Uniform Treatment of Code and Data in the Web Setting

(Thesis format: Monograph)

by

Rachita Mohan

Graduate Program

in

Computer Science

A thesis submitted in partial fulfillment

of the requirements for the degree of

Master of Science

The School of Graduate and Postdoctoral Studies

The University of Western Ontario

London, Ontario, Canada

THE UNIVERSITY OF WESTERN ONTARIO

THE SCHOOL OF GRADUATE AND POSTDOCTORAL STUDIES

CERTIFICATE OF EXAMINATION

Supervisor:	Examination committee:
Dr. Stephen M. Watt	Dr. Marc Moreno Maza
	Dr. Eric Schost
	Dr. Rob Corless

The thesis by

Rachita Mohan

entitled:

Uniform Treatment of Code and Data in the Web Setting

is accepted in partial fulfillment of the

requirements for the degree of

Master of Science

Date	Chair of the Thesis Examination Board

Abstract

Over two decades, the Web has evolved into a highly successful and robust platform to share ideas, spread information, communicate and transact. This evolution has been aided by constantly evolving technologies that have enabled developers to expand the applications and utility of the Internet. In other computing settings, the duality of code and data has proven to be a powerful and productive conceptual device. However, the concept of code/data duality has not been prevalent in the web programming environment. The concept of “homoiconicity” or code is data and data is code has brought about innovative programming techniques like metaprogramming in the past.

We explore the idea of a uniform programming environment, where code and data are represented in the same manner and explore how this property of homoiconicity can be exploited in a web setting. This could also enable uniform handling of access and persistence and uniform transformation mechanisms. Such an abstraction could allow metaprogramming in web development which can provide an efficient solution to problems such as saving the web page state over a stateless web protocol.

Keywords: Code-data duality, Scheme, XML, metaprogramming

Acknowledgement

This research would not have been possible without the support of many people.

I would like to express my heartfelt thanks and respect to my supervisor, Dr. Stephen M. Watt. He not only guided me throughout my research, but also offered invaluable support and patience. Over the course of this research, through interaction and observation, I have learnt a great deal from him in my professional as well as personal life.

I wish to extend my gratitude to the members of ORCCA lab, for providing an interactive and thought-stimulating environment to work in.

I am deeply grateful to my parents for their constant love, support and encouragement to follow my interests and excel in life. Particularly to my father, for making me understand and appreciate the process of research, from his own experience as a researcher.

Special thanks to my flatmates Sneha Lele and Aditi Nadgauda, for being with me through out my course, and for providing a family and a home away from home.

Special thanks to my friends Karthick Ramachandran and Amit Regmi, for the endless technical discussions and being my moral supports.

Finally, I am indebted to Pratik Mittal, my best friend, mentor, ally and critic, for his unconditional support in all avenues of my personal and professional life.

Contents

List of Figures

List of Listings

Chapter 1

Introduction

About two decades ago, Tim Berners Lee invented the World Wide Web and the world has never been the same. The web has now become an integral part of our daily lives. However, for over ten years since its inception, the internet was mainly used for exchanging e-mails and accessing data from remote servers. It served as a medium for sharing of information and for communication of ideas over a distance. The launch of stable and user-friendly browsers in 1993 broadened the usage of the internet to people from various domains. The Web soon turned into a gigantic database of information from all over the world. With the arrival of AltaVista in 1995, a search engine that allowed natural language inquiries and multimedia searches for the first time, people were able to navigate through this maze of information. Even business was translated from the real world to the virtual world with sites like Amazon and eBay providing services online. Thus, the Web slowly penetrated into all domains of our lives.

The Web today has seen all and done all, from IRC to Twitter and Facebook, from Netscape to Chrome, from e-mail and chat to ripples and waves. It has transformed the way we live, from communication, entertainment and social networking to business, banking and media.

As the level of interactivity with users over the internet increased, the need to provide fast, efficient, aesthetically appealing and user-friendly services arose. Lee's HTML which was primarily static, could no longer meet the growing requirements of the users. Thus, the web development environment started to evolve with the advent of new languages. These new languages and concepts such as DHTML, AJAX, JavaScript, PHP and many more, brought dynamicity to the Web and provided web developers with a gamut of design options. For example, the simple *img* element in HTML could be used to add static images to a web page, whereas HTML 5 now provides the canvas element which can be manipulated to render images, graphs, shapes, animations and the like, at the client-side, using JavaScript. AJAX has enabled retrieval of data from servers without disturbing the entire page.

Despite the existence of so many technologies, some basic problems in web development still do not have an efficient and well-structured solution. One such problem is efficiently saving the state of a web page over the stateless web protocol. There exist various concepts such as homoiconicity and continuations which can provide a solution to such problems. However, neither homoiconicity nor continuations are either available in the web development environment or have not been efficiently implemented. We briefly discuss one such concept, homoiconicity, that forms the basis for this dissertation.

1.1 Homoiconicity

The concept of homoiconicity [2] has existed in the world of programming languages since the invention of Lisp. It has introduced many powerful concepts and techniques in the programming environment. In layman terms, homoiconicity means,

Code is Data and Data is Code.

This concept can be understood by looking at the two types of syntax of a programming language: Concrete Syntax and Abstract Syntax. The context-free grammar of a language defines the concrete syntax of a programming language. In other words, the concrete syntax describes how a program looks like to a programmer. On the other hand, the abstract syntax of a programming language depicts the inner representation of a language. The inner representation of a language includes how the language is implemented and what the program looks like to a compiler.

When the abstract syntax and the concrete syntax of a language are the same, the language is said to be homoiconic, that is code and data duality exists in such a language. Code and data duality is said to exist because the primary representation of code is in the same format as one of the fundamental data types in the language. An example of this is Scheme, a dialect of Lisp. The Scheme code is written as S-expressions. S-expressions are essentially lists within parentheses. The internal representation for Scheme is also in the form of lists. Since the internal and external representation of Scheme language is the same, code and data is represented in the same way in Scheme and interpreted according to the respective constructs. Thus, Scheme is a homoiconic language. Dialects of Lisp such as Clojure and Common Lisp and languages like REBOL, XSLT, Tcl and Lua are also homoiconic in nature.

In a programming language, the degree of homoiconicity depends on the correlation between the expressions in the language and the data structures used to represent them. The concept of code and data duality has brought about powerful techniques in the programming environment. Homoiconic languages lend themselves to metaprogramming and extensibility. Metaprogramming can be defined as writing programs that generate or manipulate other programs. A YACC parser generator is an example of a metaprogram. Many languages provide metaprogramming capabilities via their eval method. However, one cannot assume that a language with an eval function is homoiconic. In an eval method, the code to be evaluated is passed as a string, which is usually not the data structure of the language. Thus, code and data is not represented in the same manner in such a case.

Metaprogramming magnifies the reuse potential of large amounts of code written for frequently used features. It enables writing Domain Specific Languages (DSLs) and using a number of DSLs to write complex applications. An example of domain specific language could be LEX and YACC, which can together be used in the development of a compiler. Thus, homoiconicity and by extension, metaprogramming have revolutionized the world of programming languages.

1.2 Objectives

In the past, powerful programming concepts and techniques have been innovated based on uniform abstractions in programming languages, that is when code and data is treated the same way. However, the current web programming environment does not provide this functionality. Having realized the potential of code and data duality in a programming language, the objectives of this dissertation are:

- **Introduction of code and data duality in web programming environment.**

  This involves developing a model / prototype that provides a uniform abstraction for representing functions and data in the same manner. It also includes uniform handling of access and persistence and uniform transformations.

- **Exploring the benefits/limitations of such a model**

We have discussed the capabilities that the concept of homoiconicity has brought to the programming world. This study will explore programming ideas and techniques like metaprogramming, DSLs and the likes, in the web programming environment. It will also examine what are the limitations of exploiting this concept in a web setting.

- **Feasability of its implementation.**

  The process of developing such a model will help us experience the ease or difficulty in realizing this concept and thus analyze the challenges faced on the way.

- **Test the applicability of this implementation.**

  Finally, the applicability of this implementation will be tested in the web development environment by modeling various existing web development scenarios.

1.3 Thesis Organization

An interpreter JavaScript for an XML representation of Scheme is developed as a part of this thesis to realize the concept of homoiconicity in the web development environment. The rest of the thesis is organized as follows:

Chapter 2 talks about the various concepts and technologies that already exist in the web programming environment. It briefly contrasts the existing technologies with the implementation presented in the thesis.

In Chapter 3, we discuss the design considerations in developing a prototype to implement the concept of code and data duality in a web setting. It talks about why Scheme and JavaScript were chosen for the implementation.

Chapter 4 introduces the prototype which is an XML representation of Scheme and describes the actual design of the prototype and how various aspects of the concept were implemented.

Chapter 5 looks at some of the examples that were used to test the implementation. We further analyze and discuss the results of our tests.

In Chapter 6, we present and discuss ideas that can be built on top of the implementation presented in the thesis to further the idea of a uniform programming environment.

We conclude with a summary of our work and briefly discuss the possible future work. We also provide the specifications of the language prototype as an appendix.

Chapter 2

Background and Related Work

There is a famous saying by Heraclitus, “Change is the only constant”. Likewise, the web has been constantly evolving and has become an integral part of people’s lives today. It has changed from a unidirectional information portal which was used to broadcast information, to an interactive platform which is widely used for sharing ideas. Many programming languages, concepts and ideas have been instrumental in this evolution of the Web from primarily static content to live and dynamic content. Before describing the research from this thesis, we glance through some of the existing web programming concepts that have made an impact on the web environment and briefly look at their limitations.

2.1 Markup Languages

Markup was first seen in the publishing and printing industry. It was primarily used to provide the typesetter with instructions for the presentation of a document. IBM was the first to come up with a Generalized Markup Language (GML) which could be used to differentiate between the content and presentation in a machine-readable document. It was later accepted as an ISO standard and gave birth to Standard Generalized Markup Language (SGML). Most markup languages in existence today are derivatives of SGML. We now discuss two of the most popular markup languages.

2.1.1 HTML

The development of Hypertext Markup Language (HTML) [20], a publishing language for the World Wide Web, by Tim Berners Lee has been a major milestone in the evolution of the Web. The usage of HTML documents was further propagated by the introduction of the Mozaic Browser. Originally, HTML was a markup language that provided support for hypertext and some basic document structuring. Thus, it could be used for creating simple documents. Over the years, the capabilities of HTML have grown and authors can create web pages with all sorts of animations, sound and multimedia.

The flexibility and error-tolerance capabilities of HTML make it popular for web development but also raise some security issues. The presentation logic or stylesheets used and the dynamic generation of HTML, sometimes makes the re-usability of the document impossible. With the growing complexity of the Web, there was a need for a more powerful markup language that provided structure and meaning within documents, and facilitated the exchange of data. Thus, the eXtensible Markup Language (XML) was created.

2.1.2 XML

eXtensibleMarkup Language (XML) [6], a restricted form of SGML, is a meta-markup language. This means that using XML, one can create their own version of HTML or another markup language. The specification and entities for the new language are specified in a Document Type Definition (DTD) or Schema. XML has strict error-handling rules. Its parsers fail at the occurrence of even the simplest of syntax errors. Owing to these advantages, HTML has been reformulated in XML syntax as XHTML.

