qbook

Contents

1 The qbook lisp documentation system

qbook generates html formatted code listings of common lisp source files. Comments in the source code are rendered as html paragraphs, text is rendered in ¡pre¿ blocks. Headings are created by preceding the text of the comment with one or more #* chars.

This is inspired by Luke Gorrie's pbook.el.

1.1 Publishing

This is the core of qbook, the driver code which takes a lisp source file and generates an html file.

1.1.1 The public entry point: PUBLISH-QBOOK

Class GENERATOR

Generic Function GENERATE

Method GENERATE

Class BOOK

Method BOOK-INDEXES-SORTED

Method ALL-CODE-PARTS

Method PERMUTATED-GLOBAL-INDEX

Function COMPARE-DESCRIPTOR-NAMES

Function SORT-DESCRIPTORS

Function SORT-PARTS-WITH-DESCRIPTORS

Function CONVERT-TO-SECTIONS

Function BUILD-INDEXES

Function PUBLISH-QBOOK - Convert FILE-NAME into a qbook html file named OUTPUT-FILE with title TITLE.

1.2 Publishing internals

1.2.1 The classes

qbook parses lisp code into a list of source-file-part objects. we have an object for code parts (each top level form is considered as a single code object), for comments and for headings.

Class SOURCE-FILE-PART - A part of a source file.

Method PRINT-OBJECT

Class CODE-PART -

Generic Function CODE-PART-P

Class COMMENT-PART

Generic Function COMMENT-PART-P

Class HEADING-PART

Method PRINT-OBJECT

Generic Function HEADING-PART-P

Class WHITESPACE-PART

Method PRINT-OBJECT

1.2.2 The publishing engine

1.3 Directives

Directives are a way to control how qbook processes the lisp code. We currently only support the '@include "filename"' directive. @include allows multiple source files to be combined to form a single html file.

Generic Function PROCESS-DIRECTIVE

Method PROCESS-DIRECTIVE

Method PROCESS-DIRECTIVE

1.4 Parsing

A qbook source file is a lisp source file. Qbook uses the lisp's reader to parse the code (so any valid lisp should be usable). qbook looks for a few things in the lisp file:

1) The code. Each top level form is wrapped in ¡PRE¿ tagged as pased through to the HTML. The first line (not form) of the top level form is presented in a bold font. If the form is longer than 3 lines it will be truncated to 3 lines and readers will have to click an the form to see the hidden text.

2) ;;;; Comments - All lines which start with 4 #\; ("ˆ;;;;") and aren't within a top level form are wrapped in a ¡P¿ tag and passed through.

3) ; Comments - All comment lines with less than 4 #\; characters are ignored by qbook.

4) @ directives - Lines which start with ;;;;@ are qbook directives. These allow the developer to control how qbook processes the source files. Currently the only supported directive is include.

A decent example of a qbook'd lisp file is qbook itself. qbook.asd contains the include directives which control the order of the sections while the various .lisp files contain qbook comments, qbook headings and ignored comments (every source file contains a copyright message which we don't want to have repeated multiple times in the html)

1.4.1 qbook markup

There is none. You simply can't create tables or produce links or bold text. Patches welcome.

1.4.2 The source code reader

Function MAKE-PART-READER

Function QBOOK-SEMICOLON-READER

Function MAKE-QBOOK-READTABLE

Function WHITESPACEP

Function READ-WHITESPACE

Function PROCESS-DIRECTIVES

Source code reading consists of the following steps: 1) Read source into parts 2) Post process (merge sequential comments, setup headers, etc.) 3) Handle any directives. 4) Gather any extra source code info 5) Setup navigation elements 6) Remove all the parts before the first comment part

Function READ-SOURCE-FILE - Parse a Lisp source code file into parts

Function HEADING-TEXT-P

Function REAL-COMMENT-P

Function COLLECT-CODE-INFO - Collect specific info for each source code part using ANALYSE-CODE-PART.

Function POST-PROCESS

Function POST-PROCESS-NAVIGATION

2 ASDF Integration

The publish-op generates documentation from the ASDF system definition. The op creates a qbook html file next to the .asd file. The default values for the parameters passed to PUBLISH-QBOOK (input-file, output-file and title) are all taken from the ASDF system. Customizing the defaults is a simple matter of passing the proper keywords to asdf:oos.

Class PUBLISH-OP

Method INPUT-FILES

Method PERFORM

Method PERFORM

Method OPERATION-DONE-P

Function PUBLISH-SYSTEM-QBOOK

3 Extra Code Analysis

In the extra code analysis phase each parsed code part is enriched with a descriptor that contains specific information of the code part depending on its type (function, class, method, etc).

Variable *CODE-INFO-COLLECTORS*

Variable *KNOWN-ELEMENTS*

Function REGISTER-DESCRIPTOR

Function FIND-DESCRIPTOR

Function ANALYSE-CODE-PART - Match an info collection from *CODE-INFO-COLLECTORS* and evaluate it to fill up the code part descriptor.

Macro DEFCODE-INFO-COLLECTOR - Macro for defining code parts info collectors.

Class DESCRIPTOR

Function SUBSEQ-FIRST-SENTENCE

Generic Function DOCSTRING-FIRST-SENTENCE - Returns the first sentence of DESCRIPTOR's docstring.

3.0.1 Info collectors

Define an info collector for each type of code part (function, method, defclass, macro, etc)

Functions info collector

Class DEFUN-DESCRIPTOR

Code Info Collector DEFUN

Macros info collector

Class DEFMACRO-DESCRIPTOR

Code Info Collector DEFMACRO

Classes info collector

Class DEFCLASS-DESCRIPTOR

Code Info Collector DEFCLASS

Slots info collector

Class CLASS-SLOT-DESCRIPTOR

Function MAKE-SLOT-DESCRIPTOR

Global variables info collector

Class GLOBAL-VARIABLE-DESCRIPTOR

Code Info Collector DEFVAR

Code Info Collector DEFPARAMETER

Generic functions info collector

Class DEFGENERIC-DESCRIPTOR

Code Info Collector DEFGENERIC

Class DEFMETHOD-DESCRIPTOR

Code Info Collector DEFMETHOD

Constants info collector

Class DEFCONSTANT-DESCRIPTOR

Code Info Collector DEFCONSTANT

3.1 Example of a custom code part definition

This is an example of how to define a custom code-part

```
(in-package :qbook)
```

Define a custom descriptor

Class INFO-COLLECTOR-DESCRIPTOR

Define an code info collector that instantiate the custom descriptor

Code Info Collector DEFCODE-INFO-COLLECTOR

Finally, possibly implement custom HTML and Latex generation, specializing WRITE-CODE-DESCRIPTOR generic function.

4 The HTML Generator

Class HTML-GENERATOR

Variable *GENERATOR*

Variable *BOOK*

Method GENERATE

```
(eval-when (:compile-toplevel :load-toplevel :execute)
  (yaclml:deftag-macro <qbook-page (&attribute title file-name (stylesheet "style.css")
                                              (printsheet "print.css")
                                              &body body)
    `(with-output-to-file (*yaclml-stream*
                           (ensure-directories-exist (merge-pathnames ,file-name (output-directory *gene
                           :if-exists :supersede
                           :if-does-not-exist :create)
       (<:html
        (<:head
         (<:title (<:as-html ,title))
         (<:meta :http-equiv "Content-Type" :content "text/html; charset=utf-8")
         (<:stylesheet ,stylesheet)
         (<:link :rel "alternate stylesheet" :href ,printsheet :title "Print")
         (when (highlight-syntax *generator*)
           (<:link :rel "stylesheet"
                   :href "https://cdn.jsdelivr.net/gh/highlightjs/cdn-release@10.0.3/build/styles/defaul
        (<:body
         (<:div :class "qbook" ,@body)
         (when (highlight-syntax *generator*)
           (<:script :src "https://cdn.jsdelivr.net/gh/highlightjs/cdn-release@10.0.3/build/highlight.mi
           (<:script :src "https://cdn.jsdelivr.net/gh/highlightjs/cdn-release@10.0.3/build/languages/l
           (<:script "hljs.initHighlightingOnLoad();"))
         )))))
```

Function GENERATE-TABLE-OF-CONTENTS

Function GENERATE-INDEX

Function GENERATE-PERMUTED-INDEX

Function GENERATE-SECTION

Method MAKE-ANCHOR-LINK

Method MAKE-ANCHOR-LINK

Method MAKE-ANCHOR-NAME

Function EFFECTIVE-NAME

Method MAKE-ANCHOR-NAME

Method MAKE-ANCHOR-NAME

Method HTML-NAME

Method HTML-NAME

Function PUBLISH

Function NUM-LINES

Function WRITE-CODE

Generic Function WRITE-CODE-DESCRIPTOR - Writes the documentation of PART using DESCRIPTOR for the current GENERATOR

Method WRITE-CODE-DESCRIPTOR

Method WRITE-CODE-DESCRIPTOR

Method WRITE-CODE-DESCRIPTOR

Method WRITE-CODE-DESCRIPTOR

4.1 Writing Comments

Function WRITE-COMMENT

4.2 Standard HTML stylesheets

We include these as variables so that whene generating an html doc set we needn't know the location of qbook's source code.

