University of Edinburgh
Division of Informatics

LODE: An Online IDE for Links in Links

4th Year Project Report
Artificial Intelligence and Computer Science

Carl Andersson
s0452925@sms.ed.ac.uk

March 2, 2008

Abstract: With the rapid development of the Internet has come a large number of differing programming techniques for creating interactive web pages. The "Web 2.0" phenomenon is comprised of websites making use of these techniques to allow complex applications to be deployed. This project is concerned with the development of one such application, created in the functional web programming language Links. The application, an online development environment for programming in Links, makes use of the Links language to create an interactive method of coding in Links on the go.

Contents

1. Introduction

The ubiquity of the internet has led to a vast increase in technologies specifically designed to make interactivity online easier and more powerful. Gone are the days of the "Information Super-Highway" when the web consisted of collections of static web pages, each no more exciting than the next, with the most impressive features of a website being animated GIFs and the occasional use of the (now thankfully deprecated) `<blink>`-tag. Today's web consists of a vast amount of dynamically generated content, and the continued development and improvement of the Document Object Model (*DOM*, an object model for representing markup languages, see [14, 32]) and JavaScript has led to improved abilities of websites to respond to user events and input, e.g. by manipulating their own contents. With the abilities has come web-based applications successfully mimicking the behaviour of what has traditionally been seen as stand-alone software, for example Google's *Google Mail* and *Google Docs*.

The distinctions between online and offline applications are beginning to be rubbed out thanks to the increased user demands on their web-experience, and the introduction of development toolkits especially aimed at making highly interactive and feature-rich online applications, for example Microsoft's *Silverlight*, Adobe's *Flex* and Google's *Web Toolkit (GWT)*, is likely to drive both the capabilities of web applications and their penetration among businesses forward. As these technologies mature it is probable that migration from standalone applications to web-based equivalents, so-called *rich Internet applications* or *RIAs* (defined by Macromedia, the originators of Flash and Flex, as "[...] combining the best user interface functionality of desktop software applications with the broad reach and low-cost deployment of Web applications and the best of interactive, multimedia communication."[12]), will increase. This dissertation details the design and implementation of one such application, LODE (Links Online Development Environment), a web-based integrated development environment (*IDE*). Unlike the web applications available today, LODE is developed in the Links programming language (see section 2.2), a functional programming language currently under development at the University of Edinburgh.

1.1 Overview

The rest of the current chapter details the reasoning behind why a tool such as LODE would be useful, and provides a summary of the results. Chapter 2 is a run-through of the technologies used in the development of the IDE, including high-level descriptions of the languages used. Chapters A and 3 present and

discuss the approach taken to designing and implementing the IDE, including conceptual issues and various difficulties encountered during the implementation, while chapter 4 explores what has been accomplished and how that relates to the project's goals, in addition to providing details of project aspects that would be suitable for further work. The final chapter, 5, reflects on the project as a whole.

1.2 Motivation

The traditional domain of development tools is firmly located in that of stand-alone applications, as many of them are heavily dependent on the ability to compile, interpret and analyse the code presented to them. Add to this the feature-rich IDEs already available in the stand-alone domain (e.g. *Eclipse*[6] by IBM et al.,and Sun's open-source Java-IDE *NetBeans*[22]) and it is hardly surprising that there has been a noticeable lack of activity in developing web-based equivalents. However, the current state of the Internet and its technologies eases the migration of applications (or at least some of their functionality) to the online domain, and IDE applications are no different. One can easily imagine a number of situations where the access to the code and compiler of a particular project being made available to developers through a shared interface, regardless of the capabilities (at least to some extent) of their machines, would come in handy. One of the main motivations of the project is to attempt to remove the dependency of projects on the physical locations and available hard- and software of their developers.

A second motivation lies closer to the domain of the Links project itself. As yet, no major applications have been developed using the language, meaning that the project has yet to be tested in a real-world development scenario. Any lessons learnt during the development of this project's product regarding how a developer experiences Links development and any weaknesses or bugs found in the implementation of the language itself could therefore be useful to the Links team, providing them with valuable user-feedback which in turn could lead to a better end-product. Furthermore, tying in to what was stated above, having a Links editor available online that allows users to run Links applications remotely, regardless of what other functionality it employs, removes the need for each user to install Links on a webserver available to them, lowering the barrier of entry.

1.3 Summary of Results

The application developed for this report is successful in what it set out to do: the online development tool allows the creation, testing and deployment of Links

applications from any location through the Firefox 3 (beta 2+) web browser, and comes with syntactic highlighting and type information to aid programmers. Users are able to work with their own files through a clean interface laid out in a fashion similar to *Eclipse*.

The experience of developing LODE has highlighted a number of issues with Links, mainly due to a lack of support for a number of standard programming functions which at times requires the Links developer to complement the language, but also in terms of efficiency.

The report concludes that despite the rather bare-boned specification, the project has produced a useful programming tool with great scope for further work, and that creating large applications in Links, while sometimes problematic and conceptually tricky, is very much possible.

