Programming World Wide Web Pages in Scheme

Kurt Nørmark
Department of Computer Science
Aalborg University
Fredrik Bajers Vej 7
DK-9220 Aalborg
Denmark
Email: normark@cs.auc.dk

Abstract

In this paper we will argue that pages on the World Wide Web can be made directly as programs in a functional programming language instead of through HTML or an HTML-based authoring tool. We use the Scheme programming language from the Lisp family for WWW page production. It is concluded that a Lisp language is an attractive direct vehicle for authoring of Internet material in the sense that the source of a WWW document becomes a Lisp program. Abstraction from details in the underlying markup language constitutes the main advantage in our approach. This is consistent with the expected advantage of introducing XML as a successor of HTML. In addition we find it useful to have the power of a high quality programming language available for automation of routine tasks during the authoring process.

1 Introduction

Today HTML [3] is the primary markup language which is used to present information on the World Wide Web (WWW).

HTML is a simple, non-extensible language in the SGML family. SGML is a meta language oriented towards definition of markup languages such as HTML, XML, and domain specific markup languages. XML [2] is an extensible markup language in which new markup elements can be defined. A new document type can be introduced by writing a grammar for it. The presentation of an XML document in a browser requires a presentation scheme (known as a style sheet). As of this writing, XML has not had much practical impact on the authoring of WWW material.

In this paper we will introduce and discuss a *Lisp Abstracted Markup Language* called LAML in which the markup elements are represented as functions in the Scheme programming language [17][1]. Using the LAML approach, the source of a WWW page is a Scheme program. By executing the Scheme program the LAML document is transformed to a lower level HTML equivalent.

There are many different ways to make collections of WWW pages. Many authors use a specialized "home page" editor, which often is a structure editor [18] for HTML with a friendly, WYSIWYG user interface. Others use standard text processing systems, such as Word, from which documents can be exported to HTML. Authors who are used to traditional text formatting systems (like TeX [13] or LaTeX [14]) often write HTML documents in a plain text editor. Translation systems, typically called *X*2html for a given kind of source *X*, are also around.

Text formatting languages like TeX and LaTeX support language extensibility in terms of so-

[1] The most up to date version of the Scheme report is defined in the *Revised*[5] *Report on the Algorithmic Language Scheme* which can be found in The Internet Scheme Repository [19].

called macro facilities. However, compared with the abstraction mechanisms found in full-fledged programming languages, these macro languages are primitive and often poorly designed. It would not be difficult to create similar ad hoc facilities for HTML, but it would hardly be desirable. In that respect the XML approach is a better alternative because it follows established SGML traditions.

In this paper we will argue that it is possible to use a programming language for markup purposes instead of using HTML or XML directly. Following our approach we mirror HTML in a programming language, such that the full expressibility of HTML becomes available from the programming language. We can hereby use the abstraction mechanisms of the programming language to provide for extensibility in the markup language. More specifically we can, by means of abstractions applied on HTML details, create layers of specialized notation well-suited for authoring of documents in particular domains. This is also the main goal of XML. However, by using a programming language based approach, some of the future advantages of XML can be utilized right away, instead of waiting for XML technologies to mature. When XML becomes widely available our efforts will not be waisted, because of the possibility to translate relatively easy between our proposed notations and XML.

The power of abstraction is the key benefit of using a full programming language for authoring of WWW material. However, there is also another reason. When we are dealing with the creation of complicated material, problem solving often involve some kind of programming. Trivial routine work (such as making trivial transformations on a list of text items, or inserting material from external files) can be eliminated by programming a solution instead of forcing the author to carry out repetitive, manual, and time consuming step-by-step solutions. Having the full power of a high quality programming language available "anywhere" and "anytime" is found to be a major advantage in the practical authoring process. This is not an observation limited to the authoring of WWW pages. The observation holds in general in any professional work situation where the computer is the primary tool.

In section 2 we will discuss which programming paradigms and which programming languages can be used in a programmatic approach to WWW authoring. Following that, in section 3, we give a concrete example in order to illustrate the LAML approach. In section 4 and 5 we discuss the two central issues, abstraction and automation. Finally in section 6 we give a brief overview of the LAML software package.

2 The choice of programming paradigm and language

We will now turn to the question of which programming paradigm and which programming language to use for WWW programming.

