Using Lisp as a Markup Language
The LAML Approach

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

Abstract

Lisp is widely known as an extremely versatile language. In this paper we will demonstrate that Lisp can be used as a powerful markup language for WWW authoring and for provision of information on the Internet. Using the LAML approach, as introduced in this paper, we write textual documents directly in Lisp; Function calls serve as applications of tags, as known from the SGML family of markup languages. The easy and uniform access to abstraction is the main advantage of using a programming language as a markup language. In addition, the direct availability of a powerful programming language allows the author to automate many trivial tasks in the writing process. We describe a systematic mirroring of HTML in Scheme, and we refer to a number of different document styles based on this mirroring.

1 Introduction

Authors and information providers using the Internet need to deal with programming at several different levels. We distinguish between four categories of Internet programming each oriented towards the presentation of WWW pages. The categories are displayed in figure 1. As can be seen from the figure we have arranged the categories on an axis according the degree of dynamics; dynamics is achieved by means of programming. The program execution time is the main distinction between the categories in the figure.

At one extreme we find the pure static approaches in which the information is written and frozen at *document creation time* using a markup language such as HTML [2]. No programming is involved in this process. At the other extreme we have the dynamic approaches in which the information keeps changing until *document read time*. The changes are mediated by a program which executes in a browser (such as a Java applet, or a Javascript program). In between these two extremes we have the two other categories called *generated* and *calculated* pages respectively.

Calculated pages are frozen at *document access time*. In this context document access refers to the act of retrieving the document from a server. CGI programming [6] is a typical and widespread representative of this category.

Using generation the information presented is created by a program which translates a high level source description to a low level target format (such as HTML). The result of the translation depends on both the source description (the input) and the translation program. The *document generation time* is quite naturally located in between the document creation time and the document access time. It is not intended that the generation should take place every time the document is accessed.

Our interest in Lisp as an Internet programming language started because we had an interest in the category of calculation. More specifically we

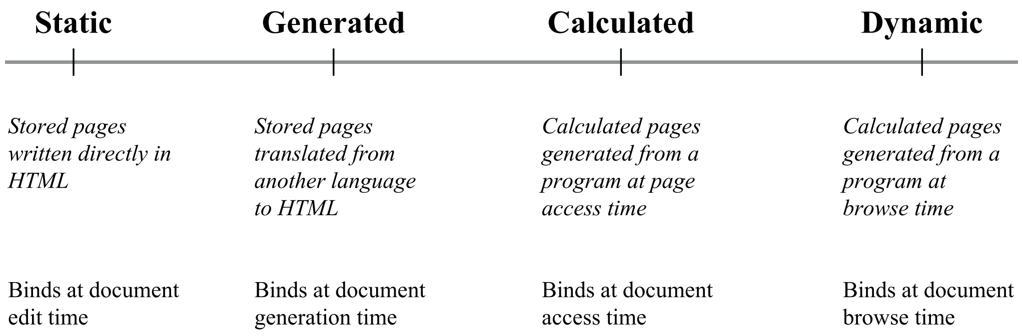

Figure 1: *Four categories of Internet Programming oriented towards presentation of WWW pages.*

started using Lisp as an applicative and (mainly) functional language for CGI programming purposes. We saw this as a natural contrast to the dominating use of imperative programming languages, such as Perl, C, and Shell script languages in this domain. It turns out that it is perfectly feasible and realistic to start a Lisp system, load some libraries and a program, and execute the program at document access time.

In this paper we will concentrate on the use of Lisp in the category of generation. More specifically we will investigate to what degree Lisp can be used as an alternative markup language for authoring of information on the World Wide Web (WWW). In the paper we will explore a number of advantages of a generative approach based on a high quality and powerful programming language like Lisp.

It is interesting to notice that Lisp also has been used in the dynamic category. Hickey el al. [7] describe SILK, which is a Scheme implementation done in Java. Using SILK it is possible to write Applets in Scheme. Other similar systems in this category (Skij, Kawa, and JaJa) are described in this reference.

In the rest of the paper we are going to explore the use of Lisp within the category of generation. In section 2 we will describe the advantages of the generative approach in comparison with the static approach. In particular, we will address the advantages of using an applicative, functional programming language for these purposes. In section 3 we will discuss the differences and similarities between programming languages and markup languages. In section 4 we go on with a detailed examination of the use of Lisp as a markup language, which can replace direct use of HTML or XML for markup purposes. In this section we introduce the idea of a *Lisp Abstracted Markup Language* (LAML). Finally, in section 5 we present a brief overview of the LAML applications crated to date by the author of this paper.

