

The Development of LAML - A Suite of Web Software for Scheme

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

May 17, 2002

Abstract

A collection of Scheme software for web authoring and programming is described. The software is called LAML. The paper gives a cohesive overview of LAML with emphasis on the lessons learned during its development. Both the LAML libraries, the document styles, the tools, and the LAML system as such are discussed. At the end of the paper we assess the use of Scheme for web authoring and programming purposes.

1 Introduction

In this paper we will describe LAML and its development. LAML represents a suite of software for web programming in Scheme.

LAML means *Lisp Abstracted Markup Language*. The name refers to the idea of forming abstractions in Lisp that mimic the elements of HTML [3] and XML [2].

In a number of other papers we have described some of the individual parts and tools of LAML [18, 20, 22, 19, 26, 27, 25]. In this paper we will give a transverse and more coherent description of the LAML software with emphasis on the lessons learned during the development process, which has lasted more than four years until now.

The work on LAML was initiated from a curiosity of applying Lisp for CGI programming purposes [6]. Both Common Lisp, Scheme, and Emacs Lisp were considered as candidate languages. We ended up using Scheme. In the conclusion of the paper we will reflect on this choice.

In the rest of the paper we will go through the individual LAML libraries, document styles, and tools. We will also describe the LAML distribution and the LAML installation procedure. Interleaved with the descriptions we will summarize the experiences we have collected in terms of a number of emphasized lessons.

2 LAML Libraries

The LAML libraries organize collections of definitions, some of which are generally useful, and others that are only relevant for more specialized purposes.

We have chosen to develop a *general function library* with a number of different extensions to the Scheme language. Many of the functions are functions that work on strings and lists, and functions that extract components of file paths. Somehow it feels wrong to re-program such functions for LAML, because several existing Scheme libraries already include a number of them. However, by including a general library in LAML we have managed to keep

```
(define (font size color x)
  (let ((ss (string-append "size = " (convert-size size)))
        (cs (string-append "color = "
                            (quote-string (apply rgb-string color) quote-string))))
    (string-append "<font " ss " " cs ">" (as-string x) "</font>")))
```

Figure 1: An example of function from the very early HTML mirror.

LAML running on most existing Scheme Systems without concerns of ‘third party’ libraries such as SLIB [9] and the PLT libraries [4] (see also section 7).

Lesson 1. LAML includes a general library of list and string functions (among others) primarily because Scheme is not standardized with comprehensive libraries.

The *library for reading and writing of text files* makes it possible to read a text file into a string and vice versa. Selective reading (based on a notion of marks) is also supported. The *time library* provides conversion functions for ‘Unix time’ (the number of seconds elapsed since January 1, 1970, 00:00:00). The *color library* supports functions that generates the hexadecimal RGB color number (which is used by HTML and CSS) from various decimal representations in different structures. The *text collection and skipping library* is the basis of several parsing tools, which play an important role both, not least internally for LAML (see section 5.3).

The *HTML mirror library* encodes a modelling of HTML in Scheme, and it supports the generation of HTML fragments from Scheme programs. This library and its descendants are described in some details in section 3.

The CGI libraries support CGI programming in Scheme. This involves *URL encoding and decoding* of characters in the so-called search part of an URL. The search part of an URL is the part following the question mark of an URL, and it represents ‘URL parameters’. We use association lists (lists of keyword values `cons` pairs) as a natural Lisp representation of these data. Similarly a decoding of form input using the POST method is supported by LAML.

The LAML CGI libraries also supports decoding of so-called *multipart encoded data*, which is used for uploading of files. The decoding of this format is more difficult to deal with than simple URL decoding. The reason is that fields are separated by unique markers which need to be identified while scanning through a stream of data. An uploaded file can be written into the web server’s file system as a side effect of multipart decoding.

The LAML *CGI libraries* offer a simple and straightforward support of CGI programming in Scheme at a relatively low level. As a contrast, it would be possible to build CGI support in Scheme at a higher level, similar to Meijer’s CGI library in Haskell [13]. The LAML CGI libraries are mature in the sense that they are the corner stones of several educational tools, which have been in real use for several years (see section 6).

