Erudite

Literate Programming System for Common Lisp

Mariano Montone

March 11, 2020

Contents

0.1 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.

0.2 Literate Programming

0.2.1 Concept

Literate programming is an approach to programming introduced by Donald Knuth in which a program is given as an explanation of the program logic in a natural language, such as English, interspersed with snippets of macros and traditional source code, from which a compilable source code can be generated.

The literate programming paradigm, as conceived by Knuth, represents a move away from writing programs in the manner and order imposed by the computer, and instead enables programmers to develop programs in the order demanded by the logic and flow of their thoughts. Literate programs are written as an uninterrupted exposition of logic in an ordinary human language, much like the text of an essay, in which macros are included to hide abstractions and traditional source code.

Literate programming tools are used to obtain two representations from a literate source file: one suitable for further compilation or execution by a computer, the "tangled" code, and another for viewing as formatted documentation, which is said to be "woven" from the literate source. While the first generation of literate programming tools were computer language-specific, the later ones are language-agnostic and exist above the programming languages.

0.2.2 Advantages

According to Knuth, literate programming provides higher-quality programs, since it forces programmers to explicitly state the thoughts behind the program, making poorly thought-out design decisions more obvious. Knuth also claims that literate programming provides a first-rate documentation system, which is not an add-on, but is grown naturally in the process of exposition of one's thoughts during a program's creation. The resulting documentation allows authors to restart their own thought processes at any later time, and allows other programmers to understand the construction of the program more easily. This differs from traditional documentation, in which a programmer is presented with source code that follows a compiler-imposed order, and must decipher the thought process behind the program from the code and its associated comments. The meta-language capabilities of literate programming are also claimed to facilitate thinking, giving a higher "bird's eye view" of the code and increasing the number of concepts the mind can successfully retain and process. Applicability of the concept to programming on a large scale, that of commercial-grade programs, is proven by an edition of TeX code as a literate program.

0.2.3 Contrast with document generation

Literate programming is very often misunderstood to refer only to formatted documentation produced from a common file with both source code and comments which is properly called documentation generation or to voluminous commentaries included with code. This is backwards: well-documented code or documentation extracted from code follows the structure of the code, with documentation embedded in the code; in literate programming code is embedded in documentation, with the code following the structure of the documentation.

This misconception has led to claims that comment-extraction tools, such as the Perl Plain Old Documentation or Java Javadoc systems, are "literate programming tools". However, because these tools do not implement the "web of abstract concepts" hiding behind the system of natural-language macros, or provide an ability to change the order of the source code from a machine-imposed sequence to one convenient to the human mind, they cannot properly be called literate programming tools in the sense intended by Knuth.

0.3 Other systems

0.3.1 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).

0.3.2 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.

0.4 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-case
      (funcall function)
    (error (e)
      (maybe-invoke-debugger e))))

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

(defmacro with-error-handling ((&optional (catch-errors-p '*catch-errors-p*))
                               &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 *debug*)
                             (verbose *verbose*)
                             (catch-errors-p *catch-errors-p*)
                             (code-indexing *code-indexing*)
                             (implicit-doc *implicit-doc*)
                             (implicit-code *implicit-code*)
                             (short-comments-prefix *short-comments-prefix*)
                             &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, :org, :markdown, :sphinx.
                       Default: :erudite"
  (with-error-handling (catch-errors-p)
    (with-destination (output destination)
      (let ((*output-type* output-type)
            (*syntax* syntax)
            (*debug* debug)
            (*verbose* verbose)
            (*implicit-doc* implicit-doc)
            (*implicit-code* implicit-code)
            (*short-comments-prefix* short-comments-prefix))
        (log:config :sane :this-console)
        (when *verbose*
          (log:config :info))
```

```
        (when *debug*
          (log:config :debug))
        (log:info "Processing ~A." file-or-files)
        (log:debug "Arguments: ~S"
                   (list :output-type output-type
                         :syntax syntax
                         :debug debug
                         :verbose verbose
                         :catch-errors-p catch-errors-p
                         :code-indexing code-indexing
                         :implicit-doc implicit-doc
                         :implicit-code implicit-code
                         :short-comments-prefix short-comments-prefix))
        (apply #'gen-doc output-type
               output
               (if (listp file-or-files)
                   file-or-files
                   (list file-or-files))
               args)
        (log:config :clear)))))
```

0.5 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.

0.5.1 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

```
              (with-input-from-string (source (file-to-string pathname))
                (write-string (expand-includes source) output))
              )))
           (t
            (write-string line output)
            (terpri output))))))
```

0.5.2 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.