Variable *PRINT.CSS* - The alternative (destined for hard copy) HTML stylesheet.

Variable *STYLE.CSS* - The default stylesheet for qbook generated html documentation.

5 The LaTeX Generator

Class LATEX-GENERATOR

Variable *LATEX-STREAM*

Function LATEX-COMMAND

Generic Function GENERATE-PART

Method GENERATE

Function SAFE-LATEX-ID

Function DESCRIPTOR-REF-ID

Function DESCRIPTOR-LINK-ID

Method GENERATE-PART - Generate link to the code

Function WRITE-SOURCE

Method GENERATE-PART

Method GENERATE-PART

Method GENERATE-PART

Function WRITE-LATEX-ESCAPED

Generic Function GENERATE-PART-REFERENCE

Method GENERATE-PART-REFERENCE

Method GENERATE-PART-REFERENCE

Method GENERATE-PART-REFERENCE

Method WRITE-CODE-DESCRIPTOR

Method WRITE-CODE-DESCRIPTOR

6 Reference

Class: GENERATOR

Slots

- TITLE
- PRINT-CASE

Hierarchy

Precedence list

- STANDARD-OBJECT

Sub Classes

- LATEX-GENERATOR
- HTML-GENERATOR

```
(defclass generator ()
  ((title :accessor title :initarg :title)
   (print-case :initarg :print-case
               :accessor print-case
               :initform *print-case*)))
```

[Source Context]

Generic Function: GENERATE

```
(defgeneric generate (book generator))
```

[Source Context]

Method: GENERATE

```
(defmethod generate :around (book generator)
  (let ((*print-case* (print-case generator)))
    (call-next-method)))
```

[Source Context]

Class: BOOK

Slots

- TITLE - The title of the book.
- CONTENTS
- INDEXES

Hierarchy

Precedence list

- STANDARD-OBJECT

```
(defclass book ()
  ((title :accessor title :initarg :title
          :documentation "The title of the book.")
   (contents :accessor contents :initarg :contents)
   (indexes :accessor indexes :initarg :indexes :initform (make-hash-table :test 'eq))))
```

[Source Context]

Method: BOOK-INDEXES-SORTED

```
(defmethod book-indexes-sorted ((book book))
  (sort (hash-table-keys (indexes book))
        #'string<
        :key (lambda (descriptor-class)
               (pretty-label-prefix (make-instance descriptor-class)))))
```

[Source Context]

Method: ALL-CODE-PARTS

```
(defmethod all-code-parts ((book book))
  (loop
    for section in (contents book)
    append (remove-if-not #'code-part-p section)))
```

[Source Context]

Method: PERMUTATED-GLOBAL-INDEX

```
(defmethod permutated-global-index ((book book))
  (let ((entries '()))
    (dolist (part (all-code-parts book))
      (when (descriptor part)
        (let ((name-string (symbol-name (effective-name (name (descriptor part))))))
          (dolist (name-part (remove-duplicates
                              (remove-if (lambda (string)
                                           (string= "" string))
                                         (cl-ppcre:split "[^a-zA-Z]" name-string))
                              :test #'string=))
            (let* ((offset (search name-part name-string :test #'char=))
                   (prefix (subseq name-string 0 offset))
                   (suffix (subseq name-string offset)))
              (if (symbolp (name (descriptor part)))
                  (push (list prefix suffix part)
                        entries)
                  (push (list (format nil "(~A ~A"
                                      (first (name (descriptor part)))
                                      prefix)
                              (format nil "~A)" suffix)
                              part)
                        entries)))))))
    (sort entries #'string< :key #'second)))
```

[Source Context]

Function: COMPARE-DESCRIPTOR-NAMES

```
(defun compare-descriptor-names (a b)
  (string< (regex-replace-all "[^A-Za-z]" (string (if (symbolp (name a))
                                                      (name a)
                                                      (second (name a)))) "")
           (regex-replace-all "[^A-Za-z]" (string (if (symbolp (name b))
                                                      (name b)
                                                      (second (name b)))) ""))))
```

[Source Context]

Function: SORT-DESCRIPTORS

```
(defun sort-descriptors (descriptors)
  (sort (copy-list descriptors) 'compare-descriptor-names))
```

[Source Context]

Function: SORT-PARTS-WITH-DESCRIPTORS

```
(defun sort-parts-with-descriptors (parts)
  (sort (remove-if #'null parts :key #'descriptor)
        'compare-descriptor-names :key #'descriptor))
```

[Source Context]

Function: CONVERT-TO-SECTIONS

```
(defun convert-to-sections (file-name parts)
  (let ((sections '()))
    (iterate
      (for p in parts)
      (if (and (heading-part-p p)
               (= 1 (depth p)))
          (push (list p) sections)
          (if (consp sections)
              (push p (car sections))
              (error "No initial heading in ~S." file-name))))
    (iterate
      (for section on sections)
      (setf (car section) (nreverse (car section))))
    (nreverse sections)))
```

[Source Context]

Function: BUILD-INDEXES

```
(defun build-indexes (book)
  (loop
    for section in (contents book)
    do (loop
         for part in section
         when (code-part-p part)
           do (when (descriptor part)
                (symbol-macrolet ((index-table
                                    (gethash (class-of (descriptor part)) (indexes book))))
                  (unless index-table
                    (setf index-table (make-hash-table :test 'eql)))
                  (setf (gethash (name (descriptor part)) index-table) part)))))
  book)
```

[Source Context]

Function: PUBLISH-QBOOK

Convert FILE-NAME into a qbook html file named OUTPUT-FILE with title TITLE.

```
(defun publish-qbook (file-name generator)
  "Convert FILE-NAME into a qbook html file named OUTPUT-FILE with title TITLE."
  (let ((book (make-instance 'book
                             :title (title generator)
                             :contents (convert-to-sections
                                        file-name
                                        (read-source-file file-name)))))
    (build-indexes book)
    (generate book generator)))
```

[Source Context]

Class: SOURCE-FILE-PART

A part of a source file. Can be code, comment, heading, etc..

Slots

- START-POSITION
- END-POSITION
- TEXT
- ORIGIN-FILE
- OUTPUT-FILE

Hierarchy

Precedence list

- STANDARD-OBJECT

Sub Classes

- WHITESPACE-PART
- COMMENT-PART
- CODE-PART

```
(defclass source-file-part ()
  ((start-position :accessor start-position :initform nil :initarg :start-position)
   (end-position :accessor end-position :initform nil :initarg :end-position)
   (text :accessor text :initform nil :initarg :text)
   (origin-file :accessor origin-file :initform nil :initarg :origin-file)
   (output-file :accessor output-file :initform nil))
  (:documentation "A part of a source file.
Can be code, comment, heading, etc.."))
```

[Source Context]

Method: PRINT-OBJECT

```
(defmethod print-object ((part source-file-part) stream)
  (print-unreadable-object (part stream :type t :identity t)
    (format stream "~S" (if (< (length (text part)) 10)
                            (text part)
                            (strcat (subseq (text part) 0 10) "...")))))
```

[Source Context]

Class: CODE-PART

Slots

- FORM
- DESCRIPTOR

Hierarchy

Precedence list

- SOURCE-FILE-PART

```
(defclass code-part (source-file-part)
  ((form :accessor form :initform nil :initarg :form)
   (descriptor :accessor descriptor :initform nil :initarg :descriptor))
  (:documentation ""))
```

[Source Context]

Generic Function: CODE-PART-P

```
(defgeneric code-part-p (object)
  (:method ((object t)) nil)
  (:method ((object code-part)) t))
```

