Erudite

Literate Programming System for Common Lisp

Mariano Montone

April 11, 2015

Contents

1 Introduction

Erudite is a system for Literate Programming in Common Lisp.

Some of its salient features are:

- Documentation is written in Common Lisp comments. This is very useful because you can work with your program as if it were not a literate program: you can load it, work from SLIME, etc, directly.
- Multiple syntaxes. Multiple type of literate syntax are supported. It is possible to choose from the default Erudite syntax, or use plain Latex or Sphinx syntax, and potentially others.
- Multiple outputs. Like Latex, Sphinx, Markdown, HTML, etc.
- Automatic indexing and cross-references.
- A command line interface.
- It is portable. You can compile and use in several CL systems.

2 Other systems

2.1 LP/Lisp

LP/Lisp is an LP system for CL by Roy M. Turner. *Erudite* shares several of its design decisions with it.

Contrary to traditional LP systems, but like *Erudite* extracts text from CL comments. That makes it possible to work with the lisp program interactively; there's no tangling needed.

But unlike *Erudite*:

- It is not portable. It runs on Allegro Common Lisp only.
- It is tightly bound to Latex, but in its input and its output.
- It is not very easily extensible in its current version (an extensible OO model is planned for its version 2).

2.2 CLWEB

CLWEB is a more traditional LP system for Common Lisp. It is not possible to work with the Lisp program in interpreter mode, as it requires previous code tangling.

3 Invocation

Erudite is invoked calling erudite function.

```
(defun call-with-destination (destination function)
  (cond
    ((null destination)
     (with-output-to-string (output)
       (funcall function output)))
    ((pathnamep destination)
     (with-open-file (f destination :direction :output
                                    :if-exists :supersede
                                    :if-does-not-exist :create)
       (funcall function f)))
    ((streamp destination)
     (funcall function destination))
    ((eql destination t)
     (funcall function *standard-output*))
    (t (error "Invalid destination: ~A" destination))))

(defun maybe-invoke-debugger (condition)
  "This function is called whenever a
condition CONDITION is signaled in Erudite."
  (if (not *catch-errors-p*)
      (invoke-debugger condition)
      (format t "ERROR: ~A~%" condition)))

(defun call-with-error-handling (catch-errors-p function)
  (setf *catch-errors-p* catch-errors-p)
  (handler-bind
      ((error #'maybe-invoke-debugger))
    (funcall function)))

(defmacro with-destination ((var destination) &body body)
  `(call-with-destination ,destination
                          (lambda (,var) ,@body)))

(defmacro with-error-handling ((&optional (catch-errors-p 't))  &body body)
  `(call-with-error-handling ,catch-errors-p (lambda () ,@body)))

(defun erudite (destination file-or-files
                &rest args &key
                             (output-type *output-type*)
                             (syntax *syntax*)
                             debug
                             verbose
                             (catch-errors-p t)
                             &allow-other-keys)
  "Processes literate lisp files and creates a document.

   Args: - destination: If NIL, output is written to a string. If T, output is written
      to *standard-output*. If a pathname, then a file is created. Otherwise, a stream
      is expected.
         - files: Literate lisp files to compile
         - args: All sort of options passed to the generation functions
         - output-type: The kind of document to generate.
```

```
                        One of :latex, :sphinx
                        Default: :latex
       - syntax: The kind of syntax used in the literate source files.
                       One of: :erudite, :latex, :sphinx.
                       Default: :erudite"
(with-error-handling (catch-errors-p)
  (with-destination (output destination)
    (let ((*output-type* output-type)
          (*syntax* syntax)
          (*debug* debug)
          (*verbose* verbose))
      (when *verbose*
        (log:config :info))
      (when *debug*
        (log:config :debug))
      (apply #'gen-doc output-type
             output
             (if (listp file-or-files)
                 file-or-files
                 (list file-or-files))
             args)))))
```

4 Algorithm

Multiple passes are run on the input files. This is because we want to be able to invoke chunks and extracts from file to file, from top to down and down to top. In a word, from everywhere without restrictions.

4.1 Includes expansion

In the first pass, *include* directives are expanded to be able to process the whole thing from a single stream.

```
(defvar *include-path* nil)

(defun expand-includes (stream)
  "Expand include directives"
  (with-output-to-string (output)
    (loop
      :for line := (read-line stream nil)
      :while line
      :do
         (cond
           ((scan "@include-path\\s+(.+)" line)
            (log:debug "~A" line)
            (register-groups-bind (path) ("@include-path\\s+(.+)" line)
              (setf *include-path* (pathname path))))
           ((scan "@include\\s+(.+)" line)
            (register-groups-bind (filename-or-path) ("@include\\s+(.+)" line)
              (let ((pathname (cond
                                ((fad:pathname-absolute-p
                                  (pathname filename-or-path))
                                 filename-or-path)
                                (*include-path*
                                 (merge-pathnames filename-or-path
                                                  *include-path*))
                                (*current-path*
                                 (merge-pathnames filename-or-path
                                                  *current-path*))
                                (t (error "No base path for include. This should not
                                   have happened")))))
                (log:debug "Including ~A" pathname)
```

Expand the included file source into output

```
                (write-string (file-to-string pathname) output)
                )))
           (t
            (write-string line output)
            (terpri output))))))
```

4.2 Chunks extraction

After includes have been expanded, it is time to extract chunks.

