AALBORG UNIVERSITY
INSTITUTE OF ELECTRONIC SYSTEMS
DEPARTMENT OF COMPUTER SCIENCE

Title: POWER - **P**rogramming **O**riented **W**eb **E**ngineE**R**ing

Main Subjects:
- Programming Oriented Web Page Authoring
- Abstracted Markup Language

Semester:
F10D

Project Period:
February 2, 1999 – June 29, 2000

Author:
Carsten Hellegaard

Supervisor:
Kurt Nørmark

Number of pages:
97

Number of copies:
6

Abstract:

Creating large and complex hypertext document systems for the World Wide Web requires tool support if we are to obtain an acceptable result. It is our belief that a creative development process leading to automatically generated page systems is the most desirable way of action.

Automated Web solutions have so far been hampered by the "page-at-a-time" approach of traditional authoring tools. Creation and maintenance of large Web systems is likely to involve a tedious and error-prone manual edition of every contained page, when working with WYSIWIGs and similar tools.

We introduce the programming oriented approach to Web authoring, where an ordinary programming language is used to generate HTML code. This method provides users with the powerful abstraction and automation mechanisms of the programming language in the authoring process.

This report presents the analysis, design and implementation of Lisp Abstracted Markup Language (LAML), which is a HTML generating authoring language based on a programming language of the LISP family, namely Scheme. The sample Web systems developed with LAML seem to prove the legitimacy of the programming oriented approach to Web authoring.

Preface

This report is the printed result of our 10th semester project at the Department of Computer Science, Aalborg University. The work presented in this report is a continuation of the project initiated at our 9th semester, which has previously been described in [Hellegaard, 99]. Knowledge of this preliminary work may be of some benefit to the reader, but it is not a necessity as the most fundamental issues raised in [Hellegaard, 99] have also been included in this report (albeit it may be rewritten at some points).

All program examples described in this report and additional ones may be found on the accompanying CD (located on the inner sleeve at the back of the report) along with other useful stuff, such as Kurt Nørmarks newest LAML distribution, GNU-Emacs for Windows, Mz-Scheme, drScheme and more.

The figures presented in this report will be numbered in succession for each chapter. Figures and figure text might not always be self explanatory, but additional discussion will be found in the surrounding paragraphs.

Aalborg, June 2000

Carsten Hellegaard

Synopsis

For at opnå acceptable resultater ved konstruktion og vedligeholdelse af større World Wide Web hypertext systemer er det en nødvendighed at anvende stærke værktøjer. Vi er af den overbevisning, at en kreativ udviklingsproces som benytter en høj grad af automatisering repræsenterer den bedste mulighed for generering af komplekse Web systemer.

Traditionelle forfatningsværktøjer har indtil videre med deres enkeltsidet baserede metoder været med til at hæmme udviklingen af automatiserede Web løsninger. Med de såkaldte WYSIWIGs eller lignende værktøjer bliver udvikling og især vedligeholdelse af store og komplekse Web systemer hurtigt en langsommelig og fejlfyldt opgave, da systemets enkelte sider skal editeres manuelt.

Derfor introducerer vi en ny fremgangsmåde baseret på programmering, hvor et almindeligt programmeringssprog bruges til at danne HTML dokumenter. Med denne metode opnår får brugeren i udviklingsprocessen råderet over programmeringssprogets kraftfulde mekanismer for abstraktion og automatisering.

I denne rapport beskrives analysen, designet og implementeringen af Lisp Abstracted Markup Language (LAML), som er et HTML producerende udviklingssprog bygget i Scheme, der er et sprog i LISP familien. Udbyggelsen af LAML til også at understøtte CSS, som er et tilhørende style sheet sprog, bliver desuden beskrevet og testet gennem et antal eksperimenter.

De eksempler på Web systemer som er udviklet med LAML og beskrevet senere i rapporten, ser umiddelbart ud til at retfærdiggøre den programmerings baserede metode til Web udvikling. Rapportens konklusion understreger, at metoden er en kærkommen tilføjelse til de eksisterende Web værktøjer, hovedsageligt på baggrund af den bedre håndtering af komplekse Web systemer.

Table of Content

1. Introduction

Following the enormous world wide expansion of the Internet in the recent years, intense scrutiny have surrounded the language for describing Web pages; HyperText Markup Language (HTML). Markup languages such as HTML allow users to pinpoint different elements of a text, simply by marking up the desired content with appropriate markup tags (e.g. *<myname>Carsten</myname>*, where Carsten is markuped using the start-tag <myname> and the end-tag </myname>). HTML provides a fixed set of tag-elements to format and display text and images on Web pages, together with possibilities for hyperlinking to other source files across the net.

Initially opting for fast network delivery HTML was defined as a small finite set of markup elements. Having the language consist of a finite set of elements meant that viewers (the so called browsers) could be built to understand the whole language, thereby eleminating the need to send a syntax defining part along with the HTML document, as is the case of SGML documents. Keeping HTML a simple language also ensured wide spread usage, because a lot of people found themselves able to write personal Web pages.

Along with the rising popularity of delivering online information the demands to the Web language rose as well. As a markup language HTML offered no possibilities of calculation through mathematical algorithms and such functionality was later incorporated by inline scripts or applets executing an alien programming language. The much criticized blending of content and layout information in HTML (HTML originally consisted of both elements for defining layout and elements for identifying different parts of the Web pages) have lately been answered by the emergence of style sheets and an on-going removal of layout elements in HTML. Also the fixed set of HTML elements eventually became a reason for concern, as many requested possibilities for constructing personalized elements. This need prompted the development of eXtended Markup Language (XML), which may be viewed as a light version of SGML, especially suitable for network usage. XML allows structured data to be interchanged through a common set of defined elements. The opinion among the top Web vendors state a Web revolution in the nearest future due to the new possibilities offered by XML (XML will be discussed in greater detail in the analysis).

Simultanenous with the ascending complexity of Web engineering, which have turned from a single language (HTML) to encompass a host of different languages, Web authoring have been subjected to extensive research. Web authoring is still commonly regarded as HTML authoring, eventhough some authoring tools may include small often used java- or CGI-scripts. One branch of authoring tools, conversion tools, have always been popular due to the minimum of effort needed. Such tools automatically convert ordinary textual documents to HTML Web pages without needing any intervention from the user. In the early days many used the *LaTeX2HTML* converting program to obtain Web versions of their publications, while today most commercial wordprocessors have built in converters to support the "save-as-HTML" mechanism. Regarding actual Web authoring (not just conversion of existing material) the commercial attention has almost solely been on WYSIWIG authoring tools.

These tools offers manipulation of different Web page content in a WYSIWIG environment and they are thereby liberating the users from knowing the HTML language.

Eventhough WYSIWIG tools receive substantial endorsement world wide some contradictory scientific voices have been raised. The prime accusations blame WYSIWIG tools for being inflexible to the user, because the full control of the more complex structures of HTML experienced on language level are lost to the user when developing with a tool. Likewise has the "page-at-a-time" approach of existing tools been heavily criticized. Web sites normally commit to a uniform page layout with a high degree of redundancy, which strongly recommends some kind of automation of the authoring process. New ideas for authoring tools capable of automatically generating Web page content or whole Web systems have been adressed by several scolars, including [Owen, 97], [Kesseler, 95], [Thimbleby, 97] and [Nørmark, 99]. Our intensions are to investigate and further research this area, where development of large Web systems should benefit significantly from a tool providing a high degree of automation. This report argues in favour of using an internal language, as described in [Rosenberg, 98], to generate HTML pages from a higher level source. The algorithmic capabilities of the programming language provides a sound basis on which to encapsulate lower level details in abstracted structures on whom automating routines may be applied.

The work efforts described in this report is the continuation of our initial 9th semester investigation, which was covered in [Hellegaard, 99]. The prior work included methods of combining the abstracted authoring language with XML structures. We have discarded this part of the prior project work and are now focusing on the authoring language with additional support for Cascading Style Sheets (CSS) and Common Gateway Interface (CGI) scripting. Concluding that XML probably wouldn't replace HTML as the de facto Web language, but merely supplement it, we decided to concentrate our work on refining the other language part with added functionality.

Onwards this report starts with an analysis of relevant Web languages and tools. The analysis discusses the three markup languages we find most applicable for network usage and whether to stick with the simple HTML or incorporate one of the more complex languages; SGML or XML. Argueing for the continued use of HTML we next investigate the existing types of Web authoring tools. Settling on a tool type we present our visions for further research. Based on the analysis some general problems will be identified in chapter 3. As probable solutions to the identified problems a couple of hypothesisses for further examination are formulated. Chapter 4 examines relevant programming language paradigms in order to find the one most suitable for the implementation of an authoring language. In chapter 5 we discuss the design criterias and implementation of the authoring language along with the presentation of several Web applications developed with the new tool. Additonal support for *CSS* and *CGI* is also presented and evaluated. Partly on account of the Web applications developed with our new authoring language we compare our system with existing tools in chapter 6, and an overall conclusion of the implemented system with a final evaluation of the formulated hypothesisses concludes the report in chapter 7.

2. Analysis

In this chapter we intend to analyse the most relevant methods for the building and maintenance of large Web systems. Having performed this analysis we should obtain enough knowledge about the existing solutions in this problem area and their different characteristics to help us choose the direction in which we want our further work to proceed.

Following the enourmous increase in popularity of the Internet in recent years, it has become obvious to most Web providers that creation and specifically maintenance of Web systems require some sort of tool support to be done properly. As adressed in [Thimbleby, 97] will the development of complex Web systems without elements of iterative design prove to be a very tedious task, where even small changes require considerable work efforts. Especially when working with hypertextual Web documents, which are of highly evolving nature, do we feel this need of constant evaluation and rebuilding. In the context of iterative design methods we find the two most central issues to be *abstraction* and *automation*. Performing iteration processes, i.e. repetitive task, by means of automated computation on high level abstracted structures presents a meaningful method of hypertext authoring, as possible changes are centralized to alteration of the automation process instead of an error proving and time consuming manual editing of a vast number of documents.

We understand abstraction to be the ability to ignore certain lower details and instead focus on a higher level description. Applying this to the hypertext domain leaves us with the desire to extend our hypertext system with new abstracted elements, which should then be available for further exhaustive use. In [Rosenberg, 98] several methods for providing extensibility in hypertext systems by the addition of generalized algorithms are presented, and below we discuss those of Rosenbergs suggestions we believe to be the most important. Rosenbergs methods are complemented in our research with the approach taking by the World Wide Web Consortium (W3C), who recommend future use of a markup language, which is itself extensible [XML, 98] :

External execution: Routines in any programming language located outside the hypertext system may be executed. A well known example of this approach is the *CGI* interface of HTML, which allows calls to any programming language with capabilities of reading from standard input and writing to standard output. This type of extensibility is in general however restricted by the circumstances under which the calls are allowed to occur. To clarify this matter again look at HTMLs *CGI* support, where CGI scripts are only executed upon command from the user, e.g. by submitting a form or by activating a hyperlink.

Internal language: With this mechanism extensibility is offered by a programming language built on top of the hypertext system. The full programming powers and additional functionality of the programming language is then made available to the user working with the hypertext system. The programming language of choice could be anyone; for instance a fully object oriented language, where the sought extensibility

would be made very apparent by the object inheritance mechanisms provided in the language.

Scripting: A hypertext system may also allow algorithms to be run in an internal scripting language. Such scripting languages will typically be unique for every hypertext system, instead of a generalized operating system level programming language. A telling example is once again HTML with its *script* tags, wherein both *javascript* and *VBscript* algorithms may reside. At first sight scripting may strike many as being very similar to the method of external execution, but a closer look reveals one key difference, as the following example, again from the HTML domain, should clarify. Externally executed algorithms through the *CGI* interface of HTML are called and executed on the server side with an optional result returned to the client side, while the above mentioned scripts are executed directly in the clients browser.

Guest algorithm: Whole areas of a host hypertext system may be under the control of an alien programming language. Once again our example steams from the HTML context, where *java* applets upholds full control in a window with an environment very different from the HTML based host. In fact the relationship between the host and the guest algorithm may be so limited, that the host language at some point could be rendered irrelevant, whereupon our language investigation should be subjected entirely to the guest algorithms.

Extensible language: Securing extensibility in hypertext system may be done apart from adding generalized algorithms according to the above mentioned methods, as the W3C suggests in [XML, 98]. Here the new markup language XML (eXtensible Markup Language) is presented, which allows for infinitely many extensions to the existing language. Constructing new markup elements, i.e. new kinds of markup *tags*, involves making a DTD (Document Type Definition), which describes the syntactical components of the new element, and a stylesheet to which the layout of the document element conforms. We delve further into the details of the XML language in the forthcomming section.

The other crucial topic for iterative design of Web documents we pinpointed earlier, was that of automation. Some form of automated solution will soon prove to be necessary, as the Web site complexity rises. Web sites consisting of a large number of pages most probably contain a lot of redundant material, which on even the slightest alteration would constitute an incomprehensible big manual task for the Web author to edit troughout the system. Locating the redundant structures (possibly expressing them as abstract components) and performing a programmed sence of repetition on these structures (automation) would present a more desirable and time saving approach. So automation could exist in the authoring process through programming capabilities, but fully automatic solutions are also possible. Whole database can be converted automatically to Web pages, where indices would be run through a loop, that created a uniform look. Another similar example is conversion from some text format to Web pages (more details on this subject are presented in section 2.2.2).

Having established the need of abstraction and automation in the Web authoring process, we quickly notice the lack of both in the HTML domain. The rest of this analysis is therefore a closer examination of possible Web authoring tools based on the above presented methods,

where the emphasis of our examination will be on the degree of abstraction and automation supported. The different approaches we believe to be of interest are divided and covered in the following three sections.

In the first section we examine the possibility of using a more powerful markup language than HTML. With all the hype today towards XML, we found it only natural to investigate the possible impact of XML conquering HTMLs spot as the de facto Web language. As XML has several benefits compared to HTML, we are wise to restrict our investigation somewhat to the field of authoring. The second category includes different kinds of existing Web tools (what we normally understand to be a Web authoring tool, e.g. WYSIWIGs and conversion tools). Eventhough this category contains tools of very different nature, they are collected as one, because of their common property as being user programs. Finally we set out to explore the possibility of using a programming language to generate HTML codes from a higher level source description.

2.1 Another Markup Language?

In this section we try to establish whether our need for extensibility and automation in the Web authoring context is provided in a satisfactory manner, when using a more powerful markup language or whether we are better of finding a solution still encompassing HTML. Our investigation below covers the three markup languages we reason to be of interest, namely HyperText Markup Language (HTML), Standard Generalized Markup Language (SGML) and eXtended Markup Language (XML).

Many of the frustrations experienced by todays Web providers are pointed towards the simplicity and inconsistent construction of the current Web language HTML. The original specification of HTML consisted of a small set of elements, whose functionality ranged both linking, structuring and layout purposes. Eventhough the HTML specification has been improved several times, there still exist a growing suspicion about its qualities, which is mostly due to its very static nature. When looking for an alternative Web language one quickly notice, that HTML is actually just one language in the SGML family. SGML is a meta language, which can be used for expressing other markup languages, besides containing a vast range of complex features for describing documents, as well as being a highly flexible and extensible language. Because SGML is such a complex language and therefore not completely ideal for network usage, a new language XML, which is a blend of HTML simplicity and some of SGML's most important capabilities, has been invented. Each language is now examined in greater detail to unveil their god and bad sides towards the Web domain.

2.1.1 Hypertext Markup Language

HTML is by far the best known markup language and it enjoys wide spread use, which the evidence of several hundred millions of Web pages easily accessible on the World Wide Web clearly proves. In fact HTML has played a significant role in the enormous growth of the WWW in recent years. Having been developed especially for net usage, HTML became a

static language of little magnitude. The static nature originating from the fixed set of HTML elements ensures excellent networking abilities. HTML documents do not need to be followed by a syntax-defining-part, because the existing web-browsers parses and displays HTML-pages according to information hard-coded into them. Since the browsers fully understand HTML, processing and display of the web-pages is fast, as opposed to high-level SGML markup languages, where a document-instance has to be parsed according to an accompanied DTD (Document Type Definition, as the language defining part is called in the SGML world). The omitance of a DTD while sending HTML documents across networks, of course, also reduce the load on the given networks. At the same time, because HTML is such a simple language it becomes easy and fast to learn and even people with little or none programming experience manage to make pages coded in HTML and viewable on the Web. Contrary to belief the simple language HTML still provides some quite satisfactory mechanisms for displaying content, text, images, and other information types, eventhough tables is used as the main method for page layout.

Language Properties

HTML has no doubt been very succesful and contributed a great deal to the Internets growing popularity, but following the global desire of keeping all kinds of information online some important functionality is discovered as being missing in the low-level HTML language. Prior to other mishaps we discuss the important issue of keeping document content separated from the layout descriptions of the document. The many advantages gained by locating document separat from layout was established over a decade ago. In [Coombs, 87] the authors present this case while arguing for the precedence of descriptive markup over procedural markup (refer to figure 2.1 for an examplication of these two markup types).

Poem	**Procedural Markup**	**Descriptive Markup**
FRA DEN ANDEN	.size 3 .upcase fra den anden	<title> fra den anden </title> <verse>
og lige i dette øjeblik savner jeg dig allermest hvor den tynde hinde imellem os brister	.size 2 .lowcase .newline og lige i dette øjeblik savner jeg dig allermest hvor den tynde hinde imellem os brister	og lige i dette øjeblik savner jeg dig allermest hvor den tynde hinde imellem os brister </verse>
og vi er så tæt at vi ånder fra den anden	.newline og vi er så tæt at vi ånder fra den anden	<verse> og vi er så tæt at vi ånder fra den anden </verse>

Figure 2.1: *The difference of procedural and descriptive markup. In procedural markup formatting commands are present in the text, where layout changes are necessary. We have presented our example of procedural markup with the fictitious "dot" commands of size, upcase, lowcase and newline. Using descriptive markup means identifying element types with the use of certain text tokens (known as tags). The text stream "fra den anden" above is identified as a title element by the start tag <title> and the corresponding end tag </title>. The poem used in this example is part of the poem collection "Universets Celle", copyright Brian Drejer & Forlaget Facet, 1998.*

As stated in [Coombs, 87], the major drawbacks of procedural markup is the formatter dependent commands inherent in the text. Modifications in layout presents a tedious and highly erroneous task, as the formatting commands might well have to be corrected throughout the whole document. Making corrections in one source file containing both the text and the layout commands also produces the unpleasant side effect of enhancing the probabilities of corrupting the actual textual content. Besides is the portability of procedural markup at a minimum, because of the system dependent formatting codes. By using descriptive markup instead, we rid ourselves of these problems. For every legal element type (i.e. tag) in a descriptive markup language, an externally located rule indicates how to format each occurence of the given element. This ensures an automatically and consistently executed layout of the text source file, while future layout modifications only involves editing the rule base. Second will the portability of the text document increase dramatically, as no system dependent codes are present, just text elements made distinct by markup, whose formatting information need only be specified in a local rule base. Further advantages gained by using descriptive as opposed to procedural markup were pointed out in [Coombs, 87]. The separate source and layout files obviously makes it a rather simple matter to obtain different views of the same text document. Shifting to the Web context this could for instance be pages displayed according to personal preferences, e.g. a person with poor vision might want an overall larger font size or maybe a colourblind person might have wishes towards the composition of colours. A final notice is giving to the inherent structure of a descriptive markup document, where the natural hierarchical structure forms a splendid basis for further processing, e.g. for writing a structure-oriented editor conforming to the markup language.

The observant and knowing reader would probably by now argue, that HTML is a descriptive markup language and therefore it should uphold the above mentioned properties. HTML was however conceived with no attention paid to the importance of keeping content and layout in separate files. Some tags in HTML are included for pure layout purposes (such as FONT, EM, STRONG and CENTER), which does not conform to the original intentions concerning descriptive markup systems. This negligence has however been acknowledged lately by the W3C with the introduction of Cascading Style Sheets [CSS, 99]. CSS is a language for writing styles (look and layout) for the different HTML elements individually or in groups (see chapter 4 for elaborating examples and further information). Both internal and external use of CSS is supported in the most recent specification of HTML (version 4.0 [HTML, 98]). The W3C clearly recommends use of externally located style sheets in order to exploit the discussed benefits of this approach. Initial steps have also been made to exclude all layout related elements and element attributes from the HTML specification, as some have already been branded *obsolete*, while others have got the predicament of *deprecated*. Obsolete elements are removed from the HTML language, while deprecated elements are in line to be made obsolete in a forthcoming version and developers are therefore advised to refrain from using them. The function of deprecated elements can be emulated by the use of style sheets and their continuing presence is purely due to reasons of backward compatibility.

When [Coombs, 87] state that descriptive markup documents become highly portable due to the strict separation of content and layout descriptions, one might suspect HTML's earlier indifferent handling of this subject could hamper its portability. This has not been the case however, because the descriptive nature of HTML ensures that local formatting rules exist for every element, being it layout elements or others. Contrary to the procedural markup approach are no machine dependent formatting codes present in HTML documents.

We now proceed to discuss another apparent lack in the HTML language. HTML is limited due to its static nature, where new customized elements cannot be added. This is the main reason for prompting the emergence of XML. Some vendors have pleaded for more elements or an extensible markup language in order to construct industry specific languages, which would ensure an easily compatible exchange of information between corporations of the same industry across the Internet. The urge to construct customized elements have most notable surfaced, when Netscape and Microsoft, the two main browser-vendors, have incorporated support of their own proprietary elements in their respective browsers. Eventhough they probably both thought it would be beneficial for the HTML language, these dual extensions have caused some confusion in the HTML specifications, as the W3C tried to keep up with both companies ideas. In ideal circumstances extensibility should be offered freely to everybody, while proprietary elements ought to be avioded for all costs. Providing total extensibility directly in the language is not possible without reverting to the scheme used in SGML, where language documents are validated according to an accompaning syntax defining instance. The XML approach also relies heavily on this mechanism and a more detailed discussion will follow in the next section. Later we consider the possibilities of creating higher level definitions on top of HTML, where extensibility are obtained either through programming language capabilities as suggested by the earlier mentioned method of an internal language (see section 2.3).

The concerns regarding HTML doesn't stop with poor distribution of formatting information and the disadvantages of a fixed set of elements, however. While HTML may be excellent for delivery across networks and capable of decent page layout, it lacks any higher level capability. It completely lacks any structuring ability (awareness of the structure through a DTD), which makes it a very poor medium for information storage. Effective searching, re-usability and validation are not very effective with HTML instances. In the section on XML we will learn of a whole new range of applications thriving on these advantageous functionalities, which become easily accessible in network environments when working with XML. Many enthusiastic Web providers and users believe that XML and the newfound powers it provide will revolutionalize the Web and with time render HTML obsolete.

Evaluation

From being cherished for its simplicity, HTML is now facing critical voices for lacking the advanced functionality required for the complexity increasing next generation of Web pages. The obvious solution of substituting HTML with a more powerful language promises to deliver such capabilities with one swift stroke. The predicted impact of using another language on the Web will be presented in the forthcoming sections. For now we want to focus on other options upholding the continuing use of HTML, instead of turning to a new language. Possible solutions based on HTML should of course try to deal with the three problem areas identified before. With the emergence of a style language (CSS) for HTML a more logical structure between formating information and content have been established, so a solution supporting CSS covers this former problem satisfactory. Concerning the missing extensibility in HTML our system could rely on the methods adressed in [Rosenberg, 98], which ensures programming capabilities in hypertext systems. Providing the client side of a network with easily accessible and customizable data structures as can be done using XML is however not easily done in HTML.

From a HTML authoring point of view support for CSS and extensibility through a higher level system on top of HTML might present an acceptable solution, while succumbing to the fact that structured information will not be available for client side manipulation[1]. Two approaches for optimizing the manual authoring task to encompass some degree of abstraction and automation will be subject for further investigation. Of the methods discussed in [Rosenberg, 98] we decide to look deeper into the construction of an internal language from which to compile to HTML code (see section 2.3). Besides the advantages and disadvantages of different types of HTML generating tools are to be considered in section 2.2.

The continuing use of HTML is at least for the foreseeable future quite reasonable by principle of inertia. Several millions pages available on the Web today are written in HTML and backward compability reasons should keep HTML in the picture for long. Besides have the somewhat more complex nature of XML until now prevented it from reaching instant world wide recognition. Moreover have people lately questioned the total extinction of HTML due to XML's emergence. It is now believed to be more likely that HTML and XML will work hand in hand in the future (Working drafts from the W3C suggest having socalled XML islands exist within HTML documents – e.i. XML code hidden in *XML*-tags analogous to java-scripts residing in *script*-tags).

