XML in LAML - Web Programming in Scheme

Kurt Nørmark

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

Abstract. The LAML software package makes XML available in Scheme and the functional programming paradigm. The elements of an XML language are mirrored as functions in Scheme. The parameter profiles of the mirror functions is designed to be easily recognizable from an XML point of view, and to make a good fit with Scheme seen as a list processing language. The paper characterizes the mirrors by means of six mirror rules. A series of practical examples illustrate the approach. The XML-in-LAML facility supports systematic mirroring of XML languages to Scheme. The facility consists of a language independent part (common for all XML languages) and language dependent parts, which are generated from XML document type definitions (DTDs).

1 Introduction

Markup languages in the XML family are static in the sense that they miss a number of "dynamic qualities" which the programming language community takes for granted:

- Language extensibility - forming and implementing new concepts.
- Encapsulation of details - forming abstractions as a measure against growing complexity.
- Availability of basic computational power - such as arithmetic expressions and file input/output.
- Conditional branching - chosing among alternatives.
- Iteration - repeated computations, in part based on processing of data collections.

It could be argued that XML should be extended to accommodate some of these needs, but we do not think it is a good idea. The relative simplicity of the core XML framework seems to be a major asset, which already is threatened by the multiplicity of facilities that grow up around the XML core ideas.

We are working on an approach where XML languages are mirrored in Scheme. As the main goal, we go for a mirror that

- preserves the flavor of XML in the programs,

– fits well with the means of expressions in Scheme.

Using a mirror of XML side by side with Scheme provides a powerful partnership. The Scheme programmer will find that the XML vocabulary is available in a straightforward Scheme syntax. The XML author will have access to the full Scheme language at any point in his or her document, and at any time in a development process. This implies that many problems encountered during authoring of complex materials can be solved by programmatic means.

An XML mirror makes the elements of a markup language available as a set of functions in Scheme. We have organized the mirrors, and other related libraries and tools, in a software package called LAML (which means "Lisp Abstracted Markup Language"). We see a good fit between the nesting of descriptive markup elements [7] and the composition of expressions in a functional program. With this basis, many problems in the XML domain can be solved by means of solutions within the functional programming paradigm. Document validation can be dealt with by means of type checking, either statically (as part of compilation) or dynamically (when the program is executed). Using Scheme as the underlying programming language it is most natural to go for dynamic XML validation.

In section 2 we will describe the conventions and rules of the mirrors, and we will discuss a number of issues related to the rules. In section 3 we illustrate applications of the mirror functions in a series of small, practical examples. We identify a need for further systematic generalizations which leads us to describe the XML-in-LAML framework for mirroring of XML languages in Scheme. This is described in section 4. Similar work is pointed out in section 5, followed by the conclusions in section 6.

2 Mirror rules

The main idea in our approach is to mirror the elements of HTML or an XML language to a set of functions in the programming language Scheme. For each element of the markup language there is a corresponding function in Scheme, of the same name as the element.

As already mentioned in the introduction we go for a preservation of the HTML/XML flavor and a good fit with the means of expression in the programming language. Needless to say, this is a trade off, which comes with a certain price.

Basically and intuitively, the HTML/XML fragment

```
<tag a1 = "v1" ... am = "vm"> contents</tag>
```

corresponds to the Scheme form

```
(tag 'a1 "v1" ... 'am "vm" contents)
```

In the Scheme form `'a1` ... `'am` are symbols and `"v1"` ... `"vm"` are strings which together represent the HTML/XML attributes. The `contents` constituent represents zero, one or more contents elements in terms of strings (PCDATA) or activations of mirror functions (children).

