

Erudite

Literate Programming System for Common Lisp

Mariano Montone

April 10, 2015

Contents

| | | |
|----------|-------------------------------|-----------|
| 1 | Introduction | 1 |
| 2 | Invocation | 3 |
| 3 | Implementation | 5 |
| 3.1 | Chunks extraction | 6 |
| 4 | Backends | 13 |
| 4.1 | LaTeX | 13 |
| 4.2 | Sphinx | 13 |
| 4.3 | Markdown | 14 |
| 5 | Command line interface | 15 |
| 5.1 | Implementation | 15 |
| 6 | Commands | 17 |
| 6.1 | Commands definition | 17 |
| 6.2 | Commands list | 17 |
| 6.2.1 | Input type | 17 |
| 6.2.2 | Output type | 18 |
| 6.2.3 | Title | 18 |
| 6.2.4 | Subtitle | 18 |
| 6.2.5 | Author | 18 |
| 6.2.6 | Chunks | 18 |
| 6.2.7 | Extraction | 18 |
| 6.2.8 | Ignore | 20 |
| 6.2.9 | Include | 20 |
| 7 | Erudite syntax | 23 |
| 7.1 | Syntax definition | 23 |
| 7.2 | Commands list | 23 |
| 7.2.1 | Section | 23 |
| 7.2.2 | Subsection | 23 |
| 7.2.3 | Subsubsection | 24 |
| 7.2.4 | Verbatim | 24 |
| 7.2.5 | Code | 24 |
| 7.2.6 | Lists | 24 |
| 7.2.7 | Emphasis | 25 |
| 7.2.8 | Bold | 25 |
| 7.2.9 | Italics | 25 |
| 7.2.10 | Inline verbatim | 25 |
| 7.2.11 | Reference | 25 |
| 7.3 | Syntax formatting | 26 |
| 8 | Tests | 27 |
| 9 | Index | 29 |

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 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))))

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

(defun erudite (destination file-or-files
               &rest args &key (output-type *output-type*)
                               (syntax *syntax*)
                               &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-destination (output destination)
    (let ((*output-type* output-type)
          (*syntax* syntax))
      (apply #'gen-doc output-type
              output
              (if (listp file-or-files)
                  file-or-files
                  (list file-or-files))
              args)))))
```


3 Implementation

First, files with literate code are 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.

```
(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 f))
            s))))))

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

(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
```

3 Implementation

```
((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))))))
```

The parser works like a custom look-ahead parser, with a whole file line being the slice looked ahead. And is implemented in Continuation Passing Style.

3.1 Chunks extraction

```
(defun extract-chunks (stream)
  "Splits a file source in docs and code"
  (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))))))
```

Tests:

```
(test chunks-test
  (is
    (equalp
      (erudite::process-file-to-string (test-file "chunk1.lisp"))
```

```

    "This is a good chunk
\\begin{code}
<<<chunk1>>>
\\end{code}
"))
  (is
    (equalp
      (erudite::process-file-to-string (test-file "chunk2.lisp"))
      "This is a good chunk
\\begin{code}
<<<chunk2>>>
\\end{code}
This is the chunk:
\\begin{code}
<<chunk2>>=
(+ 1 1)

\\end{code}
"
    ))
  (signals error
    (erudite::process-file-to-string (test-file "chunk3.lisp")))
  (is
    (equalp
      (erudite::process-file-to-string (test-file "chunk4.lisp"))
      "\\begin{code}
<<chunk4>>=
(print \"Start\")

\\end{code}
The end
\\begin{code}
<<<chunk4>>>
\\end{code}
"))
  (is (equalp
    (erudite::process-file-to-string (test-file "factorial.lisp"))
    "This is the factorial function:
\\begin{code}
(defun factorial (n)
  (if (<= n 1)
    <<<base-case>>>
    <<<recursive-case>>>
    ))

\\end{code}
The base case is simple, just check for \\verb|n=1| less:
\\begin{code}
<<base-case>>=
  1

\\end{code}
The recursive step is \\verb|n x n - 1|:
\\begin{code}
<<recursive-case>>=
  (* n (factorial (1- n)))

\\end{code}
"))))

(defun split-file-source (str)
  "Splits a file source in docs and code"
  (with-input-from-string (stream str)

```

3 Implementation

```
(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)
        (register-groups-bind (comment-line) ("\\#\\|\\s*(.*)" line)
          (write-string comment-line 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-to-end (thing list)
  (cond
    ((null list)
     (list thing))
    (t
     (setf (cdr (last list))
           (list thing))
     list)))