[Source Context]

Class: COMMENT-PART

Hierarchy

Precedence list

- SOURCE-FILE-PART

Sub Classes

- HEADING-PART

```
(defclass comment-part (source-file-part)
  ())
```

[Source Context]

Generic Function: COMMENT-PART-P

```
(defgeneric comment-part-p (obj)
  (:method ((obj t)) nil)
  (:method ((obj comment-part)) t))
```

[Source Context]

Class: HEADING-PART

Slots

- DEPTH
- NEXT-PART
- PREV-PART
- UP-PART

Hierarchy

Precedence list

- COMMENT-PART

```
(defclass heading-part (comment-part)
  ((depth :accessor depth :initarg :depth)
   (next-part :accessor next-part :initform nil)
   (prev-part :accessor prev-part :initform nil)
   (up-part :accessor up-part :initform nil)))
```

[Source Context]

Method: PRINT-OBJECT

```
(defmethod print-object ((h heading-part) stream)
  (print-unreadable-object (h stream :type t :identity nil)
    (format stream "~D ~S" (depth h) (text h))))
```

[Source Context]

Generic Function: HEADING-PART-P

```
(defgeneric heading-part-p (obj)
  (:method ((obj t)) nil)
  (:method ((obj heading-part)) t))
```

[Source Context]

Class: WHITESPACE-PART

Hierarchy

Precedence list

- SOURCE-FILE-PART

```
(defclass whitespace-part (source-file-part)
  ())
```

[Source Context]

Method: PRINT-OBJECT

```
(defmethod print-object ((part whitespace-part) stream)
  (print-unreadable-object (part stream :type t :identity t)))
```

[Source Context]

Generic Function: PROCESS-DIRECTIVE

```
(defgeneric process-directive (part))
```

[Source Context]

Method: PROCESS-DIRECTIVE

```
(defmethod process-directive ((part source-file-part))
  (list part))
```

[Source Context]

Method: PROCESS-DIRECTIVE

```
(defmethod process-directive ((part comment-part))
  (declare (special *source-file*))
  (multiple-value-bind (matchp strings)
      (cl-ppcre:scan-to-strings (load-time-value (cl-ppcre:create-scanner "^@include (.*)"))
                                (text part))
    (if matchp
        (return-from process-directive
          (read-source-file
           (merge-pathnames (let ((*readtable* (copy-readtable nil)))
                              (read-from-string (aref strings 0)))
                            (truename *source-file*))))
        (return-from process-directive (list part)))))
```

[Source Context]

Function: MAKE-PART-READER

```
(defun make-part-reader (function type)
  (lambda (stream echar)
    (let ((part (make-instance type)))
      (setf (start-position part) (file-position stream))
      (funcall function stream echar)
      (setf (end-position part) (file-position stream))
      part)))
```

[Source Context]

Function: QBOOK-SEMICOLON-READER

```
(defun qbook-semicolon-reader (stream char)
  (declare (ignore char))
  (with-output-to-string (line)
    (loop
      for next-char = (read-char stream nil stream t)
      if (or (eq next-char stream)
             (char= next-char #\Newline))
        do (return)
      else
        do (write-char next-char line))))
```

[Source Context]

Function: MAKE-QBOOK-READTABLE

```
(defun make-qbook-readtable ()
  (let ((r (copy-readtable nil)))
    (multiple-value-bind (function non-terminating-p)
        (get-macro-character #\( *readtable*)
      (set-macro-character #\( (make-part-reader function 'code-part) non-terminating-p r))
    (multiple-value-bind (function non-terminating-p)
        (get-macro-character #\; *readtable*)
      (set-macro-character #\; (make-part-reader 'qbook-semicolon-reader 'comment-part) non-terminating-
    r))
```

[Source Context]

Function: WHITESPACEP

```
(defun whitespacep (char)
  (and char
       (member char '(#\Space #\Tab #\Newline) :test #'char=)))
```

[Source Context]

Function: READ-WHITESPACE

```
(defun read-whitespace (stream)
  (iterate
    (with part = (make-instance 'whitespace-part))
    (initially (setf (start-position part) (1+ (file-position stream))))
    (while (whitespacep (peek-char nil stream nil nil)))
    (read-char stream)
    (finally (setf (end-position part) (file-position stream)))
    (finally (return-from read-whitespace part))))
```

[Source Context]

Function: PROCESS-DIRECTIVES

```
(defun process-directives (parts)
  (iterate
    (for part in parts)
    (appending (process-directive part))))
```

[Source Context]

Function: READ-SOURCE-FILE

Parse a Lisp source code file into parts

```
(defun read-source-file (file-name)
  "Parse a Lisp source code file into parts"
  (let ((*evaling-readtable* (copy-readtable nil))
        (*evaling-package* (find-package :common-lisp-user)))
    (flet ((eval-part (part)
             (etypecase part
               (code-part
                (let* ((*readtable* *evaling-readtable*)
                       (*package* *evaling-package*)
                       (*load-pathname* (pathname file-name))
                       (*load-truename* (truename *load-pathname*)))
                  (ignore-errors
                   (setf (form part) (read-from-string (text part) nil))
                   (eval (form part)))
                  (setf *evaling-readtable* *readtable*)
                  (setf *evaling-package* *package*)))
               (t part))))
      (let* ((*readtable* (make-qbook-readtable))
             (*source-file* file-name)
             (parts (with-input-from-file (stream file-name)
                      (iterate
                        (for part in-stream stream using #'read-preserving-whitespace)
                        (collect part)
                        (when (whitespacep (peek-char nil stream nil nil))
                          (collect (read-whitespace stream)))))))
        (declare (special *source-file*))
        (with-input-from-file (stream file-name)
          (let ((buffer nil))
            (dolist (part parts)
              (file-position stream (1- (start-position part)))
              (setf buffer (make-array (1+ (- (end-position part) (start-position part)))
                                       :element-type 'character))
              (read-sequence buffer stream)
              (setf (text part) buffer
                    (origin-file part) file-name)
              (eval-part part))))
        ;; step 1: post process (merge sequential comments, setup headers, etc.)
        (setf parts (post-process parts))
        ;; step 2: handle any directives.
        (setf parts (process-directives parts))
        ;; step 3: gather any extra source code info
        (setf parts (collect-code-info parts))
        ;; step 4: setup navigation elements
        (setf parts (post-process-navigation parts))
        ;; step 5: remove all the parts before the first comment part
        (setf parts (iterate
                      (for p on parts)
                      (until (comment-part-p (first p)))
                      (finally (return p))))
        ;; done!
        parts))))
```

[Source Context]

Function: HEADING-TEXT-P

```
(defun heading-text-p (text)
  (scan "^;;;;\\s*\\*+" text))
```

[Source Context]

Function: REAL-COMMENT-P

```
(defun real-comment-p (text)
  (scan "^;;;;" text))
```

[Source Context]

Function: COLLECT-CODE-INFO

Collect specific info for each source code part using ANALYSE-CODE-PART.

```
(defun collect-code-info (parts)
  "Collect specific info for each source code part using ANALYSE-CODE-PART."
  (mapcar (lambda (part)
            (typecase part
              (code-part
               ;; punt all the work to collect-code-info
               (analyse-code-part part))
              (t part)))
          parts))
```

[Source Context]

Function: POST-PROCESS

```lisp
(defun post-process (parts)
  ;; convert all the comments which are acutally headings to heading
  ;; objects
  (setf parts
        (iterate
          (for p in parts)
          (typecase p
            (comment-part
             (multiple-value-bind (match strings)
                 (scan-to-strings (load-time-value
                                   (create-scanner ";;;;\\s*(\\*+)\\s*(.*)" :single-line-mode nil))
                                  (text p))
               (if match
                   (collect (make-instance 'heading-part
                                           :depth (length (aref strings 0))
                                           :text (aref strings 1)
                                           :start-position (start-position p)
                                           :end-position (end-position p)
                                           :origin-file (origin-file p)))
                   (multiple-value-bind (match strings)
                       (scan-to-strings (load-time-value
                                         (create-scanner ";;;;(.*)" :single-line-mode t))
                                        (text p))
                     (if match
                         (collect (make-instance 'comment-part
                                                 :start-position (start-position p)
                                                 :end-position (end-position p)
                                                 :text (aref strings 0)
                                                 :origin-file (origin-file p))))))))
            ((or code-part whitespace-part) (collect p)))))
  ;;;; merge consequtive comments together
  (setf parts
        (iterate
          (with comment = (make-string-output-stream))
          (for (p next) on parts)
          (cond
            ((heading-part-p p) (collect p))
            ((and (comment-part-p p)
                  (or (not (comment-part-p next))
                      (heading-part-p next)
                      (null next)))
             (write-string (text p) comment)
             (collect (make-instance 'comment-part :text (get-output-stream-string comment)))
             (setf comment (make-string-output-stream)))
            ((comment-part-p p)
             (write-string (text p) comment))
            (t (collect p)))))
  parts)
```