Lesson 2. The LAML libraries have been developed to stay independent of specific and other non-standard Scheme libraries. The libraries have been designed conservatively, only demanding an R4RS Scheme system and a few non-standard and well-documented compatibility functions which access system resources.

3 HTML Mirrors in Scheme

Although the starting point of LAML was a number of libraries for server side programming using CGI, it soon became clear that especially the HTML library was useful for authoring of static web content as well. We use the term *programmatically authoring* for such endeavors [25]. The original HTML library (now called *the ad hoc HTML library*) was hand made and incomplete compared with the vocabulary of HTML4. Functions were introduced when

Mirror name	HTML version	Based on text or tree structure	Attribute validation	Well-formedness guaranty	Full validation relative to DTD	Complete mirror
<i>Ad hoc HTML mirror</i>	HTML4	Text	No	Yes	No	No
<i>Systematic HTML4 mirror</i>	HTML4.0	Text	Yes	Yes	No	Yes
<i>Validating mirrors</i>	HTML4.01 and XHTML1.0	Tree structure	Yes	Yes	Yes	Yes

Figure 2: Three generations of the HTML mirrors in Scheme.

needed, with more or less arbitrary signatures (and often a fixed number of parameters). The function `font` in figure 1 is a good representative.

First of all the `font` function does not cover the generality of the `font` element (in terms of the available HTML attributes). This will easily lead to an almost endless number of variants of the function. Second, the textual contents of the `font` function need to be string appended to a single string, causing a lot of explicit string concatenation. Third, the use of string concatenation calls for heavy garbage collection when forming a large document.

Lesson 3. The mirror functions need to cover the full generality of corresponding markup elements. Also, the mirror functions must accept multiple expressions of string types the values of which should be implicitly concatenated by the mirror functions.

It soon became clear that a more systematic approach was needed. We have developed three generations of HTML libraries, the properties of which are summarized in figure 2. The use of these HTML libraries are described in other papers [25, 27, 26] and the interested readers should consult one of these for details ([26] is preferable). Here we will only give a brief account.

We use the metaphor of *mirroring the markup language in the programming language*. As an essen-

tial observation, the idea of mirroring makes HTML available as individual, named functions each with a very flexible parameter profile. The following expression

```
(a 'href "http://www.x.y"
  "X" "and"
  'target "new" (em "Y"))
```

which is rendered as

```
<a href = "http://www.x.y" target = "new">
  X and <em>Y</em>
</a>
```

serves as an example. The rather flexible parameter conventions of the mirror are described in details elsewhere [26, 27].

We find it important to represent the HTML mirror as functions, and not as macros. The reason is that macros cannot be passed as parameters to higher-order functions nor returned as the result from such functions. Using the LAML approach, HTML is available as first class objects “in the center” of the programming language.

Some authors do not think it is feasible to represent the textual content of a programmatic web document as quoted strings. The argument is that it is too complicated to split and join strings in order to embed a substring into a new element. We

have developed a set of Emacs editor commands which takes care of string splitting and embedding. The inverse commands are also supported.

Lesson 4. The textual content of a web document can be represented as quoted strings provided that the editing environment supports specialized commands to split, join, embed and unembed substrings.

The latest and most sophisticated HTML libraries in LAML are exact mirrors of a particular version of HTML or XHTML, with full knowledge of the attributes and the document content model (grammar) [27, 26]. The parameter profiles of the functions generalize the syntactic conventions of HTML, and they allow the Scheme programmer to work with first class attribute lists and contents list. As an example, the expression

```
(let ((contents-list (list "and" "Y"))
      (attr-list
       (list 'href "http://www.x.y"
             'target "new")))
  (a "X" contents-list attr-list))
```

is rendered as an HTML fragment which is identical to the one shown above.

Equally important, a validation process goes on behind the scene while synthesizing the HTML structures. The mirror functions work in terms of syntax trees as a contrast to use of simple string concatenation.

The set of HTML mirror functions does not make up an *embedded language* [1] in the sense that we need to write a new interpreter for it. Rather, the HTML vocabulary can be used side by side with all other Scheme functions. We have demonstrated that higher-order functions together with the HTML mirror functions form a very useful partnership [26]. As a simple example, the expression

```
(ol (map li "i1" "i2" "i3"))
```

is rendered as

```
<ol>
  <li>i1</li> <li>i2</li> <li>i3</li>
</ol>
```

Other examples (in the domain of table composition) are discussed in [26].