`@chunk` definitions are extracted from the source, and added to the **chunks** list for later processing. The chunk name is printed via *write-chunk-name* when a chunk is found.

```
(defun extract-chunks (string)
  "Splits a file source in docs and code"
  (with-input-from-string (stream string)
    (with-output-to-string (output)
      (loop
        :with current-chunk := nil
        :for line := (read-line stream nil)
        :while line
        :do
           (cond
             ((scan "@chunk\\s+(.+)" line)
              (register-groups-bind (chunk-name) ("@chunk\\s+(.+)" line)
                (setf current-chunk (list :name chunk-name
                                          :output (make-string-output-stream)))
                (write-chunk-name chunk-name output)
                (terpri output)))
              (push (cons (getf current-chunk :name)
                          (getf current-chunk :output))
                    *chunks*)
              (setf current-chunk nil))
             (current-chunk
              (let ((chunk-output (getf current-chunk :output)))
                (write-string line chunk-output)
                (terpri chunk-output)))
             (t
              (write-string line output)
              (terpri output)))))))
```

Once both includes have been expanded, and chunks have been pre proccessed, the resulting output with literate code is parsed into *fragments*. Fragments can be of type *documentation* or type *code*. *documentation* is the text that appears in Common Lisp comments. *code* fragments are the rest. This is done via the split-file-source function.

```
(defun split-file-source (str)
  "Splits a file source in docs and code"
  (with-input-from-string (stream str)
    (append-source-fragments
     (loop
       :for line := (read-line stream nil)
       :while line
       :collect
       (parse-line line stream)))))

(defun parse-line (line stream)
  (or
   (parse-long-comment line stream)
   (parse-short-comment line stream)
   (parse-code line stream)))

(defun parse-long-comment (line stream)
  "Parse a comment between #| and |#"
```

TODO: this does not work for long comments in one line

```
  (when (equalp (search "#|" (string-left-trim (list #\  #\tab) line))
                0)
```

We've found a long comment Extract the comment source

```
    (let ((comment
```

```
            (with-output-to-string (s)
```

First, add the first comment line

```
              (register-groups-bind (comment-line) ("\\#\\|\\s*(.+)" line)
                (write-string comment-line s))
```

While there are lines without `|#`, add them to the comment source

```
              (loop
                :for line := (read-line stream nil)
                :while (and line (not (search "|#" line)))
                :do
                   (terpri s)
                   (write-string line s)
                :finally
```

Finally, extract the last comment line

```
                   (if line
                       (register-groups-bind (comment-line) ("\\s*(.+)\\|\\#" line)
                         (when comment-line
                           (write-string comment-line s)))
                       (error "EOF: Could not complete comment parsing"))))))
      (list :doc comment))))

(defun parse-short-comment (line stream)
  (when (equalp
         (search *short-comments-prefix*
                 (string-left-trim (list #\  #\tab)
                                   line))
         0)
```

A short comment was found

```
    (let* ((comment-regex (format nil "~A\\s*(.+)" *short-comments-prefix*))
           (comment
             (with-output-to-string (s)
               (register-groups-bind (comment-line) (comment-regex line)
                 (write-string
                  (string-left-trim (list #\; #\ )
                                    comment-line)
                  s)))))
      (list :doc comment))))

(defun parse-code (line stream)
  (list :code line))

(defun append-source-fragments (fragments)
  "Append docs and code fragments"
  (let ((appended-fragments nil)
        (current-fragment (first fragments)))
    (loop
      :for fragment :in (cdr fragments)
      :do
         (if (equalp (first fragment) (first current-fragment))
```

The fragments are of the same type. Append them

```
             (setf (second current-fragment)
                   (with-output-to-string (s)
                     (write-string (second current-fragment) s)
                     (terpri s)
                     (write-string (second fragment) s)))
```

else, there's a new kind of fragment

```
              (progn
                (setf appended-fragments (append-to-end current-fragment
                   appended-fragments))
                (setf current-fragment fragment))))
    (setf appended-fragments (append-to-end current-fragment appended-fragments))
    appended-fragments))

(defun process-fragments (fragments output)
  (when fragments
    (let ((first-fragment (first fragments)))
      (process-fragment (first first-fragment) first-fragment
                        output
                        (lambda (&key (output output))
                          (process-fragments (rest fragments) output))))))

(defgeneric process-fragment (fragment-type fragment output cont))

(defmethod process-fragment ((type (eql :code)) fragment output cont)
  (when (not
         (zerop (length
                 (remove #\  (remove #\newline (second fragment)))))))
```