[Source Context]

Function: POST-PROCESS-NAVIGATION

```
(defun post-process-navigation (parts)
    ;;;; setup the prev and next links in the header objects
  (iterate
    (with last-heading = nil)
    (for part in parts)
    (when (heading-part-p part)
      (when last-heading
        (setf (prev-part part) last-heading
              (next-part last-heading) part))
      (setf last-heading part)))
  ;;;; setup the up links
  (iterate
    (for (this . rest) on (remove-if-not #'heading-part-p parts))
    (iterate
      (for r in rest)
      (while (< (depth this) (depth r)))
      (setf (up-part r) this)))
  parts)
```

[Source Context]

Class: PUBLISH-OP

Slots

- GENERATOR
- INPUT-FILE

Hierarchy

Precedence list

- OPERATION

```
(defclass publish-op (asdf:operation)
  ((generator :initarg :generator :accessor generator)
   (input-file :initform nil :initarg :input-file :accessor input-file)))
```

[Source Context]

Method: INPUT-FILES

```
(defmethod input-files ((op publish-op) (system asdf:system))
  (let ((x (or (input-file op) (asdf:system-source-file system))))
    (and x (list x))))
```

[Source Context]

Method: PERFORM

```
(defmethod asdf:perform ((op publish-op) (system asdf:system))
  (publish-qbook (first (input-files op system)) (generator op)))
```

[Source Context]

Method: PERFORM

```
(defmethod asdf:perform ((op publish-op) (component t))
  t)
```

[Source Context]

Method: OPERATION-DONE-P

```
(defmethod asdf:operation-done-p ((op publish-op) (component t))
  nil)
```

[Source Context]

Function: PUBLISH-SYSTEM-QBOOK

```
(defun publish-system-qbook (system-name generator-name &rest options)
  (let ((generator (apply #'make-instance generator-name
                           options)))
    (publish-qbook (asdf:system-source-file system-name) generator)))
```

[Source Context]

Variable: *CODE-INFO-COLLECTORS*

```
(defvar *code-info-collectors* (make-hash-table))
```

[Source Context]

Variable: *KNOWN-ELEMENTS*

```
(defvar *known-elements* (make-hash-table :test #'equal))
```

[Source Context]

Function: REGISTER-DESCRIPTOR

```
(defun register-descriptor (descriptor)
  (push (cons (name descriptor) descriptor)
        (gethash (label-prefix descriptor) *known-elements*)))
```

[Source Context]

Function: FIND-DESCRIPTOR

```
(defun find-descriptor (label name)
  (when-bind elements-of-type (gethash label *known-elements*)
    (cdr (assoc name elements-of-type
                :test (lambda (a b)
                        (if (symbolp a)
                            (eq a b)
                            (and (eq (first a) (first b))
                                 (eq (second a) (second b)))))))))
```

[Source Context]

Function: ANALYSE-CODE-PART

Match an info collection from *CODE-INFO-COLLECTORS* and evaluate it to fill up the code part descriptor.

```
(defun analyse-code-part (code-part)
  "Match an info collection from *CODE-INFO-COLLECTORS* and evaluate it to fill up the code part descrip
  (awhen (gethash (first (form code-part)) *code-info-collectors*)
    (setf (descriptor code-part) (funcall it (cdr (form code-part))))
    (register-descriptor (descriptor code-part)))
  code-part)
```

[Source Context]

Macro: DEFCODE-INFO-COLLECTOR

Macro for defining code parts info collectors. You can use it to generate descriptors for your custom macros.

```
(defmacro defcode-info-collector (operator args &body body)
  "Macro for defining code parts info collectors.
You can use it to generate descriptors for your custom macros."
  (with-unique-names (form)
    (let ((function-name (intern (strcat operator :-descriptor) (find-package :it.bese.qbook))))
      `(progn
         (defun ,function-name (,form)
           (destructuring-bind ,args ,form
             ,@body))
         (setf (gethash ',operator *code-info-collectors*)
               ',function-name)))))
```

[Source Context]

Class: DESCRIPTOR

Slots

- NAME
- DOCSTRING

- LABEL-PREFIX
- PRETTY-LABEL-PREFIX

Hierarchy

Precedence list

- STANDARD-OBJECT

Sub Classes

- GLOBAL-VARIABLE-DESCRIPTOR
- CLASS-SLOT-DESCRIPTOR
- DEFCLASS-DESCRIPTOR
- DEFUN-DESCRIPTOR

```
(defclass descriptor ()
  ((name :accessor name :initarg :name)
   (docstring :accessor docstring :initarg :docstring)
   (label-prefix :accessor label-prefix :initarg :label-prefix)
   (pretty-label-prefix :accessor pretty-label-prefix :initarg :pretty-label-prefix)))
```

[Source Context]

Function: SUBSEQ-FIRST-SENTENCE

```
(defun subseq-first-sentence (string limit)
  (with-output-to-string (first-sentence)
    (flet ((ret ()
             (return-from subseq-first-sentence
               (get-output-stream-string first-sentence))))
      (loop
        for char across string
        for count below limit
        if (member char (list #\. #\? #\!))
          do (write-char char first-sentence)
          and do (ret)
        else
          do (write-char char first-sentence)
        finally (ret)))))
```

[Source Context]

Generic Function: DOCSTRING-FIRST-SENTENCE

Returns the first sentence of DESCRIPTOR's docstring. Returns at most LIMIT characters (if the first sentence is longer than LIMIT characters it will be simply truncated. If DESCRIPTOR's docstring is NIL this function returns nil.

```
(defgeneric docstring-first-sentence (descriptor &optional limit)
  (:documentation "Returns the first sentence of DESCRIPTOR's
docstring. Returns at most LIMIT characters (if the first
sentence is longer than LIMIT characters it will be simply
truncated. If DESCRIPTOR's docstring is NIL this function
returns nil.")
  (:method ((descriptor descriptor) &optional (limit 180))
    (when (not (null (docstring descriptor)))
      (subseq-first-sentence (docstring descriptor) limit))))
```

[Source Context]

Class: DEFUN-DESCRIPTOR

Slots

- LAMBDA-LIST
- BODY

Hierarchy

Precedence list

- DESCRIPTOR

Sub Classes

- INFO-COLLECTOR-DESCRIPTOR
- DEFMETHOD-DESCRIPTOR
- DEFGENERIC-DESCRIPTOR
- DEFMACRO-DESCRIPTOR

```
(defclass defun-descriptor (descriptor)
  ((lambda-list :accessor lambda-list :initarg :lambda-list)
   (body :accessor body :initarg :body))
  (:default-initargs
   :label-prefix "function"
   :pretty-label-prefix "Function"))
```

[Source Context]

Code Info Collector: DEFUN

```
(defcode-info-collector cl:defun (name lambda-list &body body)
  (multiple-value-bind (lambda-list env)
      (arnesi::walk-lambda-list lambda-list nil nil)
    (multiple-value-bind (body docstring declarations)
        (handler-bind ((arnesi::return-from-unknown-block
                         (lambda (c)
                           (declare (ignore c))
                           (invoke-restart 'arnesi::add-block))))
          (arnesi::walk-implict-progn nil body env :docstring t :declare t))
      (declare (ignore declarations))
      (make-instance 'defun-descriptor
                     :name name
                     :lambda-list lambda-list
                     :body body
                     :docstring docstring))))
```

[Source Context]

Class: DEFMACRO-DESCRIPTOR

Hierarchy

Precedence list

- DEFUN-DESCRIPTOR

```
(defclass defmacro-descriptor (defun-descriptor)
  ()
  (:default-initargs
   :label-prefix "macro"
   :pretty-label-prefix "Macro"))
```

[Source Context]