2.1.2 Standard Generalized Markup Language

SGML is the mother of all markup languages. It is with a fine word called a meta-language, meaning that it is a collection of generalized rules, that can be used to form other more specific markup languages. HTML is actually itself created by use of SGML. Markup documents written in SGML require the accompany of both a stylesheet for specifying the formatting style of each element and a Document Type Definition (DTD) for syntactical validation of the enclosed elements in a given document.

SGML is currently enjoying widespread usage as a versatile language for information storage. One example is the world's top leading technology companies, who quickly discovered the advantages of defining a common industry specific markup language. Especially the increased possibilities for cooperation, as companies sharing a industry specific markup language will not be confronted with problems concerning information compability when exhanging information.

A few notable examples of already existing industry specific markup languages are: TIM (Telecommunications Interchange Markup), which is the telecommunications industry specific markup language, is primarily used for information storage. The semiconductor industry, where each manufacturer has to maintain a lot of technical information on their produced ICs. In order to enable smooth exchange of these data, an industry consortium (the Pinnacles Group) was formed by leading companies; Intel, National Semiconductor, Philips, Texas Instruments and Hitachi with the task of designing an industry specific SGML markup language. The specification was finished by the consortium in 1995, and the implementation phase is well under way in the companies. Other examples are Microsoft, who use a SGML

[1] Two out of three ain't bad – Meat Loaf

markup language to store and display information in their Encarta Encyclopedia[2] and Newbridge Networks[3], who are using a SGML markup language to store and display their electronic documentation.

Language Properties

SGML is in possession of several strong features, which could make it suitable for network usage. First of all is SGML very flexible as it offers any range of complexity from the simple HTML to the much more complex TIM. SGML is extensible and can handle any degree of structure, thereby making it an excellent store of data. Great structurability also provides the opportunity of re-using information elements, thus promoting efficiency and economy in document creation. Many optional complex options for extensive information processing also exist in SGML. SGML is an open standard (ISO 8879-1986), making it available for everybody, and it is already well established worldwide as a storage of information. Besides is SGML system- and platform-independent, ensuring that SGML documents created on one computer system can be viewed and manipulated on another machine without any information conversion and possibility of loss between them. Totally different computer environments have no difficulties in working with exactly the same SGML documents securing excellent portability of SGML.

But imagining SGML replacing HTML as the language of choice on the Web, becomes a bit harder when facing the following disadvantages in the Web domain of SGML compared to HTML. The most striking difference is the complexity of SGML and the simplicity of HTML. Even people with very little programming experience find it possible to write Web pages in HTML. A completely other issue is the difficulties involved in programming software to accomodate SGML. This is mainly due to the complexity of SGML's options, which only a small percentage of users need anyway. Writing SGML-compliant software would be significantly easier if the number of options in high-level markup languages were considerable decreased (this is precisely the step performed in the realisation of XML). The complexity of SGML has also meant that only a few sporadic tries have been made to develop a SGML compatible browsers for the web. None so far have succeeded in challenging the dominance of the HTML browsers provided by Netscape and Microsoft. Breaking the vendor support and inertia of HTML would require something revolutionary new, which SGML doesn’t seem to have.

SGML will also contribute to a heavier deliverance load over networks, because of the included specification files for style and syntax. Usually HTML documents need none of those, but considering the recent recommendation of using stylesheets with HTML documents might turn this aspect into a rather insignificant matter.

Evaluation

Since XML is developed with the intensions of constructing a Web version of SGML it holds SGML’s excellent structural advantages, but is ridden of the complex high level features that distorts the easiness of use. This has convinced us to discard SGML as a serious contender for the leading role as Web language and concentrate on XML’s contribution.

[2] More information available on http://encarta.msn.com/EncartaHome.asp
[3] Further details on http://newbridge.com/index.html

2.1.3 Extended Markup Language

As stated earlier has the growing frustrations with the limitations of HTML urged the W3C to think of a solution for further and improved possibilities regarding Web pages. A working group was formed by the W3C and they eventually came up with the specification for a new markup language, namely the eXtended Markup Language (XML). XML can be seen as a light version of SGML, customized for the World Wide Web. Like SGML, but contrary to HTML, is XML a meta-language, meaning that XML is capable of defining other markup languages (notice that we regard an in XML defined set of markup elements as a specific markup language). XML is only using the most common parts of SGML, all of the more advanced and rarely used properties of SGML has been stripped off. As a consequence XML is now much easier to understand and better suited for network delivery.

The parts of SGML that XML conserves are structurability, extensibility, validation and re-usability. And like SGML does XML preserve system- and platform-independence, it is non-propietary and an excellent store of information. XML is seen to hold SGML prime assets and is ridden of any unnnecessary and seldom needed functionality to ensure excellent network properties.

Language Properties

Apart from introducing the basis for a whole new range of Web applications XML has several advantages compared to HTML. Below we try to outline the most important possibilities gained from using XML. Some of XML's qualities can be directly derived from the "10 commandments", that XML was designed to uphold [XML, 98]. The most noteworthy being:

- XML is straightforwardly usable over the Internet.
- XML supports a wide variety of applications.
- XML is compatible with SGML.
- Programs to process XML documents are easy to write.
- XML documents are human-legible, reasonably clear and easy to create.

Another important design issue liberates XML from the need to always send along the description of the syntactical composition of elements occuring in a given XML document, as is the case with SGML. In XML it is made optional whether or not to send along a Document Type Definition (DTD), in order to resemble the use of HTML documents. A XML document that is parsed according to an accompaning DTD is called a valid XML document. Whereas an XML document, that is only parsed for minor syntactical subjects (such as proper nesting and obligatory end tags) is called a well-formed XML document. Unless the data structures within a XML document are validated they cannot be used for further manipulation.

Eventhough XML documents cannot be validated without an accompaning DTD, future use could be handled more intelligently. Industries with a common industry specific markup language could exchange DTD's from time to time, when updates have been performed. XML documents written in this specific language could then be sent without the DTD on every delivery and still be validated, thereby minimizing the network load.

Importantly XML opens for a whole new range of Web applications. Presented in [Bosak, 97] a member of the W3C workgroup, who are responsible for the XML specification, has grouped the most important application types in the following 4 categories:

1. Applications that require the Web client to mediate between two or more heterogeneous databases.

2. Applications that attempt to distribute a significant proportion of the processing load from the Web server to the Web client.

3. Applications that require the Web client to present different views of the same data to different users.

4. Applications in which intelligent Web agents attempt to tailor information discovery to the needs of individual users.

The first category offers important improvements, as information exchange is easily obtained through a shared XML-language (use of the same DTD), regardless of any internal database formats. Bosak provides an example from the American health care system, where access to a patients medical records is available in a web-based interface. These records are represented by a folder icon, which after possible alterations can be dragged over to the internal database application and dropped, whereafter the updates are performed automatically in the database trough interfaces to the common language XML.

This resembles the use of three-tier architectures, which Microsoft states in [MS, 98] will also thrive on the emergence of XML. Microsoft feels that the classic client/server architectures are increasingly being replaced by three-tier models, in which a front-end browser or other application communicates with a perhaps heterogeneous back-end storage through a middle-tier Web server, see figure 2.2.

Three-tier architectures holds several benefits compared to the well-known client/server architectures, including better scalability, as several servers can be assigned storages with extensive hit rates, and better security, as transactions happen through the server and not directly at the source. XML succeeds in separating the user interface from the underlying data structures in the storages, allowing integration of data from multiple (maybe incompatible) sources. The underlying data formats are converted to XML (if necessary) on the middle-tier and delivered to client applications by HTTP over the net, where it can be manipulated and viewed as desired.

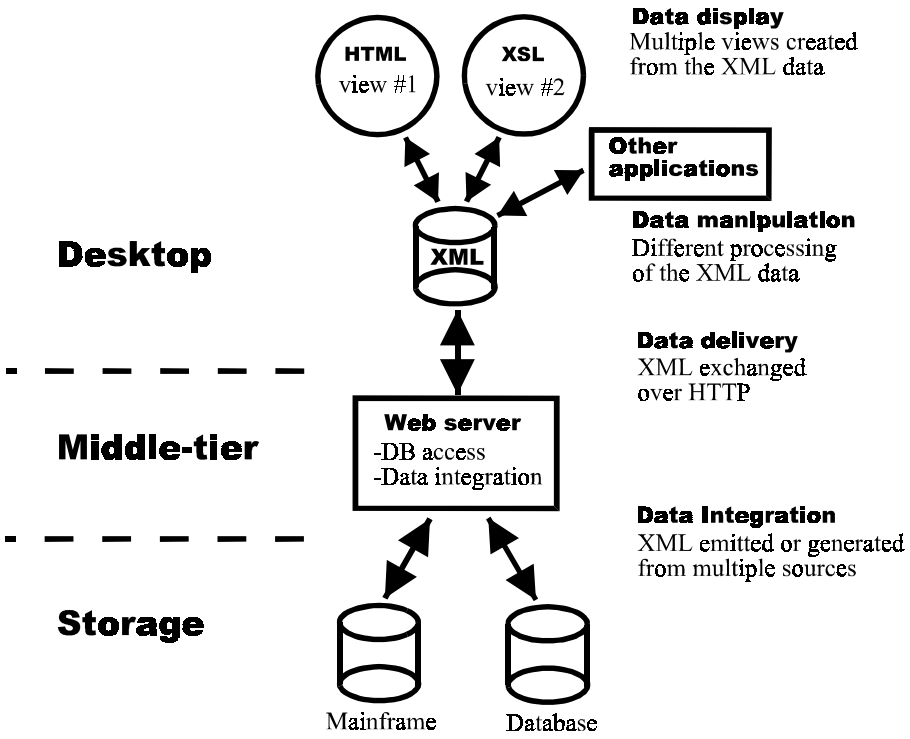

Figure 2.2: *Overview of a three-tier architecture (adopted from [MS, 98]). Data structures are gathered from different (possible heterogeneous) sources and interfaces to the common language XML provides easy access and delivery across the network. Further manipulation of the XML data is now possible on any client machine residing anywhere on a network.*

Bosak has headlined the second category *'XML gives Java something to do'*. This slogan is originated from the fact that XML provides a way to distribute heavy workloads from the web server to the web client. After a brief interaction with the server, where the XML documents are downloaded, are the web client able to perform complex computations on the XML data trough *java* applets or script languages such as *javascript*. The server is ridden of enormous resource hits and the client might deliver quicker responses to interaction from the user, when the data is on the client and ready to be manipulated at once. Bosaks main example of a second category application is the Pinnacles Group, who have constructed their own specific SGML language for IC-circuits. It could be a possibility for the different companies engineers to enter a manufacturer's web-site and download both the viewable data and a Java applet, that allows the user to model different combinations of the available circuits.

Bosak is further expecting this kind of application to grow in importance, as users come to expect and demand this kind of interoperability in manipulating their data. XML is partly designed to accomodate this kind of application, as the information provider's servers can hardly be expected to keep up with these increasingly computation intensitive tasks.

The third category presents applications in which the user can choose between different views of the uploaded data, without having to download it again in a different form. An example application is a dynamic table of contents, which enables the user to traverse a large data

collection by expanding portions of the table of contents in order to reveal more detailed levels of the document structure (similar to the directory structures presented in the Windows Explorer). Should the same be done on the Web today, a time expensive server retrievel would be necessary for each expansion or contraction of the table of contents. XML offers a far better solution, as the entire table of contents structure is downloaded once on the client, enabling it to perform the manipulations on the table of contents locally as they are requested. Another example is some kind of manual that surplies notes in multiple languages, where user selection determine which language to show.

Bosaks last category covers appplications, where it is possible for intelligent web agents to provide different information to individuals based on personal preferences described in a standardized markup language. Every imaginable information product will become completely customizable and dynamic, as users become able to choose languages, levels of comprehension, graphical user interfaces and content. This kind of application is expected to trigger a growing interest in the "push" technology paradigm. Push technology is the opposite of the familiar "pull" technology, where the user clicks on a hyperlink to retrieve the desired information. Push technology enables the information providers to deliver information to the users instead. The appearance and content of this information is again determined by individually set preferences. Examples of push techonology already exist, but XML is expected to take it a step further and provide complete customizability, e.g. a future push-enabled browser could be set to download all "Metalized Magazine" record reports of the Norwegian black metal scene as they appear online.

Others visions for improved web facilities due to the emergence of XML exist. The precise description of information in XML documents presents a better base for searching than the currently available primitive text searches in search engines like Excite and Altavista. Using the above mentioned search engines current search methods to find books on "H.C. Andersen" would probably yield both books by and about H.C. Andersen. This problem would be eliminated in an XML environment, as books could easily be categorized in a standard way by author, title, about and other criteria. A search in an XML environment would therefore yield a clear distinction between <author>H.C. Andersen<author/> and <subject>H.C. Andersen<subject/>. Query languages providing advanced search capabilities, similar to SQL, in markup languages are currently subject to further research. This research area has prompted the W3C to publish a note (being only a note it is the earliest stage of a W3C document, but research is in progress) regarding an XML query language called XML-QL[4].

Microsoft are heavily involved in the specifications of XML and its auxiliary components and they have some ideas about XML's future usage as well. Microsoft expect that XML and HTML will be used together in the future with XML delivering the structured data and HTML displaying it, and that an XSL language could be used to generate HTML from XML. Besides they are now working closely together with the W3C on a format for encapsulating XML data (so called XML islands) in HTML pages. The data stored in these XML islands are then available to processing through scripting. Microsoft also intends to make XML the only storage media for all applications in the future "MS Office" packages in order to obtain the highest degree of data integration.

[4] Consult http://www.w3.org/TR/NOTE-xml-ql/ for further information

Eventhough XML might look perfectly suited for network usage by now, it actually has a few drawbacks. The first being the lack of many of SGML's complex features which were omitted in XML to ensure better network delivery. Some providers might still need the full power of SGML and therefore won't see XML as a better alternative than HTML. It would be necessary for them to employ a multiple-tier process, where data is stored in a high level SGML-language and converted to XML instead of HTML for display purposes.

Others fear that the emergence of XML will cause increasing stress on the networks. Eventhough an XML document is not particular larger in size than its HTML counterpart, it is expected that the greater capabilities of XML will cause web suppliers to incorporate more advanced features in their web pages. Especially multimedia components, such as specialized font sets, graphics and video will be encountered more frequently. The conclusion being that XML instances require more network bandwidth than the plain old HTML instances.

A point that could show to be troublesome for XML is widespread acceptance, which of course is crucial for its usage. It is not easy to make people change to new standards, but in [Heinemann, 98] Adam Denning (Microsofts group manager for XML) identifies the following 3 key factors as crucial to the task of getting people to start using and developing XML applications:

1. People have to understand the values of XML. No one will adopt to XML just because it is a new technology.

2. Tools, such as browsers and parsers, that makes it easy to utilize the powers of XML should be available for the users.

3. The different industries have to agree upon specific XML vocabularies, e.g. booksellers should agree upon a specific XML language for books. So everybody knows there is a uniform way to describe books.

If these 3 factors comply, XML should gain widespread adoptance according to Adam Denning. Such things are however quite difficult to predict, but the fact that XML is an open standard and many leading technology companies are involved in shaping the XML specifications, promises realistic hopes for a break trough for XML.

Evaluation

A whole new range of Web applications could be a reality if XML achieved wide-spread usage. As our discussion above stated would validated XML structures on the client-side present an excellent basis for advanced manipulation and computation on the client without consulting the server further. This subject area most clearly shows XML's superiority over HTML, wherein such higher level functionality is impossible. Several factors however still speak for preserverence of HTML. The somewhat more complex nature of XML, unavailable XML tool and/or language implementations and the inertia of HTML's world wide appreciation all question whether or when XML will conquer the Web. Considering all the hype and vendor support we do however believe XML will have some impact on the Web in the future. But feeling the need of a powerful authoring system right away, we decide to look beyond XML and concentrate on HTML for now. Since we are developing an authoring

system for markup languages a future translation from HTML-tool to XML-tool may not present a major task either. Besides have recent voices from the Microsoft Corporation argued for future cooperation between HTML and XML, which would secure the longevity of our tool. As a consequence the rest of this analysis will concentrate on systems for authoring large HTML systems.

2.2 Markup Producing Tools

This section covers the different types of markup producing tools. For every type of tool we evaluate different existing implementations ranging from commercial programs enjoying wide-spread use to purely scientific prototypes. The knowledge hereby obtained hopefully enables us to find an area of focus, where we can supplement the markup tool world with further research.

2.2.1 Authoring Tools

An authoring tool is understood to be any application that is specifically designed to aid users in editing markup. The editing processes covered by this definition may range from direct hand coding (perhaps in structure editors with automated syntax support) to WYSIWYG editors that do not present the actual underlying markup to the author for editing (some tool implementations may offer a view of the resulting HTML code for "hands-on" edition). In every case we regard an authoring tool as being a full-fledged program for the user to interact with. Word processors which allow textual content to be conceived and ”saved-as-HTML” are not included in this definition, but have instead been included in the section on conversion tools, see section 2.2.2.

Examples

Structure editors with support only for highlighting and indention of syntactical content are less used today, as newer WYSIWYG (What You See Is What You Get) tools are preferred by a majority of people. A major part of the existing structure editors have besides evolved to include additional functionality and one example of such is the *CoffeeCup HTML Editor ++* [5]. The User Interface of *CoffeeCup*, which primarily consist of a structural editor and secondly buttons and menu items for further functionalities, is depicted below in figure 2.3.

[5] Shareware version available on http://www.coffeecup.com

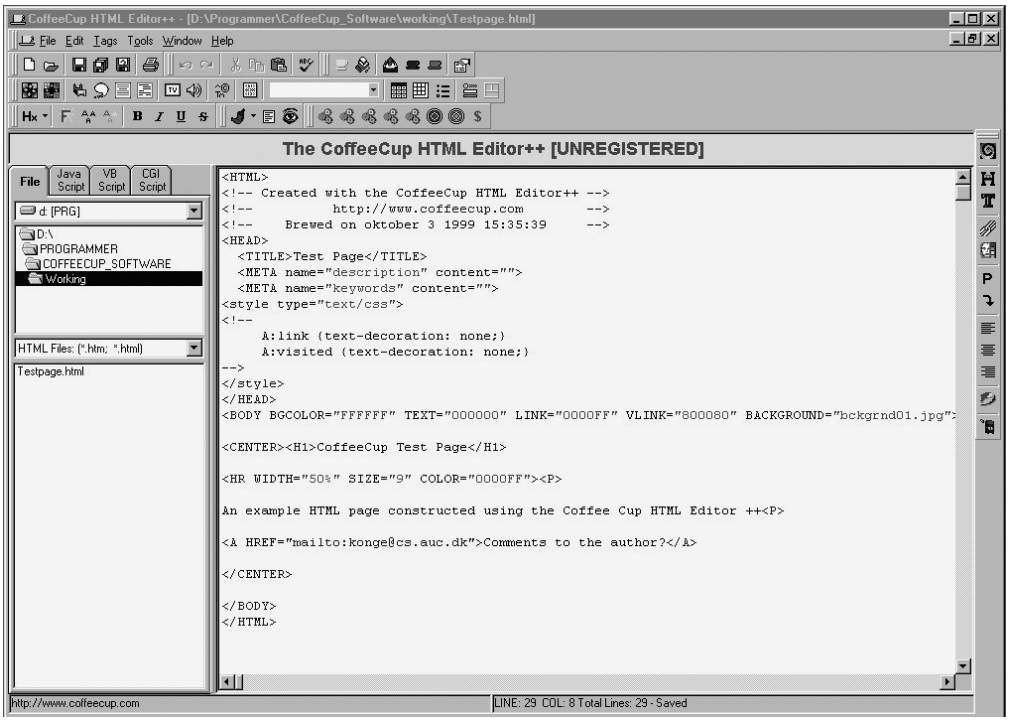

Figure 2.3: *Screenshot of the CoffeeCup HTML Editor ++. The editor is loaded with a sample page, which partly shows the structural dependent appearance of the HTML code (indented elements stands out clearly, but the colour highlightning is unfortunately only partly visible in this greyscale image).*

The editor part of *CoffeeCup* offer full customizability on highlighting details, together with a generally customazible User Interface. An accompaning package contains several hundreds animated icons, images, background images and sound files, besides including small and often used preprogrammed Java-, VB- and CGI-scripts. A range of helpful stand-alone programs can be associated with the HTML editor, in particularly a FTP program for easy uploading of newly constructed HTML pages and programs for making image maps and Cascading Style Sheets, which have all been developed by the CoffeeCup programmers team. Related external programs and packages of graphics and sounds are one thing and the actual functionalities of the HTML tool are something else entirely. The *CoffeeCup* editor makes heavily use of automated markup insertion functions, which are the features of an authoring tool that allow the user to produce markup without directly typing it. This includes a wide range of tools from simple markup insertion aids (such as a bold button on a toolbar) to wizards that construct higher level elements (e.g. a table or a form) based on a series of user inputs. The editor in question offers both buttons for often used presentational tags (e.g. italic, headers and paragraphs) and alignment settings (left, center and right), together with easy access to every existing HTML tag. Wizards and small designer functions exist for both forms, lists, tables, frames, hyperlinks and E-mail links, while a quick start function allows the user to quickly type the initial (items such as title and meta-data) part of a HTML page. A macro editor offers possibilities to constructs personal snippets of often used HTML code (e.g. a page footer to be used in several pages needing a uniform look), which can easily be accessed later during the authoring process. So the basis of the CoffeeCup HTML Editor ++ is a structured editor added with comprehensive support for higher level functions.

We now turn our attention to WYSIWIGs, which contrary to structured editor do not reveal the underlying HTML code to the page developer. Only the actual Web contents are available for editing, whereas the HTML code is applied automatically according to the desired appeareance of the content. The most notable WYSIWIG Web authoring tool around is Microsofts commercial giant *Frontpage*, which has lately become an integrated part of Microsofts *Office* package. Many significant similarities exist between *CoffeeCup* and *Frontpage*, e.g. the heavy use of wizards for constructing the somewhat more complicated Web page elements, such as tables and forms. *CoffeeCup's* startup function is also paralleled in *Frontpage* as illustrated below in figure 2.4, which shows the use of wizards in *Frontpage's* startup dialogue.