XML is not just used to create new document markup systems, but it also provides syntax for document markup. One of the main goals in designing XML was transporting and storing data, with emphasis on the data and not on its presentation. XML provides extensibility, structure and validation in comparison to HTML. Most importantly it allows late binding of presentation logic. However, XML is also verbose and thus, not preferred by many developers.

2.2 Client-Side Technologies

Rich client applications were needed for transforming the web from a static information portal to an interactive platform. This need resulted in the creation of many powerful technologies that have completely changed the face of the web.

2.2.1 JavaScript

Netscape pioneered the development of rich-client applications with the creation of JavaScript [10]. JavaScript made it possible to perform tasks such as calculations and form validation on the browser side, instead of the server-side. JavaScript soon became a preferred tool in web development environment however, it was a trademark of Sun Microsystems. For this reason, Microsoft came up with another client-side scripting language called JScript. However, both JavaScript and JScript [18] are dialects of a standardized scripting language ECMAScript and are essentially the same language with a few different constructs. ECMAScript [15] is a standard scripting language which is mainly used in Web applications. Popular dialects of ECMAScript include ActionScript [14], JavaScript and JScript . These dialects usually provide APIs or extensions to the language and thus enable specialized web computations. For example, JavaScript and JScript provide the DOM API for dynamic HTML generation and manipulation. However, ActionScript, an ECMAScript implementation from Adobe is a language, used primarily for development using Adobe Flash Player and provides an API for multimedia capabilities.

JavaScript is a multi-paradigm, prototype-based, functional, imperative, dynamic and weakly-typed scripting language. It provides scripting access to embedded objects in an HTML page through the Document Object Model (DOM) interface that is discussed later in the chapter. Together, JavaScript and DOM brought about the widely used concept of Dynamic HTML (DHTML). DHTML allows dynamically generating, manipulating and deleting HTML elements in a web page.

Although, JavaScript has been popular mainly because of its use in Web development as a scripting language, it is also a powerful programming language. It has C-like syntax, but is similar to Lisp or Scheme, which are functional languages. JavaScript's *eval* method has provided a lot of flexibility in Dynamic HTML element generation. The *eval* method and other JavaScript features like closures give it metaprogramming capabilities, but they are not as powerful as metaprogramming capabilities in languages such as Scheme.

2.2.2 Document Object Model

With the advent of client-side scripting languages like JavaScript came the possibility of dynamically generating HTML. An interface was required for handing the HTML elements to the JavaScript programs. The Document Object Model (DOM) [9] was introduced to serve this very purpose. The DOM is a platform and language-neutral interface that provides a standard object model for structuring XML and HTML documents. It also provides an API for updating the content, structure and style of an XML or HTML document. In addition, it also provides an interface for dealing with events and enables capture of user events or browser actions. Initially, each browser had its own version of the DOM. As a result of this, the web applications developed were often not compatible across various browsers. As a step towards standardization of the object model, the World Wide Web Consortium published their specification for the DOM, which is now widely used by browsers. The DOM is split into two sections: DOM Core and DOM HTML. Each DOM compliant browser supports both sections of DOM.

- *XML DOM.* DOM Core, also known as XML DOM, defines a set of XML object interfaces. It defines a tree-like structure of *Node* objects for the XML document, each representing an element, an attribute, content or some other object. The interfaces defined can be accessed through these *Node* objects. Each of the object types in the DOM not only has its own methods and properties, but also implements the *Node* interface. The interface provides some common methods and properties related to the document tree structure. It also enables navigation and provides reference for various elements in the tree. The top most *Node* object is the *Document* object that serves as the root for the document tree. It also implements the *Document* interface which provides methods like *getElementsByTagName()*, *createElement()* and *createTextNode()*.
- *HTML DOM.* The HTML DOM extends the Core DOM and describes interfaces specific to HTML documents. Capabilities like navigation through the document tree, access to tree elements and updation of nodes are available through

the DOM Core, however, functionality that depends on specific HTML elements is defined here.

2.2.3 AJAX

Another key concept that has helped make the web more dynamic is AJAX, Asynchronous JavaScript and XML [12]. The core idea behind AJAX is to make the communication with the server asynchronous, so that data is transferred and processed in the background. The user's page display remains undisturbed and the server hit is not visible to the user. Jesse James Garrett, who introduced the term AJAX, concisely describes AJAX as a collection of technologies that incorporates standards-based presentation using XHTML and Cascading Style Sheets (CSS); dynamic display and interaction using the DOM; data interchange and manipulation using XML and XSLT; asynchronous data retrieval using XMLHttpRequest; and JavaScript binding everything together. Various applications like Google Maps, Gmail and Google Suggest have exploited the concept of AJAX and are very successful today.

2.2.4 XSLT

JavaScript alone is not responsible for the dynamic nature of web content. Technologies including Cascading Style Sheets (CSS) and eXtensible Stylesheet Language(XSL) have also played a key role in making Web content dynamic. XSL Transformations (XSLT) [8] is a component of XSL, a language used for expressing stylesheets, that can be used independently. It is used to transform one type of XML into another. XSLT is often used to write the presentation logic for XML also. XML along with XSLT becomes heavy and is often not used by developers. XSLT is a Turing-complete language. It provides metaprogramming capabilities which make it more powerful and flexible than CSS. However, its specification states that it is designed primarily for the kinds of transformations that are needed when XSLT is used as part of XSL. The computational model of XSLT, however, is not apt for general-purpose programming.

2.3 Server-Side Technologies

As client-side technologies evolved, new ground was being covered in the server-side technologies. Client-side processing and server-side processing are both equally important in web development. Client-side technologies help provide responsive applications, but true dynamicity comes from server-side programming. Being able to write a single function that shows customized content to each user, depending on who has signed in, is a simple example of the dynamicity that server-side programming can provide.

2.3.1 PHP

PHP [19] is a server-side, cross-platform, HTML embedded scripting language that lets you create dynamic web pages. PHP-enabled web pages are treated just like regular HTML pages and you can create and edit them the same way you normally create regular HTML pages. However, PHP provides the option of using procedural programming, object-oriented programming or both. PHP programs are processed on servers and generate HTML, JavaScript and CSS, which implies that it provides limited metaprogramming capabilities. However, there is a possibility of incorrect HTML being generated, if programs are not written properly.

2.4 Google Web Toolkit

JavaScript, XML, DOM, AJAX and PHP have together provided developers with robust tools for user, as well as developer friendly web development. However, the clutter of so many technologies can be difficult to comprehend at times. Thus, it would be desirable to have an easy-to-use environment where all the technologies are hidden by abstractions in some other language which the developer is familiar with. Based on this precise idea, Google came up with the Google Web Toolkit.

Google Web Toolkit (GWT) [13], is a web development platform that provides a Java to JavaScript compiler which directly compiles code written in Java into HTML, Cascaded Style Sheets (CSS) and JavaScript. Thus, it manages to eradicate the confusion between the gamut of programming concepts and simplifies it with Java. Despite the popularity of GWT in recent times, there are certain limitations. Web pages have to be fast, but compiling to JavaScript in GWT takes considerable time. Also, switching from dynamic scripting languages which provide flexibility, back to strictly-typed Java-like language is not appreciated by experienced developers. Further, using Java for client-side web development can be confusing as the programming environment is very different from Java's environment. Lastly, GWT also does not support metaprogramming due to its optimization considerations.

2.5 Scheme

The Revised(5) Report on the Algorithmic Language Scheme [16] describes the idea behind Scheme precisely as,

Programming languages should be designed not by piling feature on top of feature, *but by removing the weaknesses and restrictions that make additional features appear necessary.*

Scheme is a lexically-scoped dialect of Lisp, which has only six constructs at its core. The rest of the expressions or constructs are built from the core forms using macros. It is a strongly homoiconic language where code and data are represented in the same way, as S-expressions. Scheme's features such as first-class continuations, quotations, closures and macros give it extremely powerful and versatile metaprogramming capabilities.

2.5.1 Quotation

The Scheme language has a powerful concept called quotation, which enhances its homoiconic nature. As was mentioned earlier, Scheme is a strongly homoiconic language in which code and data is represented in the same manner internally and externally. However, the compiler and interpreter must be able to distinguish between which data it should perceive as code and which as data. To make this distinction the keyword *quote* is used.

When the *quote* keyword is applied to an expression, the normal evaluation of the expression is inhibited and the expression becomes a list containing lists or immutable data. Literal constants such as booleans, characters, numbers and strings evaluate to themselves. However, literal identifiers evaluate to symbols, which are objects representing strings. A procedure expression evaluates to a list object. The *quote* expression is abbreviated as {'} . Listing 2.1 and Listing 2.2 show examples of non-quoted and quoted Scheme code where the former is interpreted as code and the latter as data.

```
(+ 2 3) => 5

7 => 7

"hello" => hello
```

Listing 2.1: Evaluated Scheme expressions

```
(quote (+ 2 3)) = '(+ 2 3) => (+ 2 3)

'7 => 7

'"hello" => hello

'a => a
```

Listing 2.2: Quoted Scheme expressions

A quote expression reads its arguments and writes them without evaluating them. These quote expressions can be evaluated using the *eval* method of Scheme language. The *eval* method takes an object that represents an expression as its argument. It evaluates a quoted expression, which is nothing but code represented as data. Listing 2.3 shows examples of *eval* method.

```

(eval '(+ 2 3)) => 5

(let ((f (eval '(lambda (g x) (g x x))))) (f + 10))
                                                 => 20
```

Listing 2.3: Examples of eval

2.5.2 Quasiquotation

The concept of Quotation ,in Scheme, enables representing code as data and allows evaluation of quoted data using *eval*. Nonetheless, one would also like to generate code at runtime. The keywords *quasiquote*, *unquote* and *unquote-splicing* together enable this and thus provide a strong base for metaprogramming.According to Alan Bawden [5], *Quasiquotation is the parameterized version of normal quotation where instead of specifying a value exactly, some holes are left to be filled later.*

The *quasiquote* keyword allows part of the expression to be “unquoted”. Without an *unquote* expression in it, a simple *quasiquote* expression would evaluate to the same result as a *quote* expression. However, if there is an *unquote* expression within the *quasiquote* expression, then the *unquote* expression will be evaluated and inserted into the result. Listing 2.4 gives examples of *quasiquote* expressions with *unquote* expressions embedded in them.

```
‘(a b ,(reverse ’(c d e)) f g)  => (a b (e d c) f g)
‘(a b ,@(reverse ’(c d e)) f g) => (a b e d c f g)

(define (eval-formula formula)
      (eval ‘(let ([x 2]
                   [y 3])
                 ,formula)))

(eval-formula ’(+ x y))      => 5
(eval-formula ’(- x (+ x y)) => -3

(eval ‘(+ ,@(cdr ‘(* 2 3)))) => 5
```

Listing 2.4: Quasiquoted Scheme expression

quasiquote is also represented as {‘} , *unquote* as {,} and *unquote-splicing* as {,@}. In case of *quote*, the evaluated result replaces the *unquote* expression in the parent expression, whereas in *unquote-splicing*, the result is spliced into the surrounding list.

