Scheme Program Documentation Tools

Kurt Nørmark
Department of Computer Science
Aalborg University
Denmark
normark@cs.aau.dk

Abstract

This paper describes and discusses two different Scheme documentation tools. The first is SchemeDoc, which is intended for documentation of the interfaces of Scheme libraries (APIs). The second is the Scheme Elucidator, which is for internal documentation of Scheme programs. Although the tools are separate and intended for different documentation purposes they are related to each other in several ways. Both tools are based on XML languages for tool setup and for documentation authoring. In addition, both tools rely on the LAML framework which—in a systematic way—makes an XML language available as a set of functions in Scheme. Finally, the Scheme Elucidator is able to integrate SchemeDoc resources as part of an internal documentation resource.

1 Introduction

Program documentation tools are important for all kinds of non-trivial programming tasks. In a general sense, program documentation tools make it possible to produce important information for programmers who apply a program library, and for future developers of a program. In this paper we are concerned with program documentation for Scheme developers. End user documentation is not an issue in this paper.

We discuss two documentation tools for Scheme. The first, SchemeDoc, is a tool for documenting library interfaces—also known as application programmers interfaces (APIs). The documentation produced by SchemeDoc is intended for Scheme programmers who apply the documented Scheme library. The second, the Scheme Elucidator, is a tool for documentation of the internal details of a Scheme program. The documentation produced by the Scheme Elucidator—called an elucidative program—is typically intended for future maintainers of the program. Elucidative Scheme programs may, however, be targeted towards any reader with an interest in understanding the program. As such, the Scheme Elucidator can be used whenever there is a need to *write about* a Scheme program, for educational, tutorial, or scientific purposes.

SchemeDoc and the Scheme Elucidator share a number of properties. The input formats of both tools are defined as XML languages, with use of XML DTDs (Document Type Definitions) [1]. In simple cases, the input formats are relatively small setup files that hold a number of processing attributes, whereas in other cases, substantial amounts of documentation is authored within the XML documents. Both tools are part of LAML (see Section 2) and as such the full expressiveness of Scheme can be used in the XML-related parts of both SchemeDoc documentation and in elucidative programs. Finally, both tools generate web output, in terms of XHTML files.

In this paper we give overall and concise descriptions of the tools. More detailed descriptions can be found on the web [16, 20]. As part of the discussions we point out relevant details in the implementation of the tools. In addition we compare the tools with other similar documentation tools for Scheme.

The paper is structured as follows. In Section 2 we summarize the most basic properties of LAML, which is the common underlying platform of both tools. In Section 3 we discuss SchemeDoc. This includes a discussion of similar tools for documentation of Scheme libraries. In Section 4 we discuss the Scheme Elucidator. The main contributions and the conclusions are summarized in Section 5. It is possible to skip Section 3 in case the reader is only interested in documentation of internal programs aspects with the Scheme Elucidator. The programs and documentation that are discussed in this paper are all available as web resources [21].

2 LAML Background

Both tools described in this paper rely on LAML (Lisp Abstracted Markup Language), and we will therefore in this section provide a brief summary of LAML. For more information about LAML please consult the paper *Web Programming in Scheme with LAML* [17] and the LAML home page [18].

From an overall perspective, LAML attempts to come up with natural Scheme-based counterparts to the most important aspects of XML. The main purpose of LAML is to make XML languages available as sets of Scheme functions. With this, an XML document becomes a Scheme expression. As a consequence, the power of Scheme is available anywhere in a document, and at any time during the authoring process. We refer to this situation as *programmatic authoring* [15].

The set of Scheme functions that corresponds to the elements of an XML language *L* is called a *mirror of L in Scheme*. Each element of an XML language is represented as a Scheme function. When applied, these functions generate an internal format (ASTs represented as lists) and they carry out a comprehensive documentation validation at run time (document processing time).

LAML provides an XML DTD parser and a mirror generation tool. These tools have been used to generate validating mirrors of XHTML, SVG and a number of more specialized XML languages (such as the SchemeDoc language and the Elucidator language discussed in this paper).

Web authoring with LAML is supported by a set of convenient Emacs editor commands. No specialized lexical Scheme conventions are used. As an example, the sample XML fragment

```
<book id = "sicp">
  <title>Structure and Interpretation
         of Computer Programs</title>
  <authors>
    <author>Abelson</author>
    <author>Sussman</author>
  </authors>
</book>
```

can be written as the Scheme expression

```
(book 'id "sicp"
  (title "Structure and Interpretation
          of Computer Programs")
  (authors (author "Abelson") (author "Sussman"))
)
```

provided that the mirror of the book description language is loaded on beforehand.

The parameters to each mirror function are interpreted relative to the *LAML parameter passing rules* [17], which can be summarized as follows: An attribute is a symbol; an attribute value is the string following a symbol; other strings represent textual element content items; lists are recursively unfolded. If relevant, white space is always provided in between element content items unless explicitly suppressed by a distinguished value (`#f` usually bound to the variable named _). As a consequence of the list unfolding rule, the expression

```
(authors (map author (list "Abelson" "Sussman")))
```

is equivalent to

```
(authors (author "Abelson") (author "Sussman"))
```

The definition of XML languages, and their mirrors in Scheme, can be seen as a *linguistic abstraction process*. With use of the higher-order function `xml-in-laml-abstraction` it is, in addition, possible for the author to define functions that use LAML parameter passing rules. Seen in contrast to the linguistic abstractions, such functions are called *ad hoc abstractions*.

LAML works on a variety of different Scheme Systems on Unix and Windows. Therefore the documentation tools discussed in this paper can be used together with many different Scheme systems on both platforms.

3 SchemeDoc

As stated in the introduction, SchemeDoc is a tool for creation of web documentation of programmatic interfaces of Scheme programs, most notable the interfaces of program libraries. Many programmers are familiar with web documentation of programmatic interfaces from the success of Javadoc [2, 29]. As Javadoc, SchemeDoc supports extraction of documentation from distinguished *documentation comments* in source programs. In addition, SchemeDoc allows manual authoring of the documentation, and documentation of XML mirror functions in Scheme. In the section 3.1 below we describe these possibilities.

3.1 SchemeDoc operational modes

SchemeDoc can be used in four operational modes:

- **Source Extraction mode.**
  The documentation is extracted from distinguished documentation comments in a Scheme source program.
- **Manual mode.**
  The documentation is authored manually, in an XML format with use of LAML.
- **XML DTD mode.**
  The documentation is extracted from a parsed XML DTD, typically with the purpose of documenting the mirror of the XML language in Scheme.
- **Augmented XML DTD mode.**
  A mixture of the XML DTD mode and the manual mode. Documentation, which is not present in the DTD is authored manually and merged with the extracted DTD documentation.