```
(defvar *parsing-doc* nil)

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

When splitting the source in fragments, we can parse either a long comment, a short comment, or lisp code:

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

Depending on the value of *implicit-comments* we treat the comment as documentation or code

```
(defun parse-long-comment (line stream)
  "Parse a comment between #| and |#"
  (if *implicit-doc*
      (parse-long-comment-implicit line stream)
      (parse-long-comment-explicit line stream)))

(defun parse-long-comment-implicit (line stream)
```

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

```
  (let ((comment-start "#|")
        (comment-end "|#"))
    (when (equalp (search comment-start (string-left-trim (list #\  #\tab) line))
                  0)
      (setf *parsing-doc* t)
```

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)
                   (`(:SEQUENCE ,comment-start
                                (:GREEDY-REPETITION 0 NIL :WHITESPACE-CHAR-CLASS)
                                (:REGISTER (:GREEDY-REPETITION 0 NIL (:sequence (:
                                   negative-lookahead ,comment-end) :everything)))
                                (:GREEDY-REPETITION 0 NIL :WHITESPACE-CHAR-CLASS)
                                (:GREEDY-REPETITION 0 1 ,comment-end)) line)
                 (write-string (string-right-trim '(#\space) comment-line) s))
```

While there are lines without comment end string, add them to the comment source

```
               (when (not (search comment-end line))
                 (loop
                    :for line := (read-line stream nil)
                    :while (and line (not (search comment-end line)))
                    :do
                    (terpri s)
                    (write-string line s)
                    :finally
```

Finally, extract the last comment line

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

(defun parse-long-comment-explicit (line stream)
```

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

```
(when (scan "^\\s*\\#\\|\\s+@doc" line)
```

We've found a long comment explicit comment

```
(setf *parsing-doc* t)
```

Extract the comment source

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

First, add the first comment line

```
(register-groups-bind (comment-line)
    ("^\\s*\\#\\|\\s+@doc\\s+(.+)" line)
  (when comment-line
    (write-string comment-line s)))
                              ; While there are lines without `|#` or `@end
                                  doc`, add them to the comment source
(loop
   :for line := (read-line stream nil)
   :while (and line (not (or (search "|#" line)
                             (search "@end doc" line))))
   :do
   (terpri s)
   (write-string line s)
   :finally
```

Finally, extract the last comment line

```
                (if line
                    (when (not (search "@end doc" 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)
  (if *implicit-doc*
      (parse-short-comment-implicit line stream)
      (parse-short-comment-explicit line stream)))

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

A short comment was found

```
    (setf *parsing-doc* t)
    (let* ((comment-regex (format nil "~A\\s*(.+)" *short-comments-prefix*))
           (comment
            (or
             (register-groups-bind (comment-line) (comment-regex line)
               (string-left-trim (list #\; #\space)
                                 comment-line))
             "")))
      (list :doc comment))))

(defun parse-short-comment-explicit (line stream)
```

```
  (let ((regex (format nil "^\\s*~A\\s+@doc\\s*(.*)"
                       *short-comments-prefix*)))
    (cond
      ((and *parsing-doc*
            (search *short-comments-prefix*
                    (string-left-trim (list #\space #\tab)
                                      line)))

       (list :doc (string-left-trim (list #\; #\space)
                                    line)))
      ((ppcre:scan regex line)
```

A short comment was found

```
       (setf *parsing-doc* t)
       (let ((comment
              (or
               (register-groups-bind (comment-line) (regex line)
                 (string-left-trim (list #\; #\space)
                                   comment-line))
               "")))
         (list :doc comment))))))

(defun parse-code (line stream)
  (setf *parsing-doc* nil)
  (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)
```

Ensure that this is not an empty code fragment first

```
  (when (not
         (zerop (length
                 (remove #\  (remove #\newline (second fragment))))))
```

Extract and output indexes if it is enabled

