Erudite Developer Manual

Mariano Montone

April 8, 2015

Contents

Chapter 1

Introduction

Erudite is a very simple 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.

- There are no chunks weaving or special directives like in original LP systems. This is not so cool, as there's no flexible way of controlling the order of the comments and code, like in other systems. But like Haskell (and its LP support), Lisp code is also pretty easy to sort without too much problems (I think...).

Chapter 2

Invocation

Erudite is invoked calling @referudite function.

```
(defun erudite (pathname files  &rest args &key (output-type *output-type*)
                                           (input-type *input-type*)
                                           &allow-other-keys)
  (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.

```lisp
")
(defvar *input-type* :erudite)
(defvar *output-type* :latex)
(defvar *current-path* 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 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 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 f)
          s))))))
```

5

```
(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
                             *output-type*))))
           ((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 split-file-source (stream)
  "Splits a file source in docs and code"
  (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 (search *short-comments-prefix*
                (string-left-trim (list #\   #\tab)
                                  line))
```

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 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 (output-type (eql :latex)))
  (write-string "<<<" stream)
  (write-string chunk-name stream)
  (write-string ">>>" stream)
  (terpri stream))

(defmethod write-chunk (chunk-name chunk stream (output-type (eql :latex)))
  (write-string "<<" stream)
  (write-string chunk-name stream)
  (write-string ">>=" stream)
  (terpri stream)
  (write-string chunk stream))
```

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 &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 ((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))))
```