The Source Extraction mode relies on the concepts of comment blocks and documentation comments. A *comment block* is a sequence of consecutive Scheme comment lines (each of which is initiated with a semicolon). A *documentation comment* is a comment block which, by means of a given commenting style, is set apart from "ordinary comments".

Documentation comments are classified as either definition comments, documentation sections, or documentation abstracts. A *definition comment* precedes and documents a Scheme definition. A *documentation section* describes common properties of the set of definitions that follows the section comment. A *documentation abstract* gives an initial and overall description of a Scheme source file.

SchemeDoc supports two different *commenting styles* for identification of documentation comments: multi-semicolon style and documentation-mark style. Using *multi-semicolon style*, each documentation comment line is initiated with two, three or four semicolons, supporting definition comments, documentation sections, and documentation abstracts respectively. Using *documentation-mark style,* a documentation comment is identified with occurrences of a distinguished character (per default '!') at the start of the first comment line in a comment block. Definition comments use a single mark, documentation sections use two marks, and documentation abstracts use three exclamation marks. Until recently, all LAML software has been documented using multi-semicolon style.

Within documentation comments, a *little markup language* is used to provide additional structure. SchemeDoc uses dot-initiated documentation keywords together with a line-oriented organization. These elements of SchemeDoc are, to a large degree, modelled directly after similar systems, such as Javadoc [2, 29] and Doxygen [30].

As a concrete illustration of SchemeDoc in Source Extraction mode with use of multi-semicolon documentation comments, the Scheme

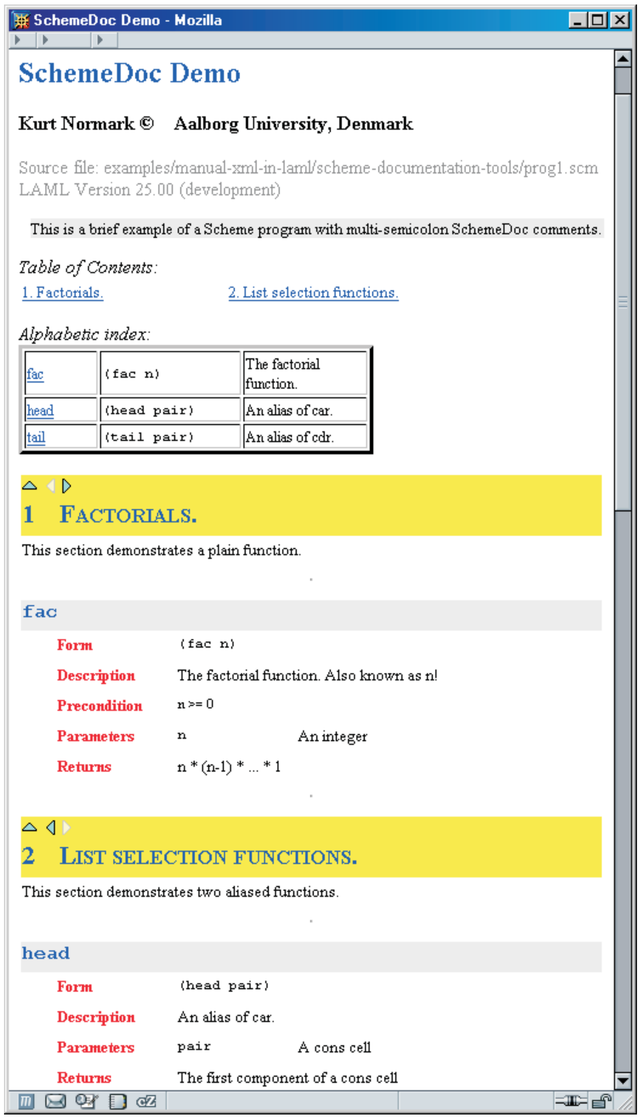
```
;;;; .title SchemeDoc Demo
;;;; .author Kurt Normark
;;;; .affiliation Aalborg University, Denmark
;;;; This is a brief example of a Scheme
;;;; program with multi-semicolon SchemeDoc comments.

; This comment is not extracted.

;;; Factorials.
;;; .section-id fac-stuff
;;; This section demonstrates a plain function.

;; The factorial function. Also known as n!
;; .parameter n An integer
;; .pre-condition n >= 0
;; .returns n * (n-1) * ... * 1
(define (fac n)
 (if (= n 0) 1 (* n (fac (- n 1)))))

;;; List selection functions.
;;; .section-id list-stuff
;;; This section demonstrates two aliased functions.

;; An alias of car.
;; .returns The first component of a cons cell
;; .form (head pair)
;; .parameter pair A cons cell
(define head car)

;; An alias of cdr.
;; .returns The second component of a cons cell
;; .form (tail pair)
;; .parameter pair A cons cell
(define tail cdr)
```

Figure 1: *A Scheme program with documentation comments in multi-semicolon style.*

Program in Figure 1 gives rise to the extracted documentation, shown partially in Figure 2. (The same example is shown with use of documentation-mark style at the web resource page [21] of this paper). The figure illustrates a single documentation abstract, two documentation sections, and three definition comments. SchemeDoc ignores one-semicolon comments. In Figure 1 we illustrate the `title`, `author`, and `affiliation` tags in the documentation abstract. In the section comments, we illustrate the `section-id` tag, which is used for generation of an anchor name in HTML. In the definition comments, we illustrate the `form`, `parameter`, `pre-condition`, and `returns` tags. The `form` tag is used in situations where the actual calling form does not appear as a constituent of the definition.

SchemeDoc can deal with nested documentation comments. More specifically, definition comments and documentation sections are extracted from the definitions, which are documented by means of definition comments. In the current version of SchemeDoc, we only handle two levels of nested documentation comments.

XML DTD mode can, in general, be used for documentation of an XML DTD, which has been parsed with the LAML DTD parser [18]. The documentation of an XML DTD presents the DTD after full expansion of the parameter entities [1] (textual macros in the DTD). Use of parameter entities is convenient in order to reduce the complexity of the DTD authoring process, but they make


SchemeDoc Demo - Mozilla

SchemeDoc Demo

Kurt Normark © Aalborg University, Denmark

Source file: examples/manual-xml-in-laml/scheme-documentation-tools/prog1.scm
LAML Version 25.00 (development)

This is a brief example of a Scheme program with multi-semicolon SchemeDoc comments.

Table of Contents:
1. Factorials. 2. List selection functions.

Alphabetic index:

fac	(fac n)	The factorial function.
head	(head pair)	An alias of car.
tail	(tail pair)	An alias of cdr.