Lesson 5. The use of functions instead of macros is useful to provide for higher-order functional programming in the web domain. The mirror functions do not form a ‘closed embedded language’ in Scheme. Rather, the mirror manifests itself as individual functions.

The HTML mirror functions relies on ‘dynamic typing’ and inhomogeneous lists. With dynamic typing we identify the types of data at run time, and we use this type information to determine the meaning of the mirror functions.

The use of dynamic typing in Scheme is less safe than static typing, as used in many other functional programming languages. In some situations runtime errors are caused by the fact that we rely on dynamic typing. On the other hand, much of the flexibility of the mirrors is due to the quite relaxed use of types in Scheme. In many situations it has been possible to issue useful and domain specific error messages, as opposed to generic error messages as generated by the type checker of the compiler.

Lesson 6. The dynamic typing in Scheme has been important for the definition of the HTML mirror functions in LAML. Explicitly programmed error checking makes it possible to issue domain-specific error messages.

There is a relatively large body of literature about HTML modelling, type checking, and document validation using functional programming languages [32, 13, 14, 15, 30, 8, 33]. Most of the literature is related to Haskell. The reader should consult [26] for a detailed comparison of LAML with the related work referred above.

4 LAML Document Styles

We have used the various HTML libraries as the primitive building blocks to form *domain specific languages*, primarily for construction of educational materials. In the early attempts we formed a collection of functions with fixed, positional parame-

```

(manual-section
 (section-title "Section 1")
 (section-body "Text of first section")
 )

(manual-page 'f
 (title "f")
 (form
 '(f a b)
 )
 (description "Description of function f")
 (parameters
 (parameter "a" "Explanation of a")

 (parameter "b" "Explanation of g")
 )
 (pre-condition "What to ensure before f is called")
 (misc "What else to to say about f")
 )

(make-manual)

```

Figure 3: An example of `manual-section` and `manual-page` clauses.

ter profiles, hereby replicating the problems identified with the early HTML libraries (see section 3). More recently we have introduced the idea of *XML-in-LAML* [27] with which we can automatically generate a generalized XML mirror in Scheme from an XML document type definition (DTD). Like the latest generation of the HTML mirrors, such XML mirrors are validating in relation to the DTDs. Most validation predicates can be created automatically from the DTD. Some content models, however, still demand explicitly programmed validation predicates. As an illustration, we had to write 4 validation predicates for XHTML1.0 strict [3].

Lesson 7. It is straightforward to form domain specific languages by means of a set of functional abstractions. The use of fixed, positional parameters in such abstractions makes them hard to maintain. The introduction of the XML conventions in Scheme via XML-in-LAML contributes with comprehensive error checking, and validation of the documents. This kind of checking adds much value to the domain specific languages of LAML.

We will now describe and discuss the major domain specific languages in the LAML package. We call these *document styles*.

4.1 The manual document style

The manual document style is used to document the programmatic interfaces of libraries and document styles. The manual document style is used in concert with the SchemeDoc tool (see section 5.1) which is able to extract interface documentation from comments of Scheme source files.

Figure 3 shows a small example of a LAML manual document. `manual-section` and `manual-page` collect and add the manual information to a list structure, which is rendered as HTML by the procedure `make-manual`.

A manual is rendered as a number of parts: (1) An introduction, (2) an overview of the manual sections, (3) an alphabetically sorted table of the interface definitions in manual, and (4) a sectioned listing of the manual entries in the same order as encountered in the manual document source (or in the order encountered by SchemeDoc in the Scheme source).

The individual manual clauses are defined by functions, many of which take a fixed number of parameters. In a forthcoming version of LAML the manual facility will be improved by an XML-in-LAML interface, based on an XML document type definition of the manual language.

4.2 The questionnaire document style

The questionnaire document style is a simple language which allows us to ask questions via web forms. The questionnaire document style is backed by a generic web server program, which accepts and organizes answers as files in a given directory structure.