The code introduced by the *quasiquote* operator is evaluated at compile-time instead of at run-time. Thus, code and code in the form of data is represented in the same manner and evaluated in the same manner. This induces simplicity and power in the metaprogramming abilities in Scheme.

2.6 Summary

As can be seen from the above discussion, none of the popular web programming languages employ the concept of code and data duality coupled with the ability to move between the two. This is a powerful concept that still remains unexplored in the web programming environment. We propose to explore the possibility of exploiting this concept in the web development environment. This thesis introduces an environment that provides an abstraction where all underlying technologies and concepts are hidden and where code and data is represented in the same manner. The environment would also allow accessing, transforming or generating objects in the same way.

Chapter 3

Homoiconic Language for the Web: Design considerations

The concept of homoiconicity has enabled development of metaprogramming, a powerful programming technique, which helps generate elegant programs that can exhibit varying behavior at run-time depending on the input. It is evident from the discussion of existing technologies that this concept is either not available or not fully implemented in the web programming environment. The aim of this research is to introduce a web development environment where code, data, access and persistence are represented in the same manner and further explore the areas in web programming, where this concept could be exploited.

Many languages have been developed and introduced in the web development environment to simplify programming for inexperienced web developers and at the same time, to provide tools for experienced web developers to write powerful and efficient code. Often, the various concepts in these languages are not in tandem with each other and have complex representation, thus confusing the developers. Hence, the best approach for achieving our aim was to implement our concept in an existing language at the most basic level instead of writing a new representation for a new language. We chose to introduce the concept by augmenting the basis of the entire web, the markup language XHTML itself.

A prototype is developed to introduce code/data duality to a web development environment. We create an interpreter JavaScript for evaluating an XML representation of Scheme code embedded in an XHTML web page. Scheme constructs are written as XHTML tags. This is evaluated at the browser using the interpreter JavaScript. It provides a uniform abstraction for representing code and data. In other words, the Scheme code in its XML representation and the HTML presentation logic are represented in the same manner. The language also has the same uniform representation for constructs to access and manipulate HTML elements and their values. The process of developing the prototype has helped identify the challenges in the implementation of the proposed concept.

Metaprogramming magnifies the reuse potential of large features. For example, adding/deleting rows to a table dynamically becomes trivial in such an environment. Metaprogramming can be enabled in HTML due to concepts like macros that are present in Scheme language. This means that a web developer will be able to define new HTML tags and write varying transformation logic for HTML in the new environment. Writing Domain Specific Web Languages will become possible. Imagine being able to create a web language using our language definition, with unique constructs defined for specific business applications or social networking websites.

3.1 Design Considerations

In the following section, we discuss the design considerations and how they effect our implementation of the interpreter for the new language.

3.1.1 Why Scheme?

In section 2.5, we briefly looked at some of the key features of the Scheme language. We now look more closely at these features and discuss how they make Scheme an apt choice for our implementation in web space.

Scheme, a statically scoped, multi-paradigm and homoiconic language, has clear and simple semantics and syntactic constructs that are written as S-expressions. These S-expressions can be represented by XML tags which can easily be embedded into HTML web pages. Syntactic constructs like define, lambda, set! and the like can be represented as XML tags with the body nested between these tags.

Scheme is a widely popular programming language. Using Scheme represented as XML enables a developer to use and build upon his basic knowledge of Scheme constructs and concepts in web development. Thus, constructs in the new XML language are familiar and more constructs can be created along the way in the same manner. An example of Scheme code represented as MATHML [21] for calculating the factorial of a number is shown in Listing 3.1 and Listing 3.2.

```
(define fact
        (lambda (n)
          (if (= n 0)
            1
            (* n (fact (- n 1))))))
(fact 3)
```

Listing 3.1: Scheme equivalent of XML representation

The main consideration behind choosing to write a Scheme interpreter is its property of homoiconicity. The presence of this property enables meta-programming, writing domain-specific applications/scripts and reflection. The homoiconicity in Scheme is due to the uniform representation of code and data in it. By representing Scheme

```
<apply>
        <define></define>
        <ci>fact</ci>
        <apply>
                <lambda></lambda>
                <apply>
                        <ci>n</ci>
                </apply>
                <apply>
                        <if></if>
                        <apply>
                                <ci>=</ci>
                                <ci>n</ci>
                                <cn>0</cn>
                        </apply>
                        <cn>1</cn>
                        <apply>
                                <ci>*</ci>
                                <ci>n</ci>
                                <apply>
                                        <ci>fact</ci>
                                        <apply>
                                                <ci>-</ci>
                                                <ci>n</ci>
                                                <cn>1</cn>
                                        </apply>
                                </apply>
                        </apply>
                </apply>
        </apply>
</apply>
<apply>
        <ci>fact</ci>
        <cn>3</cn>
</apply>
```

Listing 3.2: XML representation of Scheme for evaluating factorial

code as XML, we are able to embed it in HTML which is mostly data, thus providing a uniform representation for code and data, access and persistence and transformation.

Scheme's homoiconicity can also be attributed to another concept present in Scheme, quasiquotation. We have explained the concept of quasiquotation in section 2.5.1. In the new language, we implement *quasiquote and unquote* tags to implement the corresponding concept in Scheme. These tags enable a programmer to write HTML and Scheme code represented as XML in the same manner. HTML code is embedded in *qquote* tags and Scheme code in *unquote* tags. Together, the S-expressions and quasiquotation provide easy and flexible means for manipulating and generating programs. Thus, an HTML web page becomes a sequence of S-expressions, with or without nesting, to a developer and he no longer needs to be concerned with underlying JavaScript or AJAX or any other technology.

Another very important feature in Scheme is first-class continuation [7], which is an abstract representation of the control state. Continuations can solve problems such as maintaining page state in a web application in an uncomplicated manner. A page state can be persisted in a continuation and then later returned to by simply calling the saved continuation. Other programming constructs that can be introduced using continuations are threading, co-routines and exceptions.

3.1.2 Features of Scheme to be implemented

Over the years, Scheme has become a very popular programming language with many implementations and standards. Each implementation differs in the set of features being implemented. For the purpose of this thesis, it is feasible to implement only a subset of all the features in various implementations of Scheme. The most recent, commonly implemented Scheme standards are R5RS and R6RS. R6RS Scheme is a collection of libraries. The R6RS Scheme base library corresponds largely to R5RS Scheme. However, I/O operations, some list functions, force and delay and the like do not exist in core library. In fact, the core library implements the full numerical tower in contrast to R5RS which only allows a subset of the numerical tower to be implemented. Our implementation of Scheme, represented as XML, is a subset of the R5RS Scheme.

The idea behind Scheme is to keep it simple by implementing only the minimum number of necessary features that can later be used for expansion. Based on a similar concept, we choose to implement the features of Scheme that depict (or enable) the homoiconicity in the language and the ones that can be used as building blocks for further expansion of the language. Thus, the salient features of Scheme selected for implementation in this thesis are lexical scope, program as data, first-class continuation, hygienic macros and quasiquotation. A subset of the numerical tower will be implemented as well. However, implementing all standard Scheme procedures like environment procedures, input/output procedures or all string procedures is not feasible. More specifically, features like modules, I/O operations and the like are not implemented at this stage. They may be added later as additional libraries.

A list of standard forms, procedures and numeric procedures that have been implemented is given in the appendix.

3.1.3 Why Javascript?

JavaScript is a widely used scripting language for accessing HTML elements and DOM objects. Therefore, using JavaScript for writing a Scheme Interpreter is an obvious choice. Apart from providing client-side evaluation and accessibility to DOM objects and API, it also shares some common features with Scheme such as dynamic typing, closures and allowing functions to be first class values. Given that Scheme and JavaScript are both functional programming languages, it becomes easier to implement Scheme concepts intuitively in JavaScript.

A drawback of using JavaScript is cross-browser compatibility. JavaScript functions for Firefox and Internet Explorer are not uniform and differ in some aspects. Thus, the interpreter JavaScript should be able handle and access HTML/Scheme code represented as XML correctly, irrespective of the browser it is being rendered on.

3.1.4 Augmenting HTML: Limitations and Capabilities

As discussed earlier in chapter 2, JavaScript and PHP enable generation and manipulation of HTML dynamically on the client-side and server-side respectively. However, the HTML generated is often incorrect and gives rise to mal-formed web pages. This happens because in these languages, HTML code is usually generated as a string or as node objects, which are then embedded in HTML pages. These strings and node objects often do not represent the HTML correctly. Therefore, instead of creating another language that generates HTML, we have chosen to introduce homoiconicity at the most basic level by augmenting HTML itself. This would ensure correctness of HTML and would enable code reuse. It would also keep programming simple, yet powerful. However, this brings in some challenges as well. Despite the obvious advantages of introducing homoiconicity in HTML, writing complex programs in HTML can be verbose.

HTML also has many versions and can be strict or transitional. However, it is still not possible to access elements using JavaScript and DOM, when the HTML tags are not formed properly. An example of HTML that is correct in our language that will not be parsed properly by JavaScript is given in Listing 3.3. In this example, HTML is not parsed correctly and the *tr and td* elements are not accessible through the DOM.

This problem has been resolved by writing a new Document Type Definition (DTD) for our language which allows writing incomplete HTML within quote tags. A DTD is a set of markup specifications for defining a document type. It simply

```
<body>
    <h1>Dynamic Table</h1>
        <table id="dTable" border="1">
            <unquote>
                <apply>
                    <define></define>
                    <ci>row</ci>
                    <apply>
                        <quote></quote>
                        <tr>
                           <td width="30px">
                              <unquote>
                                  <cn>"Row"</cn>
                              </unquote>
                           </td>
                           <td width="40px">&nbsp;</td>
                        </tr>
                    </apply>
                 </apply>
            </unquote>
            <unquote>
                 <ci>row</ci>
            </unquote>
        </table>
</body>
```

Listing 3.3: HTML generating row for a table dynamically

describes the schema for a language of the SGML/XML family. Our DTD defines a document type where the document must follow the specifications of an XHTML DTD with one exception. Any markup written between quote tags is allowed. This allows a programmer to dynamically generate HTML code, but at the same time ensures that HTML written outside unquote tags is correct and will not break while parsing.

Chapter 4

Homoiconic Language for the Web: Design and Implementation

In chapter 3, we discussed the design considerations for the new language and reasons for languages choices like Scheme, XHTML and JavaScript for our implementation. In this chapter we describe the concise design of our interpreter and how it has been implemented.

4.1 Design

For an implementation to be simple and efficient, the design architecture of a language must be described elaborately with its hierarchy. More importantly, the original aim for choosing to construct a new language must not be lost during the design procedure. The aim of this research is to introduce homoiconicity in the web development environment. This goal is achieved by embedding XML-like Scheme code in an XHTML web page. Since Scheme code is represented as XML and XHTML is also of the XML family, the representation of code and data becomes identical and thus, the language becomes homoiconic. The design of this language is described in the following subsections.