Extract and output indexes first

```
    (let ((indexes (extract-indexes (second fragment))))
      (write-indexes indexes output *output-type*))
    (write-code (second fragment) output *output-type*))
  (funcall cont))

(defmethod process-fragment ((type (eql :doc)) fragment output cont)
  (with-input-from-string (input (second fragment))
    (labels ((%process-fragment (&key (input input) (output output))
               (flet ((process-cont (&key (input input) (output output))
                        (%process-fragment :input input :output output)))
                 (let ((line (read-line input nil)))
                   (if line
                       (maybe-process-command line input output #'process-cont)
                       (funcall cont :output output))))))
      (%process-fragment))))

(defmethod maybe-process-command (line input output cont)
  "Process a top-level command"
  (let ((command (find-matching-command line)))
    (if command
        (process-command command line input output cont)
        (process-doc *syntax* *output-type* line output cont))))

(defmethod process-doc ((syntax (eql :latex)) output-type line stream cont)
  (write-string line stream)
  (terpri stream)
  (funcall cont))

(defmethod process-doc ((syntax (eql :sphinx)) output-type line stream cont)
  (write-string line stream)
  (terpri stream)
  (funcall cont))

(defmethod process-doc ((syntax (eql :erudite)) output-type line stream cont)
  (let ((formatted-line line))
    (loop
      :for syntax :in *erudite-syntax*
      :while formatted-line
      :when (match-syntax syntax formatted-line)
```

```
        :do
           (setf formatted-line (process-syntax syntax formatted-line stream
              output-type))
      :finally (when formatted-line
                 (write-doc-line formatted-line stream output-type)))
    (terpri stream)
    (funcall cont)))

(defmethod write-doc-line (line stream output-type)
  (write-string line stream))

(defmethod write-code (code stream (output-type (eql :latex)))
  (write-string "\\begin{code}" stream)
  (terpri stream)
  (write-string code stream)
  (terpri stream)
  (write-string "\\end{code}" stream)
  (terpri stream))

(defmethod write-code (code stream (output-type (eql :sphinx)))
  (terpri stream)
  (write-string ".. code-block:: common-lisp" stream)
  (terpri stream)
  (terpri stream)
  (write-string (indent-code code) stream)
  (terpri stream)
  (terpri stream))

(defmethod write-code (code stream (output-type (eql :markdown)))
  (terpri stream)
  (write-string "```lisp" stream)
  (terpri stream)
  (write-string code stream)
  (terpri stream)
  (write-string "```" stream)
  (terpri stream))

(defmethod write-chunk-name (chunk-name stream)
  (write-string "<<<" stream)
  (write-string chunk-name stream)
  (write-string ">>>" stream))

(defmethod write-chunk (chunk-name chunk stream)
  (write-code (format nil "<<~A>>=~%~A" chunk-name chunk)
              stream *output-type*))
```

4.3 Chunks and extracts post processing

Once the literate code has been parsed and processed, it is time to resolve the pending chunks and extracts. This is done in *post-process-output* function.

`INSERT_CHUNK` and `INSERT_EXTRACT` are looked for and replaced by entries in *chunks* and *extracts*, respectively.

```
(defun post-process-output (str)
  "Resolve chunk inserts and extract inserts after processing"

  (with-output-to-string (output)
    (with-input-from-string (s str)
      (loop
        :for line := (read-line s nil)
```

```
        :while line
        :do
           (cond
             ((scan "^__INSERT_CHUNK__(.*)$" line)
              (register-groups-bind (chunk-name)
                  ("^__INSERT_CHUNK__(.*)$" line)
```

Insert the chunk

```
                (let ((chunk (find-chunk chunk-name)))
                  (write-chunk chunk-name
                               (get-output-stream-string (cdr chunk))
                               output))))
             ((scan "^__INSERT_EXTRACT__(.*)$" line)
              (register-groups-bind (extract-name)
                  ("^__INSERT_EXTRACT__(.*)$" line)
```

Insert the extract

```
                (let ((extract (find-extract extract-name)))
                  (write-string (get-output-stream-string (cdr extract))
                                output))))
             (t
              (write-string line output)
              (terpri output)))))))
```