```
    (when *code-indexing*
      (let ((indexes (extract-indexes (second fragment))))
        (write-indexes indexes output *output-type*)))
```

Finally write the code fragment to the output

```
    (write-code (second fragment) output *output-type*))
```

Goon with the parsing

```
  (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 :org)) output-type line stream cont)
  (write-string line stream)
  (terpri stream)
  (funcall cont))

(defmethod process-doc ((syntax (eql :markdown)) 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))

(defvar *latex-highlight-syntax* nil
  "Highlight syntax using LaTeX minted package: https://ctan.org/pkg/minted")

(defmethod write-code (code stream (output-type (eql :latex)))
  (if *latex-highlight-syntax*
      (progn
        (write-string "\\begin{minted}[fontsize=\\footnotesize]{common-lisp}" stream)
        (terpri stream)
        (write-string code stream)
        (terpri stream)
        (write-string "\\end{minted}" stream)
        (terpri stream))
      (progn
        (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-code (code stream (output-type (eql :org)))
  (write-string "#+BEGIN_SRC lisp" stream)
  (terpri stream)
  (write-string code stream)
  (terpri stream)
  (write-string " #+END_SRC" 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*))
```

0.5.3 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"
  (log:debug "Resolving chunks...")
  (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)))))))
```

0.5.4 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))))))
```

0.6 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

```
  )

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

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))))
```

0.7 Outputs

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

0.7.1 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*)
                      (highlight-syntax *latex-highlight-syntax*)
                      &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)
        (*latex-highlight-syntax* highlight-syntax))
    (let ((template (cl-template:compile-template
                     (file-to-string (or template-pathname
                                         (asdf:system-relative-pathname
                                          :erudite
                                          "resource/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))
```

0.7.2 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)))
```

0.7.3 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)))

(defmethod gen-doc ((output-type (eql :org)) output files &key prelude postlude syntax
                                                            (title *title*)
                                                            (subtitle *subtitle*)
                                                            (author *author*)
                                                            &allow-other-keys)
  "Generates Emacs org-mode 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))

  (let ((title (or title *title*)))
    (when title
      (format output "#+TITLE: ~a~%~%" title)))

  (let ((author (or author *author*)))
    (when author
      (format output "#+AUTHOR: ~A~%~%" author)))

  (write-string (process-file-to-string files) output)

  (when postlude
    (write-string (if (pathnamep postlude)
                      (file-to-string postlude)
                      postlude)
                  output)))
```

0.8 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
Default: no
Environment: VERBOSE
-(+)d, --debug[=on/off]     Turn debugging on or off.
Fallback: on
Default: off
Environment: DEBUG
-(+)id, --implicit-doc[=yes/no] Treat all comments as documentation
Fallback: yes
Default: yes
-(+)ic, --implicit-code[=yes/no] Include all code in documentation
Fallback: yes
Default: yes
-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','markdown', 'org'
Default: latex
--syntax=SYNTAX             The syntax used in source files. One of 'erudite',
'latex', 'sphinx', 'markdown', 'org'
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
```

0.8.1 Implementation

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

```
(ql:quickload :net.didierverna.clon)
(ql:quickload :erudite)

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

(eval-when (:execute :load-toplevel :compile-toplevel)
  (net.didierverna.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"
          :default-value nil
          :env-var "VERBOSE")
  (switch :short-name "d" :long-name "debug"
          :description "Turn debugging on or off."
          :argument-style :on/off
          :default-value erudite::*debug*
          :env-var "DEBUG")
  (switch :short-name "id" :long-name "implicit-doc"
          :description "Treat all comments as documentation"
          :default-value erudite::*implicit-doc*)
  (switch :short-name "ic" :long-name "implicit-code"
          :description "Include all code in documentation"
          :default-value erudite::*implicit-code*)
  (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 :markdown :org)
        :default-value erudite::*output-type*
        :description "The output type. One of 'latex', 'sphinx','markdown'")
  (enum :long-name "syntax"
        :argument-name "SYNTAX"
        :enum (list :erudite :latex :sphinx :markdown :org)
        :default-value erudite::*syntax*
        :description "The syntax used in source files. One of 'erudite', 'latex', '
           sphinx', 'markdown'")
```