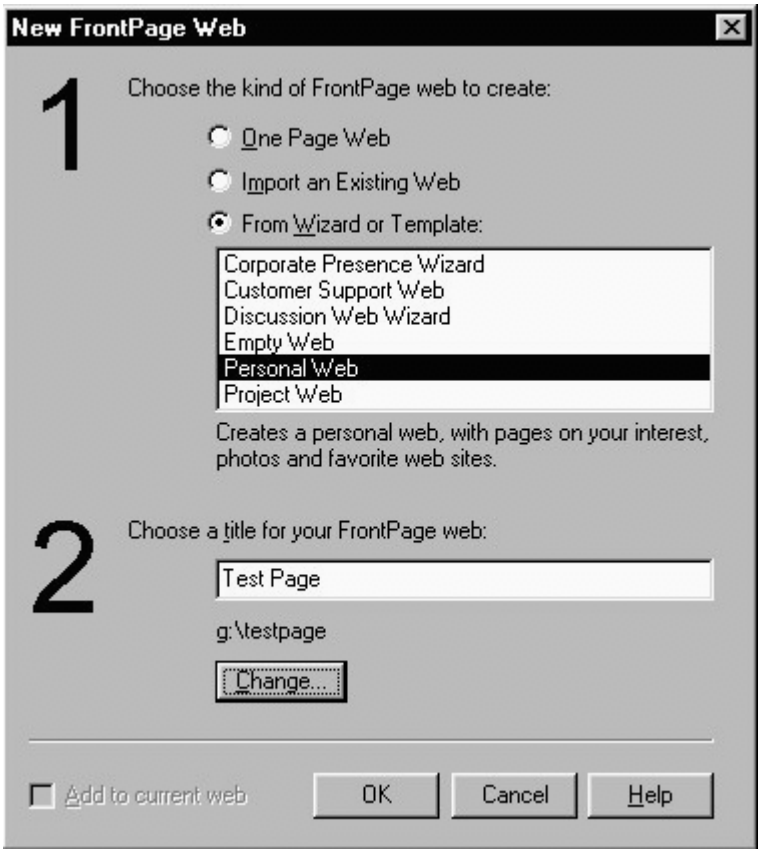

Figure 2.4: *Screenshot of Frontpage's startup dialogue. Apart from creating a single Web page or importing an existing coherent set of pages users may choose to incorporate one of Frontpage's predefined templates for a Web site (it is later possible for the user to add to this limited set of predefined site templates with personal designs). The current case depicts a selection of the Microsoft template for a personal Web page.*

Frontpage's possibilities for developing and using Web page templates easily outweighs the simple startup mechanism employed in *CoffeeCup*. The most striking difference is however *CoffeeCup's* restriction of handling only single pages, whereas *Frontpage* is able of controlling several related pages. This control gains in efficiency by the divided architecture of *Frontpage*, which consists of the two connected tools, namely the *Frontpage Explorer* and

the *Frontpage Editor*. The *Frontpage Explorer* acts as a site manager controlling a host of interconnected pages through high level descriptions. The most notable instrument being a graphical interface to a representation of the hierarchical structure and related link properties of a given site. A view of this mechanism for easy manipulation of overall site structure in *Frontpage Explorer* is presented below in figure 2.5:

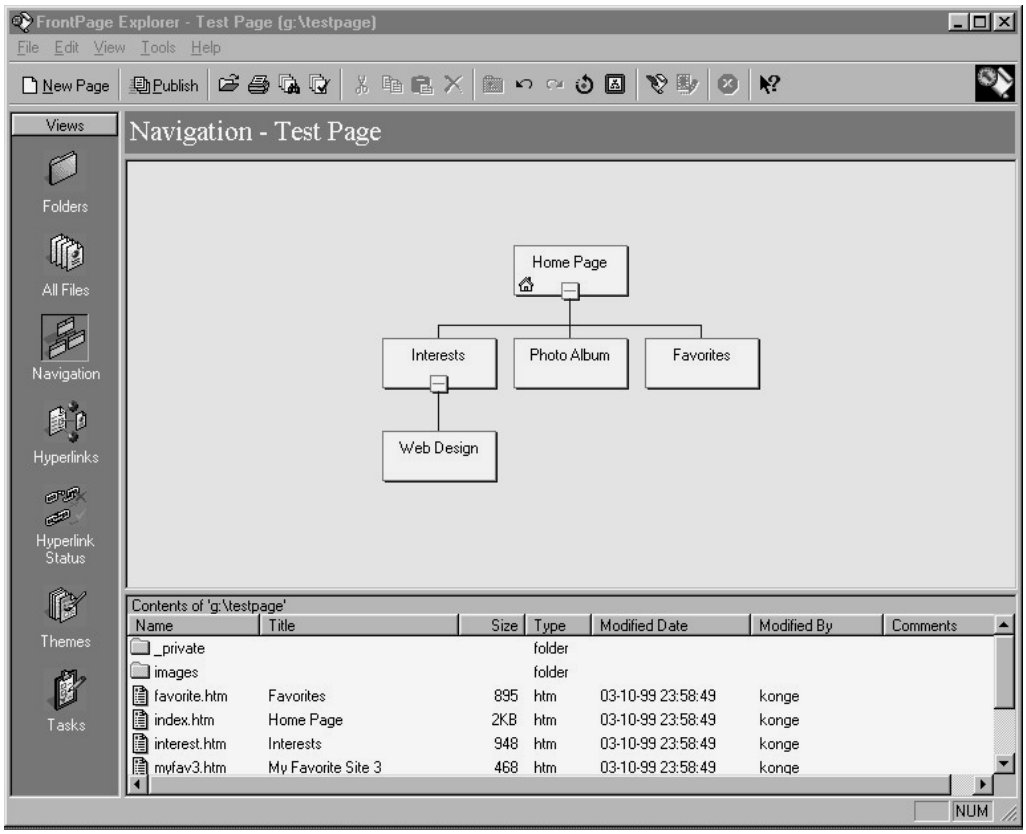

Figure 2.5: *Screenshot of the navigation part of Frontpage Explorer. Shown is the hierarchical structure of a page evolved from the"Personal Web" template. Link relations of the pages spreading from the root element "Home Page" are very evident. The page "Web Design" has been added to descent from the "Interests" page by virtue of a single mouse step.*

Further functionalities of the *Frontpage Explorer* include help for file operations as well as detailed link management, which provides views of every in- and out-going hyperlink for specific pages, besides offering possiblities for verifying the actual status of all internal and external links of the site (this mechanism is very helpful for spotting broken links). *Frontpage Explorer* also presents the Web developer with an opportunity to build a site with uniformly looking pages by the use of *themes*. Everyone of 54 predefined *themes* defines a coherent appearance of background, banners, headings, ordinary text, buttons, link buttons and textual links. Besides is a minor project managing system included, where unfinished parts of the site being build and other future tasks, such as external link status verification, can be put forward for reminding the developer, what still needs to be done. The final noteworthy mechanism in *Frontpage Explorer* is the publishing function, which allows the whole site content to be

uploaded in an integrated fashion, where related documents and/or adherent non-textual external files are automatically located correctly in the file system.

While high level site manipulations are managed in the *Frontpage Explorer*, the authoring and editing of the actual content on specific pages are performed in the *Frontpage Editor*. At first sight *Frontpage Editor* may look like an ordinary WYSIWYG word processor (true to Microsofts uniformly looking Windows applications the *Frontpage Editor* in fact resembles *Word* a great deal). For conviction refer to figure 2.6.

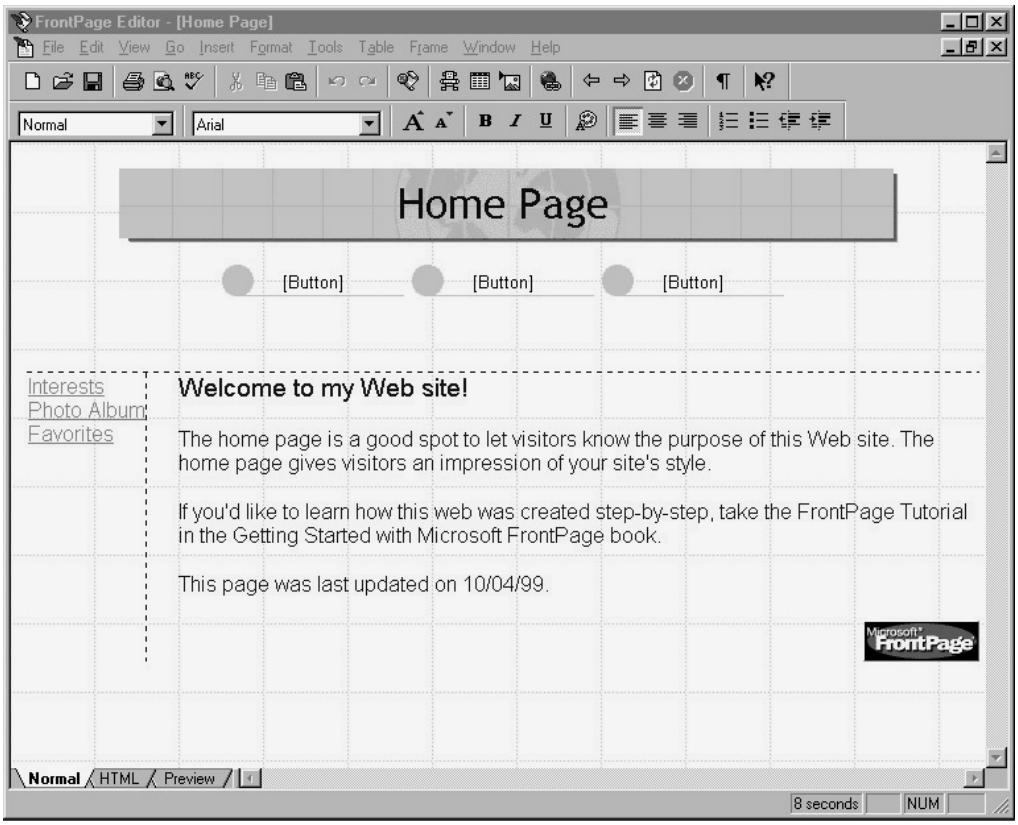

Figure 2.6: *Screenshot of the Frontpage Editor. Opened and ready for editing is the root page of the personal Web site template presented before during the discussion of the Frontpage Explorer. A quick glance shows that Frontpage Editor mainly consist of typical word processing functionality, but important additions include support for high level HTML constructs and possibilities for the developer to view the underlying HTML code, as well as previewing the constructed pages in a small built-in browser.*

However several factors make the process of writing Web pages using *Frontpage* different from writing ordinary documents in a WYSIWYG text processor. Most apparent is the hypertextual nature of Web pages, which necessarily require some degree of hyperlinking mechanism unlike static text documents. To cope with this, easy means of supplying hyperlinks to both textual and imagery elements are provided in *Frontpage*. Furthermore are wizards provided for simple development of every complex HTML construct (such as tables, forms and frames). Apart from the above mentioned traditional HTML elements *Frontpage* includes support for a few active elements, this being among other things scrolling text, hit

counters, search forms, hover buttons and video clips. Future alterations of all developed HTML elements are uncomplicated performed by the edition of a given elements properties. In summarization *Frontpage* offers a WYSIWIG editor for the creation of individual Web pages, along with a site manager capable of controlling whole site structures consisting of a vast number of underlying Web pages.

Evaluation

Tools based on structure editors are not very convenient, when larger Web systems are being developed. Of course the development process of specific pages are eased by the source code clarification provided by the indention and colouring of different structural content and by additional incorporated functionality, such as buttons, macros and wizards. But constructing larger Web systems consisting of a vast number of pages and with high probability of being subjected to extensive future updating would present an enormous task, if performed with tools upholding this single-paged focus. When talking minor Web systems most people (especially those with less experience in the field of programming) are reluctant to use this kind of tool anyway, as a certain amount of knowledge of the HTML language is required from the user.

Desirable for most people is therefore a WYSIWYG interface like the one implemented in *Frontpage*, which conceils the underlying HTML code from the user. But apart from the differences in viewing and editing, both of the tested tools share some similarities. The extensive use of Wizards and automated markup insertion functions is a trademark of both tools. Quick reference to often used HTML elements trough buttons, menu items or macros is applicable for everybody. Wizards on the contrary tend to both move the complexity from the HTML language along to the tool implementation and decrease the user's control of detailed aspects of the given HTML construct. These issues are mostly ignored by users with little programming skills, who still find the Wizard implementation easier to learn than HTML code. In our opinion both tool types are fairly easy to learn and simple pages may be developed pretty fast even by newcomers. But as briefly stated before we are seeking a tool, which offers facilities for larger Web systems and handles more than just single pages. The structured editor we examined (*CoffeCup HTML Editor ++*) was purely minded for developing pages one at the time, while the examined version of *Frontpage* did include a site level manager.

Frontpage's site manager helps in making and maintaining the tree structure of a Web site, with its pages and link relations. Referential integrity is automatically assured within the site and only outgoing hyperlinks needs to be tested for legality. To help perform this procedure *Frontpage* can deliver a list of all outgoing links with current status. When building a larger Web site a coherent look of all included pages may be needed. The *Frontpage Themes* are included for this purpose and they present a simple mechanism of altering the look of the entire site. No relations exist between similar content on the different pages and updates throughout the site must still be done manually for each page. Eventhough *Frontpage* is excellent for authoring-in-the-small (single pages or smaller Web systems) and contains some means of authoring-in-the-large , we find the following flaw to be of outmost importance. To encompass the use of *Themes* and active elements Frontpage produces a proprietary version of HTML, which includes elements not known from the HTML specification. This causes

confusion with the Netscape Navigator Browser and renders pages impossible to be contained on UNIX servers.

2.2.2 Conversion Tools

A conversion tool is any application or application feature that allows content in a different format (proprietary or not) to be converted automatically into a particular markup language. This includes stand-alone software whose primary function is to convert documents to a particular target language as well as non-markup applications with "save-as-HTML" features. This category of tools is bound to distance the developer from the Web domain, as the future Web content is produced in an alien format. The actual conversion process is then typically performed automatically, being it by a command-line program or by a built-in converter of a higher level application, and it is therefore beyond the influence of the developer.

Examples

The most noteworthy stand-alone program for conversion of text to HTML based Web pages is *LaTeX2HTML*, as described in [Drakos, 93] and [Drakos, 94]. In accordance with its name *LaTeX2HTML* performs an automatic conversion of documents written in LaTeX to HTML. The primary mechanism in this conversion is a translation of the inherent document topology consisting of chapters, sections, subsections, table of content, index list and others into a HTML counterpart. Most striking in a *LaTeX2HTML* translation is the conversion of the top most document structures (typically chapters) into individual physical HTML pages, which apart from the actual textual content are provided with a navigation bar. The navigation bar in turn offers hyperlinks to other parts of the original document (typically to the previous and next chapters along with connections to the table of contents and index). To accomodate mathemathical equations (which is a prime asset in LaTeX, but has little support in HTML) and other incompatible structures *LaTeX2HTML* images are produced and inserted accordingly in the HTML target. Other workarounds include macros for generating some of HTML more testing capabilities (forms, image-maps, etc.) by putting raw HTML in the LaTeX source. Most beneficial for users endorsing the approach taking with *LaTeX2HTML* is the easy and reliable updating, where alterations in the document source promises to preserve the integrity of hyperlinks in the HTML target files (opposite a manual editing of several HTML pages, which would prove a severe and error prone task). Alternative versions of the same document are also simple to produce by altering the target descriptions in *LaTeX2HTML* (of course the document is from birth already available both as a normal text document and as Web pages). These benefits along with the low number of available authoring tools and inexperience with a new language made *LaTeX2HTML* quite popular in the early days of the Web. For authoring material directly for the Web people have turned to the kind of authoring tools described in the previous section.

Most of the modern word processors (we have chosen *Microsoft Word* as our example) have the ability to save document files in the HTML format, which renders the need of an in-between converter program obsolete. Contrary to document splitting performed in *LaTeX2HTML* the content of a *Microsoft Word* document is translated to a single page. A few tests of *Microsoft Word* worked alright, as long as the document consisted simply of text,

images and minor tables. HTML pages with more complex content were hard or sometimes even impossible to construct.

Evaluation

Making simple Web pages with the above mentioned tools is fully automatic and saves the user the efforts of learning HTML. Pages of a more complex nature are however seldom conversed with an acceptable result. The most important obstacles are the inherent dissimilarities between the two media, as also stated in [Drakos, 94], namely the presentation information in the word-processors that cannot be reproduced in HTML and the inflexible conversion process, which does not take full advantage of HTML's powers. *LaTeX2HTML* tried to accomodate for the missing parallel in LaTeX to some of HTML higher level constructs by allowing HTML code to be inserted in the text document (the alternative is to post process the HTML code delivered by *LaTeX2HTML*). These workarounds tend to nullify most of the advantages gained in the first place. Surely the primary intention with programs of this nature is to obtain an easy conversion of content already existent in another format, while not being optimized for actual HTML authoring. Web developers will always be handicapped if using converters, as to many of HTML's high level structures will be not directly available in the alien environment. In conclusion we believe conversion tools, as the pair mentioned above, are not very applicable in Web systems development, especially as we intend to build larger Web systems.

2.2.3 Generation Tools

A generation tool is a program or script that produces markup automatically by following a template or a given set of rules. When developing with generation tools user actions may range from interacting sporadically with very advanced applications to performing actual programming in languages biased to generate certain textual content.

When working in the Web domain the generation may be performed on either the server or client side. [Nørmark, 99a] divides the existing possibilities of producing Web pages in the following three categories: generated, calculated and dynamic. *Dynamic* represents pages or page content which are calculated on the client side due to some user interaction. This calculation is either performed in Java applets or by some scripting language (such as VBscript or Javascript). *Calculated* pages are generated on the server side according to user input delivered back from the client-side browser trough the Common Gateway Interface (CGI). Server located scripts, coded in an arbitrary programming language capable of reading and interpreting the encoded CGI-stream passed to the standard input, then generates and delivers an appropriate answer in the shape of a HTML page on standard output. This resulting HTML page is then transferred back to the browser by the CGI-interface. Pages of the *generated* category denotes HTML pages constructed according to some higher level description in any given format, which upon execution expands to ordinary HTML code. Generated HTML pages are not the product of interaction from the end-user, but are instead pre-compiled static pages (it is of course possible for these static pages to contain applets or point to CGI-scripts). In our attempts to build larger Web systems we mainly focus on the generated category and sporadically touch the calculated category, as we feel Java-applets and inline scripting languages are beyond the scope of optimizing Web page authoring.

Examples

Our selection of examples on generation tools ranges from high level wizards that produce complete HTML documents on the basis of a series of user preferences to template use in environments again ranging from advanced GUIs down to almost programming language level. The chosen examples all share a scientific approach, as they are experimental prototypes only used for research purposes or locally related work. No commercial generation tools for HTML are to our knowledge visible on the market of today, as it is almost entirely occupied by authoring tools (some authoring may have generation like features (such as high level wizards), but none would be categorized overall as a generation tool).

A common target area for generation tools is larger Web sites and especially Web sites with a high degree of redundant content. Isolating these redundancies in abstracted structures (templates) then offers a platform on which to obtain a significant level of automation in the authoring process (and very importantly in future document maintenance). Both [Owen, 97] and [Thimbleby, 97] strongly argue for the need of automated solutions contrary to the "page-at-a-time" behaviour of existing commercial tools. Evidently simple pages or sites without common page layout schemes gain very little from this approach. Many of the tools presented have been developed especially for generating online course material, which contain highly redundant material (e.g. literature references, exercises and exercise solutions for every lecture). The sequential nature of course lectures also invites the use of standard sequential linkage between lecture pages.

A program for automating development of complete hypermedia courses, *HTML Course Creator (HCC)*, is presented in [Curtis, 96]. *HCC* operates on a very high level, where knowledge of the HTML language is entirely irrelevant. Users only need to present the program with a few preferences stating preliminary variables such as compilation directory and different course information. Next and last step for the user is to plot in the media to be contained on the different course pages using an easy to use GUI. *HCC* then compiles the entered information and generates HTML pages according to some predefined HTML templates, while upholding the sequential course structure with automatically produced link mechanisms. The authoring process becomes very easy and fast with *HCC* and no programming capabilities or knowledge HTML are necessary. A major drawback is however the lack of flexibility of the program, because only course material may be produced and this only according to two slightly different pre-defined templates.

Another developer program operating on a high level is the *W3DT Web-Designer* described in [Bichler, 96]. Unlike *HCC* does *W3DT Web-Designer* not limit itself to only constructing courseware. A graphical user interface allows users to construct the hierarchical structure of any type of Web site (very similar to the one incorporated in the *Frontpage Explorer*). Efforts have also been made in [Bichler, 96] to encompass calculated pages by means of CGI-scripts capable of querying databases. Curiously enough have design primitives for depicting dynamic content in the graphical representation of the site structure been named templates in [Bichler, 96]. By setting preferences on header, footer and other attributes the user may create a layout scheme to be used uniformly troughout the site. Compilating the hierarchical site structure along with the desired layout settings yields a site skeleton consisting of Web pages for every node in the site tree structure with navigational content to accomodate any node

relations. The different pages in the site skeleton will have to be pre-processed in a HTML-editor in order to add actual page content. No doubt this pre-processing hampers future alterations, due to the fact that structural changes would require a new compilation of the site, thereby overwriting the editions made on the old site skeleton.

The work discussed in [Kesseler, 95] resembles the approach just presented, but it delivers a more refined tool, partly because some of the important shortcommings of the *W3DT Web-Designer* have been avoided. The implemented program *Hypertext Structure Description Language (HSDL)* covered in [Kesseler, 95] also adhere to a graphical interface for development of site structures. In *HSDL* a class schema is created to model the different objects existing in the target domain and the relationships between them. This class schema then acts as a template for creating an instance schema of all instances of the different classes defined in the class schema. Again referential integrity is a priority when compiling the instance schemas to the target domain of Web pages, with link instances inducing ordered collections made traversable on multiple sequential paths by previous/next links. So upon compilation *HSDL* produces a Web site skeleton like the *W3DT Web-Designer* does. The *HSDL* skeleton is however provided with page content by filling the empty instances in the instance schema. This renders any pre-processing unnecessary, as content and site structure will be compiled together. Besides is *HSDL* the more flexible tool, because the author may create new or alter existing schemas according to individual needs. [Kesseler, 95] refers to this as *schema evolution*, for which a wide area of operations are possible on classes, instances and link classes. The target domain appearance of instances are also fullly customizable. Customizing is done by redefining the so called *expanders*, which perform the translation of instances to the target language HTML. The expanders in *HSDL* is written in the functional programming language *Scheme*, indicating that defining a individual page layout requires programming skills in this language (refer to figure 2.7 for a sample view of a scheme expander). People without these skills would have to settle with the default expanders built into the system.

```
(define (title node)
  (emit "<H1 ALIGN=CENTER><B>"
        (name (class node)) ": " (name (instance node))
        "</B></H1>))
```

Figure 2.7: *An example of an expander written in Scheme (adopted from [Kesseler, 95]). The depicted expander is the default one for generating the title of a HTML node. The function emit writes all arguments to the HTML document currently being built. Arguments are strings representing HTML snippets together with strings extracted by the name function from both the nodes class and instance.*

Another template based approach, *Gentler*, is discussed in [Thimbleby, 97]. Like the preceding examples does *Gentler* provide a simple mean of outlining Web site structures, which upon execution produces a site skeleton with inserted navigational mechanisms. A basic text-oriented HTML editor is provided for actual content editing. Any number of style specifications, which controls the target domain appearance, are supported in *Gentler*. Developers may create individual layout styles by intervening with *Gentlers* page design language, which is a basic macro language (see figure 2.8 for an example on the use of this macro language).

•homepage? •pagebody *Conditional pointing to pagebody, which just copies the current body to a standard page, if current page is the root page.*

```
•!homepage?                 Points to every other page than root page.
 [ •menubar
   <p>
   •has text?               Include next code snippet if page has textual content
   [ <table><tr>
     <td valign = top> •macro? [logo image file]</td>
     <td valign = top> •section? <h1> •title </h1>
                       •subsection? <h2> •title </h2>
                       <hr> •pagebody </td>
     </tr></table>]
   ] ...
```

Figure 2.8: *A sample use of Gentlers page design language (adopted from [Thimbleby, 97]). We have added a few explanatory remarks in italics to clarify the use of conditionals in the macro language. Notable is the intertwined use of macros and HTML codes.*

Gentler posseses a wide range of pre-fabricated macros, but developers are free to create custom ones for individual needs. Several useful features have been implemented in Gentler, for instance an awareness of page edition, which allows for automatically generated and annotated "What's New" pages. Extra careful attention have been paid to links in Gentler as described in section 3.3 of [Thimbleby, 97]. Navigational links are as mentioned before maintained automatically and need not concern authors. The authoring process is also improved by the ability to follow navigational links between every page in *Gentlers* document database, while editing page content. Index and index entries for documents are created automatically if the author only flags pages, words or phrases as indexable. In addition a summary page may be constructed showing relevant information on every external link existing in a site. The summary page is in HTML format allowing authors to easily check the reliability of outgoing links.

A system for managing and delivering distance courseware is presented in [Johnson, 96]. The paper adresses many issues regarding network usage, but our focus is on the special authoring language developed to facilitate the generation of the systems Web pages. Page templates are defined in the *ANDES Text Markup Language* (ATML), which extends HTML with a new set of special-tags (see figure 2.9 for examplification).

```
<@template>
<@type=open-exam>
<@title=Exam 1>
<@workshop=1A>
<@duration=60mins>
<@text=Exam on the first five lectures of the course. You have one hour to complete this exam.>
<@questions=Describe the special effects that can be used in film making....>
<@template>
```

Figure 2.9: A *template written in ATML (extracted from [Johnson, 96]). This example shows the template of type open-exam with a collection of field entries listed.*

Processing the above template yields a page of type open-exam generated on account of the specified attributes and corresponding values. One Web page implementation of the open-exam template might consist of the title, the elaborating text, the exam questions, text forms for typing answers and a submit button, as well as a built-in timer set to lock the page after an

one hour duration. Source descriptions are permitted to include both *ATML* tags and ordinary HTML-code.