4.4 Conclusion

The whole process is invoked from process-file-to-string function.

```
(defmethod process-file-to-string ((pathname pathname))
  (let ((*current-path* (fad:pathname-directory-pathname pathname)))
    (with-open-file (f pathname)
      (post-process-output
       (with-output-to-string (s)
         (process-fragments
          (split-file-source
           (extract-chunks
            (expand-includes f)))
          s))))))

(defmethod process-file-to-string ((files cons))
  (post-process-output
   (with-output-to-string (s)
     (let ((*current-path*
             (fad:pathname-directory-pathname (first files))))
       (process-fragments
        (loop
          :for file :in files
          :appending
          (with-open-file (f file)
            (split-file-source
             (extract-chunks
              (expand-includes f)))))
        s)))))

(defmethod process-file-to-string :before (pathname)
  (setf *chunks* nil
        *extracts* nil))

(defmethod process-file-to-string :after (pathname)
```

```
  (setf *chunks* nil
        *extracts* nil))

(defun process-string (string)
  (let ((*chunks* nil)
        (*extracts* nil))
    (post-process-output
     (with-input-from-string (f string)
       (with-output-to-string (s)
         (process-fragments
          (split-file-source
           (extract-chunks
            (expand-includes f)))
          s))))))
```

5 Source code indexing

```
(defun parse-definition-type (str)
  (case (intern (string-upcase str))
    (defun :function)
    (defmacro :macro)
    (defclass :class)
    (defvar :variable)
    (defparameter :variable)
    (defmethod :method)
    (defgeneric :generic)
    (otherwise (intern (string-upcase str) :keyword))))

(defun extract-indexes (code)
  (let ((indexes))
    (loop
      :for line :in (split-sequence:split-sequence #\newline code)
      :do
         (do-register-groups (definition-type name)
             ("^\\((def\\S*)\\s+([^\\s(]*)" line)
           (push (list (parse-definition-type definition-type)
                       name)
                 indexes)))
    indexes))

(defgeneric write-indexes (indexes output output-type))

(defmethod write-indexes (indexes output (output-type (eql :latex)))
  (when indexes
                                        ; (format output "\\lstset{~{index={~A}~^,~}}"
                                        ;            (mapcar (alexandria:compose #'
                                        ;                    escape-latex #'second)
                                        ;                    indexes))
    (loop for index in (remove-duplicates indexes :key #'second :test #'equalp)
          do
             (format output "\\index{~A}~%" (escape-latex (second index)))
             (format output "\\label{~A}~%" (latex-label (second index))))
    (terpri output)))

(defmethod write-indexes (indexes output (output-type (eql :sphinx)))
```

TODO: implement

```
  )

(defmethod write-indexes (indexes output (output-type (eql :markdown)))
```

TODO: implement

```
  )

(defun escape-latex (str)
  (let ((escaped str))
    (flet ((%replace (thing replacement)
             (setf escaped (regex-replace-all thing escaped replacement))))
      (%replace "\\\\" "\\textbackslash")
```

```
      (%replace "\\&" "\\&")
      (%replace "\\%" "\\%")
      (%replace "\\$" "\\$")
      (%replace "\\#" "\\#")
      (%replace "\\_" "\\_")
      (%replace "\\{" "\\{")
      (%replace "\\}" "\\}")
      (%replace "\\~" "\\textasciitilde")
      (%replace "\\^" "\\textasciicircum")
      escaped)))

(defun latex-label (str)
  (let ((escaped str))
    (flet ((%replace (thing replacement)
             (setf escaped (regex-replace-all thing escaped replacement))))
      (%replace "\\\\" "=")
      (%replace "\\&" "=")
      (%replace "\\%" "=")
      (%replace "\\$" "=")
      (%replace "\\#" "=")
      (%replace "\\_" "=")
      (%replace "\\{" "=")
      (%replace "\\}" "=")
      (%replace "\\~" "=")
      (%replace "\\^" "=")
      escaped)))
```

Code blocks in Sphinx are indented. The indent-code function takes care of that:

```
(defun indent-code (code)
  "Code in sphinx has to be indented"
  (let ((lines (split-sequence:split-sequence #\newline
                                              code)))
    (apply #'concatenate 'string
           (mapcar (lambda (line)
                     (format nil "     ~A~%" line))
                   lines))))
```

6 Outputs

Erudite supports LaTeX, Markdown and Sphinx generation at the moment.