1.4 Related Work

As mentioned previously, there have been no major applications developed using Links, and as a result there is no published related work in this area. There are, however, a number of web-based development tools available, with varying purposes and implementations, from which to seek inspiration.

CodeIDE[28] is aimed at teaching programming languages, and features code editing, syntax highlighting and code execution for a variety of languages, including JavaScript, Prolog and SQL, as well as user accounts with personal tasks and files. The interface used is slightly non-intuitive and the implementation seems buggy (many of the examples don't appear to work at time of writing), but some of the features are interesting and useful.

Tide4Javascript[27] is a JavaScript-based IDE for code-example in JavaScript. The interface is intuitive and similar to those of stand-alone IDEs, with the exception of code-editing which is done in a separate window, and features include a hierarchical project-browser, syntax-highlighting on the fly, and debugging using breakpoints and step-throughs.

rainbow9[15] is an open-source online IDE for the development of web projects, and allows for the editing of HTML, XML, JavaScript, CSS, XAML and SVG files, and includes a JavaScript "shell" for code-testing. Users are able to sign up and create/edit their own projects, as well as preview and publish them. However, no syntax highlighting is provided, and the editor bears little resemblance to stand-alone IDEs in terms of interface.

WWWorkspace[26] is a fully-fledged online IDE for Java written in JavaScript, and includes features such as syntax highlighting, a familiar user interface,

and the ability to create, develop and execute projects remotely. The aims of WWWorkspace mirror to some extent the goals of LODE in that development is meant to be flexible, portable and easy to use, but the implementation is vastly different as although the front-end is pure JavaScript, many of the features of WWWorkspace plug into an Eclipse-based webserver at the back-end.

CodePress[11] was created for the web-based IDE *ECCO* and as such contains features for undoing and redoing, and cross-browser support. Regular expressions are used in order to highlight code, making extensible language support both quick and easy, at the expense of expressive power.

CodeMirror[10] is an advanced take on interactive syntax highlighting built entirely in JavaScript, using a full lexer and parser (with continuations keeping "state") rather than regular expressions to allow for more accurately highlighted code. Originally rather bare-boned, CodeMirror was intended to be used as part of an online book on teaching JavaScript, but has since evolved into a more powerful interactive highlighter, including useful features such as undo/redo and search/replace.

2. Background and Technology

2.1 Web Development

The face of the web has changed a great deal since the early days, with an ever-growing audience of Internet users pushing technology forward at an unprecedented pace and in unexpected ways, making the web experience much more interactive than was perhaps anticipated. With these advances has, quite naturally, come a number of changes in how web pages are developed, with a large amount of programming languages and tools being designed specifically for web development use. This section describes some of the aspects of modern web development, with focus on the technologies that are relevant to the LODE project. For reasons of brevity and clarity, this section is kept relatively high-level, but pointers to relevant further reading material are made where appropriate.

2.1.1 Client vs. Server-side Programming

One of the most fundamental aspects of web development is that of the separation of application logic and display. In order for web applications to be as lightweight and responsive as possible while at the same time providing extensive capabilities in a state-less environment (*HTTP*, the Hypertext Transfer Protocol, does not allow websites to keep state — instead, updates are achieved by passing `POST` or `GET` messages from the client to the server), the functionality is split into two sections, a *client* and a *server* side. The client side is what is visible to the user and generally contains, in addition to the markup (see 2.1.1.1), the functions that interact with the user and alter the appearance of the website, e.g. the DOM manipulation functions, as well as utility functions, e.g. for sanity-checking user input. The main language for client-side scripting is *Javascript*, an interpreted language with facilities influenced by both functional and imperative programming paradigms. Data processing and more advanced functionality, e.g. file I/O or database operations, are performed on the server side of the application, and are therefore independent of the capabilities of the users. This in turn means that it is up to the developer to decide what language to use, and that a vast number of programming languages can be used.

2.1.1.1 Hypertext Markup

In order to display more than just barebone text in a document, a method of providing more information about the components is needed. One way of attaching this information to document entities is by using a *markup language*, which allows pieces of text or other elements to be tagged with information guiding the rendering of the document. For the web, the main markup languages used are Hypertext Markup Language (HTML) and its successor extensible HTML (XHTML), both of which are used to allow non-textual items such as images, tables and lists to appear in webpages. (X)HTML and related extensible markup languages (XML) are organised in a tree-like fashion, making it easy to navigate the tree programmatically using the nodes' parent-child relationships. The tree-like structure is also ideal for DOM scripting, i.e. manipulating the various nodes of the tree in order to change their ordering or particular attributes. DOM manipulation is commonly used to allow a webpage to react to user actions, for example by displaying a menu when a user clicks a particular node in the tree.