[Owen, 97] considers an approach similar to *ATML*, where templates are defined on language level rather than upholding a high level object description. Keeping template descriptions on this low level is definite to avoid the application specific approach to site level authoring employed in high level object tools. The implementation considered in [Owen, 97] allows templates to be defined in the *Automatic Site Markup Language (ASML)*, which consist of a set of tags supplementary to HTML. The elements of *ASML* contains both translations of existing HTML elements (e.g. the *img* element), different high level elements performing advanced functionality (e.g. *index* and *search* elements, which will be discussed further below) and elements for describing additional templates. An illustrative example of the use of *ASML* is presented in figure 2.10.

```
{-- Sample ASML File --}
{base system="/user/konge/asmlpages" server="/asmlpages"}

{define name="top"}        { -- top of page template --}
<html>
<head><title>{title}</title></head>
<body bgcolor ="black">
{/define}

{define name="bottom"}        { -- bottom of page template --}
</body></html>
{/define}

{ -- Generate a page -- }
{page file="index.html"}
{top title="My page"}
<p>Content of page</p>
{bottom}
{/page}

{ -- Generate a second page -- }
{page file="index2.html"}
{top title="My page 2"}
<p>Content of second page</p>
{bottom}
{/page}
```

Figure 2.10: *Sample ASML source file (extracted from [Owen, 97]). The given source file actually generates two HTML pages, which share a common page header and footer defined in the top and bottom templates. Notice the mixture of ASML and HTML content in the source file and the syntactical resemblance of ASML and HTML tags.*

As clarified in the example above are new templates easily constructed using the *{define}* tag. Template variables are inserted enclosed in braces and are later assigned values by means of an attribute in the calling form (see the *{top}* template in figure 2.10). A basic set of tags are predefined in *ASML*. This set includes the *{page}* tag, which may be used to write content to an external file with the *file* attribute delivering the chosen filename. The *{base}* tag also used above ensures location independence, as generated Web sites may be produced at or relocated to any location in the file system. Strong capabilities for indexing content on Web sites are

provided, which may automatically generate a table of content for either the entire site, a group of pages or just a single page. Consequently is it possible to incorporate *ASML's* built-in search engine, which performs its searches based on a site index file built on generation time, for every page in the site. A highlighting function then allow search terms to appear highlighted on the locations of their disclosure. Other significant functions perform iterations and conditionals (foreach, if, else and elseif), while an import function makes the inclusion of content contained in an external format possible on a Web page (currently only Rich Text Format (rtf) is supported in *ASML*). Another interesting property of *ASML* is the ability to act as a CGI language. *ASML* source files may act as a traditional Unix script, by using the #!-notation to point to the path of the *ASML* executable (e.g. #!/usr/local/bin/asml). Conditional content delivered through HTML's form construct are automatically converted to templates in *ASML*, thereby enabling further processing by interacting with the textual value of these templates. [Owen, 97] claims to have made a tool with great flexibility and a high degree of automation, without requiring users to rely on strictly programmed solutions. We find the visions behind the tool excellent, but tend to disagree that users are not programming, when extending the template set and performing iterations and conditional checks.

Evaluation

Given that tools of the generating kind are suitable for larger Web systems, where a high degree of automation is often required, we intend to focus our further work on this area of tools. Generation tools should also prove be the area most in need of research, since almost every widely used Web-builder is an authoring tool or a conversion tool. The relatively sparse usage of generation tools have been proved by the examples discussed in this section, which have all been of experimental nature and have only been used in a local and minor educational community.

As we learned from the above set of examples do this category of tool range from highly specialized tools, which only needs a few user preferences to construct a site of related Web pages, to very flexible tools operating almost on programming language level. We experience a trade-off between flexibility of the tool and involvement of the Web designer. At one extreme we have a tool like *HTML Course Creator* from [Curtis, 96], where the users work is very limited and quickly done. The drawback is however the single area of use and the predefined appearance. Opposite we have tools like the *Automatic Site Markup Language* presented in [Owen, 97], which may produce every kind of Web application with any individual layout scheme applied. Of course this approach forces the user to both define templates for the given application and the desired appearance of such.

Seeking a flexible tool our investigation is bound to follow the path of language level tools such as *ATML* and *ASML*. The primary aim of *ATML* was not automating the constructing of large Web systems, it was merely seeking an appropriate way to deliver page information, which could for instance be used to provide additional functionality (such as the time duration put on an examination page). Contrary *ASML* presents an attractive way to automate generation processes, with a few conditional and iterative elements present in the language and unlimited abstraction powers offered by the ability to define new templates. But we do not totally agree with [Owen, 97], when they suggest no programming is required for building Web pages with *ASML*. We believe the work processes in *ASML* is indeed very similar to programming. Therefore should our future intentions be to take Web engineering one step

further to programming language level. This should give a purer language than *ASML*, where the mingling of control structures, HTML translated elements and higher level elements would be avoided. Besides would control structures be directly available instead of being accessed through element tags, while aritmethic operations and variables global for whole sites should be easily accessible. The idea of generating Web pages from ordinary programming language sources are discussed in further detail in the next section.

2.3 Programming Oriented HTML Page Generation

The idea of developing Web pages by means of programming duplicates the basic thoughts behind generation tools. In programming terms the abstraction powers are obtained through definitions of functions or procedures, which are analogous to the template creation of generation tools. This method of making abstractions corresponds to the concept of internal languages put forward in [Rosenberg, 98], where extensibility in hypertext systems may be provided by means of a programming language operating on a higher level. Programming languages are excellent to perform the automation of routine tasks, which promises a heavily reduced authoring and maintenance time of large Web systems. The possibilities of abstraction and automation are essential in the authoring process, as firmly stated in [Nørmark, 99], [Nørmark, 99a], [Owen, 97] and [Thimbleby, 97]. While programming languages inherently handles such mechanisms perfectly other advantages exist too. Non-trivial Web page content may be developed with programmed solutions, which makes heavily use of aritmethic operations or other programmical properties (e.g. an automatically generated page containing statistics of content hosted on different pages in a site). Noticing that Web pages today does not only consist of HTML code, but also scripts, applets, stylesheets and CGI scripts, we will make an effort to encompass as many as possible of these Web languages in our authoring language. We act fast however to discard script languages and Java-applets, because of the relative complexity of the involved languages. Making our abstracted language act as a CGI-script is probably a simpler task, as illustrated by the CGI support of *ASML*. Besides we intend to check whether anything is gained from incorporating both HTML and the *Cascading Style Sheet* (CSS) in our language.

Continuing with the programming oriented solution, the next major step is choosing a programming paradigm and a language suitable for an implementation. The only widely used paradigm for generating HTML pages is the imperative, which is the paradigm of the preferred CGI languages *perl* and *c*. Besides we stumbled on the functional programming language Scheme, when discussing the expanders (the actual HTML producers) of *HSDL*. The popular object oriented paradigm is also a logical contender. In chapter 4 we will study these three programming language paradigms in order to find the one most applicable for the job.

Our thoughts for both an internal language for generating Web pages and CSS and CGI-support of the same are summed up and explained in the next chapter. The problems we tend to solve towards these systems are identified and narrowed down by the formulation of a couple of hypothesisses. Our hypothesisses denote solutions to specific problem areas, which we then try to investigate and prove correct in the forthcoming implementation chapters.

3. Formulation of Problem

In the introduction we argued for the need of tool support in Web engineering and specifically for the need of authoring tools with some degree of automation available. The analysis investigated several alternative solutions for minimizing the work efforts needed for developing and maintaining larger Web systems. Early on the idea of replacing HTML with a more advanced language was discarded and consequently different types of Web tools were subjected to intense scrutiny in order to find the area most suitable for further research. Below we summarize the most important properties of the analysed tool types.

- **Authoring Tools:** Enjoys extensive commercial success due to the popularity of WYSIWIG tools, which practically render knowledge of HTML unnecessary. Some inflexibility may however be experienced, when control of the underlying HTML code is handed to the authoring tool (some tools are even guilty of generating proprietary HTML code only viewable by certain browsers). And eventhough means of site management have been included in most recent tools, we find the lack of support for automated solutions apparent in this category of tools.

- **Conversion Tools:** Had a considerable following in the early days of the World Wide Web, but have later almost exclusively been used in word processors in order to save uncomplicated textual documents as HTML files. Although converters usually present the user with a fully automatic process the exploitation rate of HTML's abilities is extremely low. Converters may come in handy when quick translation of already existing textual content to HTML is required, but using them for building complex Web systems would in the light of their poor utilization of HTML's powers prove to be a mistake.

- **Generation Tools:** No widely used or commercial HTML tool is currently of this category, they are almost exclusively used internally on scientific institutes or subjected to on-going research. Generators may range from stand-alone programs, which generates whole Web systems on the basis of a few user set preferences, to pure programming, where even basic structures should be defined and manipulated (the trade-off between user effort and system flexibility is instantly revealed). The variety of different kinds of generators were all discovered to hold excellent means for automation.

Based on the above listed observations, it seemed an obvious choice to implement a generation tool for the building of complex Web applications. The analysis further argued for keeping the Web construction on a programming language level, i.e. conforming to the mechanisms of an internal language, in order to utilize full flexibility of the HTML language. Considering this we decided to slightly rewrite the first two hypothesises from [Hellegaard, 99] (basically they mean the same):

- **Hypothesis1: Automated tools are needed in order to properly build and maintain large Web systems.** *By encapsulating common content in abstracted structures and performing an appropriate automating function upon them, we obtain material which is both consistent and easy to update. In comparison with a manual solution, where overall alterations is to be made by hand throughout a whole site, the automatic solutions would only require a slight update in the automating function and a snap recompilation.*

- **Hypothesis2: Highest level of flexibility is obtained by developing Web page systems with a programming language.** *Having a programming language working on top of HTML ensures both abstraction and automation powers through the inherent programming language abilities, while at the same time keeping at close hand every aspect of the HTML language.*

In [Hellegaard, 99] we delivered a hypothesis regarding the programming paradigm of the internal language, which we also preserve (again in a slightly rewritten form). Before and now we believe that the function oriented paradigm would be an excellent choice and this seems partly justified by the use of the functional programming language *Scheme* in the HTML expanders of *HSDL* as described in [Kesseler, 95]. In chapter 4 we intend to take a closer look and compare the imperative, object oriented and function oriented programming paradigms in order to identify the one best suited to implement an internal language for a markup language. This investigation should provide us with enough knowledge to prove or disprove the hypothesis presented below:

- **Hypothesis3: A function oriented programming language should provide an attractive solution as an internal language for a markup language.** *At first thought the function oriented paradigm seems to suit markup languages pretty well, as function calls holds a similarity to the tag applications of markup languages. And by defining functions to correspond to different markup elements, it also transpires that the tree structure of markup documents appears to be handled implicitly by the nesting of first-class functions.*

The previous report [Hellegaard, 99] suggested a co-operation between the implemented authoring language and XML, which was the reason behind two more hypothesises, but since we discarded the XML angle in this report, we no longer consider these. The desire to add additional language support to our authoring language has instead prompted the definition of another hypothesis. We have decided to investigate the possibilities for incorporating CSS and CGI in our language. These two languages are very different in nature and application. To use our language for CGI-scripts we should first of all be able to run a server side command-line interpreter of the language that we end up implementing our system in. Secondly should functionality for extracting and decoding the data structures send from the CGI-interface be build, whereupon the resulting HTML pages can be generated and send back to the client side browser. This is fairly speaking quite simple tasks, which can be carried out and checked for validity, when the authoring language is defined. CSS is another story entirely, because it concerns the whole authoring process. The fact that CSS statements may appear within HTML elements prompted our desire to encompass both HTML and CSS in the authoring language. This constellation might then prove to be a benefit for the developer, partly because he/she only needs to know one language (the authoring language) and partly because

structures with an integrated use of HTML and CSS could be build and trusted to uphold a very strict appearance. The fourth hypothesis was duly based on this speculation:

- **Hypothesis4: An integrated authoring language encompassing both HTML and CSS might deliver additional benefits for the developer.** *The obvious advantage of having HTML and CSS available in a similar syntax should not be underestimated. But the real benefit could surface when using HTML and CSS in a combined manner, where very presentational strict structures may be developed.*

The following chapters contain the proposed evaluation of the three most natural programming paradigms for implementing the authoring language, as well as the design, implementation and evaluation of the same authoring language. During the evaluations we engage in discussions regarding the truthfullness of the formulated hypothesises.

4. Choosing a Programming Paradigm

In this chapter we intend to specify the programming paradigm and programming language most suitable for the implementation of an authoring language targetting the markup domain. In order to obtain a firm grasp of the target domain we first engulf in an examination of the specific properties of markup languages. Initially we present the terminology of markup languages, which later provides a solid foundation for a comparison of the most fundamental characteristics of markup languages and customary programming languages. Equipped with this knowledge we proceed to investigate the three main programming paradigms (imperative, object oriented and function oriented) for compliance with the markup language domain. Following three sections devoted to the research of each of the suggested paradigms we conclude this chapter with a partial conclusion stating our decision on the matter and the reasoning behind.

4.1 Markup Language Properties

In order to pinpoint the most important properties of markup languages we investigate the different language components and syntactical features of this family of languages. Having established this specific type of language composition we intend to compare it to that of ordinary programming languages, which should ultimately offer suggestions for the task of modelling markup languages in a programming language.

4.1.1 Markup Language Terminology

As previously stated in [Coombs, 87] several different types of markup languages exist, but this investigation will be confined to only considering the descriptive kind. This somewhat restrictive selection policy is fully justified by the markup languages currently being used in the network domain and for general information storing (HTML, XML and SGML), which all belong to the descriptive type of markup languages. The superiority of descriptive markup systems over other types of markup systems was firmly established in [Coombs, 87]. Descriptive markup allows certain parts of a textual document to be specified by kind and optional knowledge of further functionality in a declarative manner, whereas other kinds of markup merely place system dependent coding for appearance and formatting issues at appropriate locations in the text sources.

In the markup languages recently enjoying widespread usage, e.g. HTML, textual elements are marked up with the use of tags (tag elements are distinguished in textual sources by applying angle brackets around the tag names, e.g. <sometag>). Most tags act as containers, where the desired textual content is surrounded by the easily distinguishable start- and end-tag. To serve as an example we consider the HTML element *p*, which denotes a paragraph in the text. Everything between the start-tag and the corresponding end-tag indicates a

single paragraph. Not all tags are containers though, as stand-alone elements may occur. Stand-alone elements only consist of the start-tag and accordingly do not hold any content, but merely indicate placement of specific elements in certain locations of a document source. A telling example from the HTML domain is the image (img) element, which denotes placement of graphical content at a specific point in the text. Both stand-alone and container elements may enhance their descriptive powers further by the appliance of certain optional attributes. Every element has a precise defined set of legal attributes and others are prohibited or take no effect (according to the given language implementation). Typically attributes are accompanied by a related value, but specific implementations may permit the use of minimized attributes (attributes with no value), as is the case with HTML but not with XML. Syntactically attributes reside within the angle bracket boundaries of the tags (in the case of container elements attributes belong within the start-tag), as the following example clarifies. To insert an image in a HTML document the source attribute of the *img* element requires a value of type URL (Uniform Resource Locater) to be set, as sketched below:

```
<img src="/pics/myself.gif">
```

Having described the basic components of markup languages we proceed to mention a few important properties of complete markup documents. Every markup document is obliged to have a single root element containing the rest of the document elements (HTML sources are commited to having the *html* element as the root element). Furthermore are tag elements expected to nest properly, i.e. elements cannot be allowed to overlap each other. For examplification consider the code snippet depicted below, which delivers an improper syntax:

```
<deceased-dictator>Adolf<last-name>Hitler</deceased-dictator></last-name>
```

The legal interpretation of the above code snippet should read:

```
<deceased-dictator>Adolf<last-name>Hitler</last-name></deceased-dictator>
```

Conforming documents to these regulations essentially makes the overall structure of a markup document correspond to a tree structure. For conviction we give the XML document (containing information of a collection of video cassettes) below, which corresponds to the tree structure depicted in figure 3.1.

```
<videos>
    <owner>Carsten Hellegaard</owner>
    <science_fiction>
        <video director="Ridley Scott">Alien</video>
        <video director="James Cameroun">Aliens</video>
        <video director="David Fincher">Alien 3</video>
    </science_fiction>
    <horror>
        <video director=”Brian de Palma”>Carrie</video>
        <video>The Excorcist</video>
    </horror>
</videos>
```

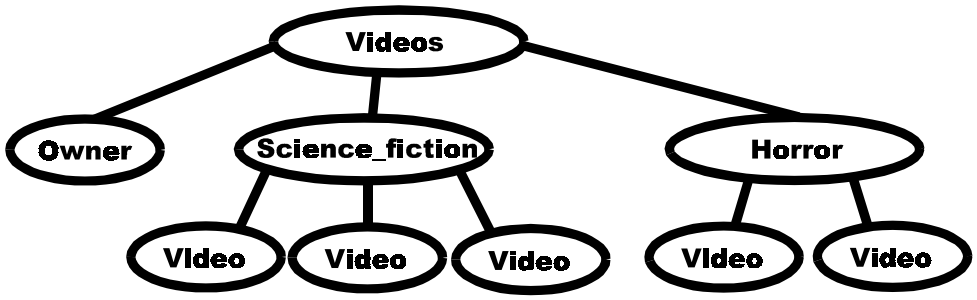

Figure 3.1: *Tree structure of an XML document. The root element (videos) of the XML document becomes the root of the corresponding tree structure. The different elements nested within the root element of the XML document duly match the branches ramifying from the root of the tree.*

4.1.2 Markup Languages versus Programming Languages

A natural starting point of our comparison of markup languages and programming languages is the lexical composition of the source documents of the two different language types, as presented in [Nørmark, 99a]. Markup documents are very much text based, while the textual content of programming language sources are limited to value types such as characters and text strings. Often text manipulation is reduced to play only a minor role in a programmed solution, because strings make up just a small part of the overall picture.

As may be deduced from the discussion in the previous section are markup documents basically a text string with certain inserted tokens. Some kind of processing of the document is later required in order to obtain a meaningful interpretation of these inserted markup elements. In other words we understand the markup language sources to be *textual documents hosting the different markup elements*. The lexical composition of program language sources, i.e. programs, is significantly different. Every part of a program is well-known, as they consist entirely of well-defined tokens, such as command, procedure or function names, operators, variables and values of a specifically defined type. Programming languages constrain pure textual content to characters or text strings, which are value types enclosed in quotation marks. Focusing on the text management of programming languages we therefore express *text strings to be hosted within programs along with other value types*. This proves an important discovery, when we come to considering the task of generating markup documents with programming languages. Accomplishing this would effectively mean the construction of one large text string (being the whole markup document) through extensive text string manipulations in the programming language of choice. Later we should discover whether the different handling of textual content in the two types of languages would have any importance in the string handling of our authoring language implementation.

Whereas programs must conform very strictly to well-defined syntactic and semantic rules of a given programming language, this is not always the case with markup languages. Markup tags are defined on the syntactical level through the Document Type Definition (DTD) mentioned before in the analysis, but the degree of strictness may vary. The DTD of HTML,

which is hardcoded into the well-known Internet browsers, may accept incorrect HTML code by simply ignoring ill-formed code or even by trying to display it anyway. With the emergence of XML it suddenly becomes possible to custom design and apply different DTDs, which put strict demands on the validness of XML documents, but still XML documents may pass as being well-formed if they oblige only certain minor syntactical rules. For viewing purposes, e.g. on the Web or on paper, different stylesheets for different presentations of the same document may also be defined. To summarize the above we find markup documents allow for very different processing by external instances, while programs holds very strict uniformly processed content.

To end this prior investigation of the two types of languages we find it appropriate to once again mention the main difference and the reason why we seek a programming paradigm for a authoring language in the first place. Markup languages offers none of the programming capabilities seen in full-fledged programming languages. Consult the analysis of chapter 2 for the discussions on how to incorporate algorithmic functionality in markup systems, as initially suggested by [Rosenberg, 98].

4.2 The Imperative Paradigm

The imperative paradigm provides a rich inheritance and it has long been viewed as the "traditional" model of computation. Further evidence of the early supremacy of the imperative paradigm is provided by the long list of heavily used imperative languages, such as FORTRAN, COMAL, BASIC, Pascal and C. The imperative model of computation closely resembles that performed by the computer on the underlying hardware. Most modern computers conform to the von Neumann model, where values are collected from a memory store to be manipulated by a processing unit, with the new calculated value being taken back and stored in the memory. A program on this low machine level represents a sequential pattern of instructions, which all alters a part of the memory. Following the computation of the instruction set the desired solutions is some or all of the calculated values now residing in the memory.

Imperative programming languages build upon this low level model. Higher abstracted datastructures (e.g. records and arrays) and value types (e.g. strings, reals and booleans) typically exist on top of the single numerical value experienced on machine level. Another abstraction very much associated with imperative programming is the variable. Variables are identifiers holding the value of a given data type, with this value being prone to extensive modification over time. The imperative style is in fact easily recognizable by the endless incremental transformation of state (memorized values) and the corresponding time aspect (changes performed over time). Initially imperative languages closely resembled the sequential program structure seen at machine level, as programs were made up of a series of commands, from which the most notable are variable assignments and control flow constructs, such as loops (either *while* loops or *for* loops) and conditionals (typically an *if - then* constellation). Later the imperative model was extended with modularization means for building and reusing chunks of often used code, such constructs are widely known as *procedures* and *functions*.

In recent years new and different ways of envisioning the computation task have been introduced (we address the two most important ones in the forthcomming sections) to accommodate a rising disbelief with the imperative languages. Negative voices were raised largely because of the close match between the imperative paradigm and the actual executions performed by the computer on the most fundamental level. At first this togetherness would appear as an advantage, but the way of thinking employed in the imperative languages is not particular reminiscent of the way human beings intuitively solve problems. Formulating problem solutions in an imperative style might therefore prove a quite difficult task, and with a difficulty prone to increase dramatically as the complexity of the problem rises.

Returning to the task of producing markup code we find the need to build a single text string representing a markup document from some high level structures in an imperative language. To illustrate how this may be done we have constructed a simple example, which generates the small HTML document depicted below:

```
<HTML>
    <HEAD><TITLE>HOME of GALACTIC TOURS</TITLE></HEAD>
    <BODY>
        <H1>GALACTIC TOURS offers you:</H1>
        <UL>
            <LI>MARS</LI>
            <LI>NEPTUNE</LI>
        </UL>
    </BODY>
</HTML>
```

We have chosen to construct the example in the imperative language *Perl*, which have been widely used to generate HTML code, especially for CGI purposes. The language Perl holds many similarities with C, but its functionality have been specifically biased towards string manipulation by the addition of several strong string mechanisms, while other advanced C functionality, such as pointers and user-defined types have been excluded. The experience from languages like Perl shows that string building in imperative languages is often a matter of continuous use of print statements (in the case of CGI programming are the HTML documents build by printing a coherent set of strings to the standard output, from where it is delivered to the client side browser). Our intentions with the Perl example were to utilize a higher degree of modularization, where re-usable units delivering markup code for different ad-hoc structures should be defined through the use of procedures. Analogous should procedures be constructed to cope with the basis elements of the target language in order to grasp the full expressive power of the target domain. A Perl program generating the HTML document presented before is given below:

```
# main
# ------

&top ("HOME of GALACTIC TOURS");
&HTMLelement ("H1", " GALACTIC TOURS offers you:");
&ULlisting ("MARS","NEPTUNE");
printf ("</BODY></HTML>");
```

```
# subroutines
# ------------

# top takes one argument: title
sub top
  {
     printf ("<HTML><HEAD><TITLE>");
     printf ($_[0]);
     printf ("</TITLE></HEAD>");
  }

# HTMLelement takes two arguments: the element name and the content
sub HTMLelement
  {
     printf ("<",$_[0],">");
     printf ($_[1]);
     printf ("</",$_[0],">");
  }

# ULlisting takes a series of list items as arguments
sub ULlisting
  {
     printf ("<UL>");
     for ($i=0;$i<@_;i++)
        {
           printf ("<LI>",$_[$i],"</LI>");
        }
     printf ("</UL>");
  }
```

The example portrayed is of course simplified, e.g. are the attributes of HTML elements not yet supported and no distinction is made between container and stand-alone elements. But the primary objective of constructing different higher level functions is obtained, which the top and ULlisting subroutines witness. While this example is probably not complex enough to fully appreciate these abstraction powers, we stress again that the advantage gains in momentum as the complexity rises.

The presented example and the extensive use of imperative languages for CGI purposes suggest that imperative languages may be useful as authoring languages. But considering the frequent use of print statements and the slight incompability between imperative languages and markup documents, we feel obliged to continue our investigation and discover whether another programming paradigm would constitute a better solution.