The parameter conventions of the mirror functions are defined by 6 rules:

- ***Mirror rule 1.*** An attribute name is a symbol in Scheme, which must be followed by an expression of type string, which plays the role as the attribute's value.
- ***Mirror rule 2.*** Parameters which do not follow a symbol are content elements (strings or instances of elements).
- ***Mirror rule 3.*** All content elements are implicitly separated by white space.
- ***Mirror rule 4.*** A distinguished data object (the boolean value false) which we conveniently bind to a variable named _ suppresses white space at the location where the value appears.
- ***Mirror rule 5.*** Every place an attribute or a content element is accepted we also accept a list, the elements of which are processed recursively and unfolded into the result.
- ***Mirror rule 6.*** An attribute with the name "css:a" refers to the a attribute in CSS [1].

We use the dynamic types of data to distinguish between element contents and attributes. Attributes are accounted for by simulated keyword parameters. The exact interpretation of the parameters of the mirror function depends on the context. (A string is an attribute value if it precedes a symbol, else it is contribution to the textual contents). Rule number 4 allows us to handle white space issues in the surround of the contents parameters, rather than inside the contents. We find it better to ask for white space suppression than white space adding, because white space separation is more frequently occurring than 'dense concatenation'.

The Scheme mirror of the HTML/XML elements allows attribute-value pairs at arbitrary locations in an activation of a mirror function. Thus,

```
(a 'href "url" 'target "win" "anchor")
```

is equivalent with both of the forms

```
(a 'href "url" "anchor" 'target "win")
(a "anchor" 'href "url" 'target "win")
```

Due to rule number 5 it is possible to work with first class attribute lists. With this, the form

```
(let ((a-list (list 'href "url" 'target "win")))
  (a "anchor" a-list))
```

is also equivalent to the three forms shown above. This provides for definition of standard attribute lists of HTML elements such as `html`, `meta`, and `body`. Without rule number 5 it would be awkward to splice an attribute list into a Scheme mirror expression. This aspect is illustrated in a practical example which is discussed in section 3 and shown in appendix A.2.

The element contents can also be passed as a list rather than as individual parameters. The following gives a simple example:

```
(ul
 (map (compose li p) list-of-strings))
```

This detail of the mirror is crucial to make a good fit between Scheme (as a functional list processing language) and HTML/XML. The reason is that structured data, to appear in web documents, typically is represented in lists when we work in languages like Scheme. It is therefore important that the mirror functions accept lists of elements.

It is also important for our approach that the markup elements are mirrored as functions, and not as macros. The reason is that macros cannot play the same role as functions when applied together with higher-order functions. In the example above we compose the `li` and `p` functions (mirrors of the HTML `li` and `p` elements) to a function which is applied on every string in a list. If the variable `list-of-strings` is bound to the list `("xml" "in" "laml")` the expression is rendered as

```
<ul>
  <li><p>xml</p></li>
  <li><p>in</p></li>
  <li><p>laml</p></li>
</ul>
```

An HTML/XML mirror function returns an instance of an internal structure (a tagged list structure) which represents an XML syntax tree of a given document fragment. The syntax tree can be rendered as an HTML/XML string, using a function called `render`. Naive and simple versions of the rendering function would recursively aggregate a string by means of string concatenation. We use a more efficient traversal that either renders directly to an output port, or into a pre-allocated and fixed-sized string, segments of which can be concatenated if needed in the end of the rendering process.

The use of the HTML/XML mirror functions validates the document while constructing the internal document structure. If validation problems are encountered there will be warnings or a fatal error (depending the mode of processing). Well-formedness is assured by the Scheme syntax, which is less redundant than HTML/XML. (The Scheme syntax does not make use of end tags). The validation is done relative to the constraints defined by a document type definition (DTD). LAML provides an ad hoc DTD parser which outputs a list representation of the DTD in which all *parameter entities* (textual macros) are expanded. From this information it is relatively easy to synthesize the mirror functions.