6.1 LaTeX

```
(defgeneric gen-doc (output-type output files &rest args))

(defmethod gen-doc ((output-type (eql :latex)) output files
                    &key
                      (title *title*)
                      (subtitle *subtitle*)
                      (author *author*)
                      template-pathname
                      (syntax *syntax*)
                      (document-class *latex-document-class*)
                      &allow-other-keys)
  "Generates a LaTeX document.

   Args: - output: The output stream.
         - files: The list of .lisp files to compile
         - title: Document title.
         - subtitle: Document subtitle.
         - author: Author of the document
         - template-pathname: A custom LaTeX template file. If none is specified, a
             default template is used."
  (let ((*latex-document-class* document-class))
    (let ((template (cl-template:compile-template
                     (file-to-string (or template-pathname
                                         (asdf:system-relative-pathname
                                          :erudite
                                          "latex/template.tex"))))))
          (body (process-file-to-string files)))
      (write-string
       (funcall template (list :title (or title
                                          *title*
                                          (error "No document title specified"))
                               :subtitle (or subtitle
                                             *subtitle*)
                               :author (or author
                                           *author*
                                           (error "No document author specified"))
                               :body body))
       output))
    t))
```

6.2 Sphinx

Sphinx is the other kind of output apart from LaTeX.

```
(defmethod gen-doc ((output-type (eql :sphinx)) output files &key prelude postlude
   syntax &allow-other-keys)
```

```
  "Generates Sphinx document.

   Args: - output: The output stream.
         - files: .lisp files to compile.
         - prelude: String (or pathname) to append before the Sphinx document.
         - postlude: String (or pathname) to append after the Sphinx document."
  (when prelude
    (write-string
     (if (pathnamep prelude)
         (file-to-string prelude)
         prelude)
     output))
  (write-string (process-file-to-string files) output)
  (when postlude
    (write-string (if (pathnamep postlude)
                      (file-to-string postlude)
                      postlude)
                  output)))
```

6.3 Markdown

Markdown is another output type.

```
(defmethod gen-doc ((output-type (eql :markdown)) output files &key prelude postlude
    syntax &allow-other-keys)
  "Generates Markdown document.

   Args: - output: The output stream.
         - files: .lisp files to compile.
         - prelude: String (or pathname) to append before the document.
         - postlude: String (or pathname) to append after the document."
  (when prelude
    (write-string
     (if (pathnamep prelude)
         (file-to-string prelude)
         prelude)
     output))
  (write-string (process-file-to-string files) output)
  (when postlude
    (write-string (if (pathnamep postlude)
                      (file-to-string postlude)
                      postlude)
                  output)))
```

7 Command line interface

It is possible to invoke *Erudite* from the command line

Run `make` to build `erudite` executable.

This is the command line syntax:

```
Usage: erudite [-hvd] [+vd] [OPTIONS] FILES...

Erudite is a Literate Programming System for Common Lisp
  -h, --help                  Print this help and exit.
  --version                   Print Erudite version
  -(+)v, --verbose[=yes/no]   Run in verbose mode
                              Fallback: yes
                              Environment: VERBOSE
  -(+)d, --debug[=on/off]     Turn debugging on or off.
                              Fallback: on
                              Environment: DEBUG
  -o, --output=OUTPUT         The output file. If none is used, result is
                              printed to stdout
  --output-type=OUTPUT-TYPE   The output type. One of 'latex', 'sphinx'
                              Default: latex
  --syntax=SYNTAX             The syntax used in source files. One of 'latex',
                              'sphinx', 'erudite'
                              Default: erudite
  --author=AUTHOR             The author to appear in the document
  --title=TITLE               The document title
```

Then run `sudo make install` to install globally in your system

Here is an example usage:

```
erudite -o erudite.tex erudite.lisp
```

7.1 Implementation

The command line is implemented via the *com.dvl.clon* library.

```
(ql:quickload :com.dvlsoft.clon)
(ql:quickload :erudite)

(defpackage erudite.cli
  (:use :cl :erudite))

(eval-when (:execute :load-toplevel :compile-toplevel)
  (com.dvlsoft.clon:nickname-package))

(clon:defsynopsis (:postfix "FILES...")
```

```
  (text :contents (format nil "Erudite is a Literate Programming System for Common Lisp
     "))
  (flag :short-name "h" :long-name "help"
        :description "Print this help and exit.")
  (flag :long-name "version"
        :description "Print Erudite version")
  (switch :short-name "v" :long-name "verbose"
          :description "Run in verbose mode"
          :env-var "VERBOSE")
  (switch :short-name "d" :long-name "debug"
          :description "Turn debugging on or off."
          :argument-style :on/off
          :env-var "DEBUG")
  (path :long-name "output"
        :short-name "o"
        :argument-name "OUTPUT"
        :type :file
        :description "The output file. If none is used, result is printed to stdout")
  (enum :long-name "output-type"
        :argument-name "OUTPUT-TYPE"
        :enum (list :latex :sphinx)
        :default-value :latex
        :description "The output type. One of 'latex', 'sphinx'")
  (enum :long-name "syntax"
        :argument-name "SYNTAX"
        :enum (list :erudite :latex :sphinx)
        :default-value :erudite
        :description "The syntax used in source files. One of 'latex', 'sphinx', '
           erudite'")
  (stropt :long-name "author"
          :argument-name "AUTHOR"
          :description "The author to appear in the document")
  (stropt :long-name "title"
          :argument-name "TITLE"
          :description "The document title"))

(defun stringp* (str)
  (and (stringp str)
       (not (equalp str ""))
       str))

(defun main ()
  (clon:make-context)
  (cond
    ((or (clon:getopt :short-name "h")
         (not (clon:cmdline-p)))
     (clon:help))
    ((clon:getopt :long-name "version")
     (print "Erudite Literate Programming System for Common Lisp version 0.0.1"))
    (t
     (let ((title (stringp* (clon:getopt :long-name "title")))
           (author (stringp* (clon:getopt :long-name "author")))
           (output-type (clon:getopt :long-name "output-type"))
           (syntax (clon:getopt :long-name "syntax"))
           (output (or (clon:getopt :long-name "output")
                       t))
           (files (mapcar #'pathname (clon:remainder))))
       (erudite:erudite output files
                        :title title
                        :author author
                        :output-type output-type
                        :syntax syntax)))))

(clon:dump "erudite" main)
```