4.3 The Object Oriented Paradigm

The principles and techniques of the object oriented programming paradigm was originally conceived in order to better model problem areas. Solving potential problems in a more human like manner, where the problem area is modelled as objects instead of the computer hardware emulating approach of the imperative paradigm, should ensure better handling of complex problems, improve the quality of programs, increase the degree of re-usability and reduce efforts required for maintenance. The object oriented approach made its first important appearance with the *Smalltalk* language, but later widely used object oriented languages such as C++ and *Java* have surfaced.

The basic object oriented concepts, which we try to clarify below, are partly deduced from [Mathiassen, 93]. The fundamental principles of object oriented programming are abstraction and encapsulation. All important entities of the problem area and their related states and behavior are encapsulated by an object. By state we refer to the attributes and properties of an object at a given time, whereas an objects behavior describes the possible actions it can perform. When all essential parts of a problem domain have been modelled by appropriate objects, these objects may interact with each other. Interactions are performed through well-defined interfaces and may for instance be the reading of an objects present state or the triggering of an object method (methods present the possible actions an object can perform). Making any outside contact adhere to the rules of the object interfaces ensures that objects are treated like black boxes. This preserves the independence of the objects in the overall system, as changes within an object remain isolated from the the rest of the system, which only see and interact with the applied interface. Besides is the object structure persistent, as only states may undergo changes, while the overall object identity remains the same.

An important notation in the object oriented paradigm is that of classes. Objects with the same properties and behavior belong to the same class. An object belonging to a certain class is said to be an instance of that class. In order to allow objects to reuse definitions of properties and behavior of other objects it is possible to organize classes in hierarchies, where classes may inherit the functionality of its parent (some languages restrict classes to inherit only from a single parent class, while others allow for inheritance from multiple parent classes). Depending on the point of view inheritance between classes are understood to be either specializations or generalizations of each other. Furthermore may objects be related through aggregation or association. Aggregations emerge when a superior object are related to a number of inferior objects, which can all be viewed as parts of the superior object. Associations represent a connection between two objects of equal stand.

The above discussed characteristics of object oriented systems are very helpful in terms of simplifying the design processes. Humans find the transition from problem domain to a potential design solution to be simpler and more tangible with the modelling of related objects in the problem area, instead of the more distant and less natural solution models experienced with other methodologies. The more intuitive modelling of the problem domain through objects also provides a sounder basis for coping with a potential increase of complexity in the system. The problems of maintenance have become less complicated too, as some changes can be done within the isolated parts of the objects without interfering with the overall

system. Likewise do the self-contained object abstractions provide for excellent re-usability not just through inheritance but also across different systems.

To show some of the possibilities for generating HTML documents with object oriented languages, we provide the following small program in the popular Java programming language, which produces the HTML document known from the example in the previous section:

```
class myPage extends Page {

    public static void main (String args[ ]) {

    myPage m = new myPage();
    m.top("HOME of GALACTIC TOURS");
    e = new htmlElement ("H1", "GALATIC TOURS offers you:");
    e.expand();
    u = new ullisting();
    u.expand ("MARS", "NEPTUNE");
    m.bottom();
   }
}

class Page {

    void top(String title) {
       System.out.println ("<HTML><HEAD><TITLE>" +
                           title + "</TITLE></HEAD>");
     }

    void bottom() {
      System.out.println ("</BODY></HTML>");
     }
}

class htmlElement {

    String name;
    String content;

    htmlElement (String n, String c) {
      name = n;
      content = c;
     }

    void expand() {
      System.out.println ("<" + name + ">" + content + "</" + name ">");
     }
```

```
// Overriding method for direct access
      void expand(String nam, String cont) {
        System.out.println ("<" + nam + ">" + cont + "</" + nam ">");
       }

}

class ullisting {

      void expand(String[ ] listing) {
        System.out.println ("<UL>");
        int i;
        for (i=0; i < listing.length; i++)
            System.out.println ("<LI>" + listing[i] + "</LI>");
        System.out.println "</UL>");
       }
}
```

Again it is a minimal and uncomplicated program example, where not all of HTML's functionality is supported. But the basic ideas are apparent and we see that the object oriented paradigm offers plenty of functionality for upholding abstracted representations of descriptive markup documents. In the more advanced case we would probably ensure that every available tag element of the target language was defined as a class in the object oriented language, instead of the minimal *htmlElement* class from the given example (which doesn't cope with attributes or distinguish between container and stand-alone elements). For ideal deployment should this collection of classes inherit basic functionality, such as methods for performing the actual HTML generation, from a parent class. Different related markup elements may even be further specialized through another common class, e.g. could the phrase definitions (such as em, strong and cite) of HTML share a superclass holding the properties of their common set of available attributes. Another obvious division of common behaviour through inheritance is the case of container versus stand-alone elements. With the basis of the target language defined by classes, every tagged element of a document source should be represented by object instances of these classes.

Certain ad-hoc functionality may be quite practical at times as the *ullisting* class shows in the above example. But the real advantages of object oriented languages are first fully utilized, when the crafty object mechanisms are exploited. The powerful means of object connectivity provided by object oriented systems presents a convenient way of describing structural relations of the target documents. Developing new higher level structures is also made easy by object aggregation of already existing objects, which may both be the basis element emulating objects or other aggregated objects. An illustration of this powerful mechanism is provided in the Java example, where the actual page generating class *myPage* inherits the methods of its superclass *Page*. In the above example *myPage* makes use of the inherited methods *top* and *bottom*.

To summarize we believe the object oriented paradigm could provide an attractive solution for the authoring language and it undoubtless offers more advanced functionality than the imperative paradigm. Some degree of consideration should however be put in the choice of

language. With limited syntactical flexibility it might soon become practically inapplicable to use certain languages. The Java example of above helps to support this view with for instance the many instantiations needed.

4.4 The Function Oriented Paradigm

The principal idea behind the function oriented paradigm is the construction of a program notation reminiscent of mathematical expressions. Making programs follow known mathematical concepts presents the function oriented programmer with a huge amount of proven manipulation techniques. Naturally function oriented programming offers the most when working with problems of a formal character. This may be the reason why traditional program development have concentrated on imperative and lately object oriented languages, while the function oriented languages have enjoyed its highest popularity in scholarly environments. Originally LISP started the function oriented trend several decades ago and LISP related languages still exist (e.g. CommonLisp and Scheme). But recently languages such as ML, Haskell and Miranda have earned themselves a worldwide reputation.

The function oriented paradigm has a range of fundamental characteristics, which separates it considerably from the two paradigms discussed prior to this section. As we briefly mentioned above are computation in the function oriented paradigm performed by applying functions to values analogous to a mathematical expression. Repetitive use of functions is commonplace with additional functions applied to the result generated by a prior function. The whole concept of values is quite different to the "traditional" one employed in the imperative and object oriented paradigms. Assignment of values to variables capable of later mutation is generally prohibited in function oriented languages. It is possible to have identifiers denoting specific values, but once created it is impossible to later modify its value. So instead of performing incremental changes on existing structures, the function oriented paradigm transforms existing values into new values, which will exist independently of the original from then on (this mechanism is usually referred to as "single assignment form"). If we for example were to add a single value to a relatively large list (one of the most important data structures of function oriented languages is the list for which they normally contain comprehensive and powerful manipulative mechanisms) it would result in a whole new list, rather than a slightly modificated version of the original list.

Also the notion of "time" is very different in function oriented programming compared to the two before mentioned computation paradigms. Functional programs do not uphold the strict sequential nature of imperative programs, nor do they treat values as states being modified over time. Instead are functional programs said to be *atemporal*, i.e. without time. Being independent of time indicates that the execution of functions may occur in any sequence. Special for function oriented languages is the freedom applied to the order of evaluation, where strict semantics signifies that functions are not allowed to be applied before the required arguments are available (as a direct consequence are other functions dependent on the outcome of this particular calculation obliged to wait for a result). Lenient semantics on the contrary permits functions to have one or more undefined arguments. Some modern function oriented languages also provides lazy evaluation, which makes it possible to reduce calculation time of large functions by only calculating identical expressions once. Regardless

of any time aspect may the function oriented programmer rest assured, that a function delivered with the same set of arguments will always produce the exact same result.

Another strong mechanism included in most function oriented languages is the possibility of considering functions as first-class data values. For a function to be regarded as first-class it should be able to be assigned to identifiers, to be passed on as argument and to be returned as the result of an execution of another function. Needless to say that first-class functions provides the function oriented programmer with a high degreee of flexibility.

For languages conforming purely to the function oriented paradigm some problems involving the imperative nature of the surrounding computer environment are almost certain to arise. For instance could providing a pure function oriented language with means for basic file input and output prove to be a quite complicated affair. This inherent frailty of the function oriented paradigm have lead to the development of function oriented languages containing a certain amount of imperative means supporting assignment, mutable data structures and input/output functionality. The benefits of this multi-paradigmatic approach, where the function oriented mechanisms are still in charge and only supplemented by a few alien commands, often outweigh the loss of paradigmatic purity.

We now proceed to considering the task of modelling markup documents in a function oriented language. Below we have constructed a program in a LISP-like language, which generates the HTML document of the previous examples.

```
;; Top level function
;; ----------------------
(page "HOME of GALACTIC TOURS"
        (string-append
             (htmlelement "H1" "GALACTIC TOURS offers you:")
             (ullisting (list "MARS" "NEPTUNE"))))

;; Sub functionality
;; ---------------------
(define (page title body)
    (string-append
       "<HTML><HEAD><TITLE>"
       title
       "</TITLE></HEAD>"
       body
       "</BODY></HTML>"))

(define (htmlelement name content)
    (string-append
       "<" name ">"
       content
       "</" name ">"))

(define (ullisting li-list)
    (string-append
```

```
        "<UL>"
        (process-list li-list "")
        "</UL>"))

(define (process-list lst res-str)
     (cond ((null? lst) res-str)
           (else (string-append
                  res-str
                  (process-list (cdr lst)
                                (string-append "<LI>" (car lst) "</LI")))))
```

This example resembles the procedure of the imperative and object oriented implementations, but we have obtained a more flexibel syntax. A deciding characteristic of function oriented language is however due to dictate some improvements to this approach. Because of the unique first-class functions supported in function oriented languages it is possible to use a syntactic composure more alike the one of markup documents. As a basis level we should construct a total mapping of the available markup elements of the target domain. Having functions, which correspond to the respective elements of the target markup language, generate markup tags and attribute-lists if necessary, offers a sound foundation on which to perform further computation (construct abstracted elements and perform automation tasks). The sample code depicted below reveals a certain similarity between tag application in a markup language and the first-class functional expressions as experienced in a function oriented programming language:

```
HTML:    <center><strong>Hello world</strong></center>

LISP:    (center (strong "Hello world"))
```

Notice how the above LISP statement is possible because function *center* takes function *strong* as argument. The *center* and *strong* elements of HTML both have an associated function in LISP, which generates the markup code particularly needed for their respective element. Because LISP evaluates the innermost function first (to accomodate the inherent functional rule of not executing a function until its arguments are available) is the function *strong* first executed with argument "Hello World". In turn this calculation should return the string "Hello world" as argument to function *center*, which will conclude the transformation to the correct HTML code.

Upon closer examination of this example we deduce that functional expressions match the hierarchical structure of markup documents very well. The nesting of elements in markup documents are duplicated by the nested activations of functions in an expression. Besides getting the structural part covered in a very natural way, we also notice the very similar syntactical composition. A future stumbling block could however be the addition of element attributes, which would require some sort of parameter identification by the involved functions in order to distinguish content from attributes (a further discussion on the subject is presented in section 5.1.2). But overall does the function oriented paradigm offer an elegant solution, because of the markup resemblance and free syntax.

4.5 The Motivated Choice

Going through the three possible programming paradigms for our Web authoring language we found the imperative paradigm to be the least attractive alternative. Imperative languages have limitations compared to both the object oriented and the function oriented languages. They haven't got the object connectivity of the object oriented paradigm or the proper nesting characteristics of function calls as experienced in the function oriented paradigm. The object oriented might offer an acceptable solution, but we are determined to concentrate on the function oriented paradigm. The motivation behind this decision is the syntactical advantages of functional languages and the already portrayed similarities between markup document structure and functional expressions.

Secondly, are the unique syntactical properties of function oriented languages especially fitting for our purposes. Because we are dealing with syntaxes of few constraints is it extremely easy to implement further functionality to the already existing language basis. A very high degree of extensibility is obtained with mechanisms for syntactical abstraction, where new functions are just added to the pool of existing functions. As a consequence will our Web authoring language end up being an actual programming language extended with functions for generating markup elements (functions generating code for higher abstracted elements and functions performing other programmed calculations are also just extensions to an existing language). Obtaining such a degree of flexibility in an object oriented system would present an overwhelming task with most languages. In the end does this intermingling of functionality mean that the high level source descriptions of Web pages are actually programs. The advantage of this approach is having the programming capabilities of the language working directly together with the markup components (both abstracted and basis elements). Compare this method with the mix of HTML code and advanced functionality being employed in ASML, as we described in section 2.2.3 (or refer to [Owen, 97]). ASML sources must first be parsed in order to extract ASML code for further processing and no actual programmability is free at hand in the ASML tags.

For the actual implementation we have chosen the language Scheme, which is a member of the LISP-family. Scheme is small in the sence that it lacks several characteristic but complex LISP functionalities. It is however still a quite powerful language. Scheme is a multi-paradigm language, thus with the main paradigm being the function oriented. The strong support of function oriented programming in Scheme is supplemented with some traditional imperative means. Scheme is common to languages in the LISP-family endorsing the parenthesized prefix notation. We partly choose Scheme because of its simplicity and partly because of the prior employment of Scheme in this area by [Kesseler, 95] and [Nørmark, 99].

The next chapter presents the Scheme implementation of an authoring language along with some developed examples of applications and thoughts concerning additional language support for CSS and CGI.

5. Lisp Abstracted Markup Language

This chapter covers the design and implementation of our new language for generating HTML based Web pages; *Lisp Abstracted Markup Language* (LAML). First a few general design criterias are presented. Then following a minor examination of the structural appearance of HTML, we investigate schemes for the syntactical composition of LAML, which provides the basis for a decision on the matter. Inherent indifferences between markup languages and programming languages are subject of a few headaches, among which the implicit nesting of strings in HTML requiring special care in Scheme presents the dominant one. Accordingly one possible solution regarding this particular problem is given. Next the basic parts of the LAML language, consisting of a complete translation of existing HTML elements to Scheme counterparts alongside several minor libraries handling time issues, file operations and other often used functionality, will be presented. We proceed to explain the use of application specific document styles, while showcasing several developed styles with accompaning examples of generated Web systems. Another important aspect of the implementation phase was the incorporation of *Cascading Style Sheets* (CSS) support. CSS language properties are examined and a LAML translation is constructed. Characteristics of LAML's duality, while encompassing both HTML and CSS, are discussed and experiments are conducted for further examination. Finally we argue in favour of also using LAML for CGI purposes.

5.1 Design

This section first describe a few design criteria we formulated for LAML. These design criteria are partly based upon the solution model drawn up in the problem chapter. Next we present the different components of the LAML system together with the implementation choices we made for the respective parts.

5.1.1 Design Criteria

One of our most important design criteria was the desire to use a function oriented programming language to implement LAML, which we established in the previous chapter. The function oriented language of our choice became *Scheme*, a very pure and simple dialect of LISP. Other members of the LISP family (e.g. Common LISP) posses considerable high level functionality, most of which have been stripped away in Scheme. However, Scheme does not suffer much from this lack, as it is still a very powerful language. Notably Scheme honours the concept of first class functions, meaning that functions can be assigned to identifiers, passed as argument to other functions and be returned as result from another function. This property makes Scheme a very flexible language.

A second fundamental issue was the desire to uphold a language level similarity with HTML. Developers already familiar with HTML would easier cope with the transition to LAML. This is beared in mind when the syntactical appearance of LAML is decided upon.

Another important design criteria for us was the flexibility of the language. LAML should cope easily with future extensions or updates to the HTML language. A future adoption to support generation of content in another target language than HTML would also benefit significantly from such flexibility (e.g. a future task of constructing a tool for XML could for instance be simplified).

A minor design criteria for LAML was the readability of our target code. Viewing the HTML source code (e.g. in a browser) is practically impossible without some means of pretty printing in our target code, because the source otherwise consist of a single line of compound text. Finally we wanted the LAML language to express meaningful descriptions of occuring errors, in order to ease eventual debugging tasks for the Web page developer.

5.1.2 LAML Syntax

The first issue coming to mind during the implementation of a new language is the syntactical appearance. As we have earlier pointed out are function oriented languages excellent to represent the different markup elements of HTML. As sketched below are the tag braces of HTML (< and >) and the corresponding ones of ATML (<@ and >) and ASML ({ and }) substituted with parenthesisses in LAML. Note also the absence of closing elements in LAML.

```
HTML:    <center><strong>Hello world!</strong></center>

Scheme:  (center (strong "Hello world!"))
```

However, a stumbling block is soon experienced by the addition of element attributes, which in HTML reside inside the tag of stand-alone elements and inside the start-tag of container elements, as is depicted below:

```
<tag-name attribute1="value1" attribute2="value2" …>

or

<tag-name attribute1="value1" attribute2="value2" …>content</tag-name>
```

Some programming languages allow such attributes to simply correspond to keyword parameters, but these are unfortunately not supported in Scheme. Some other mean of handling attributes is therefore necessary in the present case. Exploring this issue further quickly yields a simple approach when dealing with single tagged elements. Since single tagged elements never contain anything, only attributes and their corresponding values may be expected. So passing attributes and values in an optional property list to the appropriate Scheme functions are a plausible solution:

```
(function 'attribute1 value1 'attribute2 value2 ...)
```

Turning our attention to the case of container elements and their probable content, we realise that we need some way to separate content from the attributes and their values, when passing it all to an n-ary Scheme function. The following three issues can be identified as central to the syntactical variations of possible constitutions of LAML statements [Nørmark, 99a]:

The sequencing between content and attributes
To mirror HTML best, attributes should come before content

Explicit listing of attributes
Uniting all attributes and values in a list ensures easy access by the function

Explicit string concatenation of content
Uniting all content to one string ensures easy access by the function

As we adressed in the summation of our required design criteria the ideal LAML statement would preserve the HTML sequencing of content and attributes, while implicitly handling the attribute and content contributions. The syntactical variant of LAML adhering to these principles would read as the following:

```
LAML calling form:    (tag 'a1 v1 'a2 v2 "text1" "text2")

Scheme definition:    (define (tag . parameters)  ... )
```

While this variant keeps attributes before content and needs no explicit handling of either attributes or content, it places the responsibility of identifying each of the actual parameters to the Scheme functions in question. One possible way of accomplishing this constraints the developer to a somewhat strict use of parameter types. Having attributes and content conform to the Scheme types of *literal* and *string* respectively paves the way for a quick recognition of the different constituents of a parameter list.

An alternative using explicit attribute listing is given below:

```
LAML calling form:    (tag (list 'a1 v1 'a2 v2) "text1" "text2")

Scheme definition:    (define (tag attributes . contents) ... )
```

Again we have attributes appearing before content, but the explicit listing of attributes is necessary in order to only pass one parameter for all attributes and values to the Scheme function. Eventhough the Scheme functions have the required knowledge of its given parameters the explicit listing seem unfortunate, mainly because of the additonal workload of the developer while listing attribute pairs (code clarity may also suffer from this approach). The minimal case of double tags without attributes presents another obstacle in this approach, as it would require either an empty attribute list to be sent along to avoid mistaking the first textual parameter for an attribute list or more sensible methods in the Scheme functions for determining the abscence of the attribute list.

Instead we try a third alternative, which is explicit string concatenation. Notice that this approach forces the content to appear before attributes in order to take advantage of the Scheme dot-notation.

```
LAML calling form:    (tag  (string-append "text1" text2") 'a1 v1 'a2 v2)

Scheme definition:    (define (tag content . attributes)  ... )
```

This syntactical variant also provides Scheme functions with easy access to all parameters, but as mentioned it does suffer from both explicit string concatenation and wrong sequencing of attributes and content. Besides we must have in mind double tagged elements without content, as we in the previous alternative should account for elements with no attributes. Speaking in favour of explicit string concatenation is however the fact that double tags without content seldom occur and these rare cases would only have be met by passing the empty string as the first parameter, contrary to providing an empty list. Keeping in mind that LAML requires explicit string concatenation anyway, as we will elaborate on in section 5.1.3, we only feel obliged to address the sequencing problem further.

Our concern for the positioning of the attributes compared to the content may seem a bit exaggerated. But Web designers, who are already experienced in programming HTML would probably find the shifted positions akward. And more importantly do functions containing content with deeper nested elements have an irritating side-effect when attributes are present. The following table might clarify this problem (The Scheme functions in the example below are mirrors of the corresponding HTML elements. The Scheme mirrors are discussed in furtner detail in section 5.1.4):

```
(html:table
  (html:tr
    (string-append
      (html:td "Player")
      (html:td "Team")
      (html:td "Goals")
      (html:td "Assists"))) 'bgcolor 'green 'border 0)
```

It may look rather confusing that the attributes of the function *html:table* are located so far from its possesor (it would look even more confusing if attributes for *html:tr* were included, as they would reside just prior to those of *html:table*), contrary to HTML where attributes are kept close to their respective owners. So solving this problem has become a priority. A little manipulation with the powerful list mechanisms of *Scheme* is however enough to do the trick. We end up with the following syntactical appearance of LAML statements:

```
LAML calling form:    (tag 'a1 v1 'a2 v2 (string-append "text1" text2"))

Scheme definition:    (define (tag contribution1 . contribution2) ... )
```

Knowing that the content will always be the last item in a joint list of contribution1 (which naturally contains the literal *'a1*) and contribution2 (which is a list of *'v1*, *'a2*, *'v2* and the combined string of all content), we only need to extract this last item from this list and both

the content and the attribute list are easily accessible by the Scheme functions. To obtain the content from the end of the joint contribution list, we engage in a simple but somewhat time consuming list operation. By reversing the joint list, grabbing the first item and then reversing the remaining list again we manage to obtain the content and the correctly ordered attribute list. Eventhough the reverse function may not be the most time effective, we justify our simple solution by the fact that attribute lists may seldom be very long, thereby making the list operations fairly fast. The suggested algorithm is of course also correct in the axiom case of no attributes.

Should the time issue become the highest priority, another approach could be used at this important juncture of very central functionality, which is bound to be executed often. One method would be to expect developers to use a strict form of typing on the different parameters, as attribute-list and content could then be identified by their type (attributes as literals with an optional type following and content as a string). This approach would require only one loop through the whole parameter list, contrary to the extensive list manipulation carried out in the current version of LAML.

The attribute list

Having chosen a general syntactic appearance for LAML statements, we had some additional thoughts concerning the attribute lists. As mentioned earlier we expect the optional attributes to be ordered in a property-list, meaning they should occur in pairs with the attribute name first and the value second. An example:

```
(html:td 'colspan 3 (html:font 'size 2 'color 'green "Killroy was here!"))
```

A potential concern was the legality of minimized attributes in HTML (notice that they are in fact illegal in XML). Minimized attributes do not necessarily have an associated value and examples from HTML include *nowrap* and *noscroll*. Attribute names with no assigned value would of course cause havoc in an LAML attribute list, as names and values would be mixed (the property list could also suffer by an uneven number of elements causing the content to be mistaken for an attribute value). Where we to include the nowrap attribute in the example from above, it would read:

```
(html:td 'nowrap 'colspan 3 (html:font  'size 2 'color 'green "Killroy was here!"))
```

This is of course false, as the attribute named *nowrap* has the value colspan and the attribute named 3 has no value. The conclusion being that minimized attributes should be obliged to have a value. Noticing that HTML accepts the empty string as a value for minimized attributes, we might conform the example from before to the following correct form:

```
(html:td 'nowrap "" 'colspan 3 (html:font  'size 2 'color 'green "Killroy was
here!"))
```

When the attribute lists are formed, the attribute names are looked up for legality checks according to a list containing the different elements and their respective available attributes. Currently this list is deduced and edited manually from the most recent version of HTML, whereas a future version of LAML might be able to automatically detract this information

directly from a given language-DTD. Another issue for a later and enhanced version of LAML could be the inclusion of typechecking of the attribute values. This is not included in the current implementation, mainly due to some attributes ability to occur in different contexts with different elements.

