seems reasonable that we should be able to talk to it from CLSQL, too. With a little fussing to get the format right, I found something that works!

```

(connect '("127.0.0.1" "" "" "" "3307") :database-type :mysql)
(mapcar (lambda (elt) (floor (car elt)))
  (query "select * from strings where match('text')"))

```

Note A.9 (*Some added difficulty with Postgresql*). When I try to index things on the server, I get an error, as below. The question is a good one... I'm not sure *how* postgresql is set up on the server, actually...

```

ERROR: index 'strings': sql_connect: could not connect to server:
Connection refused
Is the server running on host "localhost" and accepting
TCP/IP connections on port 5432?

```

B Appendix: A simple literate programming system

Note B.1 (*The literate programming system used in this paper*). This code defines functions that grab all the Lisp portions of this document, evaluate the Emacs Lisp sections in Emacs, and save the Common Lisp sections in suitable files.³¹ It requires that the L^AT_EX be written in a certain consistent way. The function assumes that this document is the current buffer.

³¹Cf. <http://mmm-mode.sourceforge.net/>

```

(defvar lit-code-beginning-regexp
  "\\begin{elisp}\\|^\\begin{common}{\\([~]\\n)*\\}")

(defvar lit-code-end-regexp
  "\\end{elisp}\\|^\\end{common}")

(defun lit-process ()
  (interactive)
  (save-excursion
    (let ((to-buffer "*Lit Code*")
          (from-buffer (buffer-name (current-buffer)))
          (start-buffers (buffer-list)))
      (set-buffer (get-buffer-create to-buffer))
      (erase-buffer)
      (set-buffer (get-buffer-create from-buffer))
      (goto-char (point-min))
      (while (re-search-forward
              lit-code-beginning-regexp nil t)
        (let* ((file (match-string 1))
               (beg (match-end 0))
               (end (save-excursion
                     (search-forward-regexp
                      lit-code-end-regexp nil t)
                     (match-beginning 0))))
          (match (buffer-substring-no-properties
                  beg end)))
          (let ((to-buffer
                 (if file
                     (concat "*Lit Code*: " file)
                     "*Lit Code*")))
              (save-excursion
                (set-buffer (get-buffer-create
                            to-buffer))
                (insert match))))))
      (dolist
        (buffer (set-difference (buffer-list)
                               start-buffers))
          (save-excursion
            (set-buffer buffer)
            (if (string= (buffer-name buffer)
                        "*Lit Code*")
                (eval-buffer)
                (write-region (point-min)
                             (point-max)
                             (concat "~/arxana/"
                                     (substring
                                      (buffer-name
                                       buffer)
                                      12))))))
          (kill-buffer buffer))))))

```

Note B.2 (*Emacs-export?*). It wouldn't be hard to export the Emacs sections so that those who wanted to could ditch the literate wrapper.

Note B.3 (*Bidirectional updating*). Eventually it would be nice to have a code repository set up, and make it so that changes to the code can get snarfed up here.

C Appendix: Hypertext platforms

Note C.1 (*The purpose of this appendix*). I have to give a talk on "Hypertext Platforms" at MathFest 2008. This is a place for draft material that can feed into that talk.

Note C.2 (*The hypertextual canon*). There is a core library of texts that come up (or should!) in discussions of hypertext.

- The Talmud (Judah haNasi, Rav Ashi, and many others)
- Monadology (Wilhelm Leibniz)
- The Life and Opinions of Tristram Shandy, Gentleman (Lawrence Sterne)
- Middlemarch (George Eliot)
- The Nova Trilogy (William S. Burroughs)
- The Logic of Sense (Gilles Deleuze)
- Labyrinths (Jorge Luis Borges)
- Literary Machines (Ted Nelson)
- Lila (Robert M. Pirsig)
- Dirk Gently's Holistic Detective Agency (Douglas Adams)
- Pussy, King of the Pirates (Kathy Acker)

At the same time, it is somewhat ironic that none of the items on this list are themselves hypertexts in the contemporary sense of the word. It's also a bit funny that other works (even some by the same authors) aren't on this list. Finally, it seems telling that non-hypertextual canons remain mostly-non-hypertextual even today, despite the existence of catalogs and indexes.³²

Note C.3 (*A geek's guide to literature*). This title is a riff on Slavoj Žižek's "A pervert's guide to cinema". Taking Note C.2 as a jumping-off point, why don't we make a survey of historical texts from the point of view of an aficionado of hypertext! Just what does one have to do to "get on the list"? Just what is "the hypertextual perspective"? And, if Žižek is correct and we're to look for the hyperreal in the world of cinematic fictions – what's left over for the world of literature? (Or mathematics, for that matter.)