4.1.1 Writing the *Document Type Definition*

Before describing the design of the interpreter itself, the first step is to ensure that the document itself is in a form that can be correctly parsed and interpreted. For this, the following two conditions must be met.

- The document must be strictly XHTML. This means that the HTML tags must be nested properly and must have balanced tags. Each tag must have a start and an end. For example,

  ```
  <apply> <define></define> <ci>a</ci> </apply>
  ```

- The content within *quote* and *qquote* tags may not follow XHTML rules.

To ensure that the above two conditions are met, a Document Type Definition (DTD) is written that is mostly similar to an XHTML DTD, except that it allows anything to be written within *quote and quasiquote* tags.

4.1.2 Implementing *unquote* tags

Writing Scheme code as XML tags induces homoiconicity since code and data are now represented in the same manner. In Scheme, the quote/quasiquote constructs are used to represent a list as data and the unquote construct is used to evaluate code, whose result is then embedded in a data list.

```
<html xmlns="http://www.w3.org/1999/xhtml">
  <head><title>New Web Project</title></head>
  <body>
      <unquote>
        <apply>
            <define></define>
            <ci>fillBox</ci>
            <apply>
                <lambda></lambda>
                <apply><ci>key</ci></apply>
                <apply>
                    <qquote></qquote>
                    <apply>
                        <div>
                           <label>
                              <unquote><ci>key</ci></unquote>
                           </label>
                        <textarea rows="1"></textarea>
                        </div>
                    <apply>
                </apply>
            </apply>
        </apply>
       </unquote>
       <unquote>
            <apply><ci>fillBox</ci><cn>"Name"</cn></apply>
       </unquote>
       <unquote>
            <apply><ci>fillBox</ci><cn>"Age"</cn></apply>
       </unquote>
       <unquote>
            <apply><ci>fillBox</ci><cn>"Address"</cn></apply>
       </unquote>
  </body>
</html>
```

Listing 4.1: Mostly quoted web page

Borrowing from this concept, special *unquote* tags are created that have Scheme code represented as XML embedded in them. The entire XHTML page can be viewed as quoted data with unquote tags containing code within them, which are then evaluated by the interpreter. Thus, the first task of the interpreter will be to find the unquote tags in the HTML page and prepare them for evaluation. Listing 4.1 gives an example of XML-like Scheme code embedded in *unquote* tags and XHTML data embedded in *qquote* tags. Listing 4.2 is the Scheme code representation of Listing 4.1 and showcases the mapping between the two.

```
(html
  (head (title "New Web Project"))
  ‘(body
     ,(define fillbox
         (lambda
            (key)
            ‘(div
                (label ,key)
                (textarea))))
     ,(fillbox "Name")
     ,(fillbox "Age")
     ,(fillbox "Address")))
```

Listing 4.2: Scheme code for the mostly quoted web page

Listings 4.1 and 4.2 exhibit an exact correspondence between the Scheme code and its XML representation. In the actual language however, the *html* tag is not processed. XML like Scheme code can be embedded within *script* tags of *scheme-xml type*, inside the *head* tags. The *body* tag is assumed to be embedded in *qquote* tags. The XML-like Scheme code in the *head* tag and the “quoted” *body* element is then evaluated by our interpreter. As can be seen from the two representations, the *apply* tags in Listing 4.1 are analogous to the parentheses marking the start and end of an expression in Listing 4.2. Scheme keywords like `define` and `lambda` are represented as *define* and *lambda* empty tags. The variables in Scheme, like *key* and *fillbox* in our example, are embedded within *ci* tags and the constants, like *“Name”*, *“Age”* and *“Address”* are embedded within *cn* tags. Finally, in the case of quasiquotation, the XML representation of Scheme’s ‘ symbol is actually a representation of the expanded form of Scheme’s `quasiquote` operator. This means that an expression `‘(+ 2 3)`, whose expanded form is `(quasiquote (+ 2 3))` will be represented in XML as `<apply> <qquote></qquote> <apply> <ci>+</ci>`

`<ci>2</ci> <ci>3</ci> </apply> </apply>`. In Scheme language, arithmetic operators are not keywords and are treated as variables whose value can be changed in a program. Thus, they are represented in the same manner as other variables are.

A more detailed description of the language can be found in the Appendix.

4.1.3 Implementing corner features of Scheme

For evaluating the Scheme code embedded in the XHTML page, the constructs of the new language must be defined. The architecture of the interpreter has been structured into two-tiers.

The first tier consists of the core constructs, the data-types and numerical hierarchy. The second tier consists of constructs that can be defined using constructs from the first tier. We will briefly describe the constructs and data-types in each tier.

TIER 1

- Definition (`define`), assignment (`set!`), binding constructs (`lambda` and `let`), conditional expression (`if`), sequential evaluation (`begin`), quotation (`quote`, `unquote`), variable references and procedure calls form the core of Scheme implementations. Using these constructs, new constructs can be created. These constructs are implemented in tier 1 of the interpreter.
- Only Integer data-type is implemented from amongst the numeric data-types.
- Non-numeric data-types like Boolean, lists and pairs, vector, strings and symbols are implemented in this tier.
- Tier 1 also includes implementation for vector construction and manipulation expressions, equivalence predicates (`eq?`, `equal?` and `eqv?`), symbol to

string conversion, list and pair procedures (`list?`, `pair?`, `cons`, `car`, `cdr`, `length`, `set-car!`, `set-cdr!`, `assoc`), functional programming procedures (`procedure?`, `apply`) and identity predicates (`boolean?`, `pair?`, `symbol?`, `number?`, `vector?`).

- First-class continuations are also implemented in tier 1 (call-with-current-continuation).
- Lastly, basic arithmetic operators (`+, -, /, *`) and relational operators are implemented in this tier.

TIER 2

- Tier 2 is built on top of tier 1 as the constructs and procedures implemented in tier 1 are sufficient to implement tier 2 constructs.
- Syntactic extension (`syntax-rules` and `define-syntax`), conditional procedures (`case, cond`), delayed evaluation (`force, delay`), logical operators (`and , or`), iteration procedure(`do`), functional programming procedures (`map`) and quasiquote are implemented as a part of tier 2.
- Tier 2 is extendable and more procedures can be added to it later.

The entire list of procedures and constructs in their XML representation are detailed in the appendix.

4.1.4 Implementing constructs for accessing Web data

In the previous section, we described our design for evaluating XML like Scheme code, which introduces homoiconicity in our language. However, the goal of this thesis is to bring uniformity in representing code and data, as well as in accessing web data. JavaScript provides a vast number of DOM objects and methods that enable effective manipulation of HTML. We implement a subset of these methods, written in XML-like Scheme syntax that allows basic manipulation of XHTML elements using Scheme code. Listing 4.3 gives an example of JavaScript code for adding an event handler and Listing 4.4 gives the same example in our language.

```
<html>
  <head>
     <script language="javascript">
       function addEvent(){
         var btnEl = document.getElementById("btn1");
         btnEl.addEventListener("click", clickFunc, false);
       }
       function clickFunc(){
         var dvEl = document.getElementById("dv1");
         dvEl.innerHTML = "This is the clicked page.";
       }
     </script>
  </head>
  <body onload="addEvent()">
        <div id="dv1"> This is 1st page.</div>
        <button id="btn1">Click Me!</div>
  </body>
</html>
```

Listing 4.3: JavaScript code for adding event handler

The DOM manipulation methods are written in Tier 1 , described in the previous section. The `documentEl` and `bodyEl` tags are used for the *document* and *body* objects of JavaScript. The `title`, `get-element-by-id`, `create-element` and `write` methods are created for `documentEl` corresponding to *document.title, document.getElementById() and document.write()* methods from JavaScript respectively. Similarly, a subset of methods and properties, common to all HTML element objects in JavaScript, are implemented for our language. Some exam-

```
<html>
 <head>
  <title>Dynamic Table Generation - Example</title>
 </head>
 <body>
 <div id="div1">
    <div id="dv1">This is 1st page.</div>
    <button id="btn1">Click Me</button>
    <unquote>
       <apply>
           <define></define>
           <ci>clickMethod</ci>
           <apply>
              <lambda></lambda>
              <apply></apply>
              <apply>
                <inner-html></inner-html>
                <apply>
                    <get-element-by-id></get-element-by-id>
                    <cn>"dv1"</cn>
                </apply>
                <cn>"This is the clicked page."</cn>
              </apply>
           </apply>
       </apply>
    </unquote>
    <unquote>
       <apply>
          <add-event-listener></add-event-listener>
          <apply>
              <get-element-by-id></get-element-by-id>
              <cn>"btn1"</cn>
          </apply>
          <cn>"click"</cn>
          <ci>clickMethod</ci>
       </apply>
     </unquote>
  </body>
</html>
```

Listing 4.4: XML like Scheme code for adding event handler

ples are `get-elements-by-tag-name`, `add-event-listner`, `childnodes`, `tag-name`, `append-child`, `parent-node` and `remove-child`.

Many more properties and methods specific to each HTML element can be added to Tier 1 later.

4.2 Implementation

A JavaScript program is written to implement the Tier 1 of the interpreter architecture. The Tier 2 implementation consists of files containing Scheme code represented as XML. This XML like Scheme code is evaluated using the core forms implemented in tier 1. As described in the design, a Document Type Definition is written to ensure that the document containing the code to be evaluated is syntactically correct and contains complete tags. We now look at the step-wise implementation of the interpreter.

1. The document is loaded at the browser. A set of JavaScripts which contain the interpreter code are also loaded with the document. On the *onload* event of the body of the document, a JavaScript function, *interpret*, which marks the beginning of interpretation, is called. This function finds the unquote tags embedded in the body of the XHTML web page and passes them on to the parser.

2. The DOM parser is used to parse the XHTML document. However, the parsed document is not in a form where it can be easily evaluated or errors can be detected. The DOM parser parses the XHTML and returns a Document node object with child nodes. However, the child nodes could be empty text nodes as well, which makes it tough to manipulate these nodes directly. Therefore, we write another parser that parses the parsed document node, eliminates the empty text nodes, detects syntax errors and outputs a list of expressions. The

nodes are parsed into lists, which are represented by JavaScript arrays. These lists are analogous to S-expressions in Scheme. For every apply tag in the document a new list/s-expression is generated. Thus, the final output of the parser is a set of nested lists.

3. Before explaining the evaluation procedure, the core procedures' implementation and utility functions must be described. A *SchemeEnv* class is created to represent the environment. This class contains a hash-table which acts as a symbol-table and stores all variable-value pairs, maintains the current position in the nested list/s-expression, defines functions for adding and retrieving variables and their values, and functions for creating new environments and returning to parent environment. A *Tokens* class is defined to describe all the tokens in the language. Classes are written for the list, vector and symbol data types. Other data types are represented by JavaScript data types. As an example, the integer and the string data type are represented by JavaScripts number and string data types. For utility functions like car, cdr, cons of a list, functions for identifying if the given element is a string, identifier, number, vector or procedure and for printing the result, a Util class is created.