The only real challenge in the mirror synthesis process is the document validation, which is done semi-automatically from the DTD. In the current version of LAML (version 17.20) XML element content-models of the forms

1. `EMPTY`
2. `(#PCDATA)`
3. `(x | y ... | z)*`
4. `(x | y ... | z)+`

5. `(x, y, ..., z)`
6. `(x?, y?, ..., z?)`
7. Mixes of 5 and 6, such as for instance `(x?, y, v?, w)`

automatically derive validation predicates. Element content-models of other forms are handled by manually written predicate. As an example, the generation of the XHTML strict/transitional/frameset mirrors [5] called for manual generation of only three validation predicates out of 67/77/78 non-empty elements (namely for `table`, `map`, `head`). In the LAML mirror of SVG [6] we need to write validation predicates for 30 elements out of 76 non-empty elements (this has not yet been done). The end goal is naturally to support a hundred percent automated mirror generation.

The document validation would be compromised if we allow the characters '<' and '>' within the textual contents. Instead of prohibiting these characters we transform them to the HTML character entities denoted by `<` and `>` respectively. The transformation is done by a systematic mapping of all characters through a *HTML character transformation table* which is useful for other purposes as well (such as transformation of national characters like 'æ', 'ø', and 'å to `&aelig;` `&oslash;`, and `&aring;` respectively).

Several people have argued against the passing of textual contents as quoted strings to Scheme functions [14]. The following serves as an example of the problem:

```
(p "A text with" (b "bold")
    "and" (em "emphasized") "words")
```

Using the LAML Emacs commands it is easy to produce this form from the raw string

```
"A text with bold an emphasized words".
```

First, nest the whole string in a `p` form (using the **nest** editor command). Next embed the substrings "bold" and "emphasized" in the `b` and `em` form respectively, using the **embed** editor command. The inverse commands **unnest** and **unembed** are also available in the LAML Emacs environment. Likewise, string splitting and string joining editor commands are supported.

A more detailed discussion of the mirror functions, and in particular additional examples of using the mirror functions together with higher-order functions, can be found in [21].

3 Examples

In this section we will illustrate our approach by means of a number of examples. All the examples are located in appendix A and in an on-line appendix on the web [17]. It is natural to start from the level of HTML (here using XHTML). Appendix A.1 shows an initial XHTML document and appendix A.2 illustrates a similar document written in LAML.

The LAML document shown in appendix A.2 uses XHTML mirror functions, such as `html`, `head`, and `meta`. The "standard attributes" of the `html` and `meta` elements are factored out in the list `html-props` and `meta-props`. The procedure `write-html` is a LAML procedure that renders the HTML document on a file of the same proper name as the source document. In addition `write-html` controls the printing mode (pretty printing or unformatted) and the inclusion of a document prologue (an XML declaration and a document type definition) and epilogue (in terms of comments).

We can of course use the basic mechanisms of the programming language, such as conditional filtering, name binding and definition of abstractions. With this, we can refine the LAML document from appendix A.2 to that of A.3. The document in appendix A.3 is interesting in the following respects:

- **Generalizations**: We have generalized the document to include declarative knowledge about the relevance of the individual items in the list (`programmer` and `general` relevance) and based on the global variable `view` the list of items is filtered appropriately.
- **Name bindings**: The three major parts of the document (`doc-header`, `doc-substance`, and `doc-trailer`) are defined in a `let*` name binding construct side by side with a couple of minor functions.
- **Ad hoc abstractions**: The functions `kn` and `normark-url` have been defined globally in the document. The first should be moved to the LAML init file, as it is useful in many of the author's LAML documents. The latter makes it easy to redirect some links of the document to another location.

When the document is brought into the domain of Scheme it is attractive to introduce new means of expression which can be seen as *domain-specific extensions* of the set of HTML elements. For illustrative purposes we define the new `laml-li-anchor` element, which is useful for enumeration of LAML related links. This is illustrated in appendix A.4.

In our work we have experienced good use of ad hoc abstractions, such as `normark-url`. The use of ad hoc abstractions contributes with a number of qualities:

- **Maintainability**. It becomes much easier to maintain a web document if repeated pieces are represented only once in the body of a function. In particular, this holds for URLs, such as represented in the function `normark-url`.
- **Redability and terseness**. The introduction of abstractions provides for shorter documents because many of the functions can be organized in reusable "convenience libraries". We recommend development of personal convenience libraries, where LAML users organize preferred document abstractions.