As a last note on attributes, we decided to enclose attribute values in double qoutes (this is optional in HTML). We partly did this of aesthetic reasons and partly because it will become a necessity for the attributes of the forthcoming language XML.

5.1.3 Explicit String Concatenation in LAML statements

As we faintly discussed above will explicit string concatenation be needed in LAML. This inherent problem is due to the nature of LISP, which of course is somewhat different from a markup language. Markup languages handle string concatenation implicit contrary to most programming languages. This phenomenon is tried clarified below:

```
<center>Something very <strong>exciting </strong>is written here!</center>
```

This HTML snippet would be expressed as follows in LAML:

```
(html:center ”Something very (html:strong ”exciting”) is written here!”)
```

This is however wrong. The *html:center* function expects the embedded function *html:strong* to be evaluated to a string and concatenated with the other string snippets. But Scheme does not accept the string ”exciting” to be nested within the overall string. When the function *html:center* traverses its content for other Scheme expressions to evaluate it reads the following:

The string *”Something very (strong ”*
The symbol *exciting*
The string *”) is written here!”*

Having the start- and end-qoutes differ or using different sets qoutation marks for nested strings (see example below) could maybe solve the problem.

```
(html:center ”Something very (html:strong ’exciting’) is written here!”)
```

This solution is not very useful either, as nested strings within the nested string would need yet another par of different qoutes. Besides does both solutions suffer from the need to alter the Scheme interpreter to understand the use of other quotation marks.

Instead we chose to perform string concatenation explicit, on the expense of simplicity compared to HTML. But this seems to be a small price to pay for the full programmability of LAML. Besides we deemed it the best approach in the discussion of section 5.1.2. Correcting the example from above would read as follows:

```
(html:center
    (string-append "Something very " (html:strong "exciting") " is written here!"))
```

5.1.4 The LAML Language Basis

The inner core of the LAML language consist of a series of functions translated from the HTML element set and a few basic libraries. Most of the LAML language basis presented below have been derived from Kurt Nørmark's initial work on LAML. Especially the translation of HTML to LAML is largely inspired by Kurt's earlier version of LAML, but also the *time* and *input-output* libraries rely heavily on his prior makings.

HTML Element Mirrors

We decided to mirror every HTML element to a Scheme function. This allows developers to freely insert ordinary elements every in the source, contrary to only using ad-hoc produced functions. Having functions for every element also delivers a fine platform for performing attribute and attribute type checking, as this can take place in the compilation process of a given element.

To ensure flexibility of our language we started by building a routine that maps listings of HTML elements to Scheme functions. Different types of HTML elements exist as stated in section 3.1, namely elements that only contain a single tag, elements with an optional end-tag and elements, which requires both a start-tag and a end-tag. We chose to order the different types in two lists with the first containing single tag elements and the second containing double tag elements (the tags with optional end-tag was placed in the list for double tagged elements). Should new elements be invented or should old ones vanish, we only have to update and recompile these lists. Experiments have been made to generate the mirrored functions automatically by parsing the specification (Document Type Definition) of a SGML compliant language (refer to Kurt Nørmarks online LAML pages on *htttp://www.cs.auc.dk/~normark/laml* for details and further developments).

We decided to add the front 'html:' to all elements in order to avoid any namespace problems. In fact Scheme and HTML already share the *map* function/element. When we later make Scheme functions to give CSS support for all HTML elements, we greatly appreciate the avoided collision of namespaces. Eventhough function names may become bigger we believe abstracted elements will ensure minimum use of these low level functions, besides is the readability of the LAML source code enhanced. The problem of colliding namespaces is known from practically all programming languages, where reserved words are not to be used to refer to e.g. classes, methods or variable names. Especially when working with macro languages, where implicit expansion is normal, are namespace difficulties common. Here conflicts require extensive use of escape characters in order to prevent inadvertent macro expansions.

We now have a corresponding Scheme function for every HTML element, which itself represent a function (either *generate-html-tags-single* or *generate-html-tags-double*) called with the element's name as the only parameter, as can be seen below:

```
(define html:img (generate-html-tags-single "img"))
(define html:font (generate-html-tags-double "font"))
```

The *generate-html-tag-functions* yet again return a new function, which is capable of taking an additional parameter list. The function for single tagged elements takes the parameter *attributes*, which contains the optional attribute list, see below:

```
(define (generate-html-tags-single tag-name)
  (lambda attributes
    (make-tag-single tag-name attributes)))
```

The function for double tagged elements is naturally a bit more complex. *Contribution1* and *contribution2* are manipulated in order to separately obtain the content alongside the attribute list. The *make-tag-double* function is then called with tag-name, content and the optional attribute. Remember that the seldom occuring double tags without content should provide the empty string as last parameter, or else will the last item in the attribute list be taken as the content.

```
(define (generate-html-tags-double tag-name)
  (lambda (contribution1 . contribution2)
    (let ((joint-list (reverse (cons contribution1 contribution2))))
      (make-tag-double tag-name (car joint-list) (reverse (cdr joint-list))))))
```

The make-tag-double is listed below (make-tag-single is similar). First is the attribute list checked for elements, if none is present then will the content, encapsulated by start- and end-tags with the given tag name, be returned. With an attribute list present the function will call the function *attribute-handler*, which recursively traverses the list and builds a HTML attribute chain (given that the attributes verify as legal!). HTML tags are later build with the attributes occuring in the start-tag.

```
(define (make-tag-double name contents attributes)
  (if (null? attributes)
      (string-append "<" (as-string name) ">"
                     (as-string contents)
                     "</" name ">")
      (let ((html-attributes (attribute-handler attributes name)))
           (string-append "<" (as-string name) " " html-attributes ">"
                          (as-string contents)
                          "</" name ">"
                          (as-string #\newline)))))
```

Some simple means for pretty printing of the HTML destination files are obtained with newlines inserted after each element carrying attributes (experiments suggested this gave a nice spread of text). Omitting these newlines would make the HTML source consist of one single line, which is hardly very readable with for instance the *'view->Page Source'* command in the Netscape browser.

LAML Basic Libraries

Three libraries provide LAML with some basic capabilities (all three may be viewed or downloaded from [POWER, 99] and be found in the *CH-source/lib* directory of the software CD present with this report). First we consider the *time* library, which provides LAML with functions to determine date, time and weekday. This library is mostly used for automatically generating time of updates, but may also be used to control time dependent content. The *time* library and most of the *input-output* library are largely derived from Kurt Nørmarks initial work. The *input-output* library contain functionality necessary for writing to and reading from files. Writes are always needed when HTML target files are being created. Read operations may for instance be used to import foreign material from external files. Several string operations have also been added to the *input-output* library in order to supplement *Schemes* string handling capabilities. The last of the basic libraries is the *laml-lib*, which contains a wide range of useful functionality. *Scheme's* set of list operations are enhanced with a handful and a few ad-hoc functions, e.g. a hyperlink creator, are included. Common page content, such as the head, title, body and style sheet elements or linkage, can be applied by a set of similar functions, which builds the header and footer that encapsulates the content meant to reside within the HTML *body* element. Different types and layouts for update notes are available through another series of functions. Finally a bunch of table constructs are provided (example pages using these table constructs may also be found both at [POWER, 99] and on the software CD).

5.1.5 Obtaining a higher level of functionality

Given the basic LAML constituents we are able to produce any HTML system. Assistance in the authoring process is however not at a premium at present, when all we possess is a mapping of HTML elements and a few essential helping functions. The real powers of LAML only surface, when some higher degree of abstraction is achieved. Providing a collection of coherent functions for a specific application type allow developers to shift focus from low level elements to higher level components of a given application. The related set of functions, which encapsulate and hide underlying details, will be referred to as *document styles*.

Intervening with the abstractions obtained in document styles provide great means of re-usability, as the common functionality may be used exhaustively. They also present an excellent platform for automating routine tasks, which is as stated earlier highly desirable in the authoring process. Document styles typically end up consisting of both abstracted elements of the given document class and programmatic content manipulating the abstractions or calculating additonal content. To illustrate the use of document styles the forthcoming section presents different examples of developed Web applications.

5.2 Example LAML Applications

Source codes and resulting Web pages of all of the below presented examples of implemented LAML document styles may be found online [POWER, 99] or on the CD accompaning this report. We strongly recommend the reader to take a closer look at these examples and

especially the LAML source programs generating them. The discussion of the programming oriented method in the next chapter will duly evaluate the most important advantages of these sample LAML applications.

Inclusion of the *CD_DB* and *chess-lib* examples in the document styles definiton could maybe seem a bit confusing, as *CD_DB* reads and processes an external text file describing a set of CDs and *chess-lib* provides only addtional library functions for constructing and processing advanced chess tournament elements. So the implemented LAML applications span both what we understand to be document styles and other application specific functionality.

5.2.1 Simple Style

The *simple* document style is, alas the name, the most simple style developed. Primary content of the *simple* style are two similar functions for building a single HTML page. Evidence of the similarities of the two functions are presented below in the depicted function heads. Only the additonal parameter *update* in the second function differs the two. The *update* parameter may be used to specifically identify a desired form of update notation (currently possible options confine to pure text, text and LAML image, text and LAML animation or no update), e.g. would a page generated with the *simple-page-update* function and with the parameter *update* set to *anim* include an update notation with both text and animation. The *simple-page* function offers no update notation.

```
(define (simple-page title fname html-body . css-body) ...)

(define (simple-page-update title fname update html-body . css-body) ...)
```

Common for both functions is the generation of a HTML page with filename *fname*, title *title* and consisting of the *html-body*. The optional *css-body* may consist of both ordinary css-statements and elements linking to external stylesheets. Because linking elements are obliged to appear prior to ordinary css-statements, functionality for processing the css-body is provided. In itself this style may seem rather low key and it does not confront the "page-at-time" approach, which we so eagerly try to render superfluous. But authors having a consistent use of authoring tools would probably develop single pages using LAML too. Besides is the *simple* style likely to be utilized by more advanced styles in their need for actually generating different underlying pages.

5.2.2 Website Style

The *Website* style allows the developer to create a series of related pages with a coherent layout scheme. Within the bounds of this style we regard a Website to consist of pages with identical layout and a common navigation part, which by means of hyperlinking refer to all or a subset of the involved pages. Referential integrity, which is a big issue in Web engineering (as suggested in [Lennon, 96]), will always be preserved in the navigation bar linkage by consequence of the automatic generation. The functions in the *Website* style controlling the generation of navigation links recognizes and handles both imagery and textual links.

By setting an internal style parameter the *Website* style may deliver pages with different structures. Currently four different schemes for page building are provided, where the first one represents a traditional frame based approach. The navigation content will reside in a small frame on the left of the screen in this case and the different pages of a given site will be displayed in the adjacent frame. This classic frame approach have been widely used on the Web, but opinions has ever since the creation of frames been varied (frames have been heavily discussed, but some of their main disadvantages is their relative complexity and the inability to link to subpages of a site). To accomodate the people dissatisfied with frames, we offer methods of site generation based on tables. Common for all table based sites and their biggest disadvantage is the need for the navigation part to be incorporated on every sub page. This of course increases both page size and server workload, because the navigation bar has to be loaded again with every page retrievel. This is contrary to the frame based approach, where the navigation bar is always present in its own frame, with only the underlying pages needing to be retrieved from the server. With sensible built navigation bars the extra overhead should not amount to much, so the table based approach may still prove to be an excellent solution.

The table based approaches currently amount to three, where the first resembles the classical frame look, with the navigation bar on the left and page content filling the right side of the screen. The others are a little more complex with multiple tables, where one is for the navigation bar, one is for the actual page content and the rest are used for layout purposes (e.g. different colouring, indention or images). Sample pages of all styles have been developed and they may be found both online at [POWER, 99] and on this reports accompaning CD (in the *CH-source/website/* directory). Figure 5.1 below sketches the appearance of different page systems generated with the existing layout schemes of the website style.

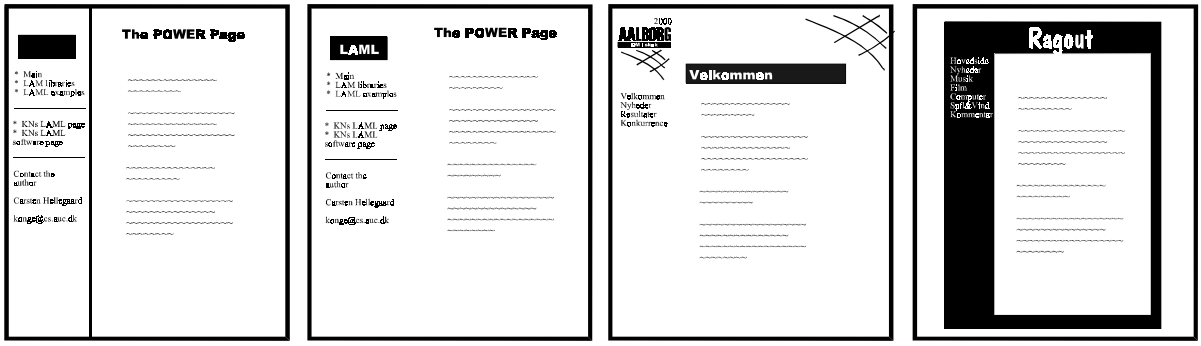

Figure 5.1: *Outlines of the four different types of Website styles developed until now. Power illustrates the traditional frame based approach incorporated in the initial version of the POWER page. Later a table based equivalent of power was conceived, namely power_noframe, which constitutes one table for navigation purposes and one for projection of the actual page content. More advanced layout mechanisms were employed in the cases of SkakDM and Ragout, where a page consists of several tables. The style used for constructing SkakDM places images, navigation bar and content in different tables for visual effects. The same applies to the fourth website style employed in the Ragout example, but this style also uses colouring (the area specific colours may easily be altered) of the different tables to obtain a paper-like look. Besides does the Ragout example utilize the option to use image buttons instead of text in the navigation bar.*

Having several available schemes for building the Web pages enables site content to be formatted to multiple targets. A single source file may compile to different looking Web sites only by altering the desired style. In our case, with only four somewhat different styles available, some content may not suit every style, but our examples showed, that shifting between the similar frame based approach and the 2 columns table approach worked very satisfactory.

5.2.3 Sport Results Style

Different kinds of sports tournaments may get online versions of result related information, such as league tables and team performance charts, generated automatically with the *sport-results* style. LAML sources using the *sport_results* style merely need to contain initializing information of compiling directories (where to locate target files and where to locate image and/or css files) and different league data (e.g. a list of participating teams, number of rounds played and which type of sport we deal with) and the results of the rounds played so far. On account of this a page is build for every participating team containing the teams performance chart (a graph picturing league positions for every played round) along with a result list of the team in question. Continuous pages containing the league tables of every round and results of the current round are also generated. Because the rounds played have a sequential nature navigation mechanisms for traversing these pages are included. Dependent on the round of the current league table arrows allow navigation to the first, previous, next and last table, e.g. the page of the last round contains no arrows for next and last page. Besides is linkage provided from team names in the league tables to the teams individual pages and back to league tables of different rounds from the result list present on the team pages. Finally an index page is built, where access to the pages of every team and every round's league table are provided. For an illustration of the properties of the *sport_results* style just mentioned we provide the sketch presented below in figure 5.2. Two actual Web systems have been developed using the sport_results style and they may duly be found online at [POWER, 99] or on the software CD in the directory *CH-source/sport_results/*.

As in the case of the *Website* style is referential integrity preserved of these automatically generated internal links. Updates to the system are handled very efficient, as new results are just entered in the source and after a recompilation new pages are generated and changes will automatically propagate around the site and alter the existing pages.

Currently the *sport-results* style supports three different kinds of sport, namely football, handball and chess. Several variables (such as points for won game and number of relegations) are set according to the type of sport, whose results need processing. Another team sorting algorithm is also assigned for chess league tables, as points scored takes precedence on match points contrary to the scheme being employed with football and handball. Later versions of *sport-results* may be extended to include goalscorers (point scorers in chess!), yellow and red cards (not used in the chess world, yet!) and match grades for each player. This additonal information should be used to automatically generate topscoring lists, card lists (with flags indicating match suspensions) and player statistics on grades and other.

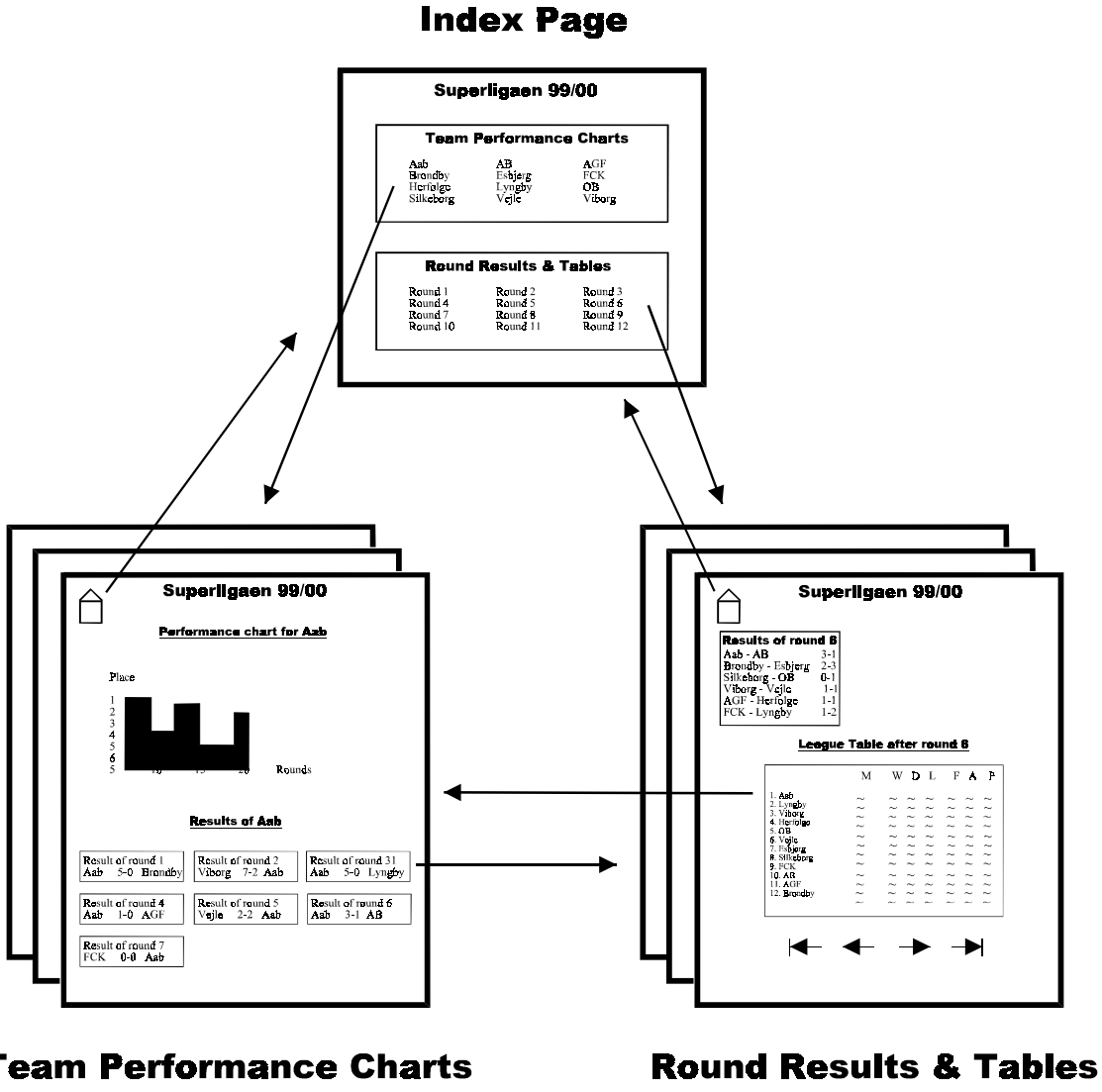

Figure 5.2: *The different constituents of Web systems generated with the sport_results style and their mutual relationships. The several pages constructed by this style amounts to an index page, a page for every participating team and a page for each round of played action. Extensive linking exists between these pages, as depicted on the figure by arrows.*

5.2.4 CD Web Base

We mentioned earlier that the *CD_DB* utility is not a regular document style. Unlike the first three examples LAML sources are not intervening with application specific functionality, instead *CD_DB* processes an external text consisting of a CD set listing. To conform with our implementation CD listings must at least be of the following minimal form, consisting of artist, title, genre, year of publication and personal rating (a scale from 0 to 10 is used) :

```
("Slayer" "Reign in Blood" "Speed Metal" 1986 10)
```

Additional details may be listed randomly afterwards, as in the case presented below.

```
("Rammstein" "Live aus Berlin" "Techno Metal" 1999 8 "x2" "Live")
```

A simple boolean variable controls whether details are wanted or not, if not they are just ignored. Besides determines a detail-list which detail categories to process and in which order. And only details fitting this list will be processed, while others are again ignored.

From the CD listing three HTML pages are generated. The first one contains a listing of all CDs sorted under the beginning letter of the artist name. Only used letters are being used, meaning if no artist begins with a K the letter K will not not be included. For every used letter a table is formed consisting of the corresponding CDs, which are primarily sorted alphabetically on artist names and secondary on publication year. An index, built of the used letters, is placed on the top of the page. An analogous page is made, where CD's are located according to their respective music genre. In accordance with the approach on the page of the alphabetical sorting the used genres are collected and formatted to represent a top page index. Cross references on genres and artist names exist between these two pages, and linkage is provided to the third page generated, which contains statistics on the processed CD list. The aritmethic powers of the programming language enables us to calcute these statistics, which currently consist of a total count of CDs (maybe not the most demanding calculation), percentage of included genres and average ratings for the albums of a particular artist.

Building this system with an ordinary authoring tool might be possible with a little patience, but performing updates would be a nightmare. For every new CD added to the list, its data will have to be included on both the alphabetical and genre sorted pages and maybe even a new artist starting letter or a new genre is encountered, which would require an alteration of the index. These updates are however nothing compared to the ones needed on the statistics page, as every probability would have to be re-calculated according to the changed CD set and re-entered. Such a manual update would take ages and be highly error-prone, whereas an CD addition in the CD list and a recompilation with the LAML *CD_DB* utility would generate a correct Web system in seconds.

5.2.5 Chess Library

The *chess-lib* isn't quite a document style either, it merely extends LAML with advanced functionality for producing online chess tournament material. Acknowledged table plans are encoded in the library for round robin tournaments of 8 or 10 players (the most common tournament schemes). Support for monrad tournaments, where future opponents are found based on prior performances, is also included. How to build a tournament table using the *chess-lib* is shown below:

```
(tournament_rr10
        "Aars IM-group 1999"
        (list
          (list "John Arni Nilssen" "FAI" 2326 0 1 .5 1 1 1 0 .5 0)
          (list "Jimmy Andersen" "DEN" 2000 0 0 .5 0 .5 .5 0 0 .5)
          (list "IM Alexei Bezgodov" "BLR" 2576 1 0 .5 1 1 1 1 .5 1)
```

```
(list "IM Vladimir Poley" "BLR" 2377 0 .5 .5 0 .5 0 .5 1 1)
(list "Jørgen Juul Kristensen" "DEN" 2310 0 .5 0 0 0 .5 0 0 .5)
(list "David Bekker Jensen" "DEN" 2264 1 1 .5 0 0 0 1 .5 0)
(list "FM Erik Hedmann" "SWE" 2348 1 .5 .5 .5 .5 0 1 1 0)
(list "Carsten Hellegaard" "DEN" 2267 0 .5 1 1 .5 .5 0 .5 .5)
(list "IM Igor Yagupov" "RUS" 2449 1 1 .5 1 1 1 1 1 1)
(list "Bent Christensen" "DEN" 2113 1 0 .5 .5 0 .5 .5 0 .5)))
```

This function builds a tournament table with assigned player number, points, placement and in the future it will be extended with routines for calculating new player ratings based on the tournament results. Unlike the *sports_results* style where every round of results will have to be entered together, unplayed matches represents no problems with *the chess-lib* (placing the escape notation ("-") instead will suffice). To add clarity to tournament tables formatting schemes allow different style sheet settings for table cells of delivered match results as opposed to cells for undetermined match result (to see for yourself consult the examples on [POWER, 99] or on the software CD attached to this report). The style sheet handling of the *chess_lib* will be discussed in greater detail in the section on CSS experiments. Entering results for monrad tournaments (characterized by the matching of opponents according to prior results) requires additional data, as is sketched below:

```
(tournament_monrad
        "Aars EMT Group1 1999" 5
        (list
          (list "Hardy Jespersen" "Aars" 1858 'B2 .5 'W5 0 'B4 1 'W3 0 'W8 0)
          (list "Irina Tetenkina" "Minsk" 2141 'W1 .5 'B3 0 'B8 1 'W6 1 'W5 0)
          (list "Rene Rasmussen" "ASF" 1889 'B4 .5 'W2 1 'B5 0 'B1 1 'W7 0)
         ... ))
```

First of all we need to deliver the number of rounds in the tournament, which equals 5 in this case. Besides must every result have a literal prior to it stating color and opponent for the next match. In the example above player number one, Hardy Jespersen, must play black against opponent number 2 in the first round.