1 FACTORIALS.

This section demonstrates a plain function.

fac

Form	(fac n)	
Description	The factorial function. Also known as n!	
Precondition	n >= 0	
Parameters	n	An integer
Returns	n * (n-1) * ... * 1	

2 LIST SELECTION FUNCTIONS.

This section demonstrates two aliased functions.

head

Form	(head pair)	
Description	An alias of car.	
Parameters	pair	A cons cell
Returns	The first component of a cons cell	

Figure 2: *A partial presentation of the SchemeDoc documentation from Figure 1.*

it difficult to read the DTD. Activation of SchemeDoc on the parsed XML DTD file leads to a straightforward presentation of the XML elements, primarily in terms of the XML content model and information about the attributes. The presentation of content models provides for easy navigation to constituent elements, and to elements in which the current element appears as a constituent. The XML DTD mode of SchemeDoc is of particular importance for documentation of the major and well-known XML languages, such as the different versions of XHTML and SVG, for which LAML provides mirrors in Scheme.

The augmented XML DTD mode makes it possible to combine manually authored contributions with the documentation extracted from the XML DTD. In that respect, this mode is a mixture of the Manual mode and the XML DTD mode, as described above. More specifically, SchemeDoc is able generate an initial documentation file (in the format used in Manual mode). By filling in the con-

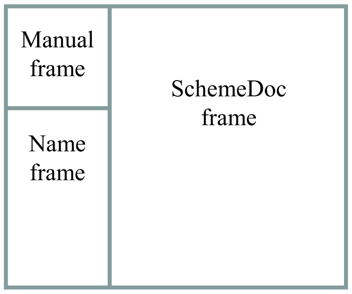


Figure 3: *The frame layout of a SchemeDoc index.*

tents of element and attribute descriptions, the intuitive meaning of the elements can be documented. It is, in addition, often helpful to add some sectioning to provide for better structure and overview. At SchemeDoc processing time, the manually authored documentation is added to the information from the parsed XML DTD. In this way, the information from the parsed XML DTD always controls the final documentation. All substantial LAML document styles, including SchemeDoc and the Scheme Elucidator (see Section 4) are documented by use of SchemeDoc in Augmented XML DTD mode. Examples of such documentation can be seen via the LAML home page [18].

3.2 SchemeDoc Indexing

A collection of SchemeDoc manuals can be indexed and organized with use of the SchemeDoc Indexing tool. Based on an enumeration of a number of SchemeDoc manuals, this tool produces a browser with three frames (see Figure 3). The browser is made available as a frameset in XHTML. The Manual frame lists the involved SchemeDoc manuals. The Name frame shows a sorted list of the defined names from a selected manual, or from all the manuals taken together. The SchemeDoc frame shows selected details from the selected manual.

The SchemeDoc indexing tool is also able to produce a useful index of the Scheme Report [6] (either R4RS or R5RS). The list of Scheme procedures and syntactic forms can either be shown separately, or it can be merged with the names from the involved SchemeDoc manuals.

3.3 Tool Support

The SchemeDoc tool can be used in several different ways. The primary way is to execute a LAML script, which parameterizes SchemeDoc appropriately. The LAML script, which extracts and creates the documentation in Figure 2 from the Scheme source program in Figure 1, is shown here:

```
(load (string-append laml-dir "laml.scm"))
(laml-style "xml-in-laml/manual/manual")

(manual
 (manual-front-matters
  'css-prestylesheet "compact"
  'css-stylesheet "original"
  'laml-resource "true"
  'documentation-commenting-style "multi-semicolon"
 )

 (manual-from-scheme-file 'src "../prog1.scm")
)
```

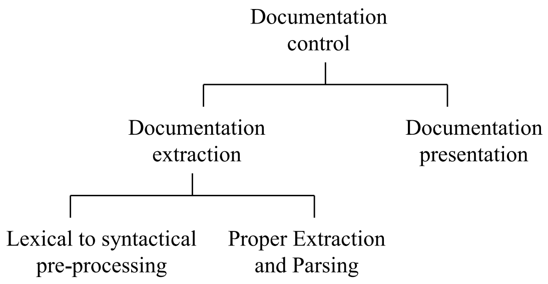

Documentation control
Documentation extraction
Documentation presentation
Lexical to syntactical pre-processing
Proper Extraction and Parsing

Figure 4: *The decomposition of the SchemeDoc tool.*

The LAML script can be executed via the operating system shell, from a Scheme read-eval-print loop, or via an Emacs command. Following the loading of `laml.scm` (first line) and the SchemeDoc manual stuff (second line) the `manual` clause contains tool setup parameters (the `manual-front-matters` clause) and specification of tool functionality (the `manual-from-scheme-file` clause). In this particular example, the element `manual-from-scheme-file` causes SchemeDoc to be used in Source Extraction mode. Typically, we organize LAML scripts, like the one shown above, in a `man` subdirectory of the Scheme source file directory.

SchemeDoc can also be used without LAML scripts. As one alternative, it is possible to activate a `schemedoc` procedure from a (LAML-enabled) Scheme read-eval-print loop. Another alternative is to activate SchemeDoc on a Scheme source file by use of the `schemedoc` command from Emacs. This can be done by `M-x schemedoc`, or via the menu attached to Scheme mode in Emacs. In these cases, the `manual-front-matters` attributes can be given in the documentation abstract comment. In that way, the SchemeDoc setup parameters (processing options) can be given as part of the Scheme source program.

3.4 Implementation Issues

The Source Extraction mode of SchemeDoc is implemented as documentation extraction followed by documentation presentation, both of which are managed by a documentation control layer. This architecture is illustrated in Figure 4. The top-level control part, the documentation extraction part, and the documentation presentation part are physically separated in the LAML software package. All parts are written in Scheme.

The documentation control layer manages the LAML authoring format (the mirror of the XML SchemeDoc language in Scheme). As it appears in Figure 4, the documentation extraction layer is subdivided in a source file pre-processing part and a proper extraction part. In the source file pre-processing part, lexical comments are transformed to syntactic comments. As an example, the lexical comment `;This is a comment` is transformed to a list like `(comment 1 "This is a comment")`. The second element of the list represents the categorization of the comment (here 1-semicolon comment).

With this pre-processing it is a matter of simple Lisp parsing (reading) to access the documentation comments in other parts of SchemeDoc. The proper extraction and parsing part examines the comment forms and parses the comment strings relative to the documentation markup language. The documentation extraction phase delivers an internal representation in terms of a list of association lists. As an example, the contribution from the `fac` function in Figure 1 is the (slightly elided) association list

```
((kind "manual-page")
 (parameters (parameter "n" "An integer."))
 (description "The factorial function...")
 (pre-condition "n >= 0.")
 (returns "n * (n-1) * ... * 1")
 (title "fac")
 (form (fac n))
)
```

The documentation presentation part generates an XHTML document (with use of CSS styling) from the information in the list of association lists. The internal manual representation is written to an auxiliary file with extension '`manlsp`' such that other tools easily can access the details of a SchemeDoc manual. This information is essential for the SchemeDoc indexing tool (see Section 3.2). The Scheme Elucidator (see Section 4) does also make use of the internal manual representation.