We have made substantial developments along this road [18]. Based on mirrors of HTML in Scheme a number of domain specific languages have been developed in terms of a set of new functions, which together make up the syntactic surface of the new languages. Execution of the program generates the underlying HTML document.

Based on our experience we have come to the conclusion that language extensions should be introduced with more care. We have made the following observations:

- **Uniform syntactic conventions**. The syntactical rules of language extensions should be compatible with rules of the HTML mirror functions (see section 2). In particular, simple positional parameter correspondence (as used in `laml-li-anchor`) is not appropriate. Also, use of attributes should be provided for in a systematic way.
- **High-level document validation**. It should be possible to validate the use of the language extensions in the same way as the document can be validated the HTML level.

Thus, development of web documents by 'free Scheme programming', and in particular use of arbitrary functions programmed in Scheme, is unwieldy. The observations from above have brought us to the conclusion that is worthwhile to more systematically introduce XML in LAML. With this we go for web documents which are tightly connected to an XML language, but still authored as a Scheme program. This is the subject of the next section of the paper.

4 XML in LAML

The XML-in-LAML tool generates a set of mirror functions from an XML document type definition (DTD). With this kind of mirroring, an XML language is made available as a set of Scheme functions. As described in section 2, the mirror functions offer flexible parameter passing conventions which fit well with the organization of data in lists. As a very important property, the mirror functions validate the XML document while synthesizing the internal document syntax tree. Validation problems can be reported as warnings, or as fatal errors (at the programmers discretion).

In a typical setup, the mirror functions of an XML language synthesize an internal syntax tree which is transformed to HTML. Mirror functions at the outer level may be associated with *action procedures*. Action procedures are supposed to initiate a transformation of the XML syntax tree. A mirror of the element `e` calls an action procedure named `e!`. Besides doing the appropriate actions, the action procedures return the syntax trees.

The transformed document typically makes use of HTML mirror functions, which in turn produce an HTML syntax tree. The HTML syntax tree is finally rendered as text and written to files. At both the XML level and the HTML level we can make use of ad hoc abstractions in terms of plain Scheme functions. As argued in section 3, the use of such functions often makes it much easier to maintain the documents.

Two XML languages *overlap* if one or more elements are defined in both languages using the same element name. XML handles the problem of overlapping languages by means *name spaces* [2]. LAML needs to handle the problem such that a number of overlapping XML languages can coexist as mirrors in LAML.

Scheme is a language with a flat name space at top level (without a package or module concept) and as such Scheme does not offer an immediate and easy solution to the problem. We could solve the problem with systematic use of *name prefixing*. With this solution, we would address the mirrors of a `title` elements in `lang1` and `lang2` as `lang1:title` and `lang2:title` respectively. We do not want to impose this naming scheme uniformly because, in practice, the set of overlapping element names is typically small, and as such, uniform name prefixing would impose an unreasonable burden on the programmer. In addition it would clutter the LAML documents.

In the XML-in-LAML framework we use the following solution:

- **Simple naming.** As the basic rule, we bind the mirror functions to simple names, hereby introducing the kind ambiguity described above.
- **Protection of the Scheme name space.** In case of a conflict between a name of a mirror function an a Scheme name (syntax or library procedure name) the mirror function is not available via a simple name.[1]
- **Language maps.** All mirror functions are available via lookup in a *language map*, which maps symbols to mirror function objects. Using the language map, `(lang1 'title)` returns the `title` mirror function in `lang1`, and `(lang2 'title)` the title mirror function of `lang2`.
- **Detection of language overlaps.** If we apply an ambiguous mirror function via a simple name, the XML-in-LAML framework will issue an error message. The error message may be a warning or a fatal error, depending on the processing mode.