Examples of LAML generated WWW pages are available from the LAML home page [10].

2 Generation of WWW pages using a functional language

Generation of WWW pages involves a transformation from a high level source description to a lower level target description. We will at the outset concentrate on a situation where the target language is HTML.

Generation of HTML pages from a high level description gives two substantial advantages compared with the purely static approach:

1. **Abstraction**. By means of abstraction it is possible isolate the WWW page author from the details of HTML. This makes it possible to take the source description to a higher and more abstract level, at which a number of decisions can be handled in the implementation of the abstractions. Changes in a document, as it appears in a browser window, can either

be performed by changing the high level description or by changing the implementation of the abstractions which carry out the translation to HTML.

2. **Automation**. In the writing process the author has to deal with many time consuming manual tasks, which have to be repeated again and again. One of the key ways to better author performance is to automate some of these tasks by means of programmed solutions. Because of the never ending demands from new kinds of routine tasks we need to make programming capabilities directly available to the author.

If we consider the use of HTML today, the idea of abstraction is already represented at several different levels. In order to avoid unnecessary bindings to particular means of presentating information in an HTML document it is possible to use style sheets, such as supported by CSS [3].

XML [1] is an interesting and much more powerful alternative than HTML. XML provides for abstraction in terms of high-level description of documents. The presentation of an XML document in a browser requires a presentation scheme (known as a style sheet). XML is believed to be the future markup language for the Internet. As of today, however, XML has almost no practical impact on authoring of WWW material.

It can be observed that neither CSS nor XML provides any support of automation in the sense that we introduced it above.

Because of the transformational nature of the document generation it is natural to investigate the use of a functional programming language. As shown in the upper part of figure 2 the input will be a high level document description which by means of a function is translated to HTML.

The main idea behind our work is to use a functional programming language as the document source language. Thus, instead of inventing a new document source language, we intend to use an existing programming language as the document source language. This brings us to the situation shown in the lower part of figure 2. Quite naturally, we intend to apply the abstraction mechanisms of the programming language for document abstraction. More interesting, however, we can use programmed solutions, in a broad sense, to support automation. Because we use the programming language as the document source language, programmed solutions can be applied everywhere in the document, and at any time in the authoring process. This is a unique opportunity which we will discuss in more detail later in the paper.

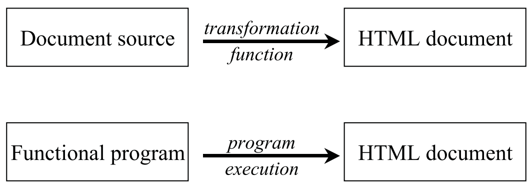

Figure 2: *Two possible transformation functions.*

The use of an existing programming language as a document source language disqualifies most programming languages because of the lack of lexical and syntactic flexibility. It is not easy to imagine classical languages such as Pascal or C, or a new language such as Java, in the role of a document source language used for markup purposes. As we will see in section 4 it is possible and attractive—although not without problems—to use Lisp because of the flexible and unique syntactic properties of the language.

As an additional argument in favor of our approach we can notice that using a programming language as the document source language brings the category of generated and calculated Internet pages closer together. The program, which servers as the document source in the generative approach, can without very many changes be used as a CGI program in the calculated category, cf. figure 1.

3 Markup languages versus programming languages

Markup denotes "detailed stylistic instructions written on a manuscript that is to be typeset"[1]. It is possible to distinguish between a number of

[1]Quote from The American Heritage Dictionary of the English Language.

```
<slide id = "LispAndMarkup">
  <title>
     Using Lisp as a Markup Language
  </title>

  <conceptList>
    <concept name = "Lisp">
        Lisp is multi paradigm programming language ...
    </concept>

    <concept name "Markup language">
       A markup language is ....
    </concept>
  </conceptList>

  <image location = "C:/images/img1.gif" caption = "Overview of ..."> </image>

  <point>
    The main point is that ...
  </point>
</slide>
```

Figure 3: *A fragment of text with XML-like descriptive markup.*

different markup systems [4]. However, the most interesting and promising kind of markup is *descriptive markup*. Using descriptive markup an author specifies the *kinds* and *roles* of individual parts of a text, in a declarative way. Using other kinds of markup the author focuses directly on the *appearance* and *formatting* of the marked text elements.