4. The procedures for evaluating core Scheme constructs that were described earlier in the design for Tier 1 and procedures for constructs for manipulation of the HTML DOM, are stored in a *reservedSymbols* array. Corresponding to each construct, for example get-element-by-id, inner-html, define, lambda, if, + , - , and the like, a builtIn object is stored. A builtIn class contains the name of the construct, a function containing JavaScript code for evaluation of the expression corresponding to the construct, and a string or a function that provides the evaluation when the construct has not been called as an application, but rather

as a variable. An example of the builtIn object for the + application/procedure is given in Listing 4.5.

```
    new builtIn('+', function(argList){

            var result = 0;
            var args = Parser.evaluateList(argList);
            while (args.length != 0) {

            var arg = Util.car(args);
            arg = arg.toString();

            if (Util.isNumber(arg))
                    result += eval(arg);
            else
                    throw incorrectTypeError;

            args = Util.cdr(args);
            }

            var res = new Number(result);
            return res;
    }, "arity-dispatched-procedure")
```

Listing 4.5: builtIn object for '+' procedure

5. The *Parser* class contains procedures for parsing, getting the next s-expression or list, evaluating an expression and evaluating a list of expressions. The *evaluate* procedure, that evaluates a list/s-expression represented internally as a JavaScript array, calls a procedure from the *action* class corresponding to whether the expression is a constant, identifier or application. The *action* class contains procedures that specify how to process a procedure call, a constant or an identifier. If the expression is an identifier, then it is first looked up in the environment's symbol-table and then in the reservedSymbols array. The value corresponding to the found variable is then returned. If its an application, then also the application is looked up in the reservedSymbols array and the environment symbol-table and the corresponding function that contains the evaluation

is called. These methods are called recursively to evaluate expressions and sub-expressions.

6. The result of the evaluation of an expression is returned from the evaluate function. This result is then processed and a text node or an element containing the result is returned. This new node which contains the expression's evaluation replaces the *unquote* tag in the document.

7. Syntax errors are caught during parsing and during evaluation and thrown to the main function where the processing began. The evaluation for the rest of the expression ceases and the error is displayed in the document.

8. For the implementation of tier 2, the simple-macro implementation for Scheme by Jonathan Rees *et.all* [17] is used. A transformer is written to transform Rees' Scheme code into its XML representation. This XML representation of the simple-macro implementation is first expanded by the tier 1 procedures and then, the expanded form is evaluated using the tier 1 implementation.

4.3 Test Cases

Testing is an important aspect of any form of software development. The purpose of testing is not to find all the bugs in the system, but to ensure that the system meets all the requirements specified in the aim for creating a new software. Our goal was to create a language that evaluates a subset of Scheme functionalities and JavaScript DOM functionalities in its XML representation. Thus, it is imperative that we test our language for these set of functionalities.

A program, written in any programming language, is an integration of expressions or statements. No single command or expression can test a concept or function completely. Thus, we perform grey-box testing at the integration and system testing levels.

Our test cases range from simple programs like calculating a factorial or finding the length of a list, to complex programs that can themselves be used in further development of the language and that can manipulate the HTML elements dynamically. These programs test the basic constructs of the Scheme language like `define, lambda, if, set, call-with-current-continuation, quasiquote, unquote, let and let*` along with list-manipulation and vector-manipulation methods and DOM manipulation capabilities. In addition to test cases, we have use cases where the program can be used for further implementation. Examples of use cases are programs defining Scheme keywords like `length, map, values and call-with-values` which can be used further, to write applications.

The tests were chosen to reflect the applications that a user would want to create using this language. Test cases were developed during the development process of the language itself. The test programs containing hundreds of lines of codes, in XML form, were evaluated successfully at the Mozilla Firefox 3.0.11 and Opera 9.62 browsers. Some examples of the language and their evaluated outputs are presented in the next chapter.

Chapter 5

Homoiconic Language for the Web: Annotated Examples

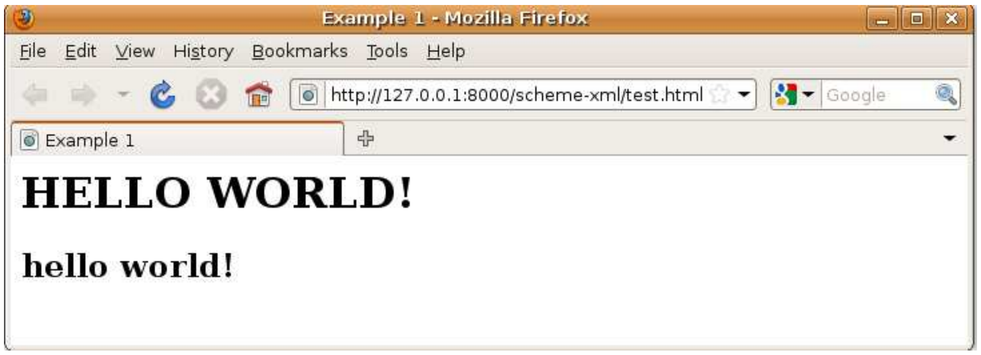
A programming language specification can be given in various forms like natural language, formal semantics, examples or test suites. Natural language alone can sometimes not describe aspects of a programming language efficiently. Thus, we choose to describe our language using a combination of examples, presented in this chapter, and BNF grammar for the language that is presented in Appendix A.

5.1 Hello World!

Listing 5.1 presents a simple program that prints "HELLO WORLD!" at the browser after evaluation. This example depicts the simple use of an *unquote* tag in the program. As mentioned earlier the *body* tag is assumed to be embedded in *quasiquote* tag for evaluation. Lines 6 and 7 are thus evaluated as quoted data and returned as is. This lines will print "HELLO WORLD!" at the browser. Line 8-13 depicts the use of *unquote* tag. Lines 9-12 contain code to convert “HELLO WORLD!” to lower case. Any kind of computation/code is embedded in *unquote* tags. *cn* tag on line 11 is used to represent strings and numbers. The output of this program is displayed in Figure 5.1. For ease of understanding, the corresponding Scheme code is also shown in Listing 5.2.

```
 1 <html>
 2     <head>
 3         <title>Example 1</title>
 4     </head>
 5     <body>
 6         <h1>HELLO WORLD!<h1>
 7         <h2>
 8           <unquote>
 9             <apply>
10                 <string-tolower></string-tolower>
11                 <cn>"HELLO WORLD!"</cn>
12             </apply>
13           </unquote>
14         </h2>
15     </body>
16 </html>
```

Listing 5.1: Example 1: Hello World


Example 1 - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://127.0.0.1:8000/scheme-xml/test.html
Google
Example 1

HELLO WORLD!

hello world!

Figure 5.1: Example 1: Hello World!

```
        (html
          (head (title "Example 1"))
          `(body
              (h1 "HELLO WORLD!")
              (h2
                ,(string-tolower "HELLO WORLD!"))))
```

Listing 5.2: Scheme code for Example 1

5.2 Nth list tail

The example given in Listing 5.4 calculates the nth tail of a list. The objective of this example is to show the usage of basic Scheme constructs like `define, lambda, if` and the like, and to show operations on a Scheme list. In line 4 of the Listing 5.4, the *define* expression begins. Variables are embedded within *ci* tags, as in line 5. Line 7 gives a lambda expression that takes two arguments *ls and n.* The *list-tail* method is called recursively on the *cdr* of the list in line 15-21, based on the condition that is evaluated in line 12. Finally, the *list-tail* method is called with suitable arguments in line 30. The Scheme code and output for the example are given in Listing 5.3 and Figure 5.2

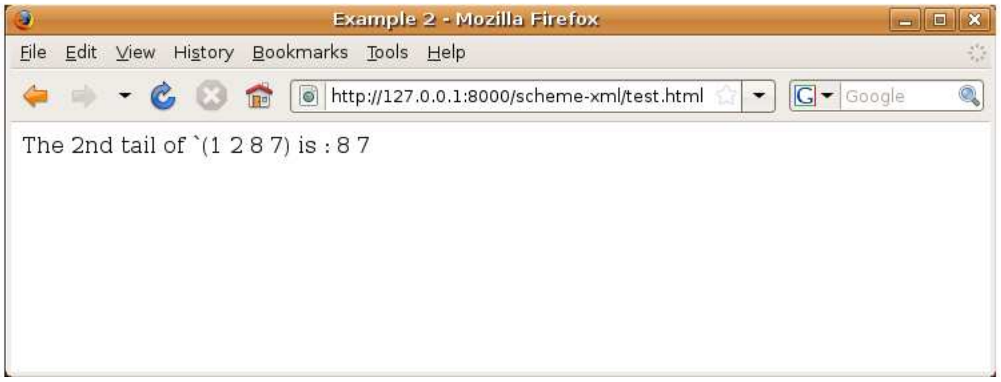
```
  (html
    (head (title "Example 2"))
    `(body
        ,(define list-tail
            (lambda (ls n)
              (if (= n 0)
                ls
                (list-tail (cdr ls) (- n 1)))))
        ,"The 2nd tail of `(1 2 8 7) is : "
        ,(list-tail `(1 2 8 7) 2)))
```

Listing 5.3: Scheme code for Example 2

```
1 <body>
2   <unquote>
3     <apply>
4       <define></define>
```

```
      <ci>list-tail</ci>
      <apply>
        <lambda></lambda>
        <apply><ci>ls</ci><ci>n</ci></apply>
        <apply>
          <if></if>
          <apply>
            <ci>=</ci><ci>n</ci><cn>0</cn>
          </apply>
          <ci>ls</ci>
          <apply>
            <ci>list-tail</ci>
            <apply><cdr></cdr> <ci>ls</ci></apply>
            <apply>
              <ci>-</ci> <ci>n</ci> <cn>1</cn>
            </apply>
          </apply>
        </apply>
      </apply>
    </apply>
  </unquote>
  <unquote>
    <cn>"The 2nd tail of `(1 2 8 7) is : "</cn>
  </unquote>
  <unquote>
    <apply>
      <ci>list-tail</ci>
      <apply>
        <qquote></qquote>
        <apply>
          <cn>1</cn><cn>2</cn><cn>8</cn><cn>7</cn>
        </apply>
      </apply>
      <cn>2</cn>
    </apply>
  </unquote>
</body>
```

Listing 5.4: Example 2: Nth list-tail


Example 2 - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://127.0.0.1:8000/scheme-xml/test.html
Google

```
The 2nd tail of `(1 2 8 7) is : 8 7
```

Figure 5.2: Example 2: Nth list-tail

5.3 Factorial

Scheme code in the form of XML can be written in the *head* element of an html document, by embedding it within *script* tags. The following examples show code to calaculate the factorial of a number. In Scheme, the factorial can be calculated using various methods. Each of those methods can be tested by keeping the *body* of the html document same and changing the factorial definition in the *head* element. As mentioned earlier, we have written a small program to convert Scheme code into its XML representation. We use the same program here, to convert various Scheme factorial definitions into XML. Listing 5.6 gives the XML output for one such factorial definition given in Listing 5.5, as processed by our program. The method calls the factorial function recursively to perform the computation.

```
(define factorial
  (lambda (n)
    (if (= n 0) 1
        (* n (factorial (- n 1))))))
```

Listing 5.5: Calculating factorial using recursion

```
<apply>
  <define></define>
  <ci>factorial</ci>
  <apply>
```

```
    <lambda></lambda>
    <apply><ci>n</ci></apply>
    <apply>
      <if></if>
      <apply>
        <ci>=<ci>
        <ci>n</ci>
        <cn>0</cn>
      </apply>
      <cn>1</cn>
      <apply>
        <ci>*</ci>
        <ci>n</ci>
        <apply>
          <ci>factorial</ci>
          <apply>
            <ci>-</ci>
            <ci>n</ci>
            <cn>1</cn>
          </apply>
        </apply>
      </apply>
    </apply>
  </apply>
</apply>
```

Listing 5.6: XML output for Scheme code

Another method that uses a loop to calculate factorial is given in Listing 5.7.

```
(define factorial
    (lambda (n)
        (do ((i 1 (+ i 1))
             (f 1 (* f i)))
           ((> i n) f))))
```

Listing 5.7: Calculating factorial using iteration

Finally, we look at an example for calculating factorials that uses continuations. Continuation is a very important concept in Scheme programming language. It represents the state of the environment or the stack, during a particular computation. These continuations can be saved to be called repeatedly or used to provide a non-local exit. In this example, shown by listing 5.9, we use continuations to calculate

```
(html
  (head (title "Example 3")
        (script
          ,(define ret #f)
          ,(define factorial
             (lambda (k)
               (if (= k 0)
                 (call/cc (lambda (s) (set! ret s) 1))
                 (* k (factorial (- k 1))))))))
  `(body
      ,"The factorial of 5 is : "
      ,(factorial 5)))
```

Listing 5.8: Scheme code for Example 3

the factorial of a number. The saved continuation can be used to find multiples of the factorial.