```
  (stropt :long-name "short-comments-prefix"
          :argument-name "SC-PREFIX"
          :default-value erudite::*short-comments-prefix*
          :description "Prefix on short comments")
  (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")
     (format t "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")))
           (short-comments-prefix (stringp* (clon:getopt :long-name "
              short-comments-prefix")))
           (output-type (clon:getopt :long-name "output-type"))
           (syntax (clon:getopt :long-name "syntax"))
           (output (clon:getopt :long-name "output"))
           (debug (clon:getopt :long-name "debug"))
           (verbose (clon:getopt :long-name "verbose"))
           (implicit-doc (clon:getopt :long-name "implicit-doc"))
           (implicit-code (clon:getopt :long-name "implicit-code"))
           (files (mapcar #'pathname (clon:remainder))))
       (if (null files)
           (format t "Error: provide the files to process~%")
           (erudite:erudite output files
                            :debug debug
                            :verbose verbose
                            :implicit-doc implicit-doc
                            :implicit-code implicit-code
                            :short-comments-prefix short-comments-prefix
                            :title title
                            :author author
                            :output-type output-type
                            :syntax syntax))))))

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

0.9 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))
```

0.9.1 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: `~A`" command line))
```

0.9.2 Commands list

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)))
```

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)))
```

Code indexing

```
(define-command code-indexing
  (:match (line)
          (scan "@code-indexing\\s+(.+)" line))
  (:process (line input output cont)
            (register-groups-bind (code-indexing) ("@code-indexing\\s+(.+)" line)
              (setf *code-indexing*
                    (let ((*package* *erudite-package*))
                      (read-from-string code-indexing))))
            (funcall cont)))
```

Package

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

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)))
```

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)))
```

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)))
```

Comments prefix

```
(define-command short-comments-prefix
  (:match (line)
          (scan "@short-comments-prefix\\s+(.+)" line))
  (:process (line input output cont)
```

```
            (register-groups-bind (prefix) ("@short-comments-prefix\\s+(.+)" line)
              (setf *short-comments-prefix* prefix))
            (funcall cont)))
```

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))))
```

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))))
```

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)))
```

Conditional output

```
(defvar *output-condition* (list t))

(define-command when
  (:match (line)
          (scan "@when\\s(.*)" line))
  (:process (line input output cont)
            (register-groups-bind (condition) ("@when\\s(.*)" line)
              (let ((value (eval (let ((*package* *erudite-package*))
                                   (read-from-string condition)))))
                (push value *output-condition*))
              (funcall cont))))

(define-command end-when
  (:match (line)
          (scan "@end when" line))
  (:process (line input output cont)
            (pop *output-condition*)
            (funcall cont)))

(define-command if
  (:match (line)
          (scan "@if\\s(.*)" line))
  (:process (line input output cont)
            (register-groups-bind (condition) ("@if\\s(.*)" line)
              (let ((value (eval (let ((*package* *erudite-package*))
                                   (read-from-string condition)))))
                (push value *output-condition*))
              (funcall cont))))

(define-command else
  (:match (line)
          (scan "@else" line))
  (:process (line input output cont)
            (let ((value (pop *output-condition*)))
              (push (not value) *output-condition*))
            (funcall cont)))

(define-command end-if
  (:match (line)
          (scan "@end if" line))
  (:process (line input output cont)
            (pop *output-condition*)
```

```
            (funcall cont)))

(defvar *in-code-section* nil)

(define-command begin-code
  (:match (line)
          (and (not *implicit-code*)
               (scan "@code" line)))
  (:process (line input output cont)
            (setf *in-code-section* t)
            (funcall cont)))

(define-command end-code
  (:match (line)
          (and (not *implicit-code*)
               (scan "@end code" line)))
  (:process (line input output cont)
            (setf *in-code-section* nil)
            (funcall cont)))
```