In this discussion it is useful to distinguish between creation of *static WWW pages* and *dynamic WWW pages*. In this context, a dynamic WWW page is produced by a program (using the CGI interface [7], for instance) instead of being fetched from a server. In other words, a dynamic WWW page is calculated at access time, not just extracted from a disk. The starting point of the calculation is a set of input, typically provided by entering data into some input form. The output is a WWW page which can be shown in a browser. As it appears, the functional programming paradigm fits perfectly in this set up: *Based on some input parameters an expression is evaluated, and the result is a text string which represents the output in HTML.*

Even though a static WWW page is stored on a server, it may often be beneficial to *generate* the page from a higher level description. The generation is not supposed to take place at document access time, but at an earlier point in time (closer to document creation time). As already pointed out, the high level description may use abstracted notation in terms of descriptive markup [5] developed to a specific domain of documents. The functional programming paradigm fits well also when we are interested in generated WWW pages: *By*

means of a function the high level document description is transformed to low level HTML representation of the document.

Today, the imperative programming paradigm is undoubtedly dominating in the areas where WWW pages are produced programmatically. As we have argued above, this is unfortunate. The concept of *expressions* is much more useful than the concept of *commands* when we model and process a text with descriptive markup. The nesting of markup elements is, in a natural way modelled by nested activations of functions in an expression. In the imperative paradigm, procedural abstractions cannot be combined in the same way. Using the imperative paradigm it is easy to program procedures that emit text with markup (by means of write statements). It is also possible for one such procedure to call another similar procedure. We do, however, not achieve an immediate and natural composition of programmed abstraction in the imperative paradigm, because procedure calls cannot be nested in the same way as function calls. This makes imperative programming less attractive for our purposes.

Like functional programming, the object-oriented programming paradigm is also attractive for representation of documents with descriptive markup. An object representation is, in general, more useful and versatile than an "expressional representation". Each kind of tag can be represented as a class, and each tagged element in a document will hereby be represented by an object. The power of specialization (inheritance) can be used conveniently to classify the tags, and the nested structure of a document can be modelled by aggregation. Functionality, such as transformation from high level to low level representations, can in a natural way be implemented as methods in the tag classes. The Document Object Model (DOM) [1], as proposed by the World Wide Web Consortium, is an example of an object-oriented, programmatic representation of WWW documents on the client side (as used in Internet browsers).

We will now turn to a discussion of particular programming language properties, which are important for programmatic authoring of WWW pages.

The syntactical properties of the programming language are important. We go for a solution where the source description of WWW documents follows the rules of a programming language. In other words, the WWW document sources *are* programs. If we use the programming language in such a way it is of primary importance that the language offers flexible syntactical solutions. Many programming languages fall short in that respect. Who can imagine Pascal, C, or Java used directly for authoring of textual documents? We are looking for a programming language with only few syntactical constraints. In addition, we look for languages with powerful abstraction mechanisms, including a mechanism for syntactic abstraction (in order to provide for ultimative language extensibility). Finally, we think that a language with an easy-to-parse syntax, without ambiguities at the syntactic level (requiring association and precedence rules) provides for the best solution.

If we go for a pure and clean functional programming language like Haskell [10; 9] some kinds of problem solving which involve the "imperative surround" (such as file output) becomes complicated. Therefore we are better off if we chose a functional programming language which also supports imperative mechanisms in terms of assignment, mutation of data structures, and not least plain file IO. In our opinion, a 90% functional solution, with 10% imperative "escape programming" at well defined and isolated program locations, is often an attractive alternative to a 100% imperative solution (or a solution where the imperative and functional paradigm are mixed arbitrarily).

Given these observations, a language in the Lisp family is an obvious choice. More specific, we chose the programming language Scheme which is a relatively small, but powerful Lisp language. Scheme is best regarded as a mixed paradigm programming language, with strong support of classic functional programming and plain imperative programming. However, Scheme lacks many of the hallmarks of contemporary functional program-

```
(html
  (string-append
    (head
      (title "Programming World Wide Web Pages in Scheme"))
    (body
     (string-append
       (h1 "Programming World Wide Web Pages in Scheme")
       (h2 (string-append "Kurt Normark" (br) "Aalborg University")) (p)

       (h2 "Abstract")
         "In this paper we will argue that pages on the World Wide Web can be
          made directly as programs in a functional programming language, instead
          of through HTML or an HTML-based authoring tool..."

       (h2 "Introduction")
       "HTML is widely known as the markup language used to represent information
        on the World Wide Web (WWW)...")

     'bgcolor (rgb-color 255 255 255)
     'text (rgb-color 0 0 0)
     'link (rgb-color 0 0 255)
     'vlink (rgb-color 255 0 255))))
```

Figure 1: *An example of a WWW page programmed in Scheme using LAML markup.*

ming languages, most notably lazy evaluation and pattern matching. Syntactically, Scheme uses parenthesized prefix notation, as all languages in the Lisp family. It turns out that the angle bracket idea of HTML/SGML and the parenthesis idea of Lisp are reasonably close to each other to provide for easy translations between the two of them.