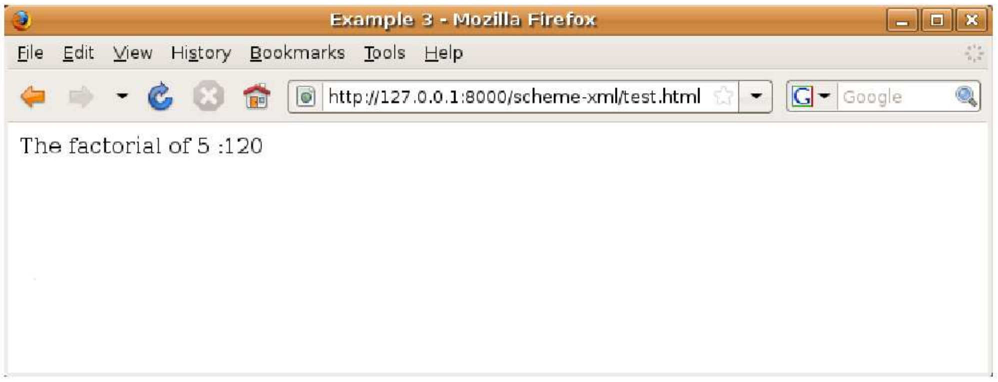
The *callcc* method, a primitive procedure in Scheme and our language, is invoked when the number, whose factorial we are calculating, is 0. This condition is shown in lines 20-23 of the listing. *Callcc* takes a procedure as its argument. This procedure itself takes one argument. *callcc* generates a representation of the current evaluation stack and environment and passes it as argument to the previously described procedure. Lines 29-33 store the current continuation in a global variable *ret* to be called later. At this stage, the computation state will consist of a stack that will be equivalent to an expression like (* 5 (* 4 (* 3 (* 2 (* 1 _))))), where, _ will be filled by the value returned or an argument. In the first case, the value returned is 1, but when the continuation, that had stored this computation state, is called repeatedly with different arguments, the _ is filled by these arguments and the computation is re-evaluated. In this way, we get the multiples of 5! repeatedly. Listing 5.8 and Figure 5.3 give the Scheme code and the output for this example.

```
1 <html>
2   <head>
3     <title>Example 3</title>
```

```
      <script type="text/scheme-xml">
        <unquote>
          <apply>
            <define></define>
            <ci>ret</ci>
            <false></false>
          </apply>
        </unquote>
        <unquote>
          <apply>
            <define></define>
            <ci>factorial</ci>
            <apply>
              <lambda></lambda>
              <apply><ci>k</ci></apply>
              <apply>
                <if></if>
                <apply>
                  <ci>=</ci><ci>k</ci><cn>0</cn>
                </apply>
                <apply>
                  <callcc></callcc>
                  <apply>
                    <lambda></lambda>
                    <apply><ci>s</ci></apply>
                    <apply>
                        <set></set>
                        <ci>ret</ci>
                        <ci>s</ci>
                    </apply>
                    <cn>1</cn>
                  </apply>
                </apply>
                <apply>
                  <ci>*</ci>
                  <apply>
                    <ci>factorial</ci>
                    <apply>
                        <ci>-</ci>
                        <ci>k</ci>
                        <cn>1</cn>
                    </apply>
                  </apply>
                  <ci>k</ci>
                </apply>
              </apply>
            </apply>
        </apply>
      </unquote>
    </script>
  </head>
<body>
    <unquote><cn>"The factorial of 5 :"</cn></unquote>
    <unquote>
        <apply><ci>factorial</ci><cn>5</cn></apply>
    </unquote>
</body>
</html>
```

Listing 5.9: Factorial example

All the above mentioned methods for calculating factorial give the same output, which is shown in Figure 5.3.


Example 3 - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://127.0.0.1:8000/scheme-xml/test.html
Google

```
The factorial of 5 :120
```

Figure 5.3: Example 3: Factorial

5.4 Dynamic Form Generation

Listing 5.11 presents an example for dynamic HTML generation. JavaScript has a rich DOM API which, among other things, provides client-side event handling and generates new HTML elements. We have incorporated the same capabilities in our language. The application in this example generates new fields for a form, based on user's inputs. Lines 1-9 render 2 textboxes for entering the name and the input type

```
(html
  (head (title "Example 4"))
  `(body
      (div
        ("Field Name" (textarea))
        ("Field Type" (textarea))
        (button))
      (div)
      ,(define clickMethod
          (lambda ()
            (append-child
             (get-element-by-id "dv2")
              `(div (table (tr
                (td
                 (label
                  ,(node-value
                  (get-element-by-id "txt1"))))
                  (td
                   ,(create-element
                    (node-value
                      (get-element-by-id "txt2"))))))))))
      ,(add-event-listener
       (get-element-by-id "btn1") "click" clickMethod)))
```

Listing 5.10: Scheme code for Example 4

of the field to be added in a form and button to add a new field to the form. The event listener is added to the button in lines 63-71. The event listener takes 3 arguments: the element on which the event will occur, the event name, and the method that will be called when the event occurs.

The method ,*clickMethod* defined on line 12, defines a procedure that appends a *div* to an existing *div*. Each new *div* contains a label with the name of the field that the user entered and an element of the type mentioned in the field type textbox. This is done using the `append-child, node-value, get-element-by-id and create-element` procedures of our language. The output of this application on subsequent clicks of the button are shown in Figures 5.4, 5.5, 5.6 and 5.7. Listing 5.10 represents the application in Scheme language.

```
<body>
  <div id="dv1" align="center">
    Field name:
    <textarea id="txt1" rows="1"></textarea>
    Field type:
    <textarea id="txt2" rows="1"></textarea>
    <button id="btn1">Add Field</button>
  </div>
  <div id="dv2"></div>
  <unquote>
    <apply>
      <define></define>
      <ci>clickMethod</ci>
      <apply>
        <lambda></lambda>
        <apply></apply>
        <apply>
            <append-child></append-child>
            <apply>
                <get-element-by-id></get-element-by-id>
                <cn>"dv2"</cn>
            </apply>
            <apply>
              <qquote></qquote>
              <div>
                <table border="1" height="40px" width="300px">
                 <tr height="40px">
                  <td width="40%">
                   <label width="50px">
                    <unquote>
                     <apply>
                      <node-value></node-value>
                      <apply>
                       <get-element-by-id></get-element-by-id>
                       <cn>"txt1"</cn>
                      </apply>
                     </apply>
                    </unquote>
                   </label></td>
                  <td width="60%">
                    <unquote>
                      <apply>
                       <create-element></create-element>
                       <apply>
                        <node-value></node-value>
                        <apply>
                         <get-element-by-id></get-element-by-id>
```

```
                                        <cn>"txt2"</cn>
                                      </apply>
                                    </apply>
                                  </apply>
                                </unquote>
                              </td>
                            </tr>
                          </table>
                        </div>
                      </apply>
                    </apply>
                  </apply>
                </apply>
  </unquote>
  <unquote>
    <apply>
      <add-event-listener></add-event-listener>
      <apply>
        <get-element-by-id></get-element-by-id>
        <cn>"btn1"</cn>
      </apply>
      <cn>"click"</cn>
      <ci>clickMethod</ci>
    </apply>
  </unquote>
</body>
```

Listing 5.11: Dynamic form generation example

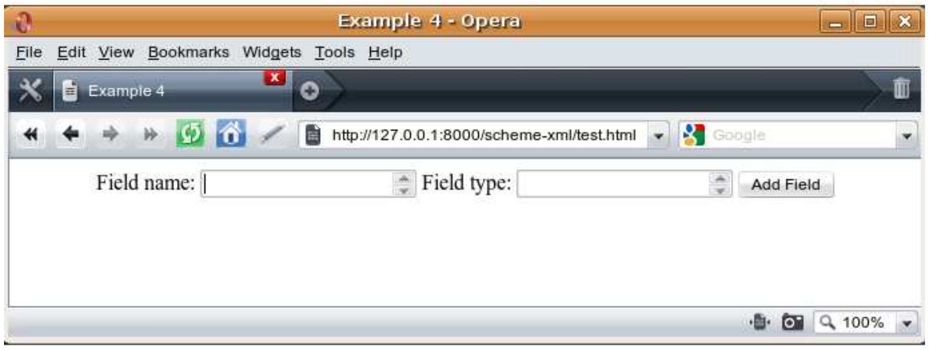
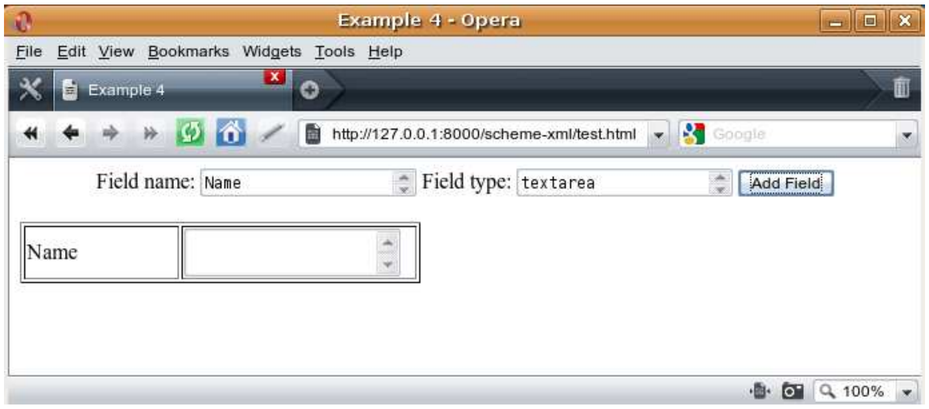