HTML [2] is a simple, non-extensible markup language, which partly supports descriptive markup and partly a more presentation specific kind of markup. XML [1] is strictly oriented towards descriptive markup. Furthermore, XML allows definition of domain specific markup. Both HTML and XML are in the SGML family of markup languages.

In this section we will focus on the differences between a markup language and a programming language. Figure 3 shows a typical text fragment with XML-like markup. The example shows an outline of a slide document with a title, a definition of two concepts, an image, and a point. The kinds of tags have to be defined in a Document Type Definition (DTD) which is a grammar-like notation (not shown here). If we want a presentation of the document in a browser, a style sheet is needed.

There is clearly a similarity between function calls and tag applications. The XML attributes (such as the id, name, and location in figure 3) and the textual content in between the start tag and end tag correspond to actual parameters. The attributes are passed as keyword parameters (as supported by Common Lisp [8] for instance). The textual contents in between start and end tags is a rather special, but essential parameter to an XML tag.

The lexical basis of a text document with SGML/XML markup is different from the lexical basis of a program in a programming language. A text document with markup is a (long) text string in which certain tokens are inserted which need interpretation during the various kind of processing of the document. *The textual document hosts the markup.* A program is structured entirely of well defined tokens, some of which may be quoted text strings. *The program hosts the text strings as well as literals of other types.*

The tags are defined at the syntactic level in XML via a grammatical formalism known as DTDs. There is no integrated semantics involved. For presentation purposes it is possible to define

```
(slide
 (string-append
   (title "Using Lisp as a Markup Language")

   (conceptList
     (string-append
        (concept "Lisp is a multi paradigm programming language..." 'name "Lisp")
        (concept "A markup language is ..." 'name "Markup language")))

   (image "" 'location "C:/images/img1.gif" 'caption "Overview of ...")

   (point "The main point is ..."))
 'id "LispAndMarkup")
```

Figure 4: *The XML document fragment from figure 3 translated to a Lisp form.*

a style sheet, but this is a matter which is external to the XML language. There is no immediate programming capability available in the language which allows for automation of routine tasks when constructing or processing the document.

We will in the following section discuss how to deal with textual markup, along the example given above, in a Lisp programming language.

4 Using Lisp as a markup language

We will now discuss how Lisp can be used as a markup language. We start with a Lisp version of the XML example shown in figure 3 together with a discussion of a number of syntactic variations of the Lisp calling form of tagging functions. In section 4.2 we continue with a discussion of semi-constant strings. In section 4.3 we describe a systematic mirroring of HTML into tagging functions in Scheme. Finally, in section 4.4 we discuss the more semantic perspectives of using Lisp as a markup language.

In the rest of this paper we will use the Scheme [11] dialect of Lisp.[2]

[2] The latest version of Scheme is defined in *Revised*^{5} *Report on the Algorithmic Language Scheme* which is available from the Internet Scheme Repository [13].

4.1 An example with syntactic variations

The example shown in figure 3 can be described lexically and syntactically in Lisp. When we use Lisp for markup purposes we talk about a *Lisp Abstracted Markup Language* (LAML). To show a concrete example to start with, figure 4 shows a possible LAML counterpart to figure 3.

There are several ways to pass the textual contents and the attributes as parameters to a Scheme function. Figure 5 outlines the most relevant of these. The issues behind the variations are the following:

1. **The sequencing between attributes and the textual contents.**
   If we are going to mirror XML, the attributes should come before the text contents.

2. **The nesting of lists in the calling form.**
   Explicitly given formal parameters in the function signature implies relatively deep nesting in the calling form.

3. **The division of text into several substrings.**
   If we need to pass the textual contents as a single parameter explicit string concatenation is needed.

The example in figure 4 uses variation number 3 from figure 5. We clearly see the consequences of explicit string concatenation, and the figure also

	Calling form	Signature
1	`(f (list 'a1 v1 'a2 v2) (string-append "Some text" "Some more text"))`	`(define (f attributes text-contents ) ...)`
2	`(f (list 'a1 v1 'a2 v2) "Some text" "more text")`	`(define (f attributes . text-contents ) ...)`
3	`(f (string-append "Some text" "Some more text") 'a1 v1 'a2 v2)`	`(define (f text-contents . attributes) ...)`
4	`(f 'a1 v1 'a2 v2 "Some text" "more text" )`	`(define (f . parameters) ...)`

Figure 5: *Four syntactic variations of tagging functions.*

illustrates the problem of separating the function "tag" name and attributes (it is difficult to figure out which attributes belong to which tag).