Lisp and Scheme have been used by others for Internet programming purposes. Hickey, Norvig, and Anderson [8] describe a Java Implementation of Scheme (SILK) which allows for programming of applets in Scheme. In the SILK paper other similar works using Scheme are described.

3 Examples of LAML documents

As mentioned in the introduction we have made a mirror of all the HTML tags in Scheme. The HTML mirror constitutes the bottom layer of LAML. In this and the following section we will illustrate the LAML approach by concrete examples.

Each HTML or XML tag application, such as

```
<tag a1=v1 a2=v2> some text </tag>
```

where `ai=vi` is an attribute value pair, corresponds to the Scheme function call

```
(tag "some text" 'a1 v1 'a2 v2)
```

The parameter profile of this `tag` function (in which there is a single textual parameter and a trailing property list of attribute value pairs) is representative of our current HTML mirror in Scheme. As discussed in [16] other parameter profiles may be possible or even preferable.

Due to the uniform nature of HTML it is easy to generate a set of HTML mirror functions in Scheme from a list of HTML single tags and double tags. Given a list of double tags such as

```
(list 'a 'em 'ol 'table 'ul ...)
```

we make Scheme functions for the `a` tag, the `em` tag, and so on. In addition, these functions are bound to the same name as the tag. Thus, we get an `a` function, an `em` function, etc. Each Scheme function generates a text string containing the

```
<html>
  <head>
    <title>Programming World Wide Web Pages in Scheme</title>
  </head>
  <body bgcolor = "#ffffff" text = "#000000" link = "#0000ff" vlink = "#ff00ff">
    <h1>
      Programming World Wide Web Pages in Scheme
    </h1>

    <h2> Kurt Normark<br>Aalborg University </h2> <p>

    <h2>Abstract</h2>
    In this paper we will argue that pages on the World Wide Web can be
    made directly as programs in a functional programming language, instead
    of through HTML or an HTML-based authoring tool...

    <h2>Introduction</h2>
    HTML is widely known as the markup language used to represent information
    on the World Wide Web (WWW)...
  </body>
</html>
```

Figure 2: *The HTML counterpart of figure 1.*

similar HTML start and end tags. These are the bottom layer primitives of LAML.

Figure 1 shows an example of a simple WWW page programmed in Scheme.[2] The page corresponds to the HTML page shown in figure 2. This HTML page is also the output from the Scheme program in figure 1 (apart from the indentation, which has been made manually for figure 2).

There are a couple of relevant observations about the LAML source in correspondence to the HTML source. First, each text contribution in the LAML document must be represented as a string constant, such as `"Programming World Wide Web Pages in Scheme"`. Thus, in figure 1 we have a number of relatively small text strings which are passed as actual parameter to the Scheme functions. In the SGML family of markup languages the tags are inserted into an implicit string, which represents the whole document. This string is semi-constant in the sense that the HTML tags are evaluated and interpretated by the Internet browser. A semi-constant string corresponds to a quasi quoted list in Lisp as supported by the backquote facility, found in many Lisp Systems (for instance in Common Lisp [11] and Scheme).

In order to alleviate this string passing problem in Scheme we might want to support semi-constant strings with evaluated substrings. As an example consider the semi-constant string

```
"A text with a
 (a "link" 'href "subsection/sec1.html")
 to a (b "subsection")
"
```

instead of the string concatenation

```
(string-append
  "A text with a "
   (a "link" 'href "subsection/sec1.html")
   " to a " (b "subsection"))
```

This corresponds to the HTML expression

```
A text with a
<a href="subsection/sec1.html">link</a>
to a <b>subsection</b>
```

which is rendered as "A text with a link to a **subsection**" in most browsers.