2.1.1.2 Cascading Style Sheets

Web markup languages are useful for describing the structure and contents of a page, but are somewhat limited in their ability to accurately describe its layout. To counter this, extra styling information is usually attached to a markup document, giving more fine-grained control over the look-and-feel of both nodes in the page and the page as a whole. The information is described using *Cascading Style Sheets*, or *CSS*, which allows styles to be assigned in a hierarchical manner (meaning that the styles of a parent *cascade* to its children) to individual nodes, groups of nodes (known as classes), and nodes of a specific type. Styles can be used to alter almost all aspects of a node, including its position, colour, background, mouse cursor on hover, and font-styles.

More information on CSS can be found at [33].

2.1.1.3 JavaScript

As mentioned, dealing with user interaction and DOM manipulation in a web page is normally achieved by using client-side capabilities using JavaScript, a highly flexible interpreted (as opposed to compiled) language. Designed at Netscape in the mid-90s, its name heralds from Sun's Java language in an attempt to become more recognisable, regardless of the fact that the languages have very little in common. Despite having its foundations standardised as ECMAScript[7], JavaScript has seen a number of different implementations with varying amounts of interoperability thanks to the "browser wars"[14] of the 1990s. The fact that

```
// Attach an anonymous function to aFun.
var aFun = function (xiVal) {
  return {
    val: xiVal,
    incr: function () {
      // Return the original value before incrementing.
      return this.val++;
    }
  };
};

// Call an object's increment function.
function increment(xiObj) {
  xiObj.incr();
}
```

Figure 2.1: JavaScript function example.

different browser support different JavaScript functions (or in some cases, the same functions but with different meaning) has resulted in the language having a bad reputation. The reason for this is mainly that it forces developers to implement multiple versions of the same application if cross-browser compatibility is to be achieved. However, browser vendors have recently begun working towards reducing the gap between different browsers in order to achieve a more uniform JavaScript platform[14]. In addition, a number of JavaScript libraries (e.g. Yahoo's *YUI*) have been developed by third parties in order to make browser-independent web applications easier to create.

The syntax of JavaScript is similar to that of Java, but beyond that there are few similarities. JavaScript is dynamic and weakly typed, meaning that no explicit type declarations are made during programming, and, with the exception of its core types, everything in JavaScript is an associative array (called *objects*). This includes functions, so that just like in functional programming[1], functions are first-class citizens and can be treated just like any other object: they can be passed around; assigned to variables; stored in arrays etc. Functions being objects also means that functions can have properties attached to them, including other functions. Other features of JavaScript include support for closures; inner and anonymous functions; and recursion.

Figure 2.1 shows a simple example of JavaScript, displaying some of the important aspects of the language. In the example, an anonymous function carrying a variable (`val`) and a function (`incr`) as properties is assigned to a variable. This variable can then be passed to the `increment` function, which calls the function attached.

More information about the features and syntax of JavaScript can be found in [14, 18, 19].

2.1.1.4 Server-side Scripting Languages

Traditional web-servers are on their own capable of little more than serving static pages upon request — anything more advanced, for example performing a database query, requires some other entity with greater abilities to be invoked. These entities are known as server-side scripts, and are used to provide customised, dynamically generated web pages based on a user's request, ranging from handling data passed to it from a form to serving encrypted webpages. Written as external applications, such scripts were originally invoked using the Common Gateway Interface (*CGI*, a protocol purposely created for interfacing with external applications from a web server, see [31, 34]), but can now normally be called directly from the web server itself.

Although in theory any programming language can be used for server-side processing for a web application, a number of languages and techniques have been developed particularly with web programming in mind. Examples of this include Java Server Pages (*JSP*), *PHP*, and Active Server Pages (*ASP*). More information about server-side scripting and the various languages can be found at [23].

2.1.2 Database Programming

Many web applications require data to be stored on the server in order to provide the desired services, e.g. user-data for log-in functionality to work. The standard way to provide this information is through the use of a server-side *database* which stores the data in some data structure and with which the application can interact by means of data *queries* written in a query language, for example the *Structured Query Language* (*SQL*). Databases are usually provided by a database management system (*DBMS*) which specifies the organisation of the data and what data structure and query language to use, as well as some transaction mechanism to ensure data integrity. Due to the nature of the data storage, the latter is one of the most important selling points of DBMSs.

From a web development point of view, DBMSs are useful as they provide a straight-forward way of accessing particular pieces of information from a (potentially very) large data set. Furthermore, depending on the DBMS used, they give the application a host of extra functionality, e.g. added security mechanisms and support for rollbacks, as well as more advanced ways of manipulating and separating data, e.g. through the use of triggers (event handlers), stored procedures and views.

Examples of database management systems include MySQL (2.1.2.1), Oracle, and Microsoft Access; more information on basic DBMS can be found at [2].

2.1.2.1 MySQL

Due to the high integrity and functionality requirements placed upon them, database management systems have traditionally been heavy-weight and expensive components, and have therefore been used mostly in industry. However, the growth of the Internet and personal websites in the 1990s led to a need for less specialised, simpler and — most of all — cheaper DBMSs. One such system is MySQL, an open-source DBMS created specifically for integration with websites and web applications. Initially a very simple database, the last few years has seen it add support for more advanced features (query caching, views, and triggers to name a few) and grow significantly in number of installations. MySQL claims to be "[...] the world's most popular open source database software."[20].