Example 4 - Opera
File Edit View Bookmarks Widgets Tools Help
Example 4
http://127.0.0.1:8000/scheme-xml/test.html
Google
Field name:
Field type:
Add Field
100%

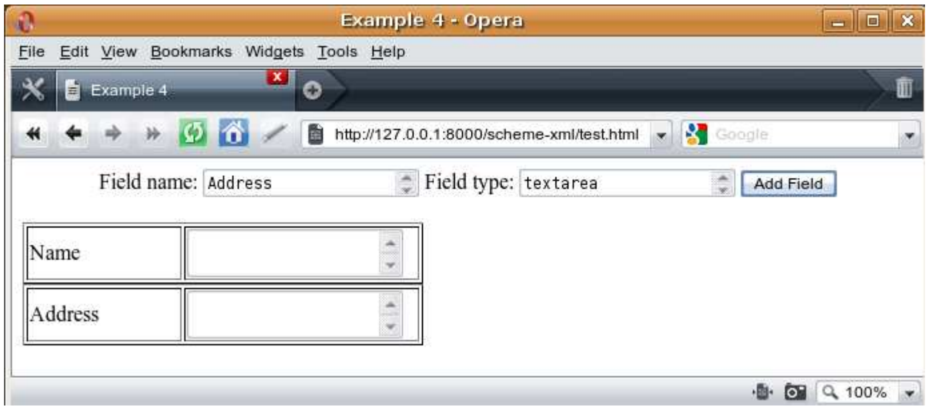
Figure 5.4: Example 4: Form : on load


Example 4 - Opera
File Edit View Bookmarks Widgets Tools Help
Example 4
http://127.0.0.1:8000/scheme-xml/test.html
Google
Add Field
100%

Field name: Name Field type: textarea

Name	

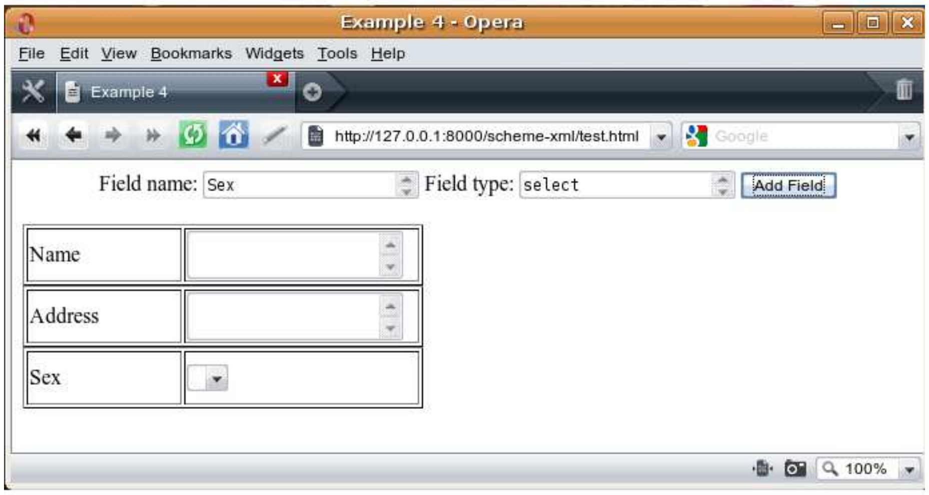
Figure 5.5: Example 4: Form : on 1st click


Example 4 - Opera
File Edit View Bookmarks Widgets Tools Help
Example 4
http://127.0.0.1:8000/scheme-xml/test.html
Google
Add Field
100%

Field name: Address Field type: textarea

Name	
Address	

Figure 5.6: Example 4: Form : on 2nd click


Example 4 - Opera
File Edit View Bookmarks Widgets Tools Help
Example 4
http://127.0.0.1:8000/scheme-xml/test.html
Google
Add Field
100%

Field name: Sex Field type: select

Name	
Address	
Sex	

Figure 5.7: Example 4: Form : on 3rd click

Chapter 6

Exploring homoiconicity in a web setting

In the previous chapters, we have discussed the effects of introducing homoiconicity in the web development environment. Better metaprogramming capabilities, multi-threading, writing domain specific languages and uniformity in representation of code, data, access and persistence are some of the key advantages highlighted in the thesis. In this chapter, we will explore the feasibility of exploiting these benefits in the web development environment. There are certain areas of web development where we have not been able to introduce homoiconicity as yet. However, we would also discuss the possible development scenarios that could emerge from introducing the concept of homoiconicity in the development environment.

Web applications are no longer dependent on single infrastructure units where they are hosted or where they retrieve data from. The latest trend on the web development block is developing applications in a cloud of resources. Systems like Memcached, Amazon Web Services and Google App Engine provide various services and resources in a cloud that can be used by a developer for application development. Memcached

[3] provides distributed memory caches and an API that allows the developer to use this memory cache. To the developer, it appears that he is manipulating or interacting with only one cache instead of multiple caches. Amazon Web Services [1] provides distributed virtual machines, storage and computation with a 'pay per use' policy. The Google App Engine provides a platform for application development in a cloud and allows developers to run their web applications on Google's infrastructure.

It isn't just the hardware and resources that are becoming distributed, the software and applications are also becoming distributed. The two commonly used distributed applications architecture are the two-tier and three-tier architectures. Within two-tier architecture itself, there could be fat-client or thin-client architecture. In fat-client architecture, most of the presentation logic, business logic and application logic resides on the client whereas database operations would occur on a data server. Thin-client applications have presentation logic on the client only, rest of the logic resides on the data-server. For more complex applications, that need a lot of processing, three-tier architecture is suitable. Having an application server, in addition to the client and data server, makes processing faster and more secure. Business logic of an application can reside on the application server and accordingly decisions can be made whether interaction with the data server is important or not.

Given the distributed nature of the web these days, it is essential to be able to interact with various applications or data stored on remote servers with ease, irrespective of which technology the application might be based on. We discuss some of the possibilities and existing technologies in the following subsections.

6.1 Remote Procedure Calls

Over the years the technologies used for web application development have evolved. In todays development environment, a multitude of web technologies like PHP, Java, JavaScript, XML, XHTML are used in creating an application. To exploit the distributed nature of the web and the infrastructure offered by various service providers, it is essential for applications written in any language or developed on any platform to be able to interact with each other.

This task has been made possible by the many Remote Procedure Call (RPC) protocols made available in the web development environment. XML-RPC [4] built on XML and the HTTP protocol, is one such protocol that allows developers to connect programs running on different computers. Java programs can interact with PHP scripts or Perl scripts or ASP.NET applications. Another protocol that is popular for RPCs in the web is SOAP. However, XML based protocols can be quite verbose.

The REST [11] architecture/protocol based on XML and HTTP has enabled utilization of the scattered resources on the web. Since the REST architecture stresses on a stateless server, distributed caching or utilizing other distributed resources becomes feasible. Nowadays, the web applications tend to be immersive in nature and need to maintain state over a period of time. The revolutionary concept AJAX, that uses JavaScript and XMLHttpRequest object, helps realize the REST architecture. AJAX enables maintaining state on the client and then communicating with the server accordingly. The server can thus remain state-less.

Allowing RPCs to be made in the same manner as access or persistence in the language we have developed, further enhances the metaprogramming capabitlities but at the same time provides a uniform environment for making RPCs. Data or Code residing in remote machines can be accessed and manipulated uniformly. By using the xmlns attribute or another attribute to specify the location of the remote procedure, the procedure call can be made in the same way as a normal procedure call would be made. The underlying use of AJAX can be abstracted away. This ensures conformity with the REST architecture and thus enables exploiting of the distributed resources on the web.

6.2 Accessing Scheme libraries

The Scheme programming language has a robust portable library, SLIB. This library provides packages that can be used with any implementation of Scheme. It has some highly useful packages that can be used and exploited in the web development environment. For example, the textual conversion package which has libraries for parsing and manipulating HTML and XML, the database package, the regular expression library and various existing libraries for sorting and searching are already proved to be efficient and can be used to create faster applications or algorithms for the web.

One can consider accessing these Scheme libraries, which have been parsed and converted to their XML representation, stored in some cloud remotely from an XHTML application containing XML-like Scheme code. Calling procedures from these libraries can be done in the same manner as accessing remotely stored data through RPCs.

6.3 Server-side evaluation

One of the reasons why server-side web programming became so popular was because it enabled developing dynamic web applications, which was not possible previously with just HTML and JavaScript. Scripting languages like PHP provided an effective way to manipulate and generate HTML. One of the advantages of having server-side programming is that these programming languages are usually compiled and thus provide more robustness. Having PHP embedded in HTML has given a lot of control to the developer for dynamic manipulation of the code.

Like PHP, one can visualize having Scheme code, represented as XML, running on the server. Different tags can be specified in a quoted XHTML page to describe whether the embedded XML-like Scheme code is to be evaluated at the server or the client. The server code would output evaluated, quoted and unquoted XHTML that will then be evaluated at the client browser. This approach in general promotes metaprogramming capabilities.

6.4 Summary

As can be seen from this chapter, there exist innumerable arenas for exploiting the concept of homoiconicity in the web setting, many of which are still unexplored. We can visualize the existence of a powerful web development system based on homoiconicity which can proliferate into and enrich many more aspects of web development.

Conclusion

The Web runs on a widely spread network of systems and is highly distributed in nature. Over the years, many technologies have been innovated to exploit this distributed nature of the Web. However, the emphasis these days, appears to be shifting towards hiding the distributed nature of the web. At the implementation level, the distributed resources and the applications can be exploited efficiently but, a centralized and coherent front is presented to the user. Applications like Google Wave, a communication tool that consolidates features from e-mail, instant messaging, blogging, multimedia management, document sharing and wiki seem to be following this philosophy. Another example is the Google Web Toolkit. GWT is a toolkit that allows developers to develop web applications in a single language, Java. The server-side code and the client-side code, both are written in Java and the developer need not worry about ,or have knowledge of JavaScript, XMLHttpRequest or browser compatibility. Nonetheless, programming tools like the GWT, though competent, have not been able to exploit some of the basic concepts in the world of programming and are found lacking in their programming capabilities.

We have presented in this thesis, a uniform web development environment where code, data, access, persistence, and transformation, all is represented in the same manner. However, the aim is not just to provide a clean and coherent programming language, but more importantly, to exploit the property of homoiconicty that is reflected in this environment. Such an environment induces strong metaprogramming capabilities in the language and can further empower web programming. One can now generate correct and well-formed web pages, create code dynamically and write domain specific languages for particular tasks or type of applications.

A simple programming language, like the one presented in this thesis, that provides a minimal set of constructs and concepts that form the basis of the programming world, is more powerful, easier to master and gives more freedom to a programmer, than all the languages that provide a vast array of functionalities and concepts.

The concept of code and data duality, which we have introduced in this thesis, can be extended to other domains of web programming like remote procedure calls, distributed application evaluation etc. Thus, we can envisage a platform for web programming where all areas of web development are represented in the same manner, making web development much easier. It would also be free from confusing and complex concepts, and still provide a powerful programming model to work with.