Lesson 8. The questionnaire document styles makes it easy to manage questionnaires on the web. We use the questionnaire facility to harvest a few basic facts about people who download LAML. A number of educational applications are also in use.

4.3 LENO

The Lecture Notes document style—LENO—is the most substantial domain specific language developed within the LAML software package. During more than four years LENO has been developed hand in hand with various web-based teaching materials (on object-oriented programming in Java, and functional programming in Scheme). LENO is currently backed by more than 10.000 lines of Scheme program.

As a basic feature, LENO supports multiple views of a teaching material (slide view, annotated slide view, an aggregated view, and a so-called thematic view). It is possible to incorporate a sound track, which plays under control of an automatically progressing slide show. The interested reader is referred to separate papers about LENO for more information about the functionality of the system [22, 23].

LENO was originally crafted with an ad hoc syntax, using functions with simple and more or less arbitrary parameter profiles. More recently we have

re-engineered LENO to comply with XML, using the XML-in-LAML facility. With this, the LENO interface is automatically derived from an XML DTD (currently consisting of 76 different elements). With this interface we also achieved document checking and error reporting, via the XML validation. We only had to hand write 3 validation predicates - 60 predicates were automatically synthesized from the DTD (and the remaining 13 elements are so-called empty elements). The DTD also contributes with attribute and content model documentation, which is merged with the LENO manual document (made by the manual document style). In that way the LENO manual is updated each time we change the underlying DTD.

In order to stay backward compatible with existing LENO source documents we both support the original syntax and the XML-in-LAML syntax. This has been done by a systematic conversion of ‘new element structures’ to ‘old element structures’ (at the level of the internal document representations). We also support proper XML syntax at a more experimental level. It will be possible to operate LENO from XML, but the XML author will not be able to take advantage of the programmatic flexibility offered by LAML. The most sophisticated aspects of LENO will, however, also in the future rely on Scheme stuff.

Lesson 9. The LENO system has been used to produce substantial and successful teaching materials over a long period of time. The programmatic authoring approach has been the key to successful management of complex materials in several editions, with multiple overlapping views, and with heavy linking, both internally and to various external information sources.

5 LAML tools

In this section we will describe the most important LAML tools, each of which automate a well-defined task related to web work in Scheme. SchemeDoc is described first, next comes the Scheme Elucidator, various parsers and pretty printers, and finally the LAML Bibtex tool.

```

;; Return a value from an alist which corresponds to key.
;; In case the key does not exist in the alist, a fatal error will occur.
;; .parameter key is a symbol.
;; .parameter a-list an association list with symbols as keys.
;; .returns the first value of key in a-list.
;; .misc Uses the function assq (based on eq? for key comparisons) internally.
;; .internal-references "similar function" "defaulted-get"
(define (get key a-list)
  ...)

```

Figure 4: An example of a Scheme function which is documented by a two semicolon comment with internal tagging.

5.1 The SchemeDoc tool

The SchemeDoc tool extracts certain comments from a Scheme source file to a list structure. The list can be presented as a web page by the `make-manual` procedure of the manual document style (see section 4.1). Taken together, SchemeDoc and the manual document style are similar to the JavaDoc tool [5], which is used to produce interface documentation of classes in Java.

Comments in Scheme source files are classified by the number of semicolons applied in front of the text of the comment. ‘One semicolon comments’ are ignored by the tool. ‘Two semicolon comments’ in front of a `define` form is used for interface documentation of the definition. ‘Three semicolon comments’ are used to start a new logical section of definitions, and ‘four semicolon comments’ serve as an introduction (abstract) to a manual. There should only be a single ‘four semicolon comment’ in a Scheme source file, which should be located in the beginning of the file.

HTML markup can be used within the comments for simple typographical and linking purposes. In addition the SchemeDoc tool supports a special markup notation, which allows us in a more structural fashion to define the characteristics of a definition. Figure 4 shows an example. Besides the documentation tags shown in the example we support a `form`, `pre-condition`, `example`, and a `reference` tag.

Lesson 10. The programmatic interface of all LAML libraries and most document styles and tools are documented by use of SchemeDoc. The proximity between the actual pieces of program and the documentation promotes the consistency between the documentation and the program. SchemeDoc has been an essential tool for the description and dissemination of factual knowledge about LAML.