[2] Using the LAML libraries as available from [15], the function names which correspond to HTML tags must all be prefixed with `html:` in order to avoid collision with existing Scheme names.

Semi-constant strings with evaluation of parenthesized sub-forms is difficult to implement in Scheme, at least with reasonable notational elegance. First notice that the semi-constant string above in reality is a sequence of subexpressions: The string `"A text with a (a "`, the symbol `link`, the string `" 'href "`, etc. As such the semi-constant string shown above does not make sense in Scheme. The underlying problem is that the string quote character both serves as string-begin and string-end. Without different characters for string-begin and string-end it is almost impossible to embed strings into other strings in such a way that the Lisp reader can parse them without ambiguities.

As a consequence of these observations we stick to explicit concatenation of strings. Some of the strings are constant, and others are stem from evaluation of LAML functions.

Compared to SGML-like markup, such a Scheme source cause aesthetic problems when reading. However, with respect to writing the expressions it is possible to find a good solution. The observation is here that the step from

```
"A string with a link to a subsection"
```

to

```
(string-append
  "A string with a link to a "
  (b "subsection"))
```

and further on to

```
(string-append
  "A string with a "
  (a "link" 'href "subsection/sec1.html")
  " to a " (b "subsection"))
```

can be carried out by roughly two editing commands on the substrings `"link"` and `"subsection"`. These commands embed the selected strings into Scheme forms, and they handle the splitting of the string into substrings together with insertion of the outer string concatenation form. We support such an editing command called `embed` via the LAML support package in the Emacs text editor.

It is important to notice that our end goal is not the Scheme expression in figure 1. If that was the case only little has been gained, and as we have seen above, a number of problems have been introduced. The Scheme form in figure 1 can, however, easily be abstracted to a higher level than HTML allows. Using Scheme as our basis, we will in the next section illustrate that it is easy to introduce Scheme functions which correspond to specialized XML-like tags. In our simple example from above the tags are oriented towards describing the structure of a scientific paper.

4 Higher level markup and abstraction

Figure 3 shows an abstracted version of the document in figure 1. As can be seen, we have introduced a few abstractions, such as `article` which is described in terms of a title, an author, an affiliation, an abstract, and a body. Further on, an article body consist of a number of sections. `Article`, `article-title`, `article-author`, and the other abstractions are programmed as Scheme functions. In the simple case, they just implement a translation to the details found in figure 1. In a more realistic case the functions would implement a more elaborate transformation (including, for instance, a table of contents and other standard article stuff). Figure 4 shows a simple implementation of the functions.

Both the HTML level description (in figure 1 and 2) and the abstract description (in figure 3) use descriptive markup. However, the abstracted description in figure 3 does not commit itself to any particular presentation details, which at least to some degree is the case in the HTML descriptions. It is, for instance, up to the implementor of the functions in figure 4 to decide on the article layout. In other words, the implementation of the abstraction is in control of the *article style*, and the source file itself contains a very clean, structural representation of the constituents of an article, without unnecessary details. If we at a later point in time want another layout we can achieve this by reimplementing just a few functions, and leave the source document unchanged. This idea is, as already pointed out in the introduction, also

```
(article
  (article-title "Programming World Wide Web Pages in Scheme")
  (article-author "Kurt Normark")
  (article-affiliation "Aalborg University")
  (article-abstract
     "In this paper we will argue that pages on the World Wide Web can be
      made directly as programs in a functional programming language, instead
      of through HTML or an HTML-based authoring tool..."
  )
  (article-body
    (section 1 "Introduction"
       "HTML is widely known as the markup language used to represent information
       on the World Wide Web (WWW)...")
    "..."))
```

Figure 3: *The document from figure 1 with high level LAML markup.*

the main asset of XML. However, XML relies on a document style to be interpreted at the "client side", in the browser. We go for a much earlier interpretation (generation) at the "server side", prior to document access.

Adjustment of the presentation style (such as font, color, and alignment) can be done by means of adjustment of HTML details in the functions which implement the high level LAML markup. Alternatively, the presentation style can be controlled by means of CSS (Cascading Style Sheet) [4] which is a language separate from HTML that controls a variety of style elements. We are currently investigating an integration of HTML and CSS into a common linguistic Scheme framework. In such a framework both HTML details and style details can be controlled by means of expressions in Scheme.