More information on MySQL can be found at [21].

2.2 Links

As previously mentioned, the language used for developing LODE is Links, a strict, typed, functional programming language for the web currently under development at the University of Edinburgh. This section will briefly go through the features of the language; a more thorough description can be found in [4, 35]. It should be noted that although at the time of writing the current official release of Links is 0.4 ("Corstorphine"), the version discussed and used for this project is taken directly from the development team's main code repository, and that some features of Links may not yet be publically available.

2.2.1 Functional Programming for the Web

Web development in general can seem like (and is) a rather daunting task. In order for any web application to be successful, its developers need to focus on making it well-designed and user-friendly, and must do so using a number of different languages: markup and styling for the front-end; a web scripting language for user-interactions; another scripting language for server interaction; and a query language for database interactions. Having to work with a number of different languages is not only a major roadblock in terms of ease of development, but it also increases the chance of *impedance mismatch*, a term borrowed from engineering and referring to the difficulties in the mapping of datatypes between

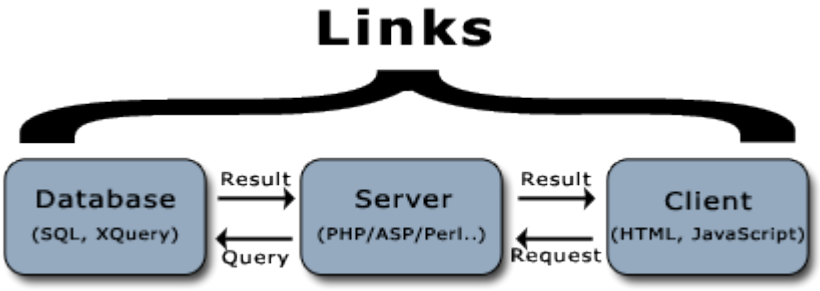


Figure 2.2: The three tiers of web programming unified by Links. (Adapted from figure 1 in [35])

system components using different data representations. How can one be certain that the data returned from a database query is of the expected type?

One of the major sources of impedance mismatch is database querying, a feature available with many programming languages (web-focused or otherwise), allowing database calls to be made from within an application through external interfaces and drivers. Historically, the interaction between the programming language compiler and the database engine has been poor, with languages lacking support for the notion of a query, choosing instead to embed them directly in strings. The results of the query may return the correct dataset, but it is unlikely to be achieved optimally as the compiler has no concept of the semantics of the query string, and is therefore unable to perform any language-specific optimisations. Furthermore, if the compiler does not know the query semantics, type checking will not be possible, thus increasing the chance of impedance mismatch between the two systems.

One of the goals of Links is to lessen the burden of development and erase the impedance mismatch by allowing the developer to use a single language for both front- and back-end scripting, as well as for database queries. During compilation, the Links compiler type-checks the code, and if need be translates relevant parts of it into JavaScript and SQL. The use of a single language means that the Links compiler is able to have complete knowledge of the semantics of the code, and is thereby able to ensure that even data returned from calls to external components, in this case the database and the client-side browser, are type safe. Figure 2.2 displays a diagram of the three *tiers* of web development, all of which are unified by Links.

In order to decrease the barrier of entry for developers wanting to migrate to Links, the syntax of the language is similar to that of JavaScript. Blocks of code are separated using curly braces ({}), and the standard control structures such as conditionals, case-statements and loops are included. Links also includes rudi-

```
# Define a table with two columns.
var sampleTable =
  table "sample" with (
    id : Int,
    text : String
  ) from (database "sampleDb");

# Insert a value into the table
insert sampleTable values [(id = 1, text = "hello")];

# Select a value
for (var lRow <-- sampleTable)
  where (lRow.text <> "")
    [lRow.text]
```

Figure 2.3: Links database example.

mentary regular expressions, as well as "standard" functional programming features, such as lists, variants, first-order functions, and structural pattern matching. For a more detailed run-through of the syntax, see [16].

Links is written in OCaml, a strict, typed, functional programming language which focuses on speed of execution and reliability. One of OCaml's main features is its attempt to include language features from both imperative and object-oriented programming paradigms. More information on OCaml can be found at [13, 29].

2.2.2 Links Features

Database Programming Links supports a subset of database operations for the *MySQL*[21], PostgreSQL[25] and SQLite[30] implementations of SQL. The operations permitted are `select`, `insert`, `update` and `delete`, and database access using these operations is achieved using list comprehensions, whereby the entries in the specified table are treated as members of a list and looped through accordingly. Figure 2.3 shows a code example which defines a table, inserts an element into that table, then selects all elements with non-empty text columns.