3.5 Similar work

There exists a number of tools which are similar to SchemeDoc. Schematics SchemeDoc [26] is work in progress, primarily oriented towards PLT Scheme, and only scarcely documented. As a novel aspect, this tool uses Scheme lists for markup purposes within documentation comments. Documentation comments are initiated with an exclamation mark. The following slightly elided example (from the web site of Schematics SchemeDoc) illustrates this:

```
;;!
;; (function map
;;   (form (map fn list) -> list)
;;   (contract  ... -> ...)
;;   (example (map (lambda (elt) ...) ...))
;;
;; Apply fn to every element of list.
(define (map fn list) ...)
```

Scmdoc [27], which is a contribution to Bigloo, uses documentation comments distinguished by an exclamation mark after the semicolons of each comment line. Scmdoc is documented clearly and concisely. Directives within a Scmdoc documentation comment are prefixed with '`@`'. The following example is from the Scmdoc documentation:

```
;! @description
;! The documentation generation function.
;! @param iport The input port.
;! @param oport The output port.
;! @return Returns <CODE>#f</CODE>.
(define (scm->html iport oport) ...)
```

Docscm [3] is another similar system, which generates DocBook XML. Docscm is implemented in the Chicken Scheme system. Here is an example, which illustrates that '`@`' is used to distinguish documentation comments from other comments.

```
;;@
;; Returns <varname> arg <varname> * 2
(define (double arg) (* arg 2))
```

In addition, Docscm supports a number of directives prefixed with '`@`', and it supports a notion of documentation sections.

It should be noticed that the documentation-mark style in source extraction mode of LAML SchemeDoc is similar to the commenting conventions supported by Schematics SchemeDoc and Docscm.

Finally, Schmooz [4] is a Texinfo markup language embedded in Scheme comments. Schmooz works with Jaffer's SCM, and it is used to extract documentation from Scheme source files for subsequent Texinfo processing. Schmooz has been used for documentation of SLIB [5].

4 The Scheme Elucidator

The Scheme Elucidator can be used to *write about* a program. Documentation generated by the Scheme Elucidator typically addresses the internal program details, as a contrast to SchemeDoc documentation of the external interface. Elucidative programs are related to literate programs [9], at least in the sense that both can be considered as *program essays*. Whereas a literate program organizes program fragments as constituents of the documentation, programs and documentation are represented separately in an elucidative program.

4.1 The basic approach

An elucidative program relies on relations between the documentation and the program. The relations are represented in the documentation, but presented as links from the documentation to the programs as well as the other way around. The initial conception of Elucidative Programming, and its relations to Literate Programming, is described in a requirements paper [14]. The paper *Elucidative Programming* [13] gives additional descriptions, including details about the original version of the Scheme Elucidator. The Java Elucidator [22] is a tool inspired by the original Scheme Elucidator.

This paper addresses the Scheme Elucidator 2, which uses an XML language as the front-end format, and XHTML (with CSS) in the back-end. The actual documentation can either be written in the special purpose markup language of the original elucidator [13] or by use of an XML documentation language (via LAML expressions in Scheme). The latter approach is recommended, because it is aligned with the approach of SchemeDoc and other XML languages in LAML, but not least because of the power of *programmatic authoring* [15]. In this paper we will stick to documentation authored via the XML language, used via LAML.

The Scheme Elucidator can handle a single documentation file and an arbitrary number of Scheme source files. Together, these files form a *documentation bundle*. In addition, an arbitrary number of SchemeDoc manuals can be taken into account. If a procedure `p`, documented by SchemeDoc, is applied in a program or mentioned in the documentation, there will be links to the interface documentation of `p` from the places where `p` is called or mentioned. In addition, all applications of R4RS/R5RS procedures and syntactic forms are linked the to appropriate locations in the Scheme Report [6].

An elucidative program is presented as a collection of frames in a web browser, using the layout shown in Figure 5. The basic and novel idea related to the presentation of an elucidative program is the *mutual navigation* between the Documentation frame and the Program frame. Given some documentation *d* shown in the Documentation frame, a program fragment described in *d* may be scrolled into view in the Program frame. Symmetrically, given a program abstraction *p* shown in the Program frame, a section of documentation which mentions or explains *p* may be scrolled into

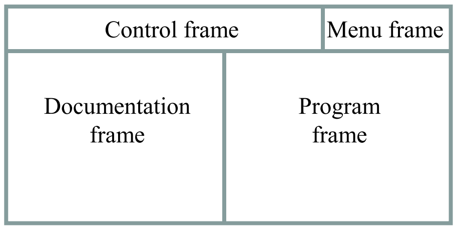


Figure 5: *The Scheme Elucidator frame layout.*

view in the Documentation frame. The Menu frame provides for selection of source programs in a documentation bundle, and the Control frame holds the main navigational icons as well as a structural index of the documentation.

4.2 An example

As a concrete illustration of the Scheme Elucidator 2 we show a small demo of an elucidative Scheme program. The demo includes a single LAML documentation file and two Scheme source files, namely `prog1.scm` from Figure 1 and another program, `prog2.scm`, with a few simple, higher-order Scheme functions. The entire documentation source file is shown in Appendix A. A snapshot of the elucidative program, which makes use of the frame layout shown in Figure 5, can be seen in Figure 6.[1] Notice that only a few links are underlined in the two large frames of Figure 6.

The documentation frame of Figure 6 contains a large number of references to abstractions in `prog1.scm` and `prog2.scm`. There are links from the documentation to the definitions of the Scheme programs. The other way around, the documented definitions in the program frame are decorated with links to the documentation sections with relevant explanations. (These links are anchored in the small icons shown just above the `define` forms). Applied names are linked to their definitions in the Scheme source programs. Reversely, the definitions are, via cross reference tables, linked to the abstractions that apply the definitions. To provide for a natural source-like appearance of the Scheme programs, the links are not underlined, and they are shown in selected dark colors.

The small colored circles, called *source markers*, denote details within a Scheme abstraction. Source markers are used for identification of program details, which are discussed in the documentation. In a Scheme source program the source markers are written as, for instance, '`@a`' in a comment. Pairs of similar source markers (in the documentation and in a source program file) provide for a visual correspondence, but they are also navigatable in both directions.