Code Info Collector: DEFMACRO

```
(defcode-info-collector cl:defmacro (name lambda-list &body body)
  (multiple-value-bind (lambda-list env)
      (arnesi::walk-lambda-list lambda-list nil nil)
    (multiple-value-bind (body docstring declarations)
        (handler-bind ((arnesi::return-from-unknown-block
                         (lambda (c)
                           (declare (ignore c))
                           (invoke-restart 'arnesi::add-block))))
          (arnesi::walk-implict-progn nil body env :docstring t :declare t))
      (declare (ignore declarations))
      (make-instance 'defmacro-descriptor
                     :name name
                     :lambda-list lambda-list
                     :body body
                     :docstring docstring))))
```

[Source Context]

Class: DEFCLASS-DESCRIPTOR

Slots

- SLOTS
- SUPERS

Hierarchy

Precedence list

- DESCRIPTOR

```
(defclass defclass-descriptor (descriptor)
  ((slots :accessor slots :initarg :slots :initform '())
   (supers :accessor supers :initarg :supers :initform '()))
  (:default-initargs
   :label-prefix "class"
   :pretty-label-prefix "Class"))
```

[Source Context]

Code Info Collector: DEFCLASS

```
(defcode-info-collector cl:defclass (name supers slots &rest options)
  (make-instance 'defclass-descriptor
                 :name name
                 :supers supers
                 :slots (mapcar #'make-slot-descriptor slots)
                 :docstring (second (assoc :documentation options))))
```

[Source Context]

Class: CLASS-SLOT-DESCRIPTOR

Hierarchy

Precedence list

- DESCRIPTOR

```
(defclass class-slot-descriptor (descriptor)
  ())
```

[Source Context]

Function: MAKE-SLOT-DESCRIPTOR

```
(defun make-slot-descriptor (slot-spec)
  (destructuring-bind (name &rest options)
      (ensure-list slot-spec)
    (make-instance 'class-slot-descriptor
                   :name name
                   :docstring (getf options :documentation))))
```

[Source Context]

Class: GLOBAL-VARIABLE-DESCRIPTOR

Hierarchy

Precedence list

- DESCRIPTOR

Sub Classes

- DEFCONSTANT-DESCRIPTOR

```
(defclass global-variable-descriptor (descriptor)
  ()
  (:default-initargs
   :label-prefix "variable"
   :pretty-label-prefix "Variable"))
```

[Source Context]

Code Info Collector: DEFVAR

```
(defcode-info-collector cl:defvar (name &optional value documentation)
  (declare (ignore value))
  (make-instance 'global-variable-descriptor
                 :name name
                 :docstring documentation))
```

[Source Context]

Code Info Collector: DEFPARAMETER

```
(defcode-info-collector cl:defparameter (name &optional value documentation)
  (declare (ignore value))
  (make-instance 'global-variable-descriptor
                 :name name
                 :docstring documentation))
```

[Source Context]

Class: DEFGENERIC-DESCRIPTOR

Hierarchy

Precedence list

- DEFUN-DESCRIPTOR

```
(defclass defgeneric-descriptor (defun-descriptor)
  ()
  (:default-initargs
   :label-prefix "generic function"
   :pretty-label-prefix "Generic Function"))
```

[Source Context]

Code Info Collector: DEFGENERIC

```
(defcode-info-collector cl:defgeneric (name arg-list &rest options)
  (multiple-value-bind (lambda-list env)
      (arnesi::walk-lambda-list arg-list nil nil :allow-specializers nil)
    (declare (ignore env))
    (make-instance 'defgeneric-descriptor
                   :name name
                   :lambda-list lambda-list
                   :docstring (second (assoc :documentation options)))))
```

[Source Context]

Class: DEFMETHOD-DESCRIPTOR

Slots

- QUALIFIER

Hierarchy

Precedence list

- DEFUN-DESCRIPTOR

```
(defclass defmethod-descriptor (defun-descriptor)
  ((qualifier :accessor qualifier :initform nil :initarg :qualifier))
  (:default-initargs
   :label-prefix "method"
   :pretty-label-prefix "Method"))
```

[Source Context]

Code Info Collector: DEFMETHOD

```
(defcode-info-collector cl:defmethod (name &rest args)
  (let ((qualifier nil)
        arguments
        body)
    (when (symbolp (first args))
      (setf qualifier (pop args)))
    (setf arguments (pop args)
          body args)
    (multiple-value-bind (lambda-list env)
        (arnesi::walk-lambda-list arguments nil nil :allow-specializers t)
      (multiple-value-bind (body docstring declarations)
          (handler-bind ((arnesi::return-from-unknown-block
                           (lambda (c)
                             (declare (ignore c))
                             (invoke-restart 'arnesi::add-block))))
            (arnesi::walk-implict-progn nil body env :docstring t :declare t))
        (declare (ignore declarations))
        (make-instance 'defmethod-descriptor
                       :name name
                       :qualifier qualifier
                       :lambda-list lambda-list
                       :body body
                       :docstring docstring)))))
```

[Source Context]

Class: DEFCONSTANT-DESCRIPTOR

Hierarchy

Precedence list

- GLOBAL-VARIABLE-DESCRIPTOR

```
(defclass defconstant-descriptor (global-variable-descriptor)
  ()
  (:default-initargs
   :label-prefix "constant"
   :pretty-label-prefix "Constant"))
```

[Source Context]

Code Info Collector: DEFCONSTANT

```
(defcode-info-collector cl:defconstant (name value &optional docstring)
  (declare (ignore value))
  (make-instance 'defconstant-descriptor
                 :name name :docstring docstring))
```

[Source Context]

Class: INFO-COLLECTOR-DESCRIPTOR

Hierarchy

Precedence list

- DEFUN-DESCRIPTOR

```
(defclass info-collector-descriptor (defun-descriptor)
  ()
  (:default-initargs
   :label-prefix "code-info-collector"
   :pretty-label-prefix "Code Info Collector"))
```

[Source Context]

Code Info Collector: DEFCODE-INFO-COLLECTOR

```
(defcode-info-collector defcode-info-collector (name lambda-list &body body)
  (multiple-value-bind (lambda-list env)
      (arnesi::walk-lambda-list lambda-list nil nil)
    (multiple-value-bind (body docstring declarations)
        (handler-bind ((arnesi::return-from-unknown-block
                         (lambda (c)
                           (declare (ignore c))
                           (invoke-restart 'arnesi::add-block))))
          (arnesi::walk-implict-progn nil body env :docstring t :declare t))
      (declare (ignore declarations))
      (make-instance 'info-collector-descriptor
                     :name name
                     :lambda-list lambda-list
                     :body body
                     :docstring docstring))))
```

[Source Context]

Class: HTML-GENERATOR

Slots

- ESCAPE-COMMENTS - If T, escape comments HTML. If NIL, output the comment as it is (useful for embedding HTML in code comments).
- HIGHLIGHT-SYNTAX - When T, highlight syntax using highlight.js library
- OUTPUT-DIRECTORY

Hierarchy

Precedence list

- GENERATOR

```
(defclass html-generator (generator)
  ((escape-comments :initarg :escape-comments
                    :accessor escape-comments
                    :type boolean
                    :initform t
                    :documentation "If T, escape comments HTML. If NIL, output the comment as it is (use
   (highlight-syntax :initarg :highlight-syntax
                     :accessor highlight-syntax
                     :type boolean
                     :initform nil
                     :documentation "When T, highlight syntax using highlight.js library")
   (output-directory :initarg :output-directory :accessor output-directory)))
```

[Source Context]

Variable: *GENERATOR*

```
(defvar *generator*)
```

[Source Context]

Variable: *BOOK*

```
(defvar *book*)
```

[Source Context]

Method: GENERATE

```
(defmethod generate (book (generator html-generator))
  (let ((*generator* generator)
        (*book* book))
    (let ((output-dir-truename (ensure-directories-exist
                                 (merge-pathnames (output-directory generator)))))
      (write-string-to-file *print.css* (make-pathname :name "print" :type "css"
                                                       :defaults output-dir-truename)
                            :if-does-not-exist :create
                            :if-exists :supersede)
      (write-string-to-file *style.css* (make-pathname :name "style" :type "css"
                                                       :defaults output-dir-truename)
                            :if-does-not-exist :create
                            :if-exists :supersede))
    (generate-table-of-contents (contents book) generator)
    (dolist (section (contents book))
      (generate-section section generator))
    (dolist (index-class (book-indexes-sorted book))
      (generate-index generator book index-class))
    (generate-permuted-index generator book)))
```