5.2 The Scheme Elucidator

The Scheme Elucidator is a tool that allows a programmer to *write about a program* in such a way that the writing is in close (navigational) proximity with the program. The Scheme elucidator can be used to maintain the understanding of a program at an internal level. As such it complements SchemeDoc and the Manual document style which produce program interface documentation. Elucidative programming is related to and inspired from Literate Programming [11, 10].

An elucidative program is presented in a two framed setup in a web browser. In one frame an augmented version of the program is shown (holding many internal links and links to the documentation). In the other frame the documentation is presented. The documentation can be authored programmatically in a LAML context, or it can be written in a special purpose markup language (designed with terseness and ease of use in mind). The processing of the elucidator (setup and parameters) is controlled by a LAML script.

The Scheme elucidator has been described in several papers [21, 20, 19, 31] and we will therefore not describe the tool in additional details in this paper. There also exists an elucidator for Java [28] which is programmed in Java, and as such it does not depend on LAML.

We have used the Scheme elucidator to maintain the understanding of critical and complex parts of the LAML software (among these, the elucidator itself). The tool has also been used for educational program explanations [29]. We are also using the elucidator as the delivery vehicle for the LAML tutorial [24] which currently is in progress. In the future, the LAML tutorial will be an integral part of the LAML distribution. We found that the Scheme elucidator is superior to other kinds of media, because of the extensive linking (in the program, between the documentation and the program, between the program and the Scheme report, and between the program and SchemeDoc interface documentation).

Lesson 11. The Scheme Elucidator is useful whenever there is a need to *write about a program*. The Scheme Elucidator provides extensive linking between the program and documentation (in a broad sense). Opposite to literate programming tools, the Elucidator does not require the program to be written in any special format. We have found that the Scheme Elucidator is particularly useful for Scheme program tutorials.

5.3 Parsers and pretty printers

The LAML system includes parsers for HTML, XML, and XML DTDs. The HTML and XML parsers are utilities which are only loosely related to LAML.

The DTD parser plays an important internal role in the creation of the HTML and XML mirrors. As already described in section 3 and 4 we parse the DTDs, expand parameter entities, and produce list representations of the DTDs. The information is used to automatically generate the mirrors, including most of the validation predicates. (Some of the complicated content models cannot yet be handled

by the DTD parser).

It should be noticed that the HTML and XML parsers are not yet of product quality. Among other things, the handling of white space needs to be rethought. The DTD parser is complete enough to parse and handle the HTML4.1 DTD, the XHTML DTDs, and the DTD of LENO. We plan to refine it slightly such that it covers all XML DTD constructs. Currently the DTD is for internal use only. This implies that the DTD parser is not part of the LAML distribution. The availability of DTD information as Lisp list data structures has proven useful for many different purposes.

All the parsers in LAML depend on the text *text collection and skipping library* (see section 2). This is a library which reads text from an open input stream through a look ahead buffer. Using this buffer it is possible to ask for the characters which are ahead relative to the current position of the input stream. It is also possible to unread characters. A number of high level reading primitives makes it relatively easy to parse XML, HTML and DTD files.

The HTML pretty printer plays a role in the rendering of LAML syntax trees. In most situations LAML users do not care about the HTML target code. In the situations where the HTML target code should be read by humans, it is possible for the author to produce pretty printed HTML code directly from LAML.

Lesson 12. The DTD parser and the HTML pretty printer play important roles for LAML components. The remaining parsers and pretty printers are useful as utilities for web programming purposes and beyond. The text collection and skipping library has proven its value in all the LAML parsers that have been written.

5.4 The LAML Bibtex tool

LAML provides a Bibtex [12] parser which converts a Bibtex file to a Lisp data structure. Like the other parsers described in section 5.3 the Bibtex parser relies on the text collection and skipping library. The parsed format is an association list of key value pairs

(symbols and strings respectively). The Bibtex tool also provides a Bibtex rendering facility in HTML.

We have used the Bibtex tool to facilitate the creation of publication lists in CVs. In addition, we have written several papers in HTML and XHTML with use of ad hoc LAML abstractions in the style of Latex, and with use of the LAML Bibtex tool to produce citations and the bibliography itself. In the future we may develop this to an XML-in-LAML complying document style.