The elucidative program source in Appendix A shows an `elucidative-front-matters` clause and the documentation intro, sections, and entries in between `(begin-documentation)` and `(end-documentation)`. The `source-files` clause in `elucidative-front-matters` enumerates the Scheme source programs of the documentation bundle and the SchemeDoc files that should be taken into consideration. The `color-scheme` clause defines the background colors which are used to group related source files to each other in a visual way. The `documentation-intro`, `documentation-section`, and `documentation-entry` clauses represent the actual documentation, and they hold the references to the abstractions of the Scheme programs.

[1]For better viewing and color presentation please bring up Figure 6 in your own Internet browser using the link on the web resource page [21] of this paper.

Within the documentation it is possible to address a Scheme definition via the name of the definition, both with and without source file qualifications. The XML element mirror functions `weak-prog-ref` and `strong-prog-ref` are used for this purpose. A strong program reference is intended as a reference to a Scheme definition from a context, which explains the definition. A weak program reference is used when a definition is mentioned in other contexts. It should be noticed that a source marker in the documentation is implicitly related to the closest preceding strong program reference. The distinction between weak and strong program references is not always objective.

Due to the many occurrences of weak and strong program references, the author may choose to introduce "flexible abstractions" on top of these, either ordinary Scheme functions or XML-in-LAML abstractions (see Section 2).

The bodies of documentation entries and sections are typically HTML paragraphs. At the most detailed level, textual content is represented as string constants. As a consequence of the LAML parameter passing rules discussed in Section 2, there is white space in between element content items, unless suppressed by the underscore symbol.

4.3 Tool Support

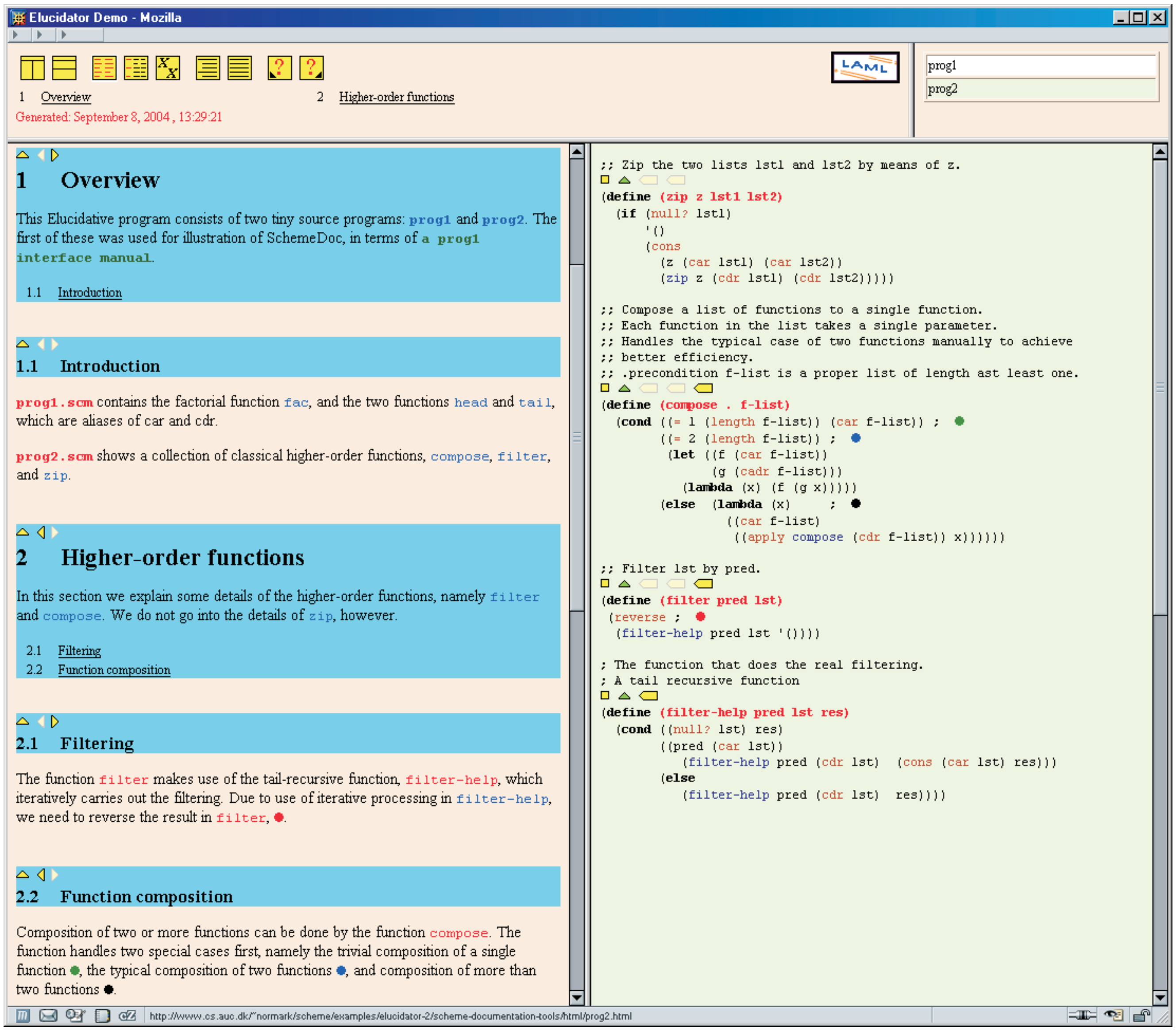
The Scheme Elucidator tool processes a documentation bundle, as defined in Section 4.1. The result of the processing is a collection of HTML files, which can be presented and explored in an Internet browser.

During program development, it is important to support elucidative programming in the programming environment. Without tool support it is difficult and error prone to manage the linking process between the documentation and the abstractions in the source programs. We have developed Emacs tools that support Elucidative Scheme programming. The tools support the creation of links and they make it possible to follow links within the editing environment.

If the programming is done in an integrated development environment (IDE) it is attractive to integrate Elucidative Programming (development as well as browsing) in the IDE. It is a non-trivial task to come up with a good integration. The integration of the Java Elucidator and the TogetherJ IDE shows how this can be done [32].

4.4 Implementation issues

Like the SchemeDoc tool, the Scheme Elucidator is implemented in Scheme. The most challenging aspect of the implementation is the rendering and the linking of the Scheme source programs, i.e., the creation of the program frames. The rendering is done by a simultaneous traversal of the textual Scheme program and the parsed Scheme program. Thus, the Scheme Elucidator processes both the textual and the structural representation of the program. The source program text holds the information about the program layout. The parsed Scheme program makes it convenient to look ahead, for instance into the actual definition following a definition comment. The handling of quotations and quasiquotations calls for particular attention during the traversals, because of differences between the textual and the structural representations.