8 Commands

Commands are held in *commands* list

```
(defvar *commands* nil)

(defun find-command (name &optional (error-p t))
  (let ((command (gethash name *commands*)))
    (when (and error-p (not command))
      (error "Invalid command: ~A" command))
    command))

(defun find-matching-command (line)
  (loop
     :for command :in *commands*
     :when (match-command command line)
     :return command))
```

8.1 Commands definition

```
(defmacro define-command (name &body body)
  (let ((match-function-def (or (find :match body :key #'car)
                                (error "Specify a match function")))
        (process-function-def (or (find :process body :key #'car)
                                  (error "Specify a process function"))))
    `(progn
       ,(destructuring-bind (_ match-args &body match-body) match-function-def
                            `(defmethod match-command ((command (eql ',name))
                                                       ,@match-args)
                               ,@match-body))
       ,(destructuring-bind (_ process-args &body process-body)
                            process-function-def
                            `(defmethod process-command ((command (eql ',name))
                                                         ,@process-args)
                               ,@process-body))
       (pushnew ',name *commands*))))

(defgeneric match-command (command line))

(defgeneric process-command (command line input output cont))

(defmethod process-command :before (command line input output cont)
  (log:debug "Processing `~A`" line))
```

8.2 Commands list

8.2.1 Input type

```
(define-command syntax
  (:match (line)
    (scan "@syntax\\s+(.+)" line))
```

```
  (:process (line input output cont)
            (register-groups-bind (syntax) ("@syntax\\s+(.+)" line)
              (setf *syntax* (intern (string-upcase syntax) :keyword)))
            (funcall cont)))
```

8.2.2 Output type

```
(define-command output-type
  (:match (line)
    (scan "@output-type\\s+(.+)" line))
  (:process (line input output cont)
            (register-groups-bind (output-type) ("@output-type\\s+(.+)" line)
              (setf *output-type* (intern (string-upcase output-type) :keyword)))
            (funcall cont)))
```

8.2.3 Title

```
(define-command title
  (:match (line)
    (scan "@title\\s+(.+)" line))
  (:process (line input output cont)
            (register-groups-bind (title) ("@title\\s+(.+)" line)
              (setf *title* title))
            (funcall cont)))
```

8.2.4 Subtitle

```
(define-command subtitle
  (:match (line)
    (scan "@subtitle\\s+(.+)" line))
  (:process (line input output cont)
            (register-groups-bind (subtitle) ("@subtitle\\s+(.+)" line)
              (setf *subtitle* subtitle))
            (funcall cont)))
```

8.2.5 Author

```
(define-command author
  (:match (line)
    (scan "@author\\s+(.+)" line))
  (:process (line input output cont)
            (register-groups-bind (author) ("@author\\s+(.+)" line)
              (setf *author* author))
            (funcall cont)))
```

8.2.6 Chunks

```
(defun find-chunk (chunk-name &key (error-p t))
  (or (assoc chunk-name *chunks* :test #'equalp)
      (error "Chunk not defined: ~A" chunk-name)))

(define-command insert-chunk
  (:match (line)
    (scan "@insert-chunk\\s+(.+)" line))
  (:process (line input output cont)
            (register-groups-bind (chunk-name) ("@insert-chunk\\s+(.+)" line)
              (format output "__INSERT_CHUNK__~A~%" chunk-name)
              (funcall cont))))
```

