Erudite Developer Manual

Mariano Montone

April 5, 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

Implementation

Implementation is very ad-hoc at the moment.

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.

```
(defun parse-lisp-source (string)
  (loop
     :with fragments = nil
     :with prev-char = nil
     :with mode = :code
     :with fragment = nil
     :for char :across string
     :do
     ;(format t "prevchar: ~A char: ~A  mode: ~A~%" prev-char char mode)
     (cond
       ((and (equalp prev-char #\#)
             (char= char #\)
             (equalp mode :code))
        ;; Documentation fragment starts
        (setf mode :doc)
        (unless (null fragment)
          (push (list :code (coerce fragment 'string)) fragments)
          (setf fragment nil)
          (setf char nil)))
       ((and (equalp prev-char #\)
             (char= char #\#)
             (equalp mode :doc))
        ;; Documentation fragment ends
        (setf mode :code)
        (unless (null fragment)
          (push (list :doc (coerce fragment 'string)) fragments)
          (setf fragment nil)
          (setf char nil)))
       ((and (equalp mode :code)
             (equalp prev-char #\#)
             (not (equalp char #\)))
        ;; False documentation start
        (setf fragment (append fragment (list prev-char char))))
       ((and (equalp mode :doc)
             (equalp prev-char #\)
             (not (equalp char #\#)))
        ;; False documentation end
        (setf fragment (append fragment (list prev-char char))))
       ((member char (list #\# #\) :test #'char=)
        ;; Dont output, could be special characters
        )
```

```
          (t
           ;; Accumulate char in current fragment
           (setf fragment (append fragment (list char)))))
       (setf prev-char char)
       :finally (unless (null fragment)
                  (push (list mode (coerce fragment 'string)) fragments))
       (return (reverse fragments))))
```

Chapter 3

Backends

Erudite support LaTeX and Sphinx generation at the moment.

3.1 LaTeX

The parsed fragments are compiled to latex code. That means embedding the code fragments found between `\begin{code}` and `\end{code}`.

```
(defun compile-latex-fragments (fragments)
  (apply #'concatenate 'string
         (loop for fragment in fragments
               collect
               (ecase (first fragment)
                 (:code (format nil "\\begin{code}~%~A~%\\end{code}"
                               (string-trim (list #\  #\newline)
                                            (second fragment))))
                 (:doc (second fragment))))))
```

To generate LaTeX, the *gen-latex-doc* function is called:

```
(defun gen-latex-doc (pathname files &key title author template-pathname)
  (let ((template (cl-template:compile-template
                   (file-to-string (or template-pathname
                                       (asdf:system-relative-pathname
                                        :erudite
                                        "latex/template.tex")))))
        (fragments
         (loop for file in files
            appending
              (parse-lisp-source (file-to-string file)))))
    (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 (compile-latex-fragments fragments)))
       f))
    t))
```

3.2 Sphinx

Sphinx is the other kind of output apart from LaTeX.

Code fragments in Sphinx must appear indented after a `.. code-block::` directive:

```
(defun compile-sphinx-fragments (fragments)
  (apply #'concatenate 'string
         (loop for fragment in fragments
            collect
              (ecase (first fragment)
                (:code (format nil ".. code-block:: common-lisp~%~%     ~A"
                               (indent-code
                                (string-trim (list #\  #\newline)
                                             (second fragment)))))
                (:doc (second fragment))))))
```

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))))
```

To generate Sphinx code, *gen-sphinx-doc* is called.

```
(defun gen-sphinx-doc (pathname files &key prelude postlude)
  (let ((fragments
         (loop for file in files
            appending
              (parse-lisp-source (file-to-string file)))))
    (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 (compile-sphinx-fragments fragments) f)
      (when postlude
        (write-string (if (pathnamep postlude)
                          (file-to-string postlude)
                          postlude)
                      f)))))
```