Erudite Developer Manual

Mariano Montone

April 9, 2015

Contents

Chapter 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 outputs. Like Latex, Sphinx, etc.
- Automatic indexing and cross-references.
- A command line interface.
- It is portable. You can compile and use in several CL systems.

Chapter 2

Invocation

Erudite is invoked calling `erudite` function.

```
(defun erudite (pathname files  &rest args &key (output-type *output-type*)
                                            (input-type *input-type*)
                                            &allow-other-keys)
  "Processes literate lisp files and creates a document.

   Args: - pathname: Pathname of the file to generate
         - 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
         - input-type: The kind of syntax used in the literate source files.
                       One of: :erudite, :latex, :sphinx.
                       Default: :erudite"

  (let ((*output-type* output-type)
        (*input-type* input-type))
    (apply #'gen-doc output-type pathname files args)))
```

Chapter 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.

```
(defvar *short-comments-prefix* ";;")
(defvar *input-type* :erudite)
(defvar *output-type* :latex)
(defvar *current-path* nil)
(defvar *chunks* nil)

(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)
         (erudite::process-fragments
          (erudite::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
           ((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.

```
(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)))
         ((scan "@end chunk" line)
          (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))))))

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

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

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

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

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

We've found a long comment Extract the comment source

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

First, add the first comment line

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

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

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

Finally, extract the last comment line

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

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

A short comment was found

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

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

(defun append-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))
```

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)
  (write-code (second fragment) output *output-type*)
  (funcall cont))

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

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

(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 *input-type* *output-type* line output cont))))


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

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

(defmethod process-doc ((input-type (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-string formatted-line stream)))
    (terpri stream)
    (funcall cont)))

(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-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*))
```

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))))

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

Chapter 4

Backends

Erudite supports LaTeX and Sphinx generation at the moment.

4.1 LaTeX

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

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

   Args: - pathname: The pathname of the .tex file to generate.
         - files: The list of .lisp files to compile
         - title: Title of the document
         - 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"))))))
      (with-open-file (f pathname :direction :output
                         :if-exists :supersede
                         :if-does-not-exist :create)
        (write-string
         (funcall template (list :title title
                                 :author author
                                 :body (process-file-to-string files)))
         f))
      t)))
```

4.2 Sphinx

Sphinx is the other kind of output apart from LaTeX.

```
(defmethod gen-doc ((output-type (eql :sphinx)) pathname files &key prelude postlude input-type
     &allow-other-keys)
  "Generates Sphinx document.
```

```
 Args: - pathname: Pathname of the .rst file to generate.
       - 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."
(with-open-file (f pathname :direction :output
                   :if-exists :supersede
                   :if-does-not-exist :create)
  (when prelude
    (write-string
     (if (pathnamep prelude)
         (file-to-string prelude)
         prelude)
     f))
  (write-string (process-file-to-string files) f)
  (when postlude
    (write-string (if (pathnamep postlude)
                      (file-to-string postlude)
                      postlude)
                  f))))
```

Chapter 5

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))
```

5.1 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*))))
```

5.2 Commands list

5.2.1 Input type

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

5.2.2 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))))
```

5.2.3 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))))
```

5.2.4 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 (input-type 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)))
```

5.2.5 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)))))
```

Chapter 6

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))
```

6.1 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*))))
```

6.2 Commands list

6.2.1 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))
```

6.2.2 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))
```

6.2.3 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))
```

6.2.4 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))
```

6.2.5 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))
```

6.2.6 Lists

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

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

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

6.2.7 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)))
```

6.2.8 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)))
```

6.2.9 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)))
```

6.2.10 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)))
```

6.2.11 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)))
```

6.3 Syntax formatting

6.3.1 Latex output

```

(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)))

(defmethod %format-syntax ((output-type (eql :latex))
                           (selector (eql :section))
                           stream
                           syntax)
  (ecase *latex-document-class*
    (:article (format stream "\\section{~A}" (second syntax)))
    (:book (format stream "\\chapter{~A}" (second syntax)))))

(defmethod %format-syntax ((output-type (eql :latex))
                           (selector (eql :subsection))
                           stream
                           syntax)
  (ecase *latex-document-class*
    (:article (format stream "\\subsection{~A}" (second syntax)))
    (:book (format stream "\\section{~A}" (second syntax)))))

(defmethod %format-syntax ((output-type (eql :latex))
                           (selector (eql :subsubsection))
                           stream
                           syntax)
  (ecase *latex-document-class*
    (:article (format stream "\\subsubsection{~A}" (second syntax)))
    (:book (format stream "\\subsection{~A}" (second syntax)))))

(defmethod %format-syntax ((output-type (eql :latex))
                           (selector (eql :begin-verbatim))
                           stream
                           syntax)
  (format stream "\\begin{verbatim}"))

(defmethod %format-syntax ((output-type (eql :latex))
                           (selector (eql :end-verbatim))
                           stream
                           syntax)
  (format stream "\\end{verbatim}"))

(defmethod %format-syntax ((output-type (eql :latex))
                           (selector (eql :inline-verbatim))
                           stream
                           syntax)
  (format stream "\\verb|~A|" (second syntax)))


(defmethod %format-syntax ((output-type (eql :latex))
                           (selector (eql :begin-code))
                           stream
                           syntax)
  (format stream "\\begin{code}"))

(defmethod %format-syntax ((output-type (eql :latex))
                           (selector (eql :end-code))
                           stream
                           syntax)
  (format stream "\\end{code}"))

(defmethod %format-syntax ((output-type (eql :latex))
                           (selector (eql :begin-list))
                           stream
                           syntax)
  (format stream "\\begin{itemize}"))

(defmethod %format-syntax ((output-type (eql :latex))
                           (selector (eql :end-list))
                           stream
                           syntax)
  (format stream "\\end{itemize}"))

(defmethod %format-syntax ((output-type (eql :latex))
                           (selector (eql :list-item))
                           stream
                           syntax)
  (format stream "\\item" (second syntax)))

(defmethod %format-syntax ((output-type (eql :latex))
                           (selector (eql :emph))
                           stream
                           syntax)
  (format stream "\\emph{~A}" (second syntax)))

(defmethod %format-syntax ((output-type (eql :latex))
                           (selector (eql :bold))
                           stream
                           syntax)
  (format stream "\\textbf{~A}" (second syntax)))

(defmethod %format-syntax ((output-type (eql :latex))
                           (selector (eql :italics))
                           stream
                           syntax)
  (format stream "\\textit{~A}" (second syntax)))

(defmethod %format-syntax ((output-type (eql :latex))
                           (selector (eql :ref))
                           stream
                           syntax)
  (format stream "\\verb#~A#" (second syntax)))
```