Concurrency The Links programming language contains resources for concurrent processing server-side, inspired by the Erlang[8] language in which processes are lightweight and share no common resources. Communication between processes is achieved through message passing. A Links programmer has the ability to spawn threads, and can pass messages to them from

```
# Thread to perform actions asynchronously.
fun actionThread()
{
  # Wait to be called.
  receive
  {
    case DoAction(xiAction) -> xiAction(); actionThread();
  }
}

# Spawn the thread.
var lHandler = spawn { actionThread() };

# Send it a message.
lHandler ! DoAction(someFunction);
```

Figure 2.4: Links concurrency example.

both client and server functions. Figure 2.4 shows a code example which creates a thread concurrent to the main process using the `spawn` primitive and passes that thread a function to call. Note the use of the `receive` primitive, which tells the thread to block until a message is received.

Form Abstraction Links includes a mechanism for abstracting over HTML forms called *formlets*, the purpose of which is to add a more unified method of form handling. Formlets allow an application developer to create reusable form components that generate a structured view of the data contained in their fields, and make use of Links' type-checking facilities to guarantee that the data generated by the form is acceptable to the function handling it. Formlet examples and more information can be found in [5, 16].

Web Interaction Links supports a wide range of facilities that allow applications to provide interactivity, including attaching code to respond to user events, e.g. when the user clicks a certain button or hovers over a particular element; and modifying the DOM, e.g. adding, deleting or modifying elements. XML is a native type in Links, allowing code to be interspersed with markup and making the modification of an active page using dynamically created XML nodes simpler than in standard JavaScript. Figures 2.5 and 2.6 show functions written in Links and JavaScript respectively for creating HTML on the fly. Notice that in the Links example, the `getDiv` function returns straight XML, whereas the JavaScript version creates each element, and their attributes, separately and orders them by explicit attachment. In both examples, the `l:onclick` and `onclick` attributes of the "myDiv"-element bind pieces of code to user click-events on that element.

```
sig getDiv : () -> Xml
fun getDiv()
{
  <div>
    <div id="myDiv" l:onclick="{toggleColour(event)}" val="off">
     Click here to change text colour.
    </div>
  </div>
}
```

Figure 2.5: Links Web interaction example.

```
  function getDiv()
  {
    // Create two divs.
    var lFirstNode = document.createElement("div");
    var lSndNode = document.createElement("div");
    lSndNode.setAttribute("id", "myDiv");
    lSndNode.setAttribute("onclick", "toggleColour(event)");
    lSndNode.setAttribute("val", "off");

    // Set the ordering and insert into the document.
    lSndNode.appendChild(
      document.createTextNode("Click here to change text colour.")
    );
    lFirstNode.appendChild(lSndNode);
    document.body.appendChild(lFirstNode);
  }
```

Figure 2.6: JavaScript interaction example.

Client/Server Programming Functions in Links may be annotated with the keywords `client` or `server`, which tells the compiler whether or not to convert the function into JavaScript. In the example in figure 2.5, the `getDiv` function is implicitly a client function, as it requires client-side functionality in order to work. If a `server`-annotated function needed to be called by the `getDiv` function, for example a function performing database lookups, the client would need to issue an asynchronous HTTP request to the server (often referred to as an Ajax-call, this allows users to request information from the server without having to wait for a response. See [9] for more information on asynchronous requests and how it fits in with modern web development.).

3. Design and Implementation

3.1 Overview

In many design and implementation descriptions of web applications, it makes sense to talk about client- vs. server-side components as they are not only physically but programmatically two separate parts of the application. Thanks to Links' tier-unifying properties this is not true for LODE — although the compilation of an application may yield both client and server code, the application may not have been developed with this separation in mind. As a result, I discuss in this chapter the design and implementation of LODE by looking at its functionality. Any mentions of client- or server-side code refers to explicit localisation declarations, i.e. when particular features of the client or server are needed for a function to be implemented.

3.2 Specification

The specification followed during the design and implementation of LODE was kept relatively basic and high-level, mainly due to the uncertainty of what the project would be able to achieve and what would be too difficult or time-consuming. The initial specification stated that "the goal of this project is to create a web-based development tool using Links that both aids newcomers to the Links programming language and increases the efficiency of those already knowledgeable in the language.", and to that end, a number of basic project requirements were drawn up based on personal experience with stand-alone development environments. The full specification can be found in appendix A.

Following the requirements, the following have been implemented for LODE:

A web-based user interface The interface separates the web page into a menu-bar with drop-down menus; a tree-structured project browser; and a code editor.

A syntax highlighting editor Code is coloured as the user types and undo/redo actions are allowed. Type information is displayed when the user hovers over a function or variable.

User accounts and sessions User are required to have registered and logged in before being able to work with LODE.

Projects and file-handling Users have the ability to create projects and files, which can only be accessed by them, as well as to load and save files.

Remote execution Links files can be executed from within LODE, with a distinction between the testing and deploying of projects.