Elucidator Demo - Mozilla
1 Overview
2 Higher-order functions
Generated: September 8, 2004 , 13:29:21
LAML
prog1
prog2
http://www.cs.auc.dk/~normark/scheme/examples/elucidator-2/scheme-documentation-tools/html/prog2.html

1 Overview

This Elucidative program consists of two tiny source programs: `prog1` and `prog2`. The first of these was used for illustration of SchemeDoc, in terms of a **`prog1 interface manual`**.

1.1 Introduction

1.1 Introduction

`prog1.scm` contains the factorial function `fac`, and the two functions `head` and `tail`, which are aliases of car and cdr.

`prog2.scm` shows a collection of classical higher-order functions, `compose`, `filter`, and `zip`.

2 Higher-order functions

In this section we explain some details of the higher-order functions, namely `filter` and `compose`. We do not go into the details of `zip`, however.

2.1 Filtering
2.2 Function composition

2.1 Filtering

The function `filter` makes use of the tail-recursive function, `filter-help`, which iteratively carries out the filtering. Due to use of iterative processing in `filter-help`, we need to reverse the result in `filter`, ●.

2.2 Function composition

Composition of two or more functions can be done by the function `compose`. The function handles two special cases first, namely the trivial composition of a single function ●, the typical composition of two functions ●, and composition of more than two functions ●.

```
;; Zip the two lists lst1 and lst2 by means of z.
(define (zip z lst1 lst2)
  (if (null? lst1)
      '()
      (cons
        (z (car lst1) (car lst2))
        (zip z (cdr lst1) (cdr lst2)))))

;; Compose a list of functions to a single function.
;; Each function in the list takes a single parameter.
;; Handles the typical case of two functions manually to achieve
;; better efficiency.
;; .precondition f-list is a proper list of length ast least one.
(define (compose . f-list)
  (cond ((= 1 (length f-list)) (car f-list)) ;  ●
        ((= 2 (length f-list)) ;  ●
         (let ((f (car f-list))
               (g (cadr f-list)))
           (lambda (x) (f (g x)))))
        (else  (lambda (x)     ;  ●
                 ((car f-list)
                  ((apply compose (cdr f-list)) x))))))

;; Filter lst by pred.
(define (filter pred lst)
 (reverse ;  ●
  (filter-help pred lst '())))

; The function that does the real filtering.
; A tail recursive function
(define (filter-help pred lst res)
  (cond ((null? lst) res)
        ((pred (car lst))
           (filter-help pred (cdr lst)  (cons (car lst) res)))
        (else
           (filter-help pred (cdr lst)  res))))
```

Figure 6: *A snapshot of an elucidative Scheme program.*

The current version of Scheme Elucidator is not aware of name binding effects caused by use of syntactic abstractions (macros). However, the Scheme Elucidator is aware of the syntactic forms that introduce local name bindings (`define`, `lambda`, `let`, `let*`, and `letrec`).

4.5 Limitations and extensions

We have a number of ideas of future improvements of the Scheme Elucidator, some of which remedy weaknesses of the current version of the tool.

As noticed in Section 4.4, the Scheme Elucidator does not expand macros during the processing of Scheme source programs. It implies, for instance, that the Scheme Elucidator does not take definitions into account which are caused by macro expansion. As a remedy, it has been proposed that the Scheme Elucidator can be told about macros that expand into definitions, and how to extract the names defined by applications of such macros [24].

The Scheme Elucidator can refer to a particular version of a Scheme source file, typically the most recent version. During a long program development process it will often be useful to address the way the program is evolving, more specifically the differences between an early version and the current version of a program. We have clearly felt the need for such facilities in the Elucidative Program that documents the Scheme Elucidator itself (accessible via the accompanying web resource page [21]). Due to this reasoning, it would be relevant to include some support of versioning, at least in a way such that an early version of a program source file can be accessed in a flexible way.

The addressing scheme, realized as a relation between the entities of the documentation and the definitions in the source programs, is not perfect. In the current tool, it is only possible to address top level entities in the source programs. It would be desirable to be able to address local name bindings as well. The main price to be paid for this would be a more complicated addressing mechanism, and a potential additional burden on the documentation writer.

The Scheme Elucidator uses mutual navigation between the Documentation frame and the Program frame (see Figure 5) based on a bidirectional linking scheme. A literate program [9] presents program fragments within sections of the documentation. In a future development of the Scheme Elucidator we wish, as a supplementary means, to be able to extract program fragments from the source program and to inline these in the documentation. Such a facility is already supported by the Java Elucidator [31], and it resembles the extraction idea of L2T [23], which we briefly review in Section 4.6 of this paper.

The Scheme Elucidator supports a single monolithic documentation node, with two levels of sectioning (sections, and subsections which are called entries). As a minor and relatively easy extension, some programmers call for a more general hypertextual structuring of the documentation in multiple nodes. The Java Elucidator [22] supports multiple documentation nodes.

4.6 Similar work

A variety of work has been done for Scheme, which loosely can be categorized under the umbrella of Literate Programming. Most of this work is oriented towards printed output, typically via use of LaTeX.

SchemeWEB [25] is characterized as "a Unix filter that allows you to generate both Lisp and LaTeX code from one source file". As the novel aspect, SchemeWEB is able to identify Scheme (Lisp) expressions in a LaTeX text. A Scheme expression starts with a '(' at the beginning of a line, and it ends with the matching ')'. The text outside Scheme expressions is considered as documentation. The SchemeWEB tool provides for simple weaving, tangling, and untangling in the web sense [10] of these words.

STOL [11] is tool for presenting a Scheme Program as a LaTeX document. STOL is described as a Literate Programming Tool, and it uses specialized markup as well as LaTeX markup in ordinary Scheme comments. During processing, Scheme code is outputted unaltered, whereas the Scheme comments are transformed relative to specialized markup rules. STOL cannot control the orderinging of the program explanations relative to the ordering of the program constituents, and it is therefore somewhat misleading to call it a literate programming tool. STOL is like a SchemeDoc tool which presents the full source code.

L2T (Lisp to Tex) [23] is a literate programming tool created by Christian Queinnec. L2T is able to extract program fragments from Scheme source files and to insert them in a TeX context, which serves as a program essay. L2T allows the source programs and the documentation to be represented separately. Program fragments are extracted and inserted in the TeX document upon preprocessing of the TeX document with the Lisp2TeX tool. L2T has been used extensively by its author (and by others) for books and papers about Scheme programs.

Mole [12] is Kirill Lisovsky's system for analyzing, repositing, and presenting Scheme source programs. Mole recognizes chapters, sections and units of Scheme definitions. The analysis leads to an SXML [8, 7] representation of a Scheme program. A variety of different queries and extracts can easily be made on the basis of the SXML representation. The presentation, which is currently supported by Mole, is targeted at HTML. The presentation makes use of outlining for presentation of the programs and the program comments at various levels of abstraction.