Lesson 13. LAML can be used to make good use of existing structured data, such as Bibtex files. The availability of Bibtex records as a Lisp data structure makes it even more attractive to invest time and efforts in maintaining a good collection of bibliographic data.

6 Server based LAML systems

The CGI libraries of LAML have been used to produce several server based systems, mostly for educational purposes. All of these actually loads `lam1.scm` (see section 7), the necessary libraries (the CGI libraries, a HTML mirror library, and others) and the specific application program. The server based systems use the Apache web server on Solaris with PLT MzScheme.

The largest server based system built with LAML is a distance education environment known as *IDAFUS*. The system supports the dialogue between teachers, supervisors, and students in the setting of problem-based distance learning. The student and teacher interfaces are fully web based, and users of IDAFUS can customize the system in many ways. The system has been in use since 1999 serving several hundreds teachers and students (among which 125 have been active enough to profile the system to their own preferences).

Another successful server based web system is the interactive *LAML calendar system*, which features a multi-month calendar with morning and afternoon appointments. The calendar allows the user to have at least six months on screen at a time. The calendars are generated as static HTML files. The LAML calendar supports a rich CGI-based user interface of

the calendar maker, featuring a possibility of calendar merging. The merging of the calendars can be automated by means of a UNIX cron job which regenerates the calendars up to four times each day. Currently there are 44 active calendars (calendars that have been updated the last 30 days), among these semester calendars for a number of educations in the Computer Science Department at Aalborg University. The LAML calendar service is available from the LAML home page [17].

Lesson 14. Despite heavy loading of Scheme source files, the LAML CGI programs performs good enough to be useful for most practical purposes. As a limitation, which prevents scaling, we have until now stored application data on many small files (not in databases).

A group of master thesis students currently develop an Apache module for LAML, called SLAML [7]. Using preloading of common LAML libraries in the server they are able to reduce the execution times by approximately 50 percents for most of the CGI programs in IDAFUS. However, for programs which read many small data files the improvements are smaller.

7 The LAML System and its distribution

In order to use LAML it must be properly installed. The installation process runs a Scheme program which generates a number of files within the distribution. The installation process is based on properties defined in an association list called `configuration`, which must contain information about the location of LAML, which Scheme engine to use, the platform and operating system, and the location of the Emacs and the LAML startup files. In addition, the LAML installation process supports updating of the Emacs init file, `.emacs`.

LAML files can be executed in three different ways. First, it is possible to execute the file via a `lam1` command in the operating system. Second, it is possible to use the `lam1` procedure in a Scheme

system (a ‘Scheme prompt’). Third, a LAML file can be executed from Emacs via an Emacs LAML mode. The latter method is the most flexible and the most well-developed mode of activation. In any case, LAML needs some *context information* in order to execute. The context information consist of the name of the LAML source file and the full directory path to the source file. The context information is used to name and place the resulting HTML files. The three different modes of LAML activation are responsible of capturing and passing the context information to the underlying Scheme program.

In an installed LAML system the file `laml.scm` plays an important role, both for generation of static HTML files and for server based CGI applications. The file defines a number of fundamental LAML functions and procedures, among those the `laml` procedure mentioned above. It also loads a selected *Scheme compatibility file* and the general LAML library (see section 2). The Scheme compatibility file depends on both the platform, the operating system, and the Scheme system. The Scheme compatibility file must define the functions `current-time`, `sort-list`, `file-exists?`, `delete-file`, `directory-exists?`, and `copy-file`. Preferably, the procedures `make-directory-in-directory`, `mail` (a mail sending procedure), `directory-list`, and `get-env` (access to OS environment variables) should also be defined. Finally, the compatibility file should define procedures that extracts and sets the LAML context information. All LAML programs can be executed in a standard R4RS compliant Scheme engine with use of a proper compatibility library.

New versions of LAML are distributed approximately twice a year. A LAML distribution comes in two different variants: The *full distribution* which includes all relevant documentation, and a much smaller *slim distribution* which relies on documentation from the LAML web site.