Code evaluation

There's experimental and incomplete support for evaluating and printing code. The eval section evaluates and prints code connecting to a SWANK instance. For example: (concatenate 'string "hello" "world") : "helloworld"

```
(defvar *current-eval* nil)
(defvar *swank-connection* nil)
(defvar *swank-host* "localhost")
(defvar *swank-port* 4005)

(define-command begin-eval
  (:match (line)
          (scan "@eval" line))
  (:process (line input output cont)
            (let ((*current-eval* (list :output (make-string-output-stream)
                                        :original-output output)))
              (funcall cont :output (getf *current-eval* :output)))))

(define-command end-eval
  (:match (line)
          (scan "@end eval" line))
  (:process (line input output cont)
            (when (null *swank-connection*)
              (setf *swank-connection* (swank-client:slime-connect *swank-host* *
                 swank-port*)))
            (when (null *swank-connection*)
              (error "Cannot evaluate code. Error connecting to swank."))
            (when *swank-connection*
              (let ((result (swank-client:slime-eval (read-from-string (
                 get-output-stream-string (getf *current-eval* :output))) *
                 swank-connection*)))
                (write result :stream (getf *current-eval* :original-output))))
```

Restore the output

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

(defvar *in-doc-section* nil) (define-command begin-doc (:match (line) (and (not *implicit-doc*) (scan "@doc" line))) (:process (line input output cont) (setf *in-doc-section* t) (funcall cont))) (define-command end-doc (:match (line) (and (not *implicit-doc*) (scan "@end doc" line))) (:process (line input output cont) (setf *in-doc-section* nil) (funcall cont)))

```
(defmethod process-doc :around (syntax output-type line stream cont)
  (if (or *ignore*
          (not (every #'identity *output-condition*))
          #+nil(and (not *implicit-doc*)
                    (not *in-doc-section*)))
      (funcall cont)
      (call-next-method)))

(defmethod process-fragment :around ((type (eql :code)) fragment output cont)
  #+nil(setf *in-doc-section* nil)
  (if (or *ignore*
          (not (every #'identity *output-condition*))
          (and (not *implicit-code*)
               (not *in-code-section*)))
      (funcall cont)
      (call-next-method)))

(defmethod maybe-process-command :around (line input output cont)
  (if (or (and *ignore* (not (match-command 'end-ignore line)))
          (and (not (every #'identity *output-condition*))
               (not (or (match-command 'when line)
                        (match-command 'end-when line)
                        (match-command 'else line)
                        (match-command 'if line)
                        (match-command 'end-if line)))))
      (funcall cont)
      (call-next-method)))
```

0.10 Erudite syntax

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

```
(defvar *erudite-syntax* nil)
(defvar *latex-document-class* :article)

(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))
```

0.10.1 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*))))
```

0.10.2 Syntax elements

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))
```

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))
```

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))
```

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))
```

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))
```

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)))
```

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)))
```

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)))
```

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)))
```

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)))
```

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)))
```

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)))
```

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)))
```

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)))
```

0.10.3 Syntax formatting

```
(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)))
```

0.11 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}
")))


(test implicit/explicit-doc-test
  (is (equalp
       (let ((erudite::*implicit-doc* t))
         (erudite::process-file-to-string (test-file "implicit.lisp")))
       "This is implicit doc
\\begin{code}
(print \"Hello world\")
\\end{code}
End
"))
```

```
(is (equalp
     (let ((erudite::*implicit-doc* nil))
       (erudite::process-file-to-string (test-file "implicit.lisp")))
     "\\begin{code}
```

This is implicit doc

```
(print \"Hello world\")
```

End

```
\\end{code}
"))
(is (equalp
     (let ((erudite::*implicit-doc* nil)
           (erudite::*code-indexing* nil))
       (erudite::process-file-to-string (test-file "explicit.lisp")))
     "\\begin{code}
```

This is implicit and does not appear as doc

```
(print \"Hello world\")
\\end{code}
This is an explicit comment
This appears as doc
\\begin{code}
(defun bye ()
```

This comment goes in the code

```
  (print \"Bye\"))
\\end{code}
")))

(test explicit-code
  (is (equalp
       (let ((erudite:*implicit-code* nil)
             (erudite:*code-indexing* nil))
         (erudite::process-file-to-string (test-file "explicit-code.lisp")))
       "Explicit code test
This is the code:
\\begin{code}
(defun hello-world ()
  (print \"hello world\"))
\\end{code}
")))
```

1 Index

Index