Using variation number 1 we would need to pass empty attribute lists to many tag functions; This is not attractive. Variation number 2 suffers from the same problem, but the problem in item 3 from above is alleviated. Variation number 4 is quite satisfactory from a calling point of view, but it is rather difficult to figure out the role of each of the actual parameters (parameter correspondence). If we use this variation we need to rely on the type of each of the actual parameters in order to locate the attributes and the textual contents strings.

Some readers may argue that the discussion above is of less importance because it just reflects minor syntactic issues. However, our choice regarding syntax *is* important because it affects the appearance and the "convenience of writing" of potentially large amounts of text.

The third issue mentioned above (the division of text into several substrings) is not solved satisfactorily in any of the variations in figure 5. Let us take a look at the Lisp counterpart to the following XML/HTML example[3]

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

Using variation number 3 we get:

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

Variation number 4 avoids the `string-append` form:

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

In either of the Lisp variations we see that the implicit hosting string in the XML/HTML version needs to be split into a number of substrings, which must be passed individually to the relevant Lisp functions. In a writing process, in which we need to introduce an anchor or a textual emphasis (such as in the example above) this *string splitting* comes in very awkwardly.

4.2 Semi-constant strings

We have attempted a number of solutions to the problem given above. First, take a look at the following:

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

The intention here is to embed a number of strings within each other, and more important, to allow for evaluation of Lisp forms within a string. We could talk about a *semi-constant string* with evaluated substrings. This corresponds to quasi quoted lists in Lisp as supported by the backquote facility found in many Lisp Systems (for instance

[3]The example will be rendered as "A text with a link to a **subsection**" in most browsers.

in Common Lisp [8] and Scheme). The semi constant string should, of course, be processed such that the `a` and the `b` functions are called. This processing could be called for uniformly in the tagging functions, such as `point`.

It is almost impossible to implement semi constant-strings in Scheme. The reason is that strings do not nest easily. The Scheme reader does not return a string with nested strings, but rather a strange sequence of forms, many of which do not make sense. In the example from above, we get the following sequence of sub-expressions, each of which will be evaluated:

- the string `"A text with a (a "`
- the symbol `subsection/sec1.html`
- the string `" "`
- the symbol `link`
- the string `") to a (b "`
- the symbol `subsection`
- the string `")"`

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 the string without ambiguities. As an illustration of the problem, imagine if we used the same character as both "start parenthesis" and "end parenthesis" in Lisp. It would have made life much easier if "begin quote" and "end quote" were two different ASCII characters.

We gave up the idea of dealing with semi-constant strings in Scheme, and we now stick to a fragmentation of the hosting string into separate substrings. Compared to SGML-like markup, the Scheme source causes aesthetic problems when reading. However, with respect to writing the expressions, it is possible to find relatively good solutions. The observation is here that the step from

```
(point
  "A string with a link to a subsection")
```

to

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

```
(define (generate-tag-function tag-name)
  (lambda (contents . attributes)
    (itag tag-name contents attributes)))

(define (itag name contents attributes)
 (if (null? attributes)
  (string-append "<" (as-string name) ">"
                 (as-string contents)
                 "</" name ">")
  (let ((html-attributes
         (linearize-attributes attributes)))
   (string-append
     "<" (as-string name) " " html-attributes ">"
     (as-string contents)
     "</" name ">"))))
```

Figure 6: *Two of the central Scheme functions which implement the mirroring of HTML in Scheme.*

and further on to

```
(point
  (string-append
    "A string with a "
    (a "subsection/sec.html" "link")
    " to a " (b "subsection")))
```

can be carried out by basically two editing commands on the substrings "link" and "subsection", which embed the selected strings into an LAML form. These editing commands handle the splitting of the string into substrings, and (if necessary) the outer string concatenation form. We support such an editing command in the Emacs text editor on buffers with LAML documents.

4.3 Mirroring HTML in Scheme

In order to establish a bottom layer on which to build higher level abstractions, we have mirrored HTML[4] in Scheme.