We want to preserve the simple nature of XML-in-LAML and take care of the exceptions at run time on a well-informed background. The following shows a typical use of the language map:

```
(let ((lang1:title (lang1 'title))
      (lang2:title (lang2 'title)))
  (...
    (lang1:title "t1")  ...
    (lang2:title "t2")
  )
)
```

The XML-in-LAML software is organized in a language independent part and language dependent parts. The language independent part is a LAML library called `xml-in-laml`. The language dependent parts are generated from document DTDs of individual XML languages, including a subset of the necessary content validation predicates (as already discussed in section 2). In the situations where we cannot automatically synthesize a validation predicate for an XML element, the predicate must be written in a 'runtime file' which is included in the language dependent part of the mirror.

[1] We can choose to extend the handling of name conflicts to cover also names of important LAML functions and procedures.

Besides the generation of the mirror functions and the validation predicates, some of the information in the document type definition is valuable for documentation purposes. Consequently, the LAML manual document style [18], which is used for generation of most of the LAML software documentation, can extract and merge DTD information about attributes and the content models with other manual properties.

Using the XML-in-LAML framework, we start a linguistic abstraction process by defining a new document type definition (DTD). The DTD gives rise to a mirror of the XML language in Scheme. The resulting mirror uses the mirror rules, which we described in section 2. The transformation of the XML-in-LAML document (typically to HTML) is initiated in the action procedures.

This approach gives much better documents than the ad hoc approach illustrated in section 3. First of all, the language is syntactically controlled and well-defined via a grammatical framework (the DTD). Second, the derivation of the mirror functions ensure use of uniform parameter conventions across all language constructs. This stands as a contrast to a language defined by arbitrary functions. Third, the document validation provides for a rather comprehensive syntactic error checking, which is difficult and error prone to program on an ad hoc basis.

As the primary use of XML-in-LAML until now, we have made a new programmatic front-end of LENO [16], which is a non-trivial, web-based lecture note system in LAML. The LENO DTD currently has 80 XML elements, of which 70 are non-empty. Of these 70 elements only three elements need manually written validation predicates. The new front-end of LENO has proven to be a major improvement compared to the old ad hoc front-end, not least because of the use of XML-style attributes instead of positional or 'rest parameters'. The XML validation is also a major step forward compared to ad hoc parameter checking (at run time) in the old LENO interface.

We plan to use XML-in-LAML systematically in the future for most of the domain specific languages (document styles) in the LAML software package [18].

5 Similar Work

Many developers in the XML community rely on XSLT [3] for document processing purposes. XSLT is a pattern-based programming language, developed for transformations of XML documents. XSLT is an XML language itself. The main points of interest relative the work described in this paper are the following:

- **Special-purpose or general-purpose language?** XSLT is a special-purpose language which supports concepts that fit well with the typical transformation tasks to be solved. Scheme is a general purpose, multi-paradigmatic language with solid roots in the functional paradigm. It is likely that XSLT will need to include more and more general-purpose aspects. On the other side, it may be attractive if the specialized pattern matching functionality of XSLT can be accommodated in Scheme and LAML.

- **New or old language?** XSLT is a relatively new language which is still being developed. Of that reason, XSLT is still immature in a number of areas. Scheme is an old language, which is well-proven in a number of areas, but still relatively unproven in the area of web development.
- **Uni-linguistic or multi-linguistic approach?** XSLT represents a uni-linguistic approach to web-development because only a single language framework is in use for document representation and document processing (namely XML). LAML also represents a uni-linguistic approach, because both document and document processing is expressed in Scheme. It is worth noticing that many other approaches are multi-paradigmatic, because they involve both an XML language and a programming language. This is especially the case in the domain of web server applications.