8.2.7 Extraction

```
(defvar *extracts* nil)
(defvar *current-extract* nil)

(defun find-extract (extract-name &key (error-p t))
  (or (assoc extract-name *extracts* :test #'equalp)
      (and error-p
           (error "No text extracted with name: ~A" extract-name))))

(define-command extract
  (:match (line)
    (scan "@extract\\s+(.+)" line))
  (:process (line input output cont)
            (register-groups-bind (extract-name) ("@extract\\s+(.+)" line)
```

Build and register the extracted piece for later processing Redirect the output to the "extract output"

```
              (let* ((extract-output (make-string-output-stream))
                     (*current-extract* (list :name extract-name
                                              :output extract-output
                                              :original-output output)))
                (funcall cont :output extract-output)))))

(define-command end-extract
  (:match (line)
    (scan "@end extract" line))
  (:process (line input output cont)
            (push (cons (getf *current-extract* :name)
                        (getf *current-extract* :output))
                  *extracts*)
```

Restore the output

```
            (funcall cont :output (getf *current-extract* :original-output))))

(define-command insert
  (:match (line)
    (scan "@insert\\s+(.+)" line))
  (:process (line input output cont)
            (register-groups-bind (extract-name) ("@insert\\s+(.+)" line)
              (format output "__INSERT_EXTRACT__~A~%" extract-name)
              (funcall cont))))
```

8.2.8 Ignore

```
(defvar *ignore* nil)

(define-command ignore
  (:match (line)
    (scan "@ignore" line))
  (:process (line input output cont)
            (setf *ignore* t)
            (funcall cont)))

(define-command end-ignore
  (:match (line)
    (scan "@end ignore" line))
  (:process (line input output cont)
            (setf *ignore* nil)
            (funcall cont)))
```

```
(defmethod process-doc :around (syntax output-type line stream cont)
  (if *ignore*
      (funcall cont)
      (call-next-method)))

(defmethod process-fragment :around ((type (eql :code)) fragment output cont)
  (if *ignore*
      (funcall cont)
      (call-next-method)))

(defmethod maybe-process-command :around (line input output cont)
  (if (and *ignore* (not (match-command 'end-ignore line)))
      (funcall cont)
      (call-next-method)))
```

9 Erudite syntax

Erudite formatting operations are held in *erudite-syntax* list

```
(defvar *erudite-syntax* nil)

(defun find-syntax (name &optional (error-p t))
  (let ((command (gethash name *erudite-syntax*)))
    (when (and error-p (not command))
      (error "Invalid syntax: ~A" command))
    command))
```

9.1 Syntax definition

```
(defmacro define-erudite-syntax (name &body body)
  (let ((match-function-def (or (find :match body :key #'car)
                                (error "Specify a match function")))
        (process-function-def (or (find :process body :key #'car)
                                  (error "Specify a process function"))))
    `(progn
       ,(destructuring-bind (_ match-args &body match-body) match-function-def
                            `(defmethod match-syntax ((command (eql ',name))
                                                      ,@match-args)
                               ,@match-body))
       ,(destructuring-bind (_ process-args &body process-body)
                            process-function-def
                            `(defmethod process-syntax ((command (eql ',name))
                                                        ,@process-args)
                               ,@process-body))
       (pushnew ',name *erudite-syntax*))))
```

9.2 Syntax elements

9.2.1 Section

```
(define-erudite-syntax section
  (:match (line)
    (scan "@section" line))
  (:process (line output output-type)
            (register-groups-bind (title)
                ("@section\\s+(.+)" line)
              (format-syntax output (list :section title)))
            nil))
```

9.2.2 Subsection

```
(define-erudite-syntax subsection
  (:match (line)
    (scan "@subsection" line))
  (:process (line output output-type)
            (register-groups-bind (title)
```

```
              ("@subsection\\s+(.+)" line)
            (format-syntax output (list :subsection title)))
          nil))
```

9.2.3 Subsubsection

```
(define-erudite-syntax subsubsection
  (:match (line)
    (scan "@subsubsection" line))
  (:process (line output output-type)
            (register-groups-bind (title)
                ("@subsubsection\\s+(.+)" line)
              (format-syntax output (list :subsubsection title)))
            nil))
```

9.2.4 Verbatim

```
(define-erudite-syntax begin-verbatim
  (:match (line)
    (scan "@verbatim" line))
  (:process (line output output-type)
            (format-syntax output (list :begin-verbatim))
            nil))

(define-erudite-syntax end-verbatim
  (:match (line)
    (scan "@end verbatim" line))
  (:process (line output output-type)
            (format-syntax output (list :end-verbatim))
            nil))
```