5 Conclusions

A tool with the properties of SchemeDoc is essential for communication of library interfaces. LAML SchemeDoc supports extraction of distinguished documentation comments from Scheme source programs, and presentation of these as HTML documents. The separate SchemeDoc Indexing tool supports the indexing and organization of a set of SchemeDoc manuals in a 3-framed browser. As the most novel contribution, SchemeDoc is able to document XML DTDs. Due to the difficulties of reading many XML DTDs this is a valuable facility in its own right. However, the documentation of XML DTDs is of particular importance for LAML, because XML languages are represented as libraries of Scheme functions in LAML.

There is no common agreement on the conventions, formats, and the markup of documentation comments in Scheme. This has lead to a number of mutually incompatible tools, as discussed in Section 3.5. Based on this observation it might be worthwhile for the Scheme community to come up with a recommended format for documentation comments in Scheme source programs.

Seen from the standpoint of traditional program documentation, and in comparison with SchemeDoc, the Scheme Elucidator is a tool for documentation of internal aspects of a Scheme program. From a more open minded point of view, the Scheme Elucidator is a tool for *program exploration*. The exploration can be done within a single source file, between program source files (following chains of name usages both forward and backward), between the program files and the authored documentation, between a program and SchemeDoc interface documentation, and between the program and the Scheme Reference Manual. We find that the Scheme Elucidator is a valuable contribution whenever there is a need to write about Scheme programs, for tutorial, educational or scientific reasons.

Both LAML SchemeDoc and the Scheme Elucidator are bound to the LAML software package. Both tools make use of particular XML front-end languages, as well as XHTML in the back-end. All involved XML languages are represented as mirrors in Scheme. Due to the LAML connection, both tools can be used on all the platforms and Scheme Systems where LAML is running. Thus, in contrast to many similar tools (see Section 3.5 and 4.6) the tools discussed in this paper are not bound to any particular Scheme system. Whereas some other similar systems, such as Scribe [28], support multiple back-ends (and thus multiple target formats) our documentations tools can only generate HTML files.

LAML SchemeDoc has been indispensable for the documentation of LAML libraries (including the mirrors of XML languages in Scheme). The Scheme Elucidator 2 has been used by the author for writing a comprehensive LAML tutorial [19] which currently consists of seven elucidative program parts. The original Scheme Elucidator has also had external users.[2]

[2]See Anton van Straaten's documentation of "An Executable Implementation of the Denotational Semantics for Scheme" at `http://www.appsolutions.com/SchemeDS/ds.html`.

The Scheme documentation tools discussed in this paper can be downloaded as free software from the LAML home page [18]. The details reflected in this paper pertain to LAML version 25.10.

Acknowledgements

I wish to thank the reviewers for useful comments on the initial version of this paper.

6 References

[1] World Wide Web Consortium. Extensible markup language (XML) 1.0, February 1998. `http://www.w3.org/TR/-REC-xml`.

[2] Lisa Friendly. The design of distributed hyperlinked programming documentation. In Sylvain Frass, Franca Garzotto, Toms Isakowitz, Jocelyne Nanard, and Marc Nanard, editors, *Proceedings of the International Workshop on Hypermedia Design (IWHD'95), Montpellier, France*, 1995.

[3] Tony Garnock-Jones. Docscm documentation: version 0.1. Available via `http://homepages.kcbbs.gen.-nz/∼tonyg/chicken/`, September 2002.

[4] Aubrey Jaffer. Schmooz. `http://swissnet.ai.mit.-edu/∼jaffer/Docupage/schmooz.html`, 2002.

[5] Aubrey Jaffer. SLIB - the portable Scheme library version 2d3. `http://www-swiss.ai.mit.edu/∼jaffer/-slib.pdf`, 2002.

[6] Richard Kelsey, William Clinger, and Jonathan Rees. Revised^5 report on the algorithmic language Scheme. *Higher-Order and Symbolic Computation*, 11(1):7–105, August 1998.

[7] Oleg Kiselyov. SXML specification. *Sigplan Notices*, 37(6):52–58, June 2002. Also available from `http://-okmij.org/ftp/papers/SXML-paper.pdf`.

[8] Oleg Kiselyov and Kirill Lisovsky. XML, XPath, XSLT implementations as SXML, SXPath, and SXSLT. 2002. Presented on International Lisp Conference 2002 (ILC 2002). Available from `http://okmij.org/ftp/papers/SXs.pdf`.

[9] Donald E. Knuth. Literate programming. *The Computer Journal*, May 1984.

[10] Donald E. Knuth and Silvio Levy. *The CWEB System of Structured Documentation, Version 3.0*. Addison Wesley, 1993.

[11] Daniel Kobler and Daniel Hernández. STOL—literate programming in Scheme. *Lisp Pointers*, 5(4):21–30, October-December 1992.

[12] Kirill Lisovsky. Scheme program source code as a semistructured data. In *2nd Workshop on Scheme and Functional Programming*, September 2001. `http://kaolin.unice.fr/-Scheme2001/article/lisovsky.ps`.

[13] Kurt Nørmark. Elucidative Programming. *Nordic Journal of Computing*, 7(2):87–105, 2000.

[14] Kurt Nørmark. Requirements for an elucidative programming environment. In *Eight International Workshop on Program Comprehension*, pages 119–128. IEEE, June 2000. Also available via [16].

[15] Kurt Nørmark. Programmatic WWW authoring using Scheme and LAML. In *The proceedings of the Eleventh International World Wide Web Conference - The web engineering track*, May 2002. ISBN 1-880672-20-0. Available from `http://-www2002.org/CDROM/alternate/296/`.

[16] Kurt Nørmark. The Elucidative Programming home page, 2003. `http://www.cs.auc.dk/∼normark/-elucidative-programming/`.

[17] Kurt Nørmark. Web programming in Scheme with LAML. To appear in Journal of Functional Programming, April 2003. Available via [18].

[18] Kurt Nørmark. The LAML home page, 2004. `http://www.-cs.auc.dk/∼normark/laml/`.

[19] Kurt Nørmark. The LAML tutorial. Part of the LAML system, April 2004. Available via [18].

[20] Kurt Nørmark. The SchemeDoc home page, 2004. `http:-//www.cs.auc.dk/∼normark/schemedoc/`.

[21] Kurt Nørmark. Web resources of the current paper, August 2004. `http://www.cs.auc.dk/-∼normark/scheme/examples/elucidator-2/-scheme-documentation-tools`.

[22] Kurt Nørmark, Max Rydahl Andersen, Claus Nyhus Christensen, Vathanan Kumar, Søren Staun-Pedersen, and Kristian Lykkegaard Sørensen. Elucidative programming in Java. In *The Proceedings on the eighteenth annual international conference on Computer documentation (SIGDOC)*. ACM, September 2000.