The PLT developers use XML as an example in their discussion of little languages and their programming environments [4]. XML is embedded into Scheme, hereby giving rise to a language called S-XML. Like Scheme, S-XML uses parenthesized Lisp syntax. The embedding of X-SML in Scheme is done by means of a macro `xml`, which serves as a bridge between Scheme and S-XML. A special construct of S-XML called `lmx` is an espace mechanism back to Scheme.

XML-in-LAML is a more tight integration of XML in Scheme than S-XML. The reason is that the XML vocabulary is made available as functions in LAML. We use the *mirroring* metaphor instead of *embedding*. Our approach provides for use of the XML mirror functions together with higher-order functions. Programming with higher-order functions is much more difficult using the S-XML approach. In addition, there is no need for the `lmx` escape in LAML. Scheme expressions and XML expressions (in LAML syntax) can be intermixed in arbitrary ways. This is probably the main asset of LAML compared with similar Scheme-based frameworks.

In a more recent development, the PLT group has described a 'transformation by example' approach [10], which relies on the experiences with high-level macros Scheme [9]. The 'transformation by example' work is based on an XML pattern matching language, along the lines of XSLT. In addition, the paper outlines some ideas about the representation and processing of XML data in Scheme.

Besides the PLT work, Scheme has been used for web programming purposes in Queinnec's work [22, 23], in BRL [11], and to some degree in Latte [8].

XML and web programming has also attracted the interest of other functional programming communities, not least that of Haskell. Wallace and Runciman discuss two different representations of XML documents in Haskell [25]. In addition Meijer and colleagues have in a number of papers dealt with aspects of web programming using Haskell [12–14]. Thiemann has also published related work [24]. We refer the reader to another paper [21] for a more detailed comparison of LAML and the related work in Haskell.

6 Conclusion

The Scheme mirrors of HTML4, XHTML1.0, and a number of other XML languages have been the backbone of a substantial amount of web software, developed by the author during the last four years. The software is divided between static generation of HTML-based materials, and server-side programs that rely on CGI.

We have found that it is attractive to use Scheme and LAML for *programmatic authoring* of non-trivial static materials. The ideas of programmatic authoring are described in additional details in [20]. We have found that LAML can be used as an alternative to authoring directly in HTML, or as an alternative to using a visually oriented WYSIWYG authoring tool. Using Scheme and LAML, the author is well-equipped to deal with almost arbitrary document complexity. Programmatic authoring with LAML will probably not have a broad appeal, however. The approach is only attractive to authors who prefer a programmatic approach to problem solving (and it is of primary interest to developers who care about functional programming in Scheme).

We have also found that Scheme and LAML is viable for programming of many web services. The loading time of application software, HTML/XML mirror libraries and CGI libraries (as Scheme source programs) does not prevent the software from being used for many practical purposes. Moreover, the use of Scheme provides for a very flexible development process, in which it is easy and realistic to carry out frequent changes in order to adapt the software to the expectations of the users.

The connection between the LAML and XML is important, because it binds web work in Scheme close to mainstream web work. Without this connection, LAML documents easily drift in more or less arbitrary directions. In principle, it is possible to write XML-in-LAML documents, such as most simple LENO documents, in pure XML. In reality, however, the use of XML-in-LAML gives so much flexibility and leverage that a Scheme programmer would never be attracted by solutions in pure XML.

LAML is available as free software (under the GNU general public license). LAML is supported by several Scheme systems (PLT, SCM, and Guile) on both Unix and Windows. The LAML home page [15] contains additional documents, including a LAML tutorial [19].

References

1. Bert Bos, Håkon Wium Lie, Chris Lilley, and Ian Jacobs. Cascading style sheets, level 2 CSS2 specification. Technical report, W3C, May 1998.
2. Time Bray, Dave Hollander, and Andrew Layman. Namespaces in XML. W3C recommendation in http://www.w3.org/TR/1999/REC-xml-names-19990114/, January 1999.
3. James Clark. XSL transformations (XSLT) version 1.0. W3C recommendation in http://www.w3.org/TR/xslt, November 1999.