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

```

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)

```

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))

```

3 Implementation

```
(%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))

(defmethod write-code (code stream (output-type (eql :markdown)))
  (terpri stream)
  (write-string "```\n" stream)
  (terpri stream)
  (write-string (indent-code code) stream)
  (write-string "```\n" 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*))

(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 (eq :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 (eq :sphinx)))

TODO: implement
)

(defmethod write-indexes (indexes output (output-type (eq :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 "\\$" "\\$"))
    escaped)

```

3 Implementation

```
(%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))))
```


4 Backends

Erudite supports LaTeX and Sphinx generation at the moment.

4.1 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))
```

4.2 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)))
```

4.3 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)))
```

5 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

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

5.1 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..."))
```

5 Command line interface

```
(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)
```

6 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))
```

6.1 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*)))
```

6.2 Commands list

6.2.1 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)))
```

6.2.2 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)))
```

6.2.3 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)))
```

6.2.4 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)))
```

6.2.5 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)))
```

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

6.2.7 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))))

```

Tests

```

(test extract-test
  (is
    (equalp
      (erudite::process-file-to-string (test-file "extract1.lisp"))
      "Extract test
This has been extracted
\\begin{code}
(+ 1 2)
\\end{code}
"))
  (signals error
    (erudite::process-file-to-string (test-file "extract2.lisp")))
  (is
    (equalp
      (erudite::process-file-to-string (test-file "extract3.lisp"))
      "Start
Extract 3
End
"))))

```

6.2.8 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)))

```

6.2.9 Include

```

(defvar *include-path* nil)

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

(define-command include
  (:match (line)
    (scan "@include\\s+(.*)" line))
  (:process (line input output cont)
    (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*)))
        (t (merge-pathnames filename-or-path
          *current-path*))))))

```

Process and output the included file


```
(write-string (process-file-to-string pathname) output)
(terpri output)
(funcall cont))))))
```


7 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))
```

7.1 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*)))
```

7.2 Commands list

7.2.1 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))
```

7.2.2 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))
```

7.2.3 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))
```

7.2.4 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))
```

7.2.5 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))
```

7.2.6 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))
```

```

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)))

```

7.2.7 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)))

```

7.2.8 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)))

```

7.2.9 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)))

```

7.2.10 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)))

```

7.2.11 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)))
```

7.3 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)))
```

8 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}
"))))
```


9 Index

Index

- *commands*, 17
- *current-extract*, 18
- *erudite-syntax*, 23
- *extracts*, 18
- *ignore*, 20
- *include-path*, 20
- *latex-document-class*, 26

- append-source-fragments, 8
- append-to-end, 8
- author, 18

- begin-code, 24
- begin-list, 24
- begin-verbatim, 24
- bold, 25

- call-with-destination, 3

- define-command, 17
- define-erudite-syntax, 23

- echo, 18
- emphasis, 25
- end-code, 24
- end-extract, 19
- end-ignore, 20
- end-list, 24
- end-verbatim, 24
- erudite, 3
- erudite-tests), 27
- erudite.cli, 15
- erudite.test, 27
- escape-latex, 11
- extract, 18
- extract-chunks, 6
- extract-indexes, 9

- factorial, 6
- find-chunk, 18
- find-command, 17
- find-extract, 18
- find-matching-command, 17
- find-syntax, 23
- format-syntax, 26

- gen-doc, 13, 14

- ignore, 20
- include, 20
- include-path, 20
- indent-code, 12
- inline-verbatim, 25
- insert, 19
- italics, 25

- latex-label, 11
- list-item, 24

- main, 15
- maybe-process-command, 9, 20

- output-type, 18

- parse-code, 8
- parse-definition-type, 9
- parse-line, 7
- parse-long-comment, 7
- parse-short-comment, 8
- post-process-output, 5
- process-doc, 9, 20
- process-file-to-string, 5
- process-fragment, 9, 20
- process-fragments, 9
- process-string, 5

- reference, 25
- run-tests, 27

- section, 23
- split-file-source, 7
- stringp*, 15
- subsection, 23
- subsubsection, 24
- subtitle, 18
- syntax, 17

- test-file, 27
- title, 18

- with-destination, 3
- write-chunk, 9
- write-chunk-name, 9
- write-code, 9
- write-doc-line, 9
- write-indexes, 9, 11