³²<http://www.gutenberg.org/wiki/Category:Bookshelf>

Note C.4 (*The number 3*). This is the number of things present if we count carefully the items A , B , and a connection C between them.

[Picture of $A \xrightarrow{C} B$.]

(Or even: given A and B , we use Wittgenstein counting, and *intuit* that C exists as the collection $\{A, B\}$; after all, some connection must exist precisely because we were presented with A and B together – and lest the connections proliferate infinitely, we lump them all together as one. [Picture of A , B , with the *frame* labeled C .])

Note C.5 (*Surfaces*). Deleuze talks about a theory of surfaces associated with verbs and events. His surfaces represent the evanescence of events in time, and of their descriptions in language. An event is seen as a vanishingly-thin boundary between one state of being and another.

Certainly, a statement that is true *now* may not be true five minutes from now. It is easier to think and talk about things that are coming up and things that have already happened. “Living in the moment” is regarded as special or even “Zen”.

We can begin to put these musings on a more solid mathematical basis. We first examine two types of *interfaces*:

1. $A \xrightarrow{C} B$, $A \xrightarrow{D} B$, $A \xrightarrow{E} B$ (the interface of A and B across C , D , and E);
2. $A \xrightarrow{C} B$, $D \xrightarrow{C} E$, $F \xrightarrow{C} G$ (the interface of various terms across C).

Note C.6 (*Comic books*). No geek’s guide to literature would be complete without putting comics in a hallowed place. [Framed picture of A , B next to framed picture of A , B , a .] What happened? ☺

Note C.7 (*Intersecting triples*). Diagrammatically, it is tempting to portray $(ACB)_{\text{mid}}DE$ as if it was closely related to $A(BDE)_{\text{beg}}C$, despite the fact that they are notationally very different. I’ll have to think more about what this means.

D Appendix: Computational Linguistics

Note D.1 (*What is this?*). It might be reasonable to make annotating sentences part of our writeup on hypertext platforms – but I’m putting it here for now. If hypertext is what deals with language artifacts on the “bulky” level (saying, for example, that a subsection is part of a section, and so on), then computational linguistics is what deals with the finer levels. However, the distinction is in some ways arbitrary, and many of the techniques should be at least vaguely similar.

Note D.2 (*Annotation sensibilities*). We will want to be able to make at least two different kinds of annotations of verbs. For example, given the statement

S . (“Who” “is on” “first”),

I’d like to be able to say

- I. (“is on” “means” “the position of a base runner in baseball”).

However, I’d also like to be able to say

II. (“is on” “because” “he was walked”).

Annotation I is meant to apply to the term “is on” itself (in a context that might be more general than just this one sentence). If Who is also on steroids, that’s another matter – as this type of annotation helps make clear!

Annotation II is meant to apply to the term “is on” *as it appears in sentence S*. In particular, Annotation II seems to work best in a context in which we’ve already accepted the ontological status of the verb-phrase “is on first”.

Whereas Annotation I should presumably exist before statement *S* is ever made (and it certainly helps make that statement make sense), Annotation II is most properly understood with reference to the fully-formed statement *S*. However, Annotation II is different from a statement like (*S* “has truth value” *F*) in that it looks into the guts of *S*.

Note D.3 (*Comparison of places and ontological status*). The difference between (I) a “global” annotation, and (II) the annotation of a specific sentence is analogous to the difference between (a) relationships between objects without a place, and (b) relationships between objects in specific places. (Cf. Note D.2: “global” statements are of course made “local” by the theories that scope them.)

For example, in a descriptive ontology of research documents, I might make the “placeless” statement,

a. (“Introduction” “names” “a section”)

On the other hand, the statement

b. (“Introduction” “has subject” “American History”),

seems likely to be about a specific Introduction. (And somewhere in the backend, this triple should be expressed in terms of places!)

E Appendix: Resource use

Note E.1 (*Free culture in action*). I thought it worthwhile to include this quote from a joint paper with Aaron Krowne (see Footnote 1):

“[F]ree content typically manifests aspects of a common resource as well as an open access resource; while anyone can do essentially whatever they wish with the content offline, in its online life, the content is managed in a socially-mediated way. In particular, rights to *in situ* modification tend to be strictly controlled. [...] By finding new ways to support freedom of speech within CBPP documents, we embrace subjectivity as a way to enhance the content of an intersubjectively valued corpus. In the context of “hackable” media and maintenance protocols, the semantics with which scholia are handled can be improved upon indefinitely on a user-by-user basis and a resource-wide basis. This is free culture in action.”