3.3 Implementation Tools

The fact that Links is a virtually unknown language in the development community means that there is a distinct lack of support in terms of programming tools, so before diving in to the details of LODE and Links programming in general, a short digression on the development process is in order. With the Links source code comes a (slightly broken) syntax highlighting definition for Emacs (a class of advanced text editors readily available on most UNIX-based systems) which, together with the fact that there are similar modes for both OCaml and JavaScript and that running it remotely using an X-server is both simple and fast, makes it my editor of choice for Links programming.

Links' source code is kept under version control using *Subversion* (a freely available advanced version control system, see [3]), so in order to facilitate the modification of the Links source while at the same time keeping up to date with the latest additions by the official development team, a branch was created in the main Links code repository specifically for this project. I also used this branch to version-track the code for LODE.

When developing a web application it can be difficult to visualise how changes made to the markup and style of a page will affect its appearance, and although for smaller pages making minor changes iteratively in order to see the changes is straight-forward, it can be rather time-consuming for more complex pages with longer load times. A much simple approach would be to edit the layout and styles of an active page "live", and the Firefox extension *Firebug*[24] is a tool that allows developers to do just that — and more. With Firebug, an active web page can be inspected and developers can access and alter all aspects of the page's HTML and DOM, including node, window and document properties, attributes and styles. Firebug also monitors and times all HTTP requests sent by the page, and comes with a powerful JavaScript debugger allowing fine-grained monitoring of code execution. During the development of LODE, Firebug has been extremely useful for interface design and debugging, both of which would no doubt have been much more frustrating and time-consuming without it.

Finally, while not a tool per se, I developed and tested LODE using the latest version of Firefox 3 beta, which comes with an updated JavaScript library and appears to be more compatible with the code generated by the Links compiler

(see section 4.2). Using LODE with other browsers or older versions of Firefox is likely to result in unexpected behaviour.

3.4 Links Extensions

During the design and implementation of LODE, a number of extra functions were found to be needed and therefore implemented. Links extensions are achieved by either adding functions to the language library or by using the (undocumented) `alien` feature. Links' library is extended by modifying the source code and implementing additional functions in either OCaml or JavaScript depending on the nature of the function, and is appropriate if the desired functionality needs to reside on the server, e.g. if access to OCaml libraries is necessary, or if the function is needed for more than one application. If an extended function is to be implemented in JavaScript, it is bound to be client-side only, and must be declared as such in the library.

The `alien` feature of Links allows applications to call JavaScript functions directly, treating them as normal Links client-side calls. JavaScript functions are declared as `alien javascript` in the applications source code, given explicit types, and implemented in the file *extras.js*, which is included by default by Links. For example, Links does at current not retain a web page's `<title>` if the page includes client-side code, so in order to be able to set the page title, LODE includes the following declaration:

```
alien javascript setTitle :  (String) -> ()
```

This tells the Links compiler that whenever the `setTitle`-function is called, it refers to a function in the included *extras.js*, and is therefore allowed. Links' static type system also requires type information to be explicit for external functions, which is included on the right-hand side of the declaration. In this case, `setTitle` takes a string and returns "nothing", or `()` (*unit*).

Where appropriate, the sections giving implementation details refer to particular Links extensions necessary for LODE to function; a full list of library extensions can be found in B.1.

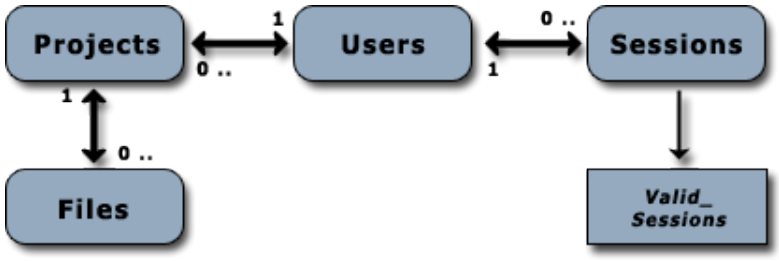


Figure 3.1: The organisation of LODE's database. `Valid_Sessions` is a view, the rest are tables.

3.5 Users and Projects

3.5.1 Database Organisation

Prior to discussing the ways in which users can work with Links code using LODE, it is sensible to detail how the users and their projects are organised. LODE makes use of a MySQL database to store information about the relationships between users, projects and files, as well as sessions. Figure 3.1 shows the organisation of this database, with the numbers on the arrows showing the nature of the relationships (e.g. users can have many projects, but any one project belongs to only one user). I decided to keep the database organisation as simple as possible, and attempted to link it into how we would normally perceive working with a project: a `user` has a number of `projects`, each of which consisting of a number of `files`, with work happening during work `sessions`. The `sessions` table contains a field detailing when the session was last active, i.e. when the user last loaded or saved a file, and the `valid_sessions` view is used to display all the sessions active in the last 20 minutes. This information is used for the session timeout feature detailed in the next section.