4. John Clements, Paul T. Graunke, Shriram Krishnamurthi, and Matthias Felleisen. Little languages and their programming environments. In *Proceedings of Monterey Workshop 2001*, 2001. Available from http://www.cs.rice.edu/CS/PLT/-Publications/mw01-cgkf.pdf.
5. World Wide Web Consortium. XHTML 1.0: The extensible hypertext markup language, January 2000. Available from http://www.w3.org/TR/xhtml1/.
6. World Wide Web Consortium. Scalable vector graphics (svg) 1.0 specification, September 2001. Available from http://www.w3.org/TR/SVG/.
7. James H. Coombs, Allen H. Renear, and Steven J. DeRose. Markup systems and the future of scholarly text processing. *Communications of the ACM*, 30(11):933–947, November 1987.
8. Bob Glickstein. Latte—the language for transforming text, 1999. Located on http://www.latte.org/.
9. Eugene E. Kohlbecker Jr. *Syntactic Extensions in the Programming Language Lisp*. PhD thesis, Indiana University, August 1986. Tecnical Report no. 199.
10. Shriram Krishnamurthi, Kathryn E. Cray, and Paul T. Graunke. Transformation-by-example for XML. In E. Pontelli and V. Santos Costa, editors, *PADL 2000*, LNCS 1753, pages 249–262. Springer Verlag, 2000.
11. Bruce R. Lewis. BRL—a database-oriented language to embed in HTML and other markup, October 2000. Located on http://brl.sourceforge.net/.
12. Erik Meijer. Server side web scripting in Haskell. *Journal of functional programming*, 10(1):1–18, January 2000.
13. Erik Meijer and Mark Sheilds. Xmλ - 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/.
14. 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.
15. Kurt Nørmark. The LAML home page, 1999. `http://www.cs.auc.dk/∼normark/-laml/`.
16. Kurt Nørmark. Web based lecture notes - the LENO approach, November 2001. Available via [15].
17. Kurt Nørmark. A collection of LAML examples, 2002. WEB material available at `http://www.cs.auc.dk/∼normark/scheme/examples/plan-x/index.html`.
18. Kurt Nørmark. The development of LAML - a suite of web software for Scheme, May 2002. Available via [15].
19. Kurt Nørmark. The LAML tutorial. Part of the LAML system, April 2002. Also available via [15].
20. 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/.
21. Kurt Nørmark. WEB programming in Scheme - the LAML approach. Submitted to Journal of Functional Programming, April 2002. Available via [15].
22. Christian Queinnec. The influence of browsers on evaluators or, continuations to program web servers. In *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, pages 23–33. ACM Press, September 2000.
23. Christian Queinnec. Inverting back the inversion of control or, continuations versus page-centric programming. Technical Report Technical Report 7, LIP6, Université Paris 6, May 2001.

24. 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.
25. 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.

A Program examples

This appendix shows a number of LAML examples which are discussed in section 3 of the paper. All the examples are available in an accompanying on-line resource on the web [17].

A.1 An initial HTML document

In this section of the appendix we show an XHTML document, the LAML counterpart of which is shown in A.2.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
   <head>
      <meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1">
      <title>LAML Info</title>
   </head>
   <body>
      <h1>LAML Info</h1>
      <p>Here you find a number of links to LAML information:</p>
      <ul>
         <li><a href="http://www.cs.auc.dk/~normark/laml/">General LAML info </a>
         </li>
         <li>
            <a href="http://www.cs.auc.dk/~normark/laml/zip-distribution/">
               LAML download
            </a>
         </li>
         <li>
            <a href="http://www.cs.auc.dk/~normark/scheme/index.html">
               Info for programmers
            </a>
         </li>
      </ul>
      <p>
         Kurt Normark
         <br>
         <span style="background-color: aqua;">normark@cs.auc.dk</span>
      </p>
   </body>
</html>
```

A.2 A similar LAML document

The LAML program in this section is section is similar to the HTML document shown in section A.1.