The starting point of this task was a list of HTML tags, separated into single tags and double tags. Given these we generate, on a textual basis, a sequence of define forms, such as

```
(define html:a (generate-tag-function "a"))
```

[4] Our concrete starting point was the collection of HTML tags described in the book *WEB-master in a Nutshell* [12]. It would be easy to mirror HTML version 3 or 4 [2] in a similar way.

for the `a` anchor tag in HTML. The function `generate-tag-function` is a higher-order function, the definition of which is shown in figure 6 together with a helping function. In the current implementation we do not check the grammatic rules (valid syntactic composition of HTML forms into each other, and the legality of the attributes). However, it would be easy to add such checks based on the Data Type Definition of HTML (as found in [2]).

Besides this systematic mirroring of HTML in Scheme we have implemented a number of very useful Scheme functions which generate HTML fragments on an ad hoc basis. The `table` function is a good example. The expression

```
(table
  (list
    (map html:b (list "Col 1" "Col 2" "Col 3"))
    (list "El 1" "El 2" "El 3")
    (list "El 4" "El 5" "El 6")))
```

generates a table with three rows, the first of which contains bold faced column headers (see figure 7). The flexible use of lists in Lisp makes it very easy and convenient to make various tables in LAML.

4.4 Semantic perspectives

Given the basic set of LAML tagging functions we are able to define *document styles* in Scheme. In this context a document style is a collection of Scheme functions which defines the LAML markup elements for documents in a particular domain.

The key idea behind a document style is that of *abstraction*. Instead of dealing with low level markup (ala HTML) we encapsulate the details into functions that transform from a high level document language to a low level language (HTML) which can be presented in a browser. The high level document language uses the syntactic conventions of Lisp, or more specifically one of the syntactic variations illustrated in figure 5.

It is our experience that it is easy to define new LAML document styles. Using LAML it is possible to bring the ideas of XML to practical use today within the category of generated WWW pages (cf. figure 1).

Col 1	**Col 2**	**Col 3**
El 1	El 2	El 3
El 4	El 5	El 6

Figure 7: *A simple table produced via LAML.*

We see *automation of routine tasks* as a particular aspect of abstraction. The work we have in mind is, for instance

- inclusion af materials from external files
- checking the targets of all links
- escaping the lexical markup such that we, e.g., are able to show literal HTML markup elements in a browser.

By using the LAML framework we are able to program Scheme functions which automate each of these tasks, and almost any other routine tasks which may be useful in future writing projects.

5 Status and conclusions

We have defined LAML document styles that support authoring of

- course home pages,
- lecture notes (slides, annotated slides, and aggregated slides),
- simple scientific papers,
- software library manuals (similar to Javadoc [5]), and
- literate Scheme programs using a variant of literate programming called *elucidative programming*.

We have used the LAML lecture notes style to author more than 400 annotated slides for a course on object-oriented programming. The manual style has been used to document most of the LAML software at the programmatic interface level. In addition we made a WWW based calendar tool and a distance education tool (via CGI programming in Scheme) both using the LAML approach.

It is our experience that the advantages in the areas of abstraction and automation (as discussed in section 4.4) outweigh the syntactic difficulties of using Lisp for markup of textual documents (as discussed in section 4.1 and 4.2).

Given the current state of the software, LAML authoring is probably most attractive to Scheme programmers. This is mainly because systematic, high level error handling has not yet been implemented in the various document styles. Also, in order to take advantage of automation of routine tasks, the author must be able to define Scheme functions by himself or herself. However, by utilizing advanced LAML-oriented editor commands (as available in Emacs) it is our judgement that the LAML approach can used by many authors, who otherwise would write directly in HTML or XML (at least when the software is brought to a state where high level error reporting is supported).

Examples of documents generated via LAML tools, and a subset of the LAML software is available from the LAML home page on **http://www.cs.auc.dk/~normark/laml/**.

An accompanying paper called *Programming World Wide Web Pages in Scheme* [9] describes our work with LAML at a more general, and less Lisp specific level.

References

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

[2] World Wide Web Consortium. HTML 4.0 specification, April 1998. http://www.w3.org/TR/-REC-html40/.

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

[4] 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.

[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 (IWHD'95), Montpellier, France*, 1995.

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

[7] 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.

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

[9] Kurt Nørmark. Programming World Wide Web Pages in scheme. 1999. Submitted to ACM Sigplan Notices. Available via http://www.cs.auc.-dk/~normark/laml/.

[10] Kurt Nørmark. The WWW home page of the LAML project. http://www.cs.auc.dk/-~normark/laml/, 1999.

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

[12] Stephen Spainhour and Valerie Quercia. *Webmaster in a nutshell: A desktop quick reference*. O'Reilly and Associates, 1996.

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