The LAML distributions are extracted automatically from the development version by means of an Emacs utility called SDC (selective directory copying). SDC could be based on Scheme as well, but Emacs was chosen because the file system can be

accessed more easily from Emacs Lisp than from Scheme. SDC copies a directory structure under control of SDC files, in which we can specify the set of files in the full and the slim distributions.

Lesson 15. The LAML installation process calls for execution of a Scheme program which reads a configuration description. This process demands some basic understanding of what it takes to run a Scheme program, and as such it is less user friendly than a typical Windows installation programs. We have stressed the fact that LAML is independent of both platform, operating system, and Scheme engine. In order to accommodate the needs of different users, we provide a wide range of possible modes of LAML activation.

8 Conclusions

It is hardly surprising that Scheme can be used for programming of web server solutions via the CGI. More interesting, we have found that it is attractive to use Scheme for elaboration of more static web materials. We call the discipline involved for *programmatically authoring*. In this paper we have given an overview of the suite of LAML software, and we have described its development over the last four years.

We have written a number of server-based systems with CGI, but the LAML distributions are primarily oriented towards domain-specific web languages and tools for generation of static web content. The programmatic approach is useful for *power users* who want to use the ultimate (conceptual) tool to deal with complexity: Programmed abstractions.

Lesson 16. In the discipline of programmatic authoring the power of Scheme is available *anywhere in a document*, and at *any time during a web development process*.

One of the main weaknesses of Scheme, as experienced during the development of LAML, is the lack of standard access to system resources. In that respect Scheme falls short of contemporary languages (such as Java) and compared with functional pro-

gramming languages such as ML and Haskell, both of which have quite extensive standard libraries. Even the most simple manipulation of files in the operating system forces the Scheme programmer to use libraries that depend on a particular Scheme implementation. Also, access to databases is tricky and difficult.

As another weakness, the flat name space of a Scheme program (without support of a standard module concept) makes it difficult to manage large scale software development in Scheme. Every now and then, name clashes between different parts of libraries and applications are experienced.

In relation to LAML, the the first mentioned weakness has forced us to operate with a *compatibility library*, which must be implemented on the ground of procedures that depend on a particular implementation of Scheme. We would recommend that the Scheme community agrees on a Standard Scheme Library, in the same vein as the Scheme language is regulated through the Revised Reports [1].

As mentioned in the introduction, we have considered both Common Lisp, Emacs Lisp, and Scheme as possible vehicles for web programming and authoring in Lisp. Appart from the problems with access to standard system resources (see above) Scheme has proven to be a very good foundation of a *Lisp Abstracted Markup Language* (LAML). Both Common Lisp and Emacs Lisp—regarded as systems—are quite difficult to approach due their size. In addition, Scheme is widely available in many different implementations on all platforms.

Lesson 17. Scheme has been used with success for both dynamic and static web programming purposes in LAML. However, we would welcome a Standard Scheme Library and a Scheme module concept in the future evolution of the language.

As mentioned earlier in the paper, LAML is available as free software (under the GNU general public license) from the LAML home page [17].

References

- [1] H. Abelson, Sussman G.J., and Sussman J. *Structure and Interpretation of Computer Programs*. The MIT Press, 1985.
- [2] World Wide Web Consortium. Extensible markup language (XML) 1.0, February 1998. <http://www.w3.org/TR/REC-xml>.
- [3] World Wide Web Consortium. XHTML 1.0: The extensible hypertext markup language, January 2000. Available from <http://www.w3.org/TR/xhtml1/>.
- [4] Matthew Flatt. PLT MzScheme: Language manual. <http://www.cs.rice.edu/CS/PLT/packages/pdf/mzscheme.pdf>, August 2000.
- [5] 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 (IWH'D'95), Montpellier, France, 1995*.
- [6] Shishir Gundavaram. *CGI Programming on the World Wide Web*. O'Reilly and Associates, Inc., 1996.
- [7] Mikael Hansen, Paw Iversen, and Jimmy Juncker. SLAML - server side LAML. Preliminary master thesis report, January 2002. Available via [17].
- [8] Michael Hanus. High-level server side web scripting in Curry. In I.V. Ramakrishnan, editor, *Practical Aspects of Declarative Languages, LNCS 1990*, Lecture Notes in Computer Science, pages 76–92, Third International Symposium, PADL 2001, Las Vegas, Nevada, 2001. Springer Verlag.
- [9] Aubrey Jaffer. SLIB - the portable Scheme library version 2d3. <http://www-swiss.ai.mit.edu/~jaffer/slib.pdf>, 2002.
- [10] Donald E. Knuth. The WEB system of structured documentation. Technical Report STAN-CS-83-980, Department of Computer Science, Stanford University, September 1983.
- [11] Donald E. Knuth. Literate programming. *The Computer Journal*, May 1984.
- [12] Leslie Lamport. *Latex user's guide and reference manual*. Addison-Wesley Publishing Company, 1986.