3.5.2 Logins and Session Handling

When working in LODE, users need to be in a session to be able to gain access to their personal projects and files, which is created when a user logs in by entering a correct username and password combination in the form on LODE's welcome page. The log-in form is a formlet, taking the data entered by the user and comparing it to what exists in the `users` table and returning the user's ID if a match is found. This ID, together with some randomly generated information, is then passed through a hash function (a deterministic one-way function generating fixed-length number from any input) and used as the session identifier. The identifier is stored in the user's browser by means of a *cookie*[17] and is used

when interacting with the database. The session ID is also used to create a temporary directory on the server in which to store files when running Links applications from LODE (see later sections).

This might seem like a rather lengthy explanation of what is in reality a rather standard method of handling user logins, but due to some lack of support in Links the process is in fact non-trivial. Hash functions, used for both identifier creation and secure storage of passwords, are not part of the standard library, meaning that one had to be created for the purposes of this project. I chose to implement an MD5-function as not only is it and its properties well-known, but the OCaml language libraries come with support for it, allowing the Links function to simply act as a middle-man. Similarly, Links did at the time of implementation (although it has been added since) not support the generation of random numbers, the use of which is necessary to add *salt* (extra, possibly non-deterministic, information) to the hashing function's input for added security. Again, I added this functionality by making use of the available OCaml libraries.

Session handling is another feature not fully supported in Links — although the language allows programmers to get and set cookies, there are no facilities for session timeouts: sessions are seen as active as long as the web browser is open. For LODE, this means that that if sessions are to be invalidated after a period of inactivity, additional session functionality is necessary. While some may argue that it is not the responsibility of the application to protect its data by timing out unattended sessions, as this is normally included in the operating system by means of screensavers etc., I felt that as LODE is meant to be used as a use-anywhere development environment the user may not always have control over how the operating system deals with sessions. Including session timeout functionality in the application adds an extra layer of security, but since it is possible users will find session timeouts obtrusive I decided to include the ability to disable them.

In LODE, session timeouts are handled in a rather novel manner. A separate thread is started when the user logs in, which attempts to select information from the `valid_sessions`-view once per minute using the session ID to check that that the user's session is active. If not, a pop-up (see 3.6.2.1) is displayed, requiring the user to re-authenticate to be able to continue working. The `active_sessions`-view is kept up to date by using a feature of MySQL known as *triggers*, which enable database functions to be attached to certain table events, in this case for the `sessions`-table. When a session is created, a new row is inserted into the `sessions`-table with a field for last access set to the creation time. Anytime a file is loaded or saved, LODE performs an update of the row in the `sessions`-table with the corresponding session ID, which fires a trigger-function updating the access time, keeping the session alive. The reason for using triggers rather than explicitly updating session times from Links is that at the time of implementation,

Links had only limited support for date and time functions.

A session is ended when the user decides to log out from LODE, at which point the session data is deleted from the `sessions`-table, the cookie created during log-in is cleared, and the temporary directory and any files therein are deleted from the server. If a user attempts to access the main editor page without having an active session, LODE will redirect the user's browser to the log-in page using a client-side redirect, rather than the explicit HTTP server responses used for redirects in Links.

There are no resources for creating new accounts in LODE, a feature intentionally left out as I felt that although the application is stable, some of the security implications (see section 4.1.3) are serious enough for it not to be viable to allow registration to be publically available.

3.6 The Web Front-end

Superficially, the design of LODE is inspired by two similar projects: the *Eclipse IDE Platform* and Will Ryan's *WWWorkspace*[26], a web-based IDE for Java. The Eclipse IDE is a good source of inspiration as it has not only been developed by a major company, but is widely used in industry meaning that its major features are not only likely to increase productivity on their own, but are also already familiar to developers. This in turn means that if LODE's IDE mimics Eclipse's behaviour, the entry threshold is lowered significantly for many developers.

WWWorkspace is a good example of a well-designed web-based application that, although the implementation is vastly different from LODE as it was mainly written in JavaScript rather than Links, provides interesting insights into the issues that affect online IDEs. The HCI analysis conducted in [26] is extensive and I have when designing LODE's interface used it as a source of inspiration and useful pointers.

3.6.1 Website Organisation

LODE is separated into two pages: an initial page displaying some information about the project; and a page containing the editor itself. The initial page states the purpose of the webpage and links to external information, and as mentioned also enables registered users to access the application by logging in.

On a successful log-in, LODE's main page is loaded; failed logins simply reload the log-in page.