5 Automation of routine tasks

When we are dealing with authoring of a complex network of hypertext pages there are many routine tasks that can be automated by means of programmed solutions. When the markup tags are mirrored in a programming language, the full power of the programming language is available to help the author solve these routine tasks.

To be concrete, we will discuss a typical problem which often appears when we are making complex WWW pages. The example deals with inclusion of external textual material, such as quotations from external files or excerpts from com-

```
(define (article ttl autr aff abstr bd)
 (html
   (string-append
     (head
       (title ttl))
     (body
       (string-append
         (h1 ttl)
         (h2 (string-append autr (br) aff))
         abstr
         bd)))))

(define (article-title ttl) ttl)
(define (article-author autr) autr)
(define (article-affiliation affl) affl)

(define (article-abstract abstr)
  (string-append
    (h2 "Abstract") abstr))

(define (article-body . bd)
  (apply string-append bd))

(define (section n header bd)
  (string-append
     (h2 (string-append (as-string n) " " header))
     bd))
```

Figure 4: *A simple implementation of the Scheme functions used in figure 3.*

puter programs. The manual solution is to copy and paste the external material into the WWW pages. During the authoring process, we typically need to make many such copies, for instance when the quoted source has been changed. In an automated solution, the copying and insertion is programmed such that the external material is taken directly from the source when the LAML document is processed in order to generate the HTML representation of the material. The steps in this process can be automated by a function which extracts a specified substring from a text file

```
(read-text-file-between-marks
  file-path
  mark)
```

If we want to change the font or color of selected substrings in the extraction, this can be done with the combined form

```
(colorize-substrings
   (read-text-file-between-marks
     file-path
     mark)
   font-and-color-specification)
```

Both `read-text-file-between-marks` and `colorize-substrings` are existing functions in the LAML libraries.

It may be argued that the solution to the problem exemplified above should be provided by a rich and powerful authoring environment instead of asking the author to program his or her own solution. If the authoring environment provides a pre-programmed facility it is of course the most ideal solution. The problem is, however, that the amount of routine tasks showing up in the future is infinite, meaning that the solutions cannot all be programmed into a fixed tool. Therefore we conclude that automation of routine tasks via programming capabilities in the markup language is very attractive in a professional authoring environment. The LAML approach makes the Scheme programming language available to the author at any place in the document and at any time in the authoring process.

6 The LAML software package

Besides the mirror of HTML in Scheme we have implemented a number of useful libraries, document styles, and tools. Taken together, we refer to these as the LAML software package. The libraries and some of the tools are available as free software from the LAML home page on the Internet [15]. Here we will give a brief overview of the most interesting pieces of the LAML software.

Our own starting point was creation of CGI programs in Scheme, mainly as a reaction against the cryptic coding style found in many imperative CGI programs in Perl. It turns out that it is realistic and without notable delay to start a Scheme system, load some libraries, and run a Scheme program as the response to a CGI request. We support a CGI library, which in a simple way helps the programmer utilize the Common Gateway Interface (CGI) from Scheme.

Our HTML library supports a useful working repertoire of Scheme functions, which mirror a relatively arbitrary subset of HTML in the programming language. In particular, the HTML library contains a number of table functions which map list structures in Scheme to tables in HTML. These functions make it easy and convenient to work with HTML tables from Scheme. It became clear, however, that we needed a better and more complete support of HTML in Scheme. As a consequence, we also support a more low-level mirroring of HTML in Scheme. Based on a list of HTML tags we are able to generate the corresponding Scheme functions automatically. Given these functions, we have established a basis which allows us to construct *any* HTML document entirely in Scheme. People who want to support particular variants or versions of HTML can easily generate their own functions via a tool in the LAML software package.

In many contexts it is useful to be able to deal with time. We have implemented a relatively complete support of time in Scheme represented as the number of seconds elapsed since January 1, 1970. Basically, we can calculate back and forth between this representation and conventional dates

and times. On top of the time library we have made a WEB calendar, which can produce arbitrary calendars after 1970. Using the table functions mentioned above we are able to present conventional calendars covering at least half a year on most screens. It is possible to feed appointments into the calendars via a CGI interface programmed in Scheme.

A number of other libraries exist, such as a library supporting hexadecimal color encoding (as required by HTML) and a library which allows us to read and write text strings from and to text files. We also support selective reading of files, and superimposing of colors and fonts on text strings (as exemplified in the previous section).