- [13] Erik Meijer. Server side web scripting in Haskell. *Journal of functional programming*, 10(1):1–18, January 2000.
- [14] Erik Meijer and Mark Shields. Xmlλ - a functional language for constructing and manipulating XML documents. Submitted to USENIX Annual Technical Conference 2000, 2000. Available via <http://www.cse.ogi.edu/~mbs/pub/xmlambda/>.
- [15] Erik Meijer and Danny van Velzen. Haskell server pages - functional programming and the battle for the middle tier. In *Electronic Notes in Theoretical Computer Science 41, no. 1*. Elsevier Science B.V., 2001. Available via <http://www.elsevier.nl/locate/entcs/volume41.html>.
- [16] Kurt Nørmark. The Elucidative Programming Home Page, 1999. <http://www.cs.auc.dk/~normark/elucidative-programming/>.
- [17] Kurt Nørmark. The LAML home page, 1999. <http://www.cs.auc.dk/~normark/laml/>.
- [18] Kurt Nørmark. Programming World Wide Web Pages in Scheme. *Sigplan Notices*, 34(12):37–46, December 1999. Also available via [17].
- [19] Kurt Nørmark. Elucidative programming. *Nordic Journal of Computing*, 7(2):87–105, 2000.
- [20] Kurt Nørmark. An elucidative programming environment for Scheme. In *Proceedings of NWP-ER'2000 - Nordic Workshop on Programming Environment Research*, pages 109–126, May 2000. Also available via [16].
- [21] 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].
- [22] Kurt Nørmark. A suite of WWW-based tools for advanced course management. In *Proceedings of the 5th annual SIGCSE/SIGCUE Conference on Innovation and Technology in Computer Science Education*, pages 65–68. ACM Press, July 2000. Also available from <http://www.cs.auc.dk/~normark/laml/>.
- [23] Kurt Nørmark. Web based lecture notes - the LENO approach, November 2001. Available via [17].
- [24] Kurt Nørmark. The LAML tutorial. Part of the LAML system, April 2002. Also available via [17].
- [25] 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/>.
- [26] Kurt Nørmark. WEB programming in Scheme - the LAML approach. Submitted to *Journal of Functional Programming*, April 2002. Available via [17].
- [27] Kurt Nørmark. XML in LAML - Web programming in Scheme. Submitted to *PLAN-X: Programming Language Technologies for XML*, May 2002. Available via [17].
- [28] 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.
- [29] Kurt Nørmark and Thomas Vestdam. Elucidative programming in computer science education, November 2001. Available via [16].
- [30] Peter Thiemann. Modeling HTML in haskell. In E. Pontelli and V. Santos Costa, editors, *Practical Aspects of Declarative Languages, LNCC 1753*, Lecture Notes in Computer Science, pages 263 – 277, Second International Workshop, PADL 2000, Boston, MA, USA, 2000. Springer Verlag.
- [31] Thomas Vestdam and Kurt Nørmark. Aspects of internal program documentation - an elucidative perspective. In *10th International Workshop on Program Comprehension*. IEEE, June 2002. Also available via <http://dopu.cs.auc.dk>.
- [32] Malcolm Wallace and Colin Runciman. Haskell and XML: Generic combinators or type-based translation? In *Proceedings of the ACM SIGPLAN International Conference on functional programming*, pages 148–159, 1999. Published in *Sigplan Notices* vol 34 number 9.
- [33] Ulf Wiger. XMerL - interfacing XML and Erlang. Sixth International Erlang/OTP User Conference, October 2000. <http://www.erlang.se/euc/00/>.