To supplement the tournament table production the *chess_lib* supports generation of a common leaderboard consisting of leaders from every implicated tournament. Customizability of the leaderboard includes layout settings and the number of leaders from every tournament wanted. To examplify the use of *chess_lib* we first implemented a single page containing both four tournament tables and an extensive leaderboard, and later we generated separate pages for every tournament and an index page with the accompaning leaderboard linking to the respective tournament pages (may also be found at [POWER, 99] and on the CD in *CH-source/chess/*).

5.3 Support for Cascading Style Sheets

As we established in the analysis is it highly desirable to keep document structure and presentational data stored separately. Following the mix-up of content and layout in the early versions of HTML, this has later been acknowledged by the World Wide Web Consortium,

who are responsible for the Web relevant language specifications. The currently recommended specification of HTML (version 4.0) has been incorporated with support for style sheets. It is possible to use almost any style sheet language together with HTML, but the W3C have themselves delivered specifications for one: Cascading Style Sheets [CSS, 99].

It is our intension to include means of CSS in LAML in order to embrace the full powers of the current HTML recommendation. It is an attractive idea to hold expression powers of both Web languages (HTML and CSS) in LAML. Apart from given full control over both languages in their own right, e.g. when making external CSS style sheets in LAML to be imported in HTML-pages also written in LAML, the unification of both languages in LAML could possibly offer further advantages when used in a combined manner. Several ways of using these unified powers in LAML will be presented through experiments and evaluated in section 5.4.

Because of the newly invented opportunities for inclusion of style sheets in HTML, several old HTML tags and some elements attributes, solely used for layout purposes, are made obsolete. Others, whose functions can now be done more methodical with the use of style sheets, are stamped deprecated by the W3C. Elements in this last category are still part of the HTML-language, but developers are urged to stop using them and turn to style sheets instead. Future HTML specifications might render elements currently stamped as deprecated for obsolete. Together with support for the style sheet language CSS, LAML still contain mappings of the deprecated HTML-tags. This inclusion is made to accomodate developers, who are used to write Web-pages without the use of style sheets and cannot part with some older and well known tags or attributes.

5.3.1 CSS components

The language CSS is quite unlike HTML, which consists of a fixed set of elements with very similar behaviour (only difference is the distinction between single and double tagged elements). CSS on the contrary consists of several different components, which behave differently. Below we sketch the different components of CSS and the corresponding implementation of them in LAML.

Basic selectors

The simplest selector in CSS is a HTML element followed by a list of properties and their desired values:

```
h1 { font-style: italic; font-size: 18pt }
h2 { font-style: normal; font-size: 16pt; color: red }
```

The CSS specification [CSS, 99] formally defines 5 categories of properties, which constitute font, color/background color, text, box and classification. Each category consist of several related properties, each with an accompaning set of possible values. An example is the *text-align* property of the text category, which may be set to either of the following values: left, right, center and justify.

The LAML implementation of the basic selectors constitute a mapping of every HTML tag to a corresponding CSS element. The above listed CSS-statements would read as follow in their LAML counterpart:

```
(css:h1 ’font-style ’italic 'font-size '18pt)
(css:h2 ’font-style ’normal 'font-size '16pt ’color ’red)
```

The procedure is analogous of the mapping of HTML-elements to LAML, and furthermore are the properties checked for validity in a similar manner as the attribute checking in pure LAML.

To accomodate often used style properties we decided to include capabilities for constructing and using predefined lists of properties. The CSS statements of LAML are able to take lists mixed with ordinary property/value pairs of properties (naturally single properties must be ordered correctly in the statement with the accompaning value directly following the property), as sketched:

```
(define font-properties (list 'font-family 'times 'font-style 'normal 'color 'black))
(define text-properties (list ’text-decoration ’underline ’text-align ’center))

(css:h1 font-properties ’font-weight ’bolder text-properties)
(css:h2 ’text-align ’center font-properties)
```

This method of handling element properties is of course available to all the different kinds of CSS selectors.

We thought of using a similar scheme with the attribute lists in the LAML statements, but we didn’t find it necessary as very few HTML elements would be having the same attributes. The more frequent use of higher level abstractions in LAML also limits this need considerably.

Class selectors

The different HTML elements can have several classes, thus allowing each element to use many different styles. Classes can both be defined to only validate for a specific element or they may be defined as a generic class available for all elements, as is sketched below:

```
p.header { color: blue }
p.special { color: black }
.note { font-size: small; color: purple }
```

This example offers two classes specific for the paragraph element (header and special) and one for all elements (note), including p. The class styles are later applied in HTML using the CLASS-attribute:

```
<p class=”header”>Mating life of the Rhinoceros</p>
<p class=”note”>Remember to buy beers for the weekend</p>
```

When representing class selectors in LAML we make the distinguishment between associated and unassociated classes in the first parameter. Given the first parameter equals the name of a HTML element we expect the next parameter to hold name of the desired class for the HTML element. On condition the first parameter does not coorespond to a HTML element a generic class is created. The classes shown before would read as follows in their LAML counterpart:

```
(class-selector 'p 'header 'color 'blue)
(class-selector 'p 'special 'color 'black)
(class-selector 'note 'font-size 'small 'color 'purple)
```

ID selectors

The ID selectors works very similar to unassociated classes, as we can define properties for a given ID and apply them to HTML elements with the ID attribute. The CSS definition looks like this (notice the #-character):

```
#owen10 { font-weight: bolder }
```

This is simply accomplished in LAML as follows:

```
(id-selector 'owen10 'font-weight 'bolder)
```

Contextual selectors

Contextual selectors offer a way to assign style to nested elements by listing elements in context separated by white spaces and followed by the desired properties and values. The cascading rules of CSS then gives contextual selectors precedence over simple selectors. To enlight this we provide the example below:

```
p em { background: white; color: blue }
```

This CSS statement means that emphasized text within a paragraph should be written in a blue color on a white background. Emphasized text within other block-elements than *p* are not affected. And due to the cascading nature of CSS will styles selected just for *p* and *em* be ignored, when they appear in the given context.

The LAML construct for contextual selectors requires a list of HTML elements as its first argument to perform the equivalent of the CSS statement:

```
(contextual-selector (list 'p 'em) 'background 'white 'color 'blue)
```

Grouping

The grouping function is included to avoid repetitious statements in the style sheets. If several elements are to be assigned identical style declarations, the time saving grouping looks as follows:

```
h1, h2, p { font-family: times; color: black }
```

The LAML equivalent is implemented similarly to the LAML contextual-selector construct:

```
(grouping (list 'h1 'h2 'p) 'font-family 'times 'color 'black)
```

Pseudo Classes

Three pseudo classes are present in the CSS language to give a wider range of control over anchors. The three pesudo classes are listed below, together with confirmation that they can can be used in contextual selectors and in combination with normal classes:

```
a:link { color: red }
a:visited {color: blue }
a:active { color: lime }
a:link img { border: solid red }      -contextual
a.myclass:link { color: brown }      -in combination with myclass
```

Pesudo classes do not behave completely like ordinary classes, meaning that user agents must handle anchor presentation automatically opposed to the manually inserted class attributes. The anchor element of class *link* in the example below will have no style effect, because *link* is no real class (the anchor then follows prior style settings for anchor elements).

```
a:link { color: red }
<a class="link">some anchor</a>
```

We have included these anchor pseudo classes in LAML, by mapping them as were they ordinary elements (like *br* or *table*). The only possibility of combining pseudo classes with normal classes in LAML is therefore to invent the classes like *.myclass:link*, but as we believe these constructs will seldom appear in practical use, we find this approach satisfactory.

Pseudo Elements

CSS also includes two pseudo elements, which can be applied to all block elements, namely first-line and first-letter. Examples are given below:

```
p:first-letter { font-size 150%; color: green }
p:first-line { color: black }
```

Because we attach little relevance to the pseudo elements and because only a limited set of properties apply to them, pseudo elements have not been implementet in LAML. Assigning special style to the first letter or the first line may besides be attained by the use of inline styles (see next subsection) or classes.

Inlining Style

Style sheets can also be applied inline using the *style* attribute, which applies to all HTML elements. An example is given below:

```
<p style=”color: red; font-size: 125%”>A sample use of inline styles </p>
```

The HTML statement presented above can easily be constructed in pure LAML, given the developer knows the syntax of CSS property lists:

```
(html:p ’style ”color: red; font-size: 125%” ”A sample use of inline styles”)
```

A more sensible solution is provided, as we find it desirable to relieve the developer of the task of producing the potentially many style properties in pure string form. The original intent was also to let LAML cover both HTML and CSS, so the developer would not be restricted by not knowing the CSS language. Since routines to handle style properties was already developed in the *CSS2LAML* library for controlling the properties of ordinary CSS statements, the following function offers a simple method for inline style property handling:

```
(define (style-inline-handler . properties)
   (let ((inline-properties (property-handler properties "")))
        (prettify inline-properties)))
```

Purely aesthetic reasons is behind the decision to prettify the output of the *property-handler* function (in fact only the last character (an empty space) of the returned string is removed). The ideas behind the *property-handler* function was presented in the section above covering the basic selectors. This new approach to inline styles in LAML is examplified below, where a paragraph with content "Pollefar" will be displayed according to both the predefined font-properties and the left alignment:

```
(define font-properties (list ’color ’white ’font-family ’courier))
...
(html:p 'style (style-inline-handler font-properties 'text-align 'left) "Pollefar"))
```

Analogous with the mirroring of the HTML elements we now possess a LAML translation of CSS (if only of the important parts). But as was also the case with the HTML mirroring not much is gained from a mere translation, the actual benefits will only surface when usage have attained a higher level. Different schemes for utilizing the joint powers of CSS and HTML in LAML will be investigated and discussed in the next chapter.

5.4 Experiments on LAML and CSS

Prior to conducting the actual experiments we investigated the different available methods for incorporating CSS in HTML. Following this investigation we pinpointed two approaches, one based on inline use of CSS and one based on external CSS files, which were to be subjected to further experimentation. Upon evaluation of the experiments we take special notice whether the combination of both HTML and CSS in LAML offers any additional advantages compared to the commonly used separation of the two languages.

5.4.1 CSS Containment in HTML

For style sheets to take effect on the presentation, awareness of their presence is required. To accomodate this need the HTML specification was extended to include ways of linking to external style sheets and methods for inserting CSS statements in HTML. The HTML code listed below shows how style sheets may be inserted in HTML.

```
<HTML>
  <HEAD>
    <TITLE>Providing style sheets in HTML</TITLE>
1.  <LINK rel="stylesheet" type="text/css" href="black.css">
2.  <STYLE type="text/css">
        H1 { font-size: 20pt; color: green; }
        P { font-style: italic; color: lightblue; }
    </STYLE>
  </HEAD>
  <BODY>
    <H1>Green Heading</H1>
    <P>Lightblue writing in italics</P>
3.  <P style="font-style: normal; color: blue">Blue writing, no italics</P>
  </BODY>
</HTML>
```

The three possible ways of combining styles and HTML may be identified in the code above (indicated by the numbers 1 to 3). CSS statements are allowed to occur either within the *STYLE* element located in the *HEAD* or *inlined* within the *style* attribute of ordinary HTML elements. Loading external CSS files are done using the LINK element (notice that HTML may easily import other types of style sheets than CSS, given that user agents supports additional style sheet languages). The HTML specification actually also allow external style sheets to be referred to by the *@import* command, which resides inside the STYLE element before any additonal CSS statements. But because the *@import* command works very similar to the LINK element and the major browsers, Netscape Navigator and MS Internet Explorer, do not yet support this command, we decided to concentrate on the remaining three possiblities for delivering CSS in HTML.

An important aspect of CSS is the cascading (hence the name) nature of the styles. The cascading refer to the order of precedence of applied styles, as it cascades from the outher most level (external style sheets), which have lowest precedence, across CSS statements located in the STYLE element with a higher precedence, to the inner most level of inline styles, which takes highest precedens. Looking back to the sample HTML code above a consequence experienced on behalf of the cascading, would be the CSS statement of the HTML element *H1* overriding a possible CSS statement on the same element located in the external *black.css* file. In the same vain is the inline use of styles on the second paragraph due to override the style settings put on the element *P* in the *STYLE* element. Special attention is however necessary regarding the overriding of style settings, because they function on property level and not on statement level. Meaning that a CSS statement does not override an entire statement of lower precedence, but only the properties explicitly given in the overriding statement. To illustrate we yet again return to the HTML code presented above, where an

additional property defined on element *P* in the style section, e.g. setting *text-decoration* to underline, would also render the *P* element using inline style underlined, unless the *text-decoration* property is explicitly set to another value in the inline statement.

5.4.2 Inline and External Use of CSS

Upon experimentation we divided the methods of providing CSS in HTML into the two categories: inline and external. This distinction was made because of the fact that external style sheets and CSS statements listed in the top element *STYLE* are essentially the same, as both mechanisms uphold layout information external to the affected elements contrary to the inline use of CSS.

Supplying HTML elements with styles using the inline approach contradicts the reasons put forward in the first place for employing style sheets, i.e. the divided keeping of content and layout descriptions. Inclusion of inline style usage in our experiments is however justified by the programmic evasion of the primary concern for mixing content and layout. Recall that alterations in documents consisting of such mixtures was time consuming and error prone, while only a simple intervention with the separately kept formatting information would otherwise be necessary. Equipped with the programming powers of LAML this problem may be solved by defining a layout specification at some point in the source and then let a programmed solution propagate the layout information to the relevant elements in the source.

Experiments

In order to get a concrete foundation for comparison we decided to implement two versions of the chess library (see section 5.2.5), with one using inline styles and the second being formatted according to an external style sheet. Different applications were then generated with each copy of the *chess_library* for easy comparison, some of which may be found on the software CD in the directory *CH-source/CSS_testing*.

The tournament tables generated by the chess library is like most complex HTML elements made up of tables, which in this case prompted different formatting for table caption, table header cells, ordinary table cells and an alternative table cell, used when different formatting of cells containing confirmed results is desired. Applying the formatting by means of CSS would ensure a homogeneous look of a tournament table (same size and colouring) regardless of the surrounding environment, i.e. the parent page being conformed to another style sheet.

In both versions of the chess library the four formatting lists are defined only one place, namely in the top of the source file. Clearly the inline version caused the most overhead in the source document, when the formatting information was to propagate as parameters through the different table generating functions. In the version based on an external style sheet the four formatting lists are defined as CSS classes and written to an external CSS file. Minimal overhead is experienced between the generating functions, as only the four class names will need to be assigned at their respective generation function. The main disadvantage compared to the inline version is the need to extend existing style sheets of parent pages with the newly defined series of classes. Two sensible solution models exist however. First may LAML automatically update an existing CSS file with the extension set of classes (this approach has

been succesfully applied within the chess library, as the optional leaderboard extend the existing class set with a few more of its own) or second may we exploit the fact that HTML pages are permitted to link to several external style sheet by adding an extra *LINK* element on the parent page pointing to the newly created CSS file.

Experimentation showed very similar behaviour of the two versions. Both had the same ease of altering page formatting, and both suffered from casual use of formatting on general table elements such as *TD* and *TH*, which distorts the cell formatting, unless unreasonable many properties are set for every class or inline element (this is due to the property overriding not happening on statement level). The most important difference between the two versions was the size of the target language files. When using only approximately three properties (e.g. background colour, font colour and font size) for each cell class, we experienced almost double sized target files of the inline version compared to the external style sheet version (this proportion would undoubtful rise with the addition of further properties). Besides did we register a minor difference in the browsers page loading time, with the pages displayed according to external style sheets being processed a little bit faster than their inline counterparts. We don't attach too much significance to this phenomenon, however (the slight difference might perhaps only have been a side-effect of the larger file sizes of inline style documents).

Learning from these first series of experiments we intend to only use inline styles for smaller constructions, where the size of target files are unlikely be significantly affected. Besides is the use of external style sheets not entirely justified, when working with smaller elements, as manipulation of existing style sheets may seem a bit exaggerated if only for including one or two new classes. Furthermore we developed a series of different update constructions varying from pure text to text and animations. The desired uniform look was satisfactory attained using inline styles.

For larger constructions, such as complex tables, we duly settled on using external style sheets for delivering the formatting information. The primary reason being the somewhat smaller HTML files generated, but also because LAML source programming becomes a simpler task due to a lesser amount of formatting information manipulation. Apart from the external style sheet version of the chess library we made extensive use of this approach when constructing the sport results style. Being made almost exclusively by tables this style uses in the region of 20 classes to control different table cell formatting. On a general note we might add that style sheets are being used exhaustively in our Web development, not only for controlling complex structures but also for commonly used HTML elements (such as H1 and anchors A, A:active, A:link and A:visited).

5.4.3 Evaluation

First lesson learned from our experiments was the considerable advantages gained when using external style sheets for large constructions. In these cases inline use of style may seem justified when speaking of keeping format information in one place for easier alteration (delivered by the programming abilities of LAML). But the resulting HTML files are magnified in size compared to the corresponding HTML files made with use of external style sheets. The sole reason behind this is the high level of rendundant formatting information

used in e.g. tables, which in the case of external style sheets only requires a class name contrary to inline styles or the formerly used *FONT* element, where a complete property list must be delivered on every occasion. The LAML applications delivered so far have proven the sense of using inline styles only for smaller constructs, while external style sheets are clearly best suited for construction of larger elements.

Determining whether the merging of HTML and CSS in LAML offers an extra dimension in Web engineering is less clear-cut. There is no question that we are convinced about the usability of style sheets, which our previous developed LAML applications may witness. So having a LAML translation of CSS must always prove a benefit, as we have the CSS components ready at hand in a syntax similar to ordinary LAML. In the bigger picture we especially notice the resemblance between LAML constructs formatted according to a set of CSS classes and XML elements formatted in correspondence with rules written in the style language XSL. Newly constructed XML elements require accompaning formatting rules to be defined in XSL, before content may be displayed. Similarly in LAML we define a set of classes to cover different parts of a larger construction and then we write a style sheet in CSS to control formatting of the involved classes. This goes a long way to show that the abstraction powers of LAML emulate those of XML very satisfactory.

5.5 Support for CGI

Back in section 2.2.3 when discussing generation tools we presented three different possibilities for generating Web pages. *Calculated* pages were defined as pages generated on the server in accordance with data derived from an user interaction performed in a client-side browser. The information travelling back and forth between client and server are exchanged through the *Common Gateway Interface* (CGI). In order for server-side applications to produce a response to user selections they must be able to interpret and process an encoded CGI-stream being passed from the client-side browser. The calculated result obtained upon processing of the input CGI-stream amounts to a HTML page, which is subsequently shipped back to the user's browser for display.

An arbitrary programming language capable of reading a text string from standard input and delivering one back to the standard output may be used to create a HTML page based on the information encoded in the CGI-stream. Typically CGI programming purposes have been handled by compiled programs written in languages such as C and C++ or by scripts written in languages like Perl. Perl is a language similar to C, but without certain high level mechanisms such as pointers and user defined types. Instead is Perl enhanced to better cope with string manipulation, which makes it more applicable for scripting purposes. Noticing that these often used CGI solutions adhere to the imperative way of thinking, we found it very interesting to try and utilize LAML in the area of CGI, too.

As it turns out is LAML excellently suitable for CGI purposes. This should come as no greater surprise, as the essence of CGI programming is the construction of HTML pages according to input received from the users browser. This formula also fits the basic reasoning of functional languages and thereby also the nature of LAML, where an output is calculated based on any given input. Several advantages are apparent when using LAML for CGI purposes: first of all are the general authoring powers of LAML available for the writer of

CGI content, which again substitutes the less suitable imperative means of the traditional approaches, as was the case in ordinary Web development. Secondly will developers most certainly welcome an united authoring language for generating both static and dynamic content on Web pages. Using one language for both issues has its obvious advantages in terms of the knowledge and experience of the developer towards the language. Binding the generated and calculated categories closer together with a common language may also have further benefits, as integrated solutions become easier to develop.

Kurt Nørmark has among other things created a system for online education purposes, which makes extensive use of dynamical content by means of CGI programming in LAML[6]. Staying with the examples described in this report we should point out the *CD_DB* of section 5.2.4, where an integration with dynamic content would be especially useful. Construction of a browser user interface for adding, deleting or editing the entries of a CD base, along with control functions at CD base level, would provide the user of the CD_DB with easy online handling of a set of CD bases. Apart from making large integrated custom applications one could also imagine a host of libraries with functionality for commonly used CGI constructions, such as search forms, guestbooks, email forms and customizable questionnaires.

In order to utilize LAML for CGI purposes we need to address certain problem areas. How do we for instance execute a server located LAML program working as a CGI script? This is simply done the same way as most other scripts are executed on a Unix system, by using the #! notation. The #! notation resides on the first line of the CGI file pencilled in for execution and it indicates the location of the interpreter program, which is supposed to process the file. In the case of LAML we require a Scheme system to be installed on the server and the top of the file notation should be pointing to the location of the Scheme system executable, as depicted below:

```
#!/pack/scm/bin/scm
```

The notation just given informs the system that the rest of the file being read should be processed by the *scm* program located in *pack/scm/bin*. The remainder of the CGI file should now contain a Scheme program, which the primitive program given below is an example of:

```
#!/pack/scm/bin/scm

(define writeln
  (lambda args
    (for-each display args)
    (newline)))

(writeln "Content-type: text/html")
(writeln "")
(writeln "Data have been registered")
(writeln "</body>")
(writeln "</html>")
(exit)
```

[6] Further details may be acquired at http://www.cs.auc.dk/~normark/laml

This sample CGI file returns a HTML page with the writing "Data have been registered" no matter what input it may receive (The phrase "Content-type: text/html" is to start every HTML page returned through CGI and the final *exit* statement halts execution and shuts down the Scheme system after processing). Such a CGI program is of course of limited value and LAML is in fact not even utilized in this example, as only pure Scheme functionality is used. In order to process input from the user, we must be able to extract information from the encoded CGI stream delivered from the browser. Fortunately has Scheme got excellent parsing abilities and decoding libraries may duly be built. Once again Kurt Nørmark is the pioneer in the field and he has built such libraries to extract data from a stream and deliver them as Scheme friendly association-lists[7]. A more advanced, but hypothetical example is given below, where several libraries are loaded in order to deliver CGI-decoding and LAML functionality:

```
#!/pack/scm/bin/scm

(load "lib/cgi.scm")              ; CGI functionality
(load "lib/LAML2HTML.scm") ; HTML mirros
(load "lib/laml-lib.scm")         ; Additional LAML functionality

(write-page
  "CGI-result"
  (con
    (html:h1 "You entered the following:")
    (table-plain 0 (process-list (get-form-list)))))
 (exit)
```

The presented CGI file makes use of the hypothetical function *get-form-list* from the CGI library, which should be able to extract all information from a CGI-stream and present it as an association-list. Running this list through the also non-existent function *process-list* should offer us a list of lists applicable for the *table-plain* function. Eventhough this example may be somewhat thought up, it shows the rich potential of using LAML as a CGI language.

LAML and its possibilities have now been presented and the next chapter will compare these discoveries with the properties of existing Web authoring tools.

[7] These libraries may be found in Kurt Normark's LAML distribution both given on the CD attached to this report or online at http://www.cs.auc.dk/~normark/laml

6. Programming Oriented Web Engineering

In this chapter we intend to prove the idea of Programming Oriented Web EngineeRing (POWER) as legitimate. We hope to obtain this goal by reflecting on the sample LAML applications and experiments performed in the previous chapter. By a comparison of the experiences gathered on LAML in chapter 5 with the characteristics of existing tools we plan to indicate the potential impact of POWER on the Web authoring community. The primary source of comparison from the existing collection of widely used Web authoring tools will be *Microsoft Frontpage*, which seems to be at the forefront of the development at the moment. *Frontpage* is also a WYSIWIG tool and it represent the group labelled as authoring tools in the analysis, which is the most popular type of tool currently in use. It makes good sense to compare our invention with the at present mostly used tool.