```
(load (string-append laml-dir "laml.scm"))
(laml-style "simple-xhtml1.0-strict-validating")

(define meta-props
  (list 'http-equiv "Content-Type" 'content "text/html; charset=iso-8859-1"))

(define html-props (list 'xmlns "http://www.w3.org/1999/xhtml"))

(write-html '(pp prolog)
 (html html-props
  (head
   (meta meta-props)
   (title "First page"))
  (body
    (h1 "LAML Info")

    (p "Here you find a number of links to LAML information:")

    (ul
      (li (a 'href "http://www.cs.auc.dk/~normark/laml/"
             "General LAML info"))
      (li (a 'href "http://www.cs.auc.dk/~normark/laml/zip-distribution/"
             "LAML download"))
      (li (a 'href "http://www.cs.auc.dk/~normark/scheme/index.html"
             "Info for programmers"))
    )

    (p  "Kurt Normark" (br)
        (span 'css:background-color "aqua" "normark@cs.auc.dk")))))

(end-laml)
```

A.3 A programmatically refined document

The program shown below is a programmatically refined generalization of the program from section A.2.

```
(load (string-append laml-dir "laml.scm"))
(laml-style "simple-xhtml1.0-strict-validating")

(define meta-props
  (list 'http-equiv "Content-Type" 'content "text/html; charset=iso-8859-1"))

(define html-props (list 'xmlns "http://www.w3.org/1999/xhtml"))

(define (kn)
  (p "Kurt Normark" (br)
     (span 'css:background-color "aqua" "normark@cs.auc.dk")))

(define (normark-url suffix)
 (string-append "http://www.cs.auc.dk/~normark/" suffix))

(define view 'programmer)

(write-html '(pp prolog)
 (let* ((as-anchor (lambda (e) (a 'href (second e) (third e))))
        (entry list)
        (doc-header (h1 "LAML Info"))
        (doc-substance
         (div
          (p "Here you find a number of"
             (cond ((eq? view 'programmer) "programmer related")
                   ((eq? view 'general) "general")
                   (else "???"))
             "links to LAML information:")
          (ul
           (map (compose li as-anchor)
            (filter (lambda (e) (eq? (first e) view))
             (list
              (entry 'general (normark-url "laml/")
                     "General LAML info")
              (entry 'programmer (normark-url "laml/zip-distribution/")
                     "LAML download")
              (entry 'programmer (normark-url "scheme/index.html")
                     "Info for programmers")))))))
        (doc-trailer (kn)))

  (html html-props
    (head (meta meta-props) (title "Second page"))
    (body doc-header doc-substance doc-trailer))))

(end-laml)
```

A.4 Introducing a new means of expression

This program, which is a variation of the program from section A.2, illustrates the use of a domain-specific element, called `laml-li-anchor`.

```
(load (string-append laml-dir "laml.scm"))
(laml-style "simple-xhtml1.0-strict-validating")

(define meta-props
  (list 'http-equiv "Content-Type" 'content "text/html; charset=iso-8859-1"))

(define html-props (list 'xmlns "http://www.w3.org/1999/xhtml"))

(define (normark-url suffix)
 (string-append "http://www.cs.auc.dk/~normark/" suffix))

(define (laml-li-anchor user-relative-url anchor-text)
  (li (a 'href (normark-url user-relative-url) anchor-text)))

(write-html '(pp prolog)
 (html html-props
  (head
   (meta meta-props)
   (title "Third page"))
  (body
    (h1 "LAML Info")

    (p "Here you find a number of links to LAML information:")

    (ul
      (laml-li-anchor "laml/" "General LAML info")
      (laml-li-anchor "laml/zip-distribution/" "LAML download")
      (laml-li-anchor "scheme/index.html" "Info for programmers")
    )

    (p  "Kurt Normark" (br)
         (span 'css:background-color "aqua" "normark@cs.auc.dk")))))

(end-laml)
```