[23] Christian Queinnec. L2T: a literate programming tool. Available via `http://www-spi.lip6.fr/∼queinnec/-WWW/l2t.html`.

[24] Matthias Radestock. Use of the Scheme Elucidator with SISC. personal correspondence, July 2003.

[25] John Ramsell. SchemeWEB. `http://www.tug.org/-tex-archive/web/schemeweb/`.

[26] Schematics SchemeDoc. `http://schematics.-sourceforge.net/schemedoc.html`.

[27] A Scheme documentation generator. Contained in `ftp://-ftp-sop.inria.fr/mimosa/fp/Bigloo/contribs/-scmdoc.tar.gz`, 1998.

[28] Manuel Serrano and Erick Gallesio. This is Scribe! Presented at the 'Third Workshop on Scheme and Functional Programming', October 2002. `http://www-sop.inria.fr/-mimosa/fp/Scribe/doc/scribe.html`.

[29] Sun Microsystems. Javadoc tool home page (sun microsystems). Available from `http://java.sun.com/products/-jdk/javadoc/index.html`, 2004.

[30] Dimitri van Heesch. Doxygen. `http://www.doxygen.org`, 2004.

[31] Thomas Vestdam. Generating consistent program tutorials. In *Proceedings of NWPER'2002 - Nordic Workshop on on Programming and Software Development Tools and Techniques*, 2002. Available via `http://dopu.cs.auc.dk/publications/`.

[32] Thomas Vestdam. Elucidative Programming in open integrated development environments for Java. In *Proceedings of the 2nd International Conference on the Principles and Practice of Programming in Java*, pages 49–54, June 2003. Available via `http://dopu.cs.auc.dk/publications/`.

A The elucidative program source

In this appendix we show the LAML source of the elucidative demo program, which we discussed in Section 4, and illustrated in Figure 6.

```
(load (string-append laml-dir "laml.scm"))
(laml-style "xml-in-laml/elucidator/elucidator")

(elucidator-front-matters

 'laml-resource "true"
 'scheme-report-version "r5rs"

 ; OVERALL attributes
 'table-of-contents "shallow" ; detailed or shallow
 'shallow-table-of-contents-columns "3"
 'detailed-table-of-contents-columns "2"
 'source-marker-presentation "image"  ; image, text, colored-text
 'source-marker-char "@"
 'browser-pixel-width "1100"
 'control-frame-pixel-height "120"

 ; INDEX attributes
 'cross-reference-index "aggregated"  ; per-letter, aggregated
 'defined-name-index "aggregated"     ; per-letter, aggregated

 ; PROGRAM attributes
 'initial-program-frame "blank" ; blank, first-source-file
 'large-font-source-file  "true"
 'small-font-source-file  "true"
 'default-source-file-font-size  "small"   ; small or large
 'program-menu "separate-frame"   ; inline-table, none, separate-frame
 'processing-mode "verbose"

 (color-scheme
   (color-entry 'group "doc" (predefined-color "documentation-background-color"))
   (color-entry 'group "index" (predefined-color "documentation-background-color"))
   (color-entry 'group "core" (predefined-color "program-background-color-1"))
   (color-entry 'group "others" (predefined-color "program-background-color-2"))
 )

 (source-files
   (program-source 'key "prog1"
                   'file-path "../../manual-xml-in-laml/scheme-documentation-tools/prog1.scm"
                   'group "core" 'process "true")
   (program-source 'key "prog2" 'file-path "src/prog2.scm"
                   'group "others" 'process "true")

   (manual-source  'key "laml-lib"
                   'file-path "../../../lib/man/general"
                    'url "../../../lib/man/general.html")
   (manual-source  'key "prog1-man"
                   'file-path "../../manual-xml-in-laml/scheme-documentation-tools/man/prog1"
                   'url "../../../manual-xml-in-laml/scheme-documentation-tools/man/prog1.html")
 )
)

(begin-documentation)

 (documentation-intro
    (doc-title "Elucidator Demo")
    (doc-author "Kurt Normark")
    (doc-affiliation "Aalborg University, Denmark")
    (doc-email "normark@cs.auc.dk")
    (doc-abstract
      (p "This is a brief demo example of an Elucidative Program")))
```

```
(documentation-section
  'id "overview-sect"
  (section-title "Overview")
  (section-body
    (p "This Elucidative program consists of two tiny source programs:"
       (weak-prog-ref 'file "prog1")"and" (weak-prog-ref 'file "prog2") _ "."
       "The first of these was used for illustration of SchemeDoc, in terms of"
       (weak-prog-ref 'file "prog1-man" "a prog1 interface manual")_ "." )
  )
)

(documentation-entry
  'id "intro"
  (entry-title "Introduction")
  (entry-body
    (p (strong-prog-ref 'file "prog1" "prog1.scm") "contains the factorial function"
       (weak-prog-ref 'name "fac")_ "," "and the two functions" (weak-prog-ref 'name "head") "and"
       (weak-prog-ref 'name "tail")_ "," "which are aliases of" (weak-prog-ref 'name "car") "and"
       (weak-prog-ref 'name "cdr")_ "." )

    (p (strong-prog-ref 'file "prog2" "prog2.scm")
       "shows a collection of classical higher-order functions," (weak-prog-ref 'name "compose") _
       "," (weak-prog-ref 'name "filter") _ "," "and" (weak-prog-ref 'name "zip") _ "." )
   )
)

(documentation-section
  'id "higher-order-sec"
  (section-title "Higher-order functions")
  (section-body
    (p "In this section we explain some details of the higher-order functions, namely"
       (weak-prog-ref 'name "filter")"and" (weak-prog-ref 'name "compose") _ "."
       "We do not go into the details of" (weak-prog-ref 'name "zip") _ "," "however." )
  )
)

(documentation-entry
  'id "filtering"
  (entry-title "Filtering")
  (entry-body
   (p "The function" (strong-prog-ref 'name "filter") "makes use of the tail-recursive function,"
      (strong-prog-ref 'name "filter-help")_ ","
      "which iteratively carries out the filtering. Due to use of iterative processing in"
      (weak-prog-ref 'name "filter-help")_ "," "we need to reverse the result in" (strong-prog-ref 'name "filter") _ ","
      (source-marker 'name "a")_ "." )
  )
)

(documentation-entry
  'id "composing"
  (entry-title "Function composition")
  (entry-body
   (p "Composition of two or more functions can be done by the function"
      (strong-prog-ref 'name "compose")_ "."
      "The function handles two special cases first, namely the trivial composition of a single
       function" (source-marker 'name "b") _ ","
      "the typical composition of two functions" (source-marker 'name "c") _ ","
      "and composition of more than two functions" (source-marker 'name "d") _ "." )
  )
)

(end-documentation)
```