The section below starts with a comparison of the most notable of LAMLs charateristics with the corresponding or missing ones of traditional Web tools (in this case mostly Frontpage). This is of course the most important part in the evaluation of LAML and its capacilities, but subsequently we engage in an investigation of the increasing demands put on a LAML Web systems developer. Next a subsection is devoted to a discussion of the different perceptions of a research tool and a commercial product already enjoying widespread use. This chapter is concluded with an estimation of the worth and usability of POWER.

6.1 LAML compared with traditional Web tools

The most important subsection of this comparison of LAML and Frontpage constitutes a comparative discussion of the most notable characteristics of LAML, which are extracted from the sample LAML applications presented in section 5.2 and the conducted experiments on CSS, as described in section 5.4. Next we stress the increased responsibilities and capability needs of a developer converting from a WYSIWIG tool to a program oriented tool like LAML. The common mistake of putting research and scientific work on equal terms with existing commercial products is elaborated in the final subsection.

6.1.1 LAML Characteristics

The single most important feature of LAML is of course the possibility to develop Web systems with a high degree of automatically generated content. This simplifies the development and especially the maintenance of large Web systems, which consists of multiple pages with similar layout schemes applied. Just recall the applications described in section 5.2, where automation playes a key role. The *CD_DB* automatically generates several pages of information based on a CD-set described in an external file, where the actual build of the specific pages is determined by the entries of the CD-set (a top index-list and the actual content tables are formatted according to existing entries, which are divided by the beginning

letter of the artists or the music categories). Besides are statistical data calculated and displayed according to the constituents of the CD-set and the range of CD details for processing may be switched off or modelled according to the developers pleasure. The *sport_results* style manipulates a rich amount of data, e.g. teams, team performances and match results. For every participating team there is generated a page with individual performance chart and result list. And for every round of match play there is constructed a page with the current rounds league table and results. Extensive hyper linking exist between these pages in order to secure the user of easy navigation through the pages. Analogous to the CD_DB are automated routines used extensively in the *sport_results* style and to a smaller extent by the chess library and the *website* style.

These applications rely heavily on automatic tasks, which would prove difficult to implement in Frontpage. The systems mentioned above could of course be built in Frontpage, but manual editing of every page would be necessary. Producing a page skeleton might ease this task, but real complications are certain to arise when maintaining the system becomes relevant. Applications built with *CD_DB* and the *sport_results* style are due to evolve considerably when the input quantity changes, because the changed or additional information propagate to almost every page in the system. With LAML these changes are automatically updated in the proper places, while Frontpage would require the user to manually edit every page affected by the changes. Professional systems of this kind could never be considered developed in a WYSIWIG tool. Instead would server constellations of databases and a dynamic front-end language such as ASP or PHP be preferred. So in LAML we hold the expression powers to both emulate such professional database constellations and create ordinary Web pages.

Web systems consisting of many pages are likely to contain a host of hyperlinks within the site hierarchy. Such system internal links are normally created automatically in LAML and referential integrity will definitely be preserved within the system and only linkage to remote sites should be an issue (the main danger being dangling or incorrect links). All of the developed LAML applications we mentioned in the previous chapter contain lots of site internal linking, which LAML generates automatically and error-free. Managing the overall structure and the internal linking is made simple in Frontpage by means of the explorer part, which is in fact a GUI site level editor. Page relations are automatically deducted from the site manager and links are duly inserted on the pages in question. Eventhough the inserted links follows a standard procedure (making it less flexibel than its LAML counterpart) they do simplify site management and allow fast prototyping. When links to remote pages are inserted on a page they may be placed on a list of remote links for later status checking. This clever feature is also included in Gentler, as described in [Thimbleby, 97], and we have implemented similar functionality in the Website style, where external links may be gathered and displayed in a sample HTML page for enhanced overview and easy status verification.

Users of Frontpage may choose from 6 different prefabricated site skeletons ranging from personal homepages to corporate Web systems. The site structure and layout schemes applied on the pages may of course be altered according to the developers preferences. These few albeit frequently used applications may be extended by user developed site skeletons, which in turn might rival the endless amount of possible LAML applications.

Frontpage offer only moderate support for CSS, as directly typed CSS statements placed within the *head* element are the only possible way to use them. Instead Frontpage uses a

unique and implementation dependent scheme called *themes*. Themes defines a coherent page layout by altering colour and appearence of common page constituents, such as buttons, banners and headings. Using a different theme on a Web site constructed in Frontpage is analogous to applying new CSS classes for certain structures in LAML. The CSS usage in LAML is however superior to the themes employed in Frontpage. Firstly because not only common content like buttons and heading, but also large abstracted structures, such as the tournament tables implemented in the *chess_lib* (as described in section 5.2.5), may be assigned to a set of CSS classes. Secondly because CSS is an open and non-proprietary language unlike the Microsoft dependent themes.

Another issue related to the overall layout alterations just discussed is the possibility with LAML of obtaining multiple views of the same source. We mentioned this property earlier when reporting about the *Website* style, where the POWER pages were generated according to both a frame based and a table based approach. Kurt Nørmark has likewise developed document styles that offer both a Web version and a version more suitable for printing on paper of the same sources files[8]. This multiple target view of page content cannot be carried out in Frontpage (naturally may different page systems holding the same content can be manually created in Frontpage, but the benefits are sparse).

Dynamic content is another issue easily compatible with LAML, but harder to utilize in Frontpage. Frontpage offers a few dynamic gadgets known as active elements. Examples are counters and search forms, but again there is a limited number of elements available and they uphold a proprietary nature. As explained in section 5.5 may LAML be used directly as a CGI-scripting language, which opens for further possibilities of building Web systems with dynamic content.

Eventhough Frontpage doesn't have any real automation functionality, where site structure edition and site level layout themes presents the only system wide manipulation schemes, one would agree to its dominance when dealing with certain kinds of applications. While LAML is clearly best suited for the construction of large and complex Web systems, we believe Frontpage would be the obvious choice for making single pages or lesser Web systems, because of its relative simplicity and fast prototyping. Having obtained a lot of experience with one tool might however persuade some users to develop every kind of application with the preferred tool. An experienced LAML developer would probably not revert to Frontpage just for the case of single pages, but instead rely on the functionality best known by him and vice versa.

6.1.2 User Acquirements

Users developing Web systems with LAML are required to be considerable more skillful than developers using WYSIWIG tools. The most apparent need is programming capabilities and in this case preferably knowledge of functional programming. In order to utilize the full flexibility offered by LAML, users should also have a certain degree of knowledge of the underlying languages, i.e. HTML and CSS. The actual developer of a document style should certainly know HTML, while other people may later make use of this document style without

[8] Further information available on http://www.cs.auc.dk/~normark/laml

knowing all of the underlying functionality. But as a consequence they lose control of the appearance of the generated pages.

Knowing HTML is not a criteria when making Web material with *Frontpage*. Only in advanced cases is it necessary to edit the HTML code produced by *Frontpage*, e.g. for insertion of alien code, such as java-scripts or code snippets generated by external programs. A vital user acquirement in the case of *Frontpage*, is of course knowledge of the tool at hand. But mastering or at least being able to use a program closely related in functionality and appearance with other Microsoft Windows applications is considerable easier for most people than programming and needing to understanding HTML.

6.1.3 It's not a Popularity Contest

A common confusion in the computer society is the (maybe unconscious) distinction between research programs and commercial products, as also discussed in [Thimbleby, 97]. It is hardly fair to directly compare attractive commercial program packages developed by large teams of programmers and given intensive marketing with research programs conducted by single or few scholars. Impressive looking commercial products may only be superior in limited ways and they might have specific problems or lack vital functionality, which could prove interesting areas for closer investigation through research.

So without being blinded by the popularity and impressive appearance of commercial giants like *Frontpage*, we ventured the domain of Web authoring without prejudice. As it turned out we discovered an apparent lack of automation functionality in widely used Web tools. But eventhough we have produced a prototype of a tool, which handles complex Web systems significantly better than ordinary WYSIWIG tools, we hold no false illusions for tools of LAMLs kind gaining any real commercial succes. The substantial knowledge required by a LAML Web developer, as discussed in the previous subsection, is bound to discourage the majority of people. But LAML should be able to obtain a reasonable high popularity in educational environments and at Web minded corporations.

There is no denying the qualities of programming oriented Web authoring, so another possibility of utilizing some of the explored authoring issues could very reasonable be an incorporation of automation functionality in a WYSIWIG tool. A possible unification could utilize LAML like powers to construct site skeleton and page dependencies, while WYSIWIG editing would later be used to fill in the actual page content.

6.2 Conclusion

Programming oriented Web engineering is definetely a method to be reckoned with. The automation processes that are so important for large system development are present at a premium in LAML, which no other known Web tool can rival. The sample LAML applications we have implemented along with the contributions of Kurt Nørmark[9] have clearly shown the potentiel of the programming oriented approach. There is no doubt that

[9] His entire range of LAML applications may be found at http://www.cs.auc.dk/~normark/laml

LAML provides developers with excellent capabilities in areas, where *Frontpage* and other widely used Web tools are less adequate.

The spreading of LAML is however deemed to be hampered by its extensive demands to the user. Only people with a solid background in programming and knowledge of HTML and CSS will be able to harvest the benefits of LAML, which probably restricts usage to educational institutes and Web-minded corporations. Later incorporation with WYSIWIG tools might however open for a broader audience.

But as we stated in the previous section is popularity not the only criteria for success. While looking at the Web applications made possible to implement in a sensible manner in LAML, we acknowledge the legitimacy of such an authoring tool. The next chapter will evaluate the collected work efforts of this report and shortly discuss the future perspectives of the programming oriented method for developing Web systems.

7. Evaluation

Originating from the problems formulated in chapter 3, we engaged in a investigation of three possible programming paradigms for our authoring language in chapter 4. Later in chapter 5 the design, implementation and usage of the Lisp Abstracted Markup Language was presented together with explanation and experimentation on the added support for CSS. Finally we put the whole idea of Programming Oriented Web EngineeRing under scrutiny in chapter 6 by comparing the most apparent characteristics of LAML with those of widely used Web tools.

In this chapter we start by evaluating the truthfulness of the hypothesises formulated before in chapter 3. The succeeding section deals with the future perspectives of POWER and LAML, as we discuss both their possible impact on the Web authoring world and prospects for future enhancement and refinement. In the final section we conclude on the result of the work carried out and presented in this report.

7.1 Hypothesises

We devote this section to a confirmation or otherwise of the four hypothesises formulated in chapter 3. Common for the statements of all four hypothesises is an ambigous nature, which makes any assumption on their correctness quite subjective. Hypothesises 1, 2 and 4 belong together, as they share a direct relation with the efficiency and usability of the developed tool. These three hypothesises should therefore be evaluated according to the impact of the invented tool.

While lacking scientific methods for obtaining clear and unambigous proof of the merits of our tool, we must engage in a process of empirical testing under different circumstances. A thorough empirical evalutation might very well present a long and tiresome task, but we believe, based on our experiences with using and experimenting with the system, that we have the ability to access the correctness of the stated hypothesises within an acceptable probability. This means that the assesments given below of these specific hypothesises are deducted from the content of chapters 5 and 6.

Regarding hypothesis 3, which focuses on the programming paradigm of the authoring language, we find an evaluation even harder to give. In order to obtain a meaningful comparison with other programming paradigms, we would have to experiment with actual implementations in other paradigms. As such implementations are not available to us, we limit our evaluation to only comprehend the conclusions of chapter 4 and the workability of our implemented authoring language.

Hypothesis1:

> *Automated tools are needed in order to properly build and maintain large Web systems.*

If we emphatize the word *large* or extend the meaning of the statement to also include complex Web systems, we regard this hypothesis as confirmed. Some of the examples portrayed in chapter 4 are of a pretty complex nature and in chapter 5, we established the inferiority of prevailing Web tools in this area, as some of our developed LAML applications might not easily be made with other tools. As also concluded in chapter 5 would simpler pages or page systems gain significantly less from our approach, they might in fact be developed quicker and easier with other tools. But concerning large or complex Web systems we have firm proof that some kind of automation is necessary. Summarization of some of the most apparent advantages would be: Future updating and adjusting may be handled considerable faster and less error prone, when not relying on manual editing. The availability of mathemathical content, which may be calculated automatically by the algorithmic capabilities of programming languages. Time dependent content. Dynamic content through CGI-scripting. Extraction of data from external files.

The concepts of abstraction and automation is in fact very central to the whole art of programming, where they present the main advantages in most programmed solution in all spectres of computer science. The conclusions of our own investigations and these general programming principles forms the basis for regarding hypothesis 1 as a truism.

Hypothesis2:

> *Highest level of flexibility is obtained by developing Web page systems with a programming language.*

The complete translation of HTMLs elements ensures that we have the full expressive power of HTML available. And because the functions representing these basis elements may be inserted freely in higher abstracted functions, we obtain a very high level of flexibility. Easy access to both higher level structures and the lowest level functions, together with the problem free intermingling, provides the developer with a vast number of opportunities. Developers are not restricted to merely interact with a wizard or some other sense of programmed notion on top of HTML, which hides the underlying code and thereby limits the developers control of the implemented structures. We established in chapter 6, that LAML is more flexibel than any of the other known Web authoring tool. This prompts us to consider hypothesis 2 as confirmed.

But as was also discussed in chapter 6 is full flexibility not entirely positive for an authoring tool. Typically there is a trade-off between flexibility and usability/simplicity. Some would undoubtedly prefer a WYSIWIG environment with cleverly designed wizards for constructing advanced structures as opposed to needing programming skills and HTML knowledge in order to utilize full flexibility. We went for flexibility, while sacrificing simple usage in the process, and as a result we believe that hypothesis 2 is satisfied.

Hypothesis3:

> *A functional programming language should provide an attractive solution as an internal language for a markup language.*

Determining whether the functional paradigm is actually the best suited for the implementation of an authoring language is made considerable more difficult by the abscence of implementations of the other relevant paradigms, which could otherwise serve as sources for direct comparison. Instead we rely on the conclusions made in chapter 4, where the imperative paradigm was soon dismissed as a serious candidate. Both the object oriented and the functional paradigm seemed to contain functionality, which could reasonable well portray markup languages. But different syntactical properties made the functional languages seem better suited. Our implementation of LAML appears to support this point of view. It is therefore our belief that this hypothesis is confirmed as well (in the end we only asked for an attractive solution!), thus an object oriented implementation might prove a close rival.

Hypothesis4:

> *An integrated authoring language encompassing both HTML and CSS might deliver additional benefits for the developer.*

Again for this hypothesis an evaluation is not too straigthforward. But if we look at CSS for instance, it's legitimacy is well documented, as we clearly stated in the analysis when discussing generic coding (i.e. the separate keeping of content and layout information). See [Coombs, 87] and [CSS, 99] for further conviction. Following this endorsement of CSS, it is only natural to support it in LAML. At least having HTML and CSS in a similar syntax in LAML simplifies the developers task. Furthermore did our experiments with the combination of HTML and CSS yield a positive result. Building larger HTML structures with CSS classes adhering to some external stylesheet proved an excellent choice, while minor constructions were occasionally better handled by using inline styles. In fact did this integration of HTML and CSS resemble the mechanisms employed by XML and it's style language XSL, as we discussed in chapter 5. But whether any given developer experiences better conditions for Web system constructing with this integration, as opposed to just writing external stylesheets on the side (and insert correct inlined CSS statements on their own) is not easily answered without a larger empirical investigation. Therefore we regard this hypothesis as partially confirmed.

The hypothesises have mostly been regarded as confirmed, so the idea of programming oriented Web engineering has proven its worth and we recommend further work in this area. In the following section we discuss the perspectives of continued use and research of POWER.

7.2 Future Perspectives of POWER

The sample LAML applications described in chapter 5 and the reflections on POWER presented in chapter 6 all support the justification of programming oriented Web development. We concluded in chapter 6, that LAML or similar Web tools are unlikely to ever acquire any real commercial success, because of the user's need to have programming abilities. But eventhough our product would never be accepted by the majority of people, it still deserves its place, because ot its apparent advantages. Certain aspects of LAML may however still be enhanced and refined in order to obtain a more mature product. A constant

improvement of our authoring language would of course strengthen the Web developers impression of LAML as a genuine possibility for a Web authoring tool.

Overall versatility would increase with the construction of the basic language elements by means of an automatically parsing of the target languages SGML compliant Document Type Definition. Directly building the LAML basis for a given language with both available elements, attributes and attribute types according to the language definiton in question ensures faster and less error prone translations, as well as easy adaption for future updates. Different versions of LAML may also be constructed in order to accomodate other languages than HTML. A future connection with the new Web language XML could be an interesting possibility prospering from this. Kurt Nørmark have done some experimentation on this approach and have so far automatically created a working version of LAML based on the DTD of a late version of HTML[10].

Another area worthy of more intensive work is the creation of dynamic content. The implementation work carried out simultaneous with writing this report have neglected the area of dynamically created pages. But the possibilities for creating dynamic content through CGI-scripting exist in LAML, as formerly stated in section 5.5. Also concerning dynamic pages have Kurt Nørmark made the initial explorations and he now uses CGI-scripting through LAML extensively, for instance in his distance education system and with his very matured course ware system[11]. The steps required for using LAML as a CGI-script language was previously explained in chapter 5.5, but additionally should LAML be expanded with a CGI-library, which conveniently offers the developer the most used mechanisms for including dynamic content in a Web system. Prefabricated functionality for adding the most frequently used dynamic structures, such as search fields, order forms and mail histories, to your Web site might also be developed for easy adaption to any LAML application.

Future work may also include the building of a huge library set with document styles for many frequently used application types, which might invitate in a few more Web developers. By asking the developer to only interact with the interface of a document styles we obtain an imitation of the generation systems described in section 2.2.3, e.g. the *HTML Course Creator* and *ATML*. Users with less or no programming experience might be able to construct Web systems with predefined document styles, but of course they have lost the flexibility and possibility for major alterations of the end product.

7.3 Conclusion

The four proposed hypothesises were to a certain extent all confirmed. Hypothesis 1 were proven by some of the complex applications portrayed in chapter 5, which according to the conclusions put forward in chapter 6 would present a overwhelming task to implement in WYSIWIGs or other tools. Specific applications may be handled very effective with dynamic languages acting as a front-end for a database, but LAML also cover this category in acceptable fashion along with other application types, where databases offers little help. Very

[10] Readers may follow this work on Kurts LAML page at http://www.cs.auc.dk/~normark/laml
[11] Again consult http://www.cs.auc.dk/~normark/laml for details

predictable was also hypothesis 2 confirmed, as the flexibility level is at its highest on the lowest level of human interaction. We understood there existed a trade-off between flexibility and usability and we went for a tool with a high degree of flexibility on the account of losing Web developers without a certain level of programming knowledge. Hypothesis 3 was also confirmed, eventhough we had no tools of the other relevant paradigms available for a direct comparison.

Relying on the credibility of the developed LAML language and our investigation in chapter 4 we however felt in a position to confirm it as well. The relatively few experiments carried out with the incorporated CSS support allowed us to only partly acknowledge hypothesis 4. It makes sense to extend LAML with CSS, but whether additional benefits is obtained is not so easily answered. But the performed experiments provided some encouraging results of the combined powers of HTML and CSS. With the overall confirmation of the stated hypothesises and with the conclusions of chapter 6 in mind we believe, that there is without a doubt a place for the programming oriented idea of Web authoring and languages like LAML.

Literature

[Bichler, 96] Martin Bichler and Stefan Nusser
'Developing Structured WWW-Sites with W3DT'
Proceedings of AACE WebNet '96, 1996

[Bosak, 97] Jon Bosak
'XML, Java, and the future of the Web'
http://sunsite.unc.edu/pub/sun-info/standards/xml/why/xmlapps.html, 1997

[CSS, 99] World Wide Web Consortium
'Cascading Style Sheets, level 1'
http://www.w3.org/TR/1999/REC-CSS1199990111

[Coombs, 87] James H. Coombs, Allen H. Renear, and Steven J. DeRose
'Markup Systems and the Future of Scholarly text Processing'
Communications of ACM 30, pp. 933-947, 1987

[Curtis, 96] Curtis A. Carver and Clark Ray
'Automating Hypermedia Course Creation and Maintenance'
Proceedings of AACE WebNet '96, 1996

[Drakos, 93] Nikos Drakos
'Text to Hypertext Conversion with LaTeX2HTML'
Baskerville, 3(2), pp. 13-15, 1993

[Drakos, 94] Nikos Drakos
'From Text to Hypertext: A Post-Hoc Rationalisation of LaTeX2HTML'
Computer Networks and ISDN Systems, 27(2), pp. 215-224, 1994

[Hellegaard, 99] Carsten Hellegaard
'WAT - Web Application Tools'
Department of Computer Science, Aalborg University, 1999

[Heinemann, 98] Charles Heinemann
'Let's Go to the Tape: Q&A with a Microsoft XML Guru'
http://www.microsoft.com/xml/articles/xml051198.asp, 1998

[HTML, 98] World Wide Web Consortium
'HTML 4.0 Specification'
http://www.w3.org/TR/1998/REC-html40-19980424

[Johnson, 96] W. Lewis Johnson, Tyler Blake and Erin Shaw
'Automated Management and Delivery of Distance Courseware'
Proceedings of AACE WebNet '96, 1996

[Kesseler, 95] Marcus Kesseler
'A Schema Based Approach to HTML Authoring'
Proceedings of the 4th International World Wide Web Conference, 1995

[Lennon, 96] Jennifer Lennon and Herman Maurer
'Aspects of Large World Wide Web Systems'
Proceedings of AACE WebNet '96, 1996

[Mathiassen, 93] Lars Mathiassen, Andreas Munk-Madsen, Peter Axel Nielsen and Jan Stage
'Objektorienteret Analyse'
Forlaget Marko Aps, 1993

[MS, 98] Microsoft Corporation
'XML: Enabling Next-Generation Web Applications'
http://www.microsoft.com/xml/articles/xmlwp2.asp, 1998

[Nørmark, 99] Kurt Nørmark
'Programming World Wide Web Pages in Scheme'
http://www.cs.auc.dk/~normark/laml/papers/programming_www_scheme.ps, 1999

[Nørmark, 99a] Kurt Nørmark
'Using Lisp as a Markup Language The LAML Approach'
http://www.cs.auc.dk/~normark/laml/papers/lugm_laml.ps, 1999

[Owen, 97] Charles B. Owen, Fillia Makedon, Glen Frank and Micael Kenyon
'ASML: Automatic Site Markup Language 1.03'
Technical Report, Dartmouth College, Computer Science, Number TR97-308, 1997

[POWER, 99] Carsten Hellegaard
'The POWER page'
htttp://www.cs.auc.dk/~konge/POWER

[Rosenberg, 98] Jim Rosenberg
'Locus Looks at the Turing Play: Hypertextuality vs. Full Programmability'
Proceedings of the ninth ACM conference on Hypertext and Hypermedia: links, objects, time and structure in hypermedia systems, pp. 152-160, 1998

[Springer, 89] George Springer and Daniel P.Friedman
'Scheme and the Art of Programming'
The MIT Press, 1989

[Thimbleby, 97] Harold Thimbleby
'Gentler: a tool for systematic web authoring'
International Journal of Human-Computer Studies, 47(1), pp. 139-168, 1997

[XML, 98] World Wide Web Consortium
'Extensible Markup Language (XML) 1.0 Specification'
http://www.w3.org/TR/REC-xml, 1998

Filnavn:	main.doc
Bibliotek:	D:\Projekt\Rapport\Samlet
Skabelon:	D:\Microsoft Office\Skabeloner\Normal.dot
Titel:	Table of Contents
Emne:	
Forfatter:	Carsten Hellegaard
Nøgleord:	
Kommentarer:	
Oprettelsesdato:	31-01-00 15:57
Versionsnummer:	40
Senest gemt:	03-07-00 23:36
Senest gemt af:	Carsten Hellegaard
Redigeringstid:	162 minutter
Senest udskrevet:	03-07-00 23:46
Ved seneste fulde udskrift	
Sider:	97
Ord:	34.785 (ca.)
Tegn:	198.276 (ca.)