[Source Context]

Function: GENERATE-TABLE-OF-CONTENTS

```
(defun generate-table-of-contents (sections generator)
  (<qbook-page :title (title generator)
               :file-name "index.html"
               (<:div :class "contents"
                      (<:h1 :class "title" (<:as-html (title generator)))
                      (<:h2 "Table of Contents")
                      (dolist (section sections)
                        (dolist (part section)
                          (when (heading-part-p part)
                            (<:div :class (strcat "contents-heading-" (depth part))
                                   (<:a :href (make-anchor-link part)
                                        (<:as-html (text part)))))))
                      (<:h2 "Indexes")
                      (dolist (index (book-indexes-sorted *book*))
                        (<:div :class "contents-heading-1"
                               (<:a :href (strcat "index/" (label-prefix (make-instance index)) ".html")
                                    (<:as-html (pretty-label-prefix (make-instance index)))
                                    " Index")))
                      (<:div :class "contents-heading-1"
                             (<:a :href "index/permutated.html" "Permuted Symbol Index")))))
```

[Source Context]

Function: GENERATE-INDEX

```
(defun generate-index (generator book index-class)
  (declare (ignore generator))
  (<qbook-page :title (strcat (pretty-label-prefix (make-instance index-class))
                              " Index")
               :file-name (strcat "index/" (label-prefix (make-instance index-class)) ".html")
               :stylesheet "../style.css"
               :printsheet "../print.css"
               (<:div :class "api-index"
                      (<:h1 (<:as-html (strcat (pretty-label-prefix (make-instance index-class))
                                               " Index")))
                      (<:div :class "contents"
                             (<:dl
                              (dolist (part (sort-parts-with-descriptors (hash-table-values (gethash inc
                                (<:dt (<:a :href (strcat "../" (make-anchor-link (descriptor part)))
                                           (<:as-html (name (descriptor part)))))
                                (when (docstring (descriptor part))
                                  (<:dd (<:as-html (docstring-first-sentence (descriptor part))))))))))
  t)
```

[Source Context]

Function: GENERATE-PERMUTED-INDEX

```
(defun generate-permuted-index (generator book)
  (declare (ignore generator))
  (<qbook-page :title "Permuted Index"
               :file-name "index/permutated.html"
               :stylesheet "../style.css"
               :printsheet "../print.css"
               (<:div :class "api-index"
                      (<:h1 (<:as-html "Permuted Index"))
                      (<:div :class "contents"
                             (<:table :class "permuted-index-table"
                                      (dolist* ((prefix suffix part) (permutated-global-index book))
                                        (<:tr
                                         (<:td :align "right"
                                               (<:a :href (strcat "../" (make-anchor-link (descriptor pa
                                                    (<:as-html prefix)))
                                         (<:td (<:a :href (strcat "../" (make-anchor-link (descriptor pa
                                                    (<:as-html suffix)))
                                         (<:td (<:a :href (strcat "../" (make-anchor-link (descriptor pa
                                                    " [" (<:as-html (pretty-label-prefix (descriptor par
```

[Source Context]

Function: GENERATE-SECTION

```
(defun generate-section (section generator)
  (<qbook-page :title (title generator)
               :file-name (make-pathname :name (make-anchor-name (text (first section)))
                                         :type "html")
               (output-directory generator)
               (<:h1 :class "title" (<:as-html (title generator)))
               (<:div :class "contents"
                      (publish section))))
```

[Source Context]

Method: MAKE-ANCHOR-LINK

```
(defmethod make-anchor-link ((h heading-part))
  (if (= 1 (depth h))
      (strcat (make-anchor-name (text h)) ".html")
      (labels ((find-level-1 (h)
                 (if (= 1 (depth h))
                     h
                     (find-level-1 (up-part h)))))
        (strcat (make-anchor-link (find-level-1 h)) "#" (make-anchor-name (text h))))))
```

[Source Context]

Method: MAKE-ANCHOR-LINK

```
(defmethod make-anchor-link ((d descriptor))
  (if (name d)
      (concatenate 'string "api/" (make-anchor-name d) ".html")
      "#"))
```

[Source Context]

Method: MAKE-ANCHOR-NAME

```
(defmethod make-anchor-name ((text string))
  (regex-replace-all "[^A-Za-z.-]" text
                     (lambda (target-string start end match-start match-end reg-starts reg-ends)
                       (declare (ignore start end match-end reg-starts reg-ends))
                       (format nil "_~4,'0X" (char-code (aref target-string match-start))))))
```

[Source Context]

Function: EFFECTIVE-NAME

```
(defun effective-name (function-name)
  (if (symbolp function-name)
      function-name
      (second function-name)))
```

[Source Context]

Method: MAKE-ANCHOR-NAME

```
(defmethod make-anchor-name ((descriptor descriptor))
  (make-anchor-name (strcat (label-prefix descriptor)
                            "_"
                            (package-name (symbol-package (effective-name (name descriptor))))
                            "::"
                            (if (symbolp (name descriptor))
                                (symbol-name (name descriptor))
                                (format nil "(~A ~A)"
                                        (symbol-name (first (name descriptor)))
                                        (symbol-name (second (name descriptor))))))))
```

[Source Context]

Method: MAKE-ANCHOR-NAME

```
(defmethod make-anchor-name ((method-descriptor defmethod-descriptor))
  (make-anchor-name (strcat (label-prefix method-descriptor)
                            "_"
                            (package-name (symbol-package (effective-name (name method-descriptor))))
                            "::"
                            (html-name method-descriptor))))
```

[Source Context]

Method: HTML-NAME

```
(defmethod html-name ((descriptor descriptor))
  (name descriptor))
```

[Source Context]

Method: HTML-NAME

```
(defmethod html-name ((descriptor defmethod-descriptor))
  (format nil "(~A~@[ ~A~]~{ ~A~})"
          (name descriptor)
          (qualifier descriptor)
          (remove-if #'null
                     (mapcar (lambda (argument)
                               (typecase argument
                                 (arnesi::specialized-function-argument-form
                                  (arnesi::specializer argument))))
                             (lambda-list descriptor)))))
```

[Source Context]

Function: PUBLISH

```
(defun publish (parts)
  (iterate
    (with state = nil)
    (for p in parts)
    (setf (output-file p) (strcat (make-anchor-name (text (first parts))) ".html"))
    (etypecase p
      (comment-part (setf state (write-comment p state)))
      (whitespace-part (setf state nil) (<:as-html (text p)))
      (code-part (setf state (write-code p state))))))
```

[Source Context]

Function: NUM-LINES

```
(defun num-lines (text)
  (iterate
    (with num-lines = 0)
    (for char in-string text)
    (when (member char '(#\Newline #\Return #\Linefeed))
      (incf num-lines))
    (finally (return num-lines))))
```

[Source Context]

Function: WRITE-CODE

```
(defun write-code (part state)
  (ecase state
    ((nil) nil)
    (:in-comment
     (setf state nil)
     (write-string "</p>" *yaclml-stream*)
     (terpri *yaclml-stream*)))
  (write-code-descriptor (descriptor part) part *generator*)
  nil)
```

[Source Context]

Generic Function: WRITE-CODE-DESCRIPTOR

Writes the documentation of PART using DESCRIPTOR for the current GENERATOR

```
(defgeneric write-code-descriptor (descriptor part generator)
  (:documentation "Writes the documentation of PART using DESCRIPTOR for the current GENERATOR"))
```

[Source Context]

Method: WRITE-CODE-DESCRIPTOR

```
(defmethod write-code-descriptor ((descriptor t) part (generator html-generator))
  (let ((text (text part)))
    (setf text (yaclml::escape-as-html text))
    (setf text (regex-replace-all "(\\(|\\))"
                                  text
                                  "<span class=\"paren\">\\1</span>"))
    (setf text (regex-replace "^.*"
                              text
                              (strcat "<span class=\"first-line\">\\&</span><span class\"body\">")))
    (<:pre (<:code :class "code" (<:as-is text) (<:as-is "</span>")))))
```

[Source Context]