Bibliography

[1] Amazon elastic compute cloud. `http://aws.amazon.com/ec2/`. (valid on June 30, 2010).

[2] Definition of homoiconic. `http://c2.com/cgi/wiki?DefinitionOfHomoiconic`. (valid on June 30, 2010).

[3] Memcached documentation. `http://code.google.com/p/memcached/wiki/Start`. (valid on June 30, 2010).

[4] *Programming Web Services with XML-RPC*. O'Reilly Media, 2001.

[5] Alan Bawden. Quasiquotation in lisp. In *Partial Evaluation and Semantic-Based Program Manipulation*, pages 4–12, 1999.

[6] T. Bray, J. Paoli, E. Maler, F. Yergeau, and C. Sperberg-McQueen. Extensible markup language (xml) 1.0 (fifth edition). `http://www.w3.org/TR/2008/REC-xml-20081126`. (valid on June 30, 2010).

[7] William E. Bryd. Web programming with continuations. `www.double.co.nz/pdf/continuations.pdf`. (valid on June 30, 2010).

[8] James Clark. XSL transformations (XSLT) version 1.0. `http://www.w3.org/TR/xslt`. (valid on June 30, 2010).

[9] The World Wide Web Consortium. Document Object Model (DOM). `http://www.w3.org/DOM`. (valid on June 30, 2010).

[10] Douglas Crockford. A survey of the Javascript Programming Language. `http://javascript.crockford.com/survey.html`. (valid on June 30, 2010).

[11] R. T. Fielding and R. N. Taylor. Principled design of the modern web architecture. In *ACM Trans. Internet Technol.*, pages 115–150, 2002.

[12] Jesse James Garrett. Ajax: A new approach to web applications. `http://www.adaptivepath.com/ideas/essays/archives/000385.php`. (valid on June 30, 2010).

[13] Google. Google web toolkit. `http://code.google.com/webtoolkit`. (valid on June 30, 2010).

[14] Adobe Systems Incorporated. ActionScript 3.0 Language Specification. `http://livedocs.adobe.com/specs/actionscript/3/wwhelp/wwhimpl/js/html/wwhelp.htm?href=000_titlepage.html`. (valid on June 30, 2010).

[15] ECMA International. ECMAScript language specification. `http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-262.pdf`. (valid on June 30, 2010).

[16] Richard Kelsey, William Clinger, and Jonathan R. (editors). Revised (5) report on the algorithmic language scheme. *ACM SIGPLAN Notices*, 33:26–76, 1998.

[17] Richard Kelsey and Jonathan Ress. Simple macros expander. `ftp://ftp.cs.indiana.edu/pub/scheme-repository/code/lang/simple-macros.tar.gz`. (valid on June 30, 2010).

[18] MSDN. JScript (Windows Script Technologies). `http://msdn.microsoft.com/en-us/library/hbxc2t98(v=VS.85).aspx`. (valid on June 30, 2010).

[19] Philip Olson. Php manual. `http://www.php.net/manual/en`. (valid on June 30, 2010).

[20] D. Ragget, A. Hors, and I. Jacobs. HTML 4.0 specification. `http://www.w3.org/TR/REC-html40-971218/`. (valid on June 30, 2010).

[21] S. M. Watt, Yuzhen Xie, and L. Padovani. A lisp subset based on MathML. In *International Conference on MathML and Math on the Web*, 2002.

Appendix A

Language Specifications

This appendix provides an informal BNF grammar for our language. Listing A.1 displays this grammar. Some of the notations used in this grammar are described below.

1. Non-terminals in the grammar are represented by upper case letters.
2. * after a grammatical phrase indicates zero or more occurrences of the phrase.
3. + indicates that atleast one occurrence of the phrase is required.
4. The empty tags in the grammar like <qquote />, <define /> are a shorter representation of tags like <qquote></qquote> and <define></define>.
5. The non-terminal EMPTY means that it is an empty production. No expansion is required.

```
WEBPAGE ::= HTML-ELEMENTS*

HTML_ELEMENTS ::= <ELEMENT-TAG>
```

```
                       HTML-ELEMENTS*
                  </ELEMENT-TAG>
                 | UNQUOTE-EXPRESSION

ELEMENT-TAG ::= any valid HTML tag except
                       for html,head and body

UNQUOTE-EXPRESSION ::= <unquote>
                         [DEFINITION | EXPRESSION]
                       </unquote>

DEFINITION ::= VAR-DEF | PROC-DEF
VAR-DEF ::= <apply>
               <define /> VARIABLE EXPRESSION
            </apply>
PROC-DEF ::= <apply>
               <apply> VARIABLE DEF-FORMALS </apply>
               BODY
             </apply>
DEF-FORMALS ::= VARIABLE* | VARIABLE* <dot /> VARIABLE


EXPRESSION ::= VARIABLE | LITERAL
               | PROCEDURE-CALL
               | LAMBDA-EXPRESSION
               | CONDITIONAL
               | ASSIGNMENT
               | CONTINUATION
               | DERIVED-EXPRESSION
               | DOM-MANIPULATION-EXPR

VARIABLE ::= any IDENTIFIER that isn't
                       also a SYNTACTIC-KEYWORD

IDENTIFIER := <ci> INITIAL SUBSEQUENT* </ci>
              | <ci> PECULIAR-IDENTIFIER </ci>

INITIAL ::= LETTER | SPECIAL

PECULIAR-IDENTIFIER ::= + | - | * | = | > | < | >= | ...
LETTER ::= a|b|c|d|...|z
SUBSEQUENT ::= INITIAL | DIGIT
DIGIT ::= 0|1|2|...|8|9
SPECIAL ::= ! | $ | % | & | * | / | : | ? | ^ | _ | ~
SYNTACTIC-KEYWORD ::= EXPRESSION-KEYWORD
                      | else | define
                      | unquote
```

```
EXPRESSION-KEYWORD ::= quote | lambda | if
                       | set | cond | and | or | let | begin
                       | let-s | quasiquote

LITERAL ::= QUOTATION | SELF-EVALUATING

SELF-EVALUATING ::= BOOLEAN | NUMBER | STRING
BOOLEAN ::= <true/>|<false/>
NUMBER ::= <cn> [DECIMAL | OCTAL | BINARY | HEXADECIMAL] </cn>
DECIMAL ::= [EMPTY|#d] DIGIT+ [EMPTY|. DIGIT*]
OCTAL ::= #o [0|1|2|3|4|5|6|7|8]+
BINARY ::= #b [0|1]+
HEXADECIMAL ::= #x [DIGIT|a|b|c|d|e|f]+
STRING ::= <cn>" STRING-ELEMENT "</cn>
STRING-ELEMENT ::= any character

QUOTATION ::= <apply> <quote /> DATUM </apply>
DATUM ::= SIMPLE-DATUM | LIST | HTML-ELEMENTS
SIMPLE-DATUM ::= BOOLEAN | NUMBER | STRING | SYMBOL
SYMBOL ::= IDENTIFIER
LIST ::= <apply> DATUM* </apply>

(In QUOTATION, the non-terminal HTML-ELEMENTS can't evaluate
 to UNQUOTE-EXPRESSION)

PROCEDURE-CALL ::= <apply> OPERATOR OPERAND* </apply>
OPERATOR ::= EXPRESSION
OPERAND ::= EXPRESSION

LAMBDA-EXPRESSION ::= <apply> <lambda /> FORMALS BODY </apply>
FORMALS ::= <apply> VARIABLE* </apply> | VARIABLE
            | <apply> VARIABLE+ <dot /> VARIABLE </apply>
BODY ::= DEFINITION* SEQUENCE
SEQUENCE ::= EXPRESSION+

CONDITIONAL ::= <apply>
                   <if /> TEST CONSEQUENT ALTERNATE
                </apply>
TEST ::= EXPRESSION
CONSEQUENT ::= EXPRESSION
ALTERNATE ::= EXPRESSION | EMPTY

ASSIGNMENT ::= <apply>
                 <set /> VARIABLE EXPRESSION
               </apply>

CONTINUATION ::= <apply>
```

```
                   <callcc /> [LAMBDA-EXPRESSION | VARIABLE]
                 </apply>

(in continuation, the FORMALS of LAMBDA-EXPRESSION must be
<apply> VARIABLE </apply> only and in case of VARIABLE, the
variable must evaluate to a lambda expression with 1 argument)

DERIVED-EXPRESSION ::= COND | AND | OR | LET | LET-S | BEGIN
                       | QUASIQUOTATION

COND ::= <apply>
          <cond />
          CLAUSE+ | [CLAUSE* <apply><else/> SEQUENCE </apply>]
         </apply>
CLAUSE ::= <apply>
               [TEST SEQUENCE] | TEST
           </apply>

LET ::= <apply>
          <let />
           [BINDING-SPEC BODY | VARIABLE BINDING-SPEC BODY]
        </apply>
BINDING-SPEC ::= <apply> BINDING* </apply>
BINDING ::= <apply> VARIABLE EXPRESSION </apply>

LET-S ::= <apply>
           <let />
             BINDING-SPEC BODY
          </apply>

AND ::= <apply> <and /> TEST* </apply>
OR ::= <apply> <and /> TEST* </apply>

BEGIN ::= <apply> <begin /> SEQUENCE </apply>

QUASIQUOTATION ::= <apply>
                      <qquote /> DATUM
                   </apply>

DOM-MANIPULATION-EXPR ::= NODE-VALUE | CREATE-ELEMENT
                          | GET-ELEMENT-BY-ID
                          | ADD-EVENT-LISTENER
                          | APPEND-CHILD | INNER-HTML

NODE-VALUE ::= <apply> <node-value /> EXPRESSION </apply>

CREATE-ELEMENT ::= <apply>
```

```
                        <create-element /> EXPRESSION
                    </apply>

GET-ELEMENT-BY-ID ::= <apply>
                        <get-element-by-id /> STRING
                      </apply>

ADD-EVENT-LISTENER ::= <apply>
                         <add-event-listener />
                         EXPRESSION STRING
                         [VARIABLE | LAMBDA EXPRESSION]
                       </apply>

APPEND-CHILD ::= <apply>
                  <append-child /> EXPRESSION EXPRESSION
                 </apply>

INNER-HTML ::= <apply> <inner-html /> EXPRESSION </apply>
```

Listing A.1: BNF Grammar for our language

This grammar covers most of the syntax of our language. However, constructs like `car, cdr, cons, vector, list, apply, assq, isnull` and related syntax is not described in the grammar. The purpose of presenting this grammar is to give a basic idea of the syntax of the language. Other constructs and syntax can be built on it.

Curriculum Vitae

Name	Rachita Mohan
Post-secondary Education	The University of Western Ontario London, Ontario, Canada M. Sc. September 2008 - April 2010
	Galgotia's College of Engg. and Tech. (GCET) Greater Noida, India B. Tech. August 2003 - May 2007
Related Work Experience	Teaching Assistant The University of Western Ontario September 2008 - December 2009
	Research Assistant Ontario Research Centre for Computer Algebra September 2008 - April 2010
	Software Engineer HCL Technologies Ltd., India June 2007 - July 2008