The LENO lecture note system is the most substantial LAML document style written to date. In LENO it is possible to make annotated slides, which are organized in a number of lectures. Besides overviews and indexes we support three different views on the material: A conventional slide view which can be presented from an Internet brower in an auditorium, annotated slides, and a holistic presentation of a lecture in which all the information is aggregated into a single page. The slides and the annotations are linked automatically in a natural way. Besides these links, it is of course possible to link to any resource on the Internet. The availability of such well-organized, interlinked material turns out to be of great value in a lecturing situation. Because the material is transformed to pure HTML it is available from any browser, independent of special plugins.

The LAML software is documented by means of a tool, which extract certain comments from a Scheme programs and present these using a manual document style, much like the Javadoc tool [6] in the Java JDK toolset. Using this tool it is easy to produce up-to-date documentation of the more than 300 external functions in the LAML Scheme libraries.

Finally, we are working on a practical literate programming [12] system based on the LAML software packages. Using this system it is possible to produce internal documentation of Scheme programs as it is known from the literate programming WEB systems pioneered by Knuth. We call our variant of literate programming for *elucidative programming* because it emphasizes the description of potentially complicated internal program details by means of explanation. The Scheme Elucidator presents program and documentation in two vertical HTML frames with heavy linking in between the two of them.

7 Concluding remarks

In this paper we have introduced a novel application of the Scheme programming language. It has been demonstrated that a WWW page can be authored as a Scheme program. The Scheme program makes up the high-level document source, which by means of program execution is translated to HTML. The availability of high-quality abstraction mechanisms is the main asset of the LAML approach. The main drawback is the splitting of a document into many relatively small strings which are passed as parameters to Scheme functions. As we have seen, this causes some problems when reading an LAML source file, but not necessarily problems when we write the document in an advanced editor. Using the LAML approach we are approximating the potential of the forthcoming XML technology, but on a much more simple basis which can be used today. In addition, the possibility of automating routine work via integrated, programmed solutions turns out to be very useful.

Much of the LAML software is available as free software from the LAML home page on **http://www.cs.auc.dk/~normark/laml/**.

An accompanying paper called *Using Lisp as a Markup Language - The LAML approach* [16] describes our work with LAML at a more Lisp specific level.

References

[1] World Wide Web Consortium. Document object model (dom) level 1 specification, October 1998. http://www.w3.org/TR/REC-DOM-Level-1/.

[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. HTML 4.0 specification, April 1998. http://www.w3.org/TR/-REC-html40/.

[4] World Wide Web Consortium. Cascading style sheets, level 1, January 1999. http://-www.w3.org/TR/REC-CSS1.

[5] 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.

[6] 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.

[7] Shishir Gundavaram. *CGI Programming on the World Wide Web*. O'Reilly and Associates, Inc., 1996.

[8] Timothy J. Hickey, Peter Norvig, and Kenneth R. Anderson. Lisp - a language for internet scripting and programming. In *Proceedings of the lisp user group meeting*. Franz Inc., November 1998. http://www.franz.com/-elugm99/conference/past.html.

[9] Paul Hudak and Joseph H. Fasel. A gentle introduction to haskell. *ACM Sigplan Notices*, 27(5), May 1992.

[10] Simon Peyton Jones and John Hughes (editors). Haskell 98: A non-strict, purely functional language, February 1999. http://haskell.systemsz.-cs.yale.edu/onlinereport/.

[11] Guy L. Steele Jr. *Common Lisp, the language, 2nd Edition*. Digital Press, 1990.

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

[13] Donald E. Knuth. *The Texbook*. Addison-Wesley Publishing Company, 1984.

[14] Leslie Lamport. *Latex user's guide and reference manual*. Addison-Wesley Publishing Company, 1986.

[15] Kurt Nørmark. The LAML homepage. http:-//www.cs.auc.dk/∼normark/laml/, 1999.

[16] Kurt Nørmark. Using Lisp as a markup language—the LAML approach. 1999. To be presented at the European Lisp User Group Meeting, Amsterdam. Available via http://www.cs.-auc.dk/∼normark/laml/.

[17] J. Rees and W. Clinger. Revised^3 report on the algorithmic language Scheme. *Sigplan Notices*, 21(11), 1986.

[18] Gerd Szwillus and Lisa Neal. *Structure-Based editors and environments*. Academic Press, 1996.

[19] John Zuckerman. The internet Scheme repository. http://www.cs.indiana.edu/-scheme-repository/home.html.