Method: WRITE-CODE-DESCRIPTOR

```
(defmethod write-code-descriptor :around ((descriptor descriptor) part (generator html-generator))
  (<:div :class (strcat "computational-element-link "
                        "computational-element-link-" (label-prefix descriptor))
         (<:p (<:a :name (make-anchor-name descriptor)
                   :href (make-anchor-link descriptor)
                   (<:as-html (pretty-label-prefix descriptor))
                   " "
                   (<:as-html (html-name descriptor)))
              " "
              (when-bind first-sentence (docstring-first-sentence descriptor)
                (<:as-html first-sentence))))
  (<qbook-page :title (strcat (pretty-label-prefix descriptor) " " (html-name descriptor))
               :file-name (make-anchor-link descriptor)
               :stylesheet "../style.css"
               :printsheet "../print.css"
               (<:div :class "computational-element"
                      (<:h1 (<:as-html (pretty-label-prefix descriptor)) ": " (<:as-html (html-name desc
                      (<:div :class "contents"
                             (when (docstring descriptor)
                               (<:h2 "Documentation")
                               (<:blockquote
                                (<:as-html (docstring descriptor))))
                             (call-next-method)
                             (<:h2 "Source")
                             (<:pre (<:code :class "code" (<:as-html (text part))))
                             (<:a :href (strcat "../" (output-file part) "#" (make-anchor-name (descript
                                  "Source Context"))))))
```

[Source Context]

Method: WRITE-CODE-DESCRIPTOR

```
(defmethod write-code-descriptor ((descriptor descriptor) part (generator html-generator))
  (declare (ignore part))
  nil)
```

[Source Context]

Method: WRITE-CODE-DESCRIPTOR

```
(defmethod write-code-descriptor ((descriptor defclass-descriptor) part (generator html-generator))
  (declare (ignore part))
  (when (slots descriptor)
    (<:h2 "Slots")
    (<:ul
     (dolist (slot (slots descriptor))
       (<:li (<:as-html (name slot))
             (when (docstring slot)
               (<:as-html " - " (docstring slot)))))))
  (<:h2 "Hierachy")
  (<:h3 "Precedence List")
  (flet ((make-class-link (class)
           (aif (find-descriptor "class" (class-name class))
                (<:a :href (strcat "../" (make-anchor-link it))
                     (<:as-html (class-name class)))
                (<:as-html (class-name class)))))
    (<:ul
     (dolist (class (mopp:class-direct-superclasses (find-class (name descriptor))))
       (<:li (make-class-link class))))
    (awhen (mopp:class-direct-subclasses (find-class (name descriptor)))
      (<:h3 "Sub Classes")
      (<:ul
       (dolist (sub it)
         (<:li (make-class-link sub)))))))
```

[Source Context]

Function: WRITE-COMMENT

```
(defun write-comment (part state)
  (etypecase part
    (heading-part
     (ecase state
       ((nil))
       (:in-comment
        ;; heading during a comment, break the current comment
        ;; and start a new one.
        (write-string "</p>" *yaclml-stream*)
        (terpri *yaclml-stream*)))
     (flet ((heading ()
              (<:a :name (make-anchor-name (text part)) (<:as-html (text part)))
              (<:as-is "&nbsp;"))
            (nav-links ()
              (<:div :class "nav-links"
                     (if (prev-part part)
                         (<:a :class "nav-link" :href (make-anchor-link (prev-part part)) "prev")
                         (<:span :class "dead-nav-link" "prev"))
                     " | "
                     (if (up-part part)
                         (<:a :class "nav-link" :href (make-anchor-link (up-part part)) "up")
                         (<:span :class "dead-nav-link" "up"))
                     " | "
                     (if (next-part part)
                         (<:a :href (make-anchor-link (next-part part)) "next")
                         (<:span :class "nav-link" "next"))
                     " | "
                     (<:a :href "index.html" "toc"))))
       (case (depth part)
         (1 (<:h2 (heading)))
         (2 (<:h3 (heading)))
         (3 (<:h4 (heading)))
         (4 (<:h5 (heading)))
         (5 (<:h6 (heading)))
         (t (error "Nesting too deep: ~S." (text part))))
       (nav-links))
     nil)
    (comment-part
        ;;;; regular comment
     (ecase state
       ((nil) (write-string "<p>" *yaclml-stream*))
       (:in-comment nil))
     (if (escape-comments *generator*)
         (<:as-html (text part))
         (<:as-is (text part)))
     :in-comment)))
```

[Source Context]

Variable: *PRINT.CSS*

The alternative (destined for hard copy) HTML stylesheet.

```
(defvar *print.css*
  "body {
  background-color: #FFFFFF;
  padding: 0px; margin: 0px;
}

.qbook {
  width: 600px;
  background-color: #FFFFFF;
  padding: 0em;
  margin: 0px;
}

h1, h2, h3, h4, h5, h6 {
  font-family: verdana;
}

h1 {
  text-align: center;
  padding: 0px;
  margin: 0px;
}

h2 {
  text-align: center;
  border-top: 1px solid #000000;
  border-bottom: 1px solid #000000;
}

h3, h4, h5, h6 {
  border-bottom: 1px solid #000000;
  padding-left: 1em;
}

h3 { border-top: 1px solid #000000; }

p { padding-left: 1em; }

pre.code {
  border: solid 1px #FFFFFF;
  padding: 2px;
  overflow: visible;
}

pre .first-line-more-link { display: none; }

pre.code * .paren  { color: #666666; }

pre.code a:active  { color: #000000; }
pre.code a:link    { color: #000000; }
pre.code a:visited { color: #000000; }

pre.code .first-line { font-weight: bold; }

pre.code .body, pre.code * .body { display: inline; }

div.contents {
  font-family: verdana;
  border-bottom: 1em solid #333333;
  margin-left: -0.5em;
}
```

[Source Context]

Variable: *STYLE.CSS*

The default stylesheet for qbook generated html documentation.

```
(defvar *style.css*
  "body {
  background-color: #FFFFFF;
  padding: 0px;
  margin: 0px;
  font-family: verdana;
}

.qbook {
  margin: auto;
  background-color: #FFFFFF;
  width: 40em;
}

h1, h2, h3, h4, h5, h6 {
  color: #990000;
}

h1 {
  text-align: center;
  padding: 0px;
  margin: 0px;
}

h2 {
  text-align: center;
  border-bottom: 5px solid #CC0000;
  margin-top: 2em;
}

h3, h4, h5, h6 {
  padding-left: 1em;
  margin-top: 2em;
}

h3 {
  border-bottom: 2px solid #CC0000;
}

h4, h5, h6 {
  border-bottom: 1px solid #CC0000;
}

pre.code {
  background-color: #eeeeee;
  border: solid 1px #d0d0d0;
  overflow: auto;
}

pre.code * .paren { color: #666666; }

pre.code .first-line { font-weight: bold; }

pre.code .body, pre.code * .body { display: none; }

div.contents {
  font-family: verdana;
}

a:active  { color: #0000AA; }
a:link    { color: #0000AA; }
```

[Source Context]

Class: LATEX-GENERATOR

Slots

- OUTPUT-FILE
- LISTINGS - When non-NIL, generate listings with LaTeX listings package.
- HIGHLIGHT-SYNTAX - When T, highlight syntax using highlight.js library

Hierarchy

Precedence list

- GENERATOR

```
(defclass latex-generator (generator)
  ((output-file :initarg :output-file :accessor output-file)
   (listings :initarg :listings :accessor listings :initform nil
             :documentation "When non-NIL, generate listings with LaTeX listings package.")
   (highlight-syntax :initarg :highlight-syntax
                     :accessor highlight-syntax
                     :type boolean
                     :initform t
                     :documentation "When T, highlight syntax using highlight.js library")))
```

[Source Context]

Variable: *LATEX-STREAM*

```
(defvar *latex-stream*)
```

[Source Context]

Function: LATEX-COMMAND

```
(defun latex-command (name &rest args)
  (declare (special *latex-stream*))
  (write-string "\\" *latex-stream*)
  (write-string name *latex-stream*)
  (dolist (arg args)
    (write-string "{" *latex-stream*)
    (write-string arg *latex-stream*)
    (write-string "}" *latex-stream*))
  (terpri *latex-stream*))
```

[Source Context]

Generic Function: GENERATE-PART

```
(defgeneric generate-part (part generator))
```

[Source Context]