9.2.5 Code

```
(define-erudite-syntax begin-code
  (:match (line)
    (scan "@code" line))
  (:process (line output output-type)
            (format-syntax output (list :begin-code))
            nil))

(define-erudite-syntax end-code
  (:match (line)
    (scan "@end code" line))
  (:process (line output output-type)
            (format-syntax output (list :end-code))
            nil))
```

9.2.6 Lists

```
(define-erudite-syntax begin-list
  (:match (line)
    (scan "@list" line))
  (:process (line output output-type)
            (format-syntax output (list :begin-list))
            nil))

(define-erudite-syntax end-list
  (:match (line)
    (scan "@end list" line))
  (:process (line output output-type)
            (format-syntax output (list :end-list))
            nil))
```

```
(define-erudite-syntax list-item
  (:match (line)
    (scan "@item" line))
  (:process (line output output-type)
            (regex-replace "@item" line
                           (lambda (match)
                             (format-syntax nil (list :list-item)))
                           :simple-calls t)))
```

9.2.7 Emphasis

```
(define-erudite-syntax emphasis
  (:match (line)
    (scan "@emph{(.*?)}" line))
  (:process (line output output-type)
            (regex-replace-all "@emph{(.*?)}" line
                               (lambda (match text)
                                 (format-syntax nil (list :emph text)))
                               :simple-calls t)))
```

9.2.8 Bold

```
(define-erudite-syntax bold
  (:match (line)
    (scan "@bold{(.*?)}" line))
  (:process (line output output-type)
            (regex-replace-all "@bold{(.*?)}" line
                               (lambda (match text)
                                 (format-syntax nil (list :bold text)))
                               :simple-calls t)))
```

9.2.9 Italics

```
(define-erudite-syntax italics
  (:match (line)
    (scan "@it{(.*?)}" line))
  (:process (line output output-type)
            (regex-replace-all "@it{(.*?)}" line
                               (lambda (match text)
                                 (format-syntax nil (list :italics text)))
                               :simple-calls t)))
```

9.2.10 Inline verbatim

```
(define-erudite-syntax inline-verbatim
  (:match (line)
    (scan "@verb{(.*?)}" line))
  (:process (line output output-type)
            (regex-replace-all "@verb{(.*?)}" line
                               (lambda (match text)
                                 (format-syntax nil (list :inline-verbatim text)))
                               :simple-calls t)))
```

9.2.11 Link

```
(define-erudite-syntax link
  (:match (line)
    (scan "@link{(.*?)}{(.*?)}" line))
  (:process (line output output-type)
            (regex-replace-all "@link{(.*?)}{(.*?)}" line
                               (lambda (match target label)
```

```
                                  (format-syntax nil (list :link target label)))
                                :simple-calls t)))
```

9.2.12 Label

```
(define-erudite-syntax label
  (:match (line)
    (scan "@label{(.*?)}" line))
  (:process (line output output-type)
            (regex-replace-all "@label{(.*?)}" line
                               (lambda (match label)
                                 (format-syntax nil (list :label label)))
                               :simple-calls t)))
```

9.2.13 Index

```
(define-erudite-syntax index
  (:match (line)
    (scan "@index{(.*?)}" line))
  (:process (line output output-type)
            (regex-replace-all "@index{(.*?)}" line
                               (lambda (match text)
                                 (format-syntax nil (list :index text)))
                               :simple-calls t)))
```

9.2.14 Reference

```
(define-erudite-syntax reference
  (:match (line)
    (scan "@ref{(.*?)}" line))
  (:process (line output output-type)
            (regex-replace-all "@ref{(.*?)}" line
                               (lambda (match text)
                                 (format-syntax nil (list :ref text)))
                               :simple-calls t)))
```

9.3 Syntax formatting

```
(defvar *latex-document-class* :article)

(defun format-syntax (destination syntax)
  (if (null destination)
      (with-output-to-string (stream)
        (%format-syntax *output-type* (first syntax) stream  syntax))
      (%format-syntax *output-type* (first syntax) destination syntax)))
```

10 Tests

```
(defpackage erudite.test
  (:use :cl :fiveam :erudite)
  (:export :run-tests))

(in-package :erudite.test)
```

Tests are run with run-tests

```
(defun run-tests ()
  (run! 'erudite-tests))

(def-suite erudite-tests)

(in-suite erudite-tests)


(defun test-file (filename)
  (merge-pathnames filename
                   (asdf:system-relative-pathname :erudite "test/")))

(test basic-processing-test
  (is
   (equalp
    (erudite::process-string ";; Hello
(print \"world\")")
    "Hello
\\begin{code}
(print \"world\")
\\end{code}
"))
  (is
   (equalp
    (erudite::process-string "#| Hello
|#
(print \"world\")")
    "Hello
\\begin{code}
(print \"world\")
\\end{code}
")))
```

11 Index

Index