Method: GENERATE

```
(defmethod generate (book (generator latex-generator))
  (with-output-to-file (*latex-stream* (output-file generator)
                                       :if-exists :supersede
                                       :if-does-not-exist :create)
    (declare (special *latex-stream*))
    (flet ((wl (s &rest args)
             (write-line (apply #'format nil s args) *latex-stream*)))
      (wl "\\documentclass[11pt,pdflatex,makeidx]{scrbook}")
      (wl "\\usepackage[margin=0.5in]{geometry}")
      (wl "\\usepackage{xcolor}")
      (wl "\\usepackage{makeidx}")
      (wl "\\usepackage{hyperref}")
      (when (highlight-syntax generator)
        (wl "\\usepackage{minted}")
        (wl "\\usepackage{mdframed}"))

      (when (listings generator)
        (latex-command "usepackage" "listings")
        (latex-command "lstset" "language=lisp"))

      (when (stringp (listings generator))
        (latex-command "lstset" (listings generator)))

      (wl "\\usepackage{courier}")

      (wl "\\definecolor{CodeBackground}{HTML}{E9E9E9}")
      (wl "\\hypersetup{colorlinks=true,linkcolor=blue}")

      (wl "\\parindent0pt  \\parskip10pt             % make block paragraphs")
      (wl "\\raggedright                            % do not right justify")

      (latex-command "title" (title generator))

      (latex-command "date" "")
      (latex-command "makeindex")

      (latex-command "begin" "document")
      (latex-command "maketitle")
      (latex-command "tableofcontents")
      (dolist (section (contents book))
        (dolist (part section)
          (generate-part part generator)))
      (terpri *latex-stream*)
      (wl "\\addtocontents{toc}{\\protect\\setcounter{tocdepth}{0}}")
      (wl "\\chapter{Reference}")
      (dolist (section (contents book))
        (dolist (part section)
          (generate-part-reference part generator)))
      (terpri *latex-stream*)
      (terpri *latex-stream*)
      (wl "\\chapter{Index}")
      (wl "\\printindex")
      (latex-command "end" "document"))))
```

[Source Context]

Function: SAFE-LATEX-ID

```
(defun safe-latex-id (string)
  (with-output-to-string (stream)
    (iterate
      (for char in-string string)
      (case char
        ((#\& #\$ #\% #\# #\_ #\{ #\} #\^ #\\ )
         (write-char #\- stream))
        (t (write-char char stream))))))
```

[Source Context]

Function: DESCRIPTOR-REF-ID

```
(defun descriptor-ref-id (descriptor)
  (safe-latex-id
   (strcat (string (label-prefix descriptor))
           ":"
           (princ-to-string (name descriptor)))))
```

[Source Context]

Function: DESCRIPTOR-LINK-ID

```
(defun descriptor-link-id (descriptor)
  (safe-latex-id
   (strcat "link:" (string (label-prefix descriptor))
           ":" (princ-to-string (name descriptor)))))
```

[Source Context]

Method: GENERATE-PART

Generate link to the code

```
(defmethod generate-part ((part code-part) (generator latex-generator))
  "Generate link to the code"
  (if (descriptor part)
      (progn
        (latex-command "label" (descriptor-link-id (descriptor part)))
        (format *latex-stream*
                "\\hyperref[~a]{~a ~a}"
                (descriptor-ref-id (descriptor part))
                (pretty-label-prefix (descriptor part))
                (safe-latex-id (princ-to-string (name (descriptor part)))))
        (terpri *latex-stream*)
        (when (docstring (descriptor part))
          (write-line " - " *latex-stream*)
          (write-line (docstring-first-sentence (descriptor part)) *latex-stream*)))
      (write-source (text part) generator)))
```

[Source Context]

Function: WRITE-SOURCE

```
(defun write-source (source generator)
  (if (highlight-syntax generator)
      (write-line "\\begin{minted}[fontsize=\\footnotesize, framesep=2mm,baselinestretch=1.2, bgcolor=Co
      (latex-command "begin" (if (listings generator) "lstlisting" "verbatim")))
  (write-line source *latex-stream*)
  (latex-command "end"
                 (cond
                   ((highlight-syntax generator) "minted")
                   ((listings generator) "lstlisting")
                   (t "verbatim"))))
```

[Source Context]

Method: GENERATE-PART

```
(defmethod generate-part ((part whitespace-part) (generator latex-generator))
  (write-string (text part) *latex-stream*))
```

[Source Context]

Method: GENERATE-PART

```
(defmethod generate-part ((part heading-part) (generator latex-generator))
  (write-string (ecase (depth part)
                  (1 "\\chapter{")
                  (2 "\\section{")
                  (3 "\\subsection{")
                  (4 "\\subsubsection*{")
                  (5 "\\subsubsubsection*{"))
                *latex-stream*)
  (write-latex-escaped (text part) *latex-stream*)
  (write-string "}" *latex-stream*)
  (terpri *latex-stream*))
```

[Source Context]

Method: GENERATE-PART

```
(defmethod generate-part ((part comment-part) (generator latex-generator))
  (write-latex-escaped (text part) *latex-stream*))
```

[Source Context]

Function: WRITE-LATEX-ESCAPED

```
(defun write-latex-escaped (string stream)
  (iterate
    (for char in-string string)
    (case char
      ((#\& #\$ #\% #\# #\_ #\{ #\} #\^)
       (write-char #\\ stream)
       (write-char char stream)
       (write-string "{}" stream))
      (#\\ (write-string "$\\backslash$" stream))
      (t (write-char char stream)))))
```

[Source Context]

Generic Function: GENERATE-PART-REFERENCE

```
(defgeneric generate-part-reference (part generator))
```

[Source Context]

Method: GENERATE-PART-REFERENCE

```
(defmethod generate-part-reference ((part code-part) (generator latex-generator))

  (when (docstring (descriptor part))
    (write-latex-escaped (docstring (descriptor part)) *latex-stream*)
    (terpri *latex-stream*)
    (write-line "\\vskip 0.1in" *latex-stream*))

  (write-code-descriptor (descriptor part) part generator)

  (write-source (text part) generator)
  (terpri *latex-stream*)
  (format *latex-stream* "\\hyperref[~a]{[Source Context]}"
          (descriptor-link-id (descriptor part)))
  (terpri *latex-stream*)
  (terpri *latex-stream*))
```

[Source Context]

Method: GENERATE-PART-REFERENCE

```
(defmethod generate-part-reference (part generator)
  )
```

[Source Context]

Method: GENERATE-PART-REFERENCE

```
(defmethod generate-part-reference :around ((part code-part) generator)
  (when (descriptor part)
    (latex-command "section*" (strcat (pretty-label-prefix (descriptor part))
                                      ": "
                                      (safe-latex-id (princ-to-string (name (descriptor part))))))
    (latex-command "label" (descriptor-ref-id (descriptor part)))
    (latex-command "index" (strcat (pretty-label-prefix (descriptor part)) " "
                                    (safe-latex-id (princ-to-string (name (descriptor part))))))
    (call-next-method)))
```

[Source Context]

Method: WRITE-CODE-DESCRIPTOR

```
(defmethod write-code-descriptor ((descriptor t) part (generator latex-generator)))
```

[Source Context]

Method: WRITE-CODE-DESCRIPTOR

```
(defmethod write-code-descriptor ((descriptor defclass-descriptor) part (generator latex-generator))
  (flet ((write-class-link (class)
           (format *latex-stream*
                   "\\hyperref[class:~a]{~a}"
                   (class-name class)
                   (class-name class))))
    (when (slots descriptor)
      (latex-command "subsection*" "Slots")
      (latex-command "begin" "itemize")
      (dolist (slot (slots descriptor))
        (write-string "\\item " *latex-stream*)
        (princ (name slot) *latex-stream*)
        (when (docstring slot)
          (write-string " - " *latex-stream*)
          (write-string (docstring slot) *latex-stream*))
        (terpri *latex-stream*))
      (latex-command "end" "itemize"))

    (latex-command "subsection*" "Hierarchy")
    (latex-command "subsubsection*" "Precedence list")

    (latex-command "begin" "itemize")
    (dolist (class (mopp:class-direct-superclasses (find-class (name descriptor))))
      (write-string "\\item " *latex-stream*)
      (write-class-link class)
      (terpri *latex-stream*))
    (latex-command "end" "itemize")

    (awhen (mopp:class-direct-subclasses (find-class (name descriptor)))
      (latex-command "subsection*" "Sub Classes")
      (latex-command "begin" "itemize")
      (dolist (sub it)
        (write-string "\\item " *latex-stream*)
        (write-class-link sub)
        (terpri *latex-stream*))
      (latex-command "end" "itemize"))))
```

[Source Context]

7 Index

Index