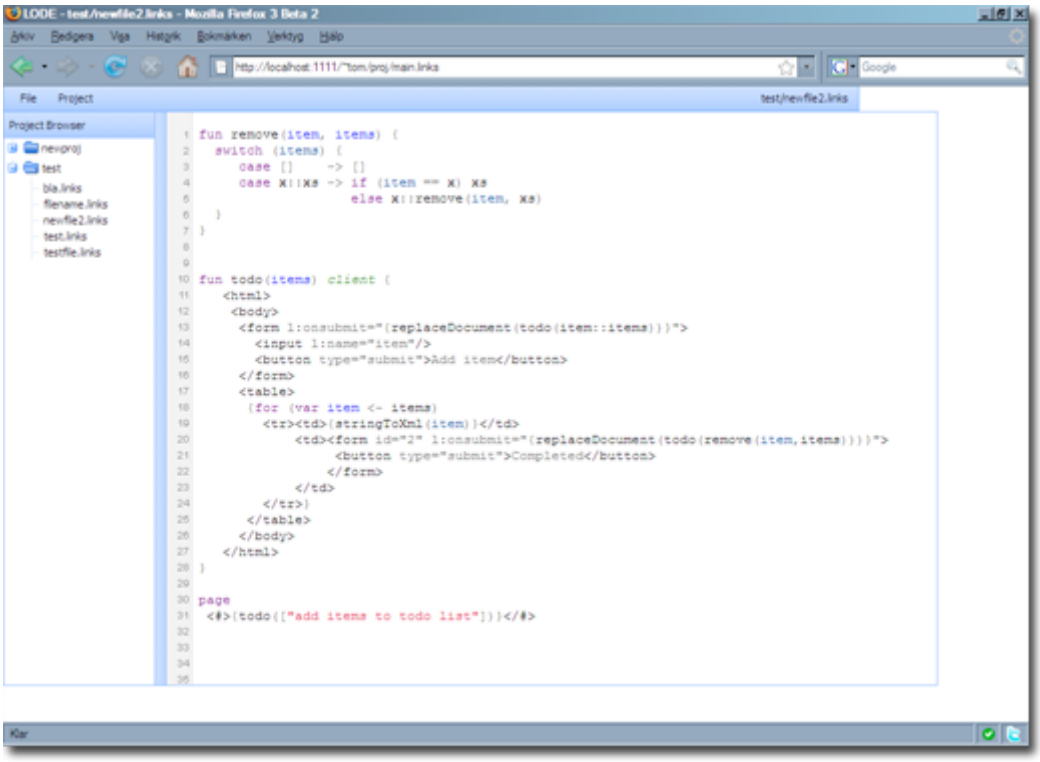


Figure 3.2: LODE's user interface.

3.6.2 User Interface

As stated above, the features of the Links IDE are to be as similar to those of Eclipse as possible in order to make the transition from a stand-alone IDE application as simple as possible. This quite naturally extends to the user interface, which needs to provide a familiar way for developers to use the application. Throughout development, I have laid the main focus for the interface on ensuring its simplicity and clarity.

The interface can be seen in figure 3.2, and more detailed descriptions of its components can be found below.

3.6.2.1 The Menu Bar

Consisting of two drop-down menus, *File* and *Project*, the menu bar allows the user to perform file and control operations. Drop-down menus are realised by creating elements with their `display`-style set to `none` when the page loads, and attaching event handlers (using Links' `l:onclick` mechanism) to the appropriate elements, toggling the style attribute between `none` and `block`.

The *File*-menu (seen in 3.3(c)) gives access to the file operations *Save*, *Save As* and *New*, the latter two of which yield an internal "pop-up" window requiring further user interaction when clicked. The *New* pop-up enables the creation of new projects, the names of which must be unique, and new files, which must be assigned to a project. The *Save As* pop-up behaves in the same way as the *New* pop-up, and allows users to move files between projects. The *File*-menu also contains options for toggling the *Auto-save* (see 3.7.1) and *Session timeout* functions, as well as exiting the application by means of *Logging out*.

The *Project*-menu contains two options, *Run* and *Deploy*. Selecting *Run* means LODE copies all files belonging to that project into a temporary directory on the server accessible by the webserver (and therefore the Links compiler), and opens a new browser window pointing to that location, whereas *Deploy* copies a project's files into the user's "live" directory on the server. Copying all files from the active project rather than just the active file means that any in-project dependencies, for example links or imports (although Links does not yet support the latter), are left intact and that the files being used are always the most recent ones.

When a file is being edited, its full name (including its project) is displayed on the far end of the menu bar, as well as in the browser title.

Pop-up windows, one of which can be seen in figure 3.3(d), are toggled in the same way as the drop-down menus. When a pop-up is visible, all other sections of the page are disabled using an all-encompassing element which, when visible, lies on top of all elements bar the pop-up. LODE uses this type of user interaction to simulate the traditional dialogue boxes seen in stand-alone applications wherever appropriate. Although there are other ways of achieving similar effects, for example using pure JavaScript dialogues, I felt that keeping the dialogues completely within the web applications not only allows LODE to retain complete control over their behaviour and appearance, but also lies more in line with the goal of "porting" stand-alone applications.

3.6.2.2 The Project Browser

The *File*-menu omits one very important file operation: loading files. In LODE, files can only be loaded by clicking on the desired file-name in the project browser, seen in figure 3.3(a). As the files are organised by project, I felt this was a more direct and intuitive way of accessing them. LODE builds the contents of the project browser from the `projects`-table in the database upon loading, creating the tree-like structure by iterating through a user's projects. For each project a number of images are added and a list-element is created, and by selecting and looping through all files in that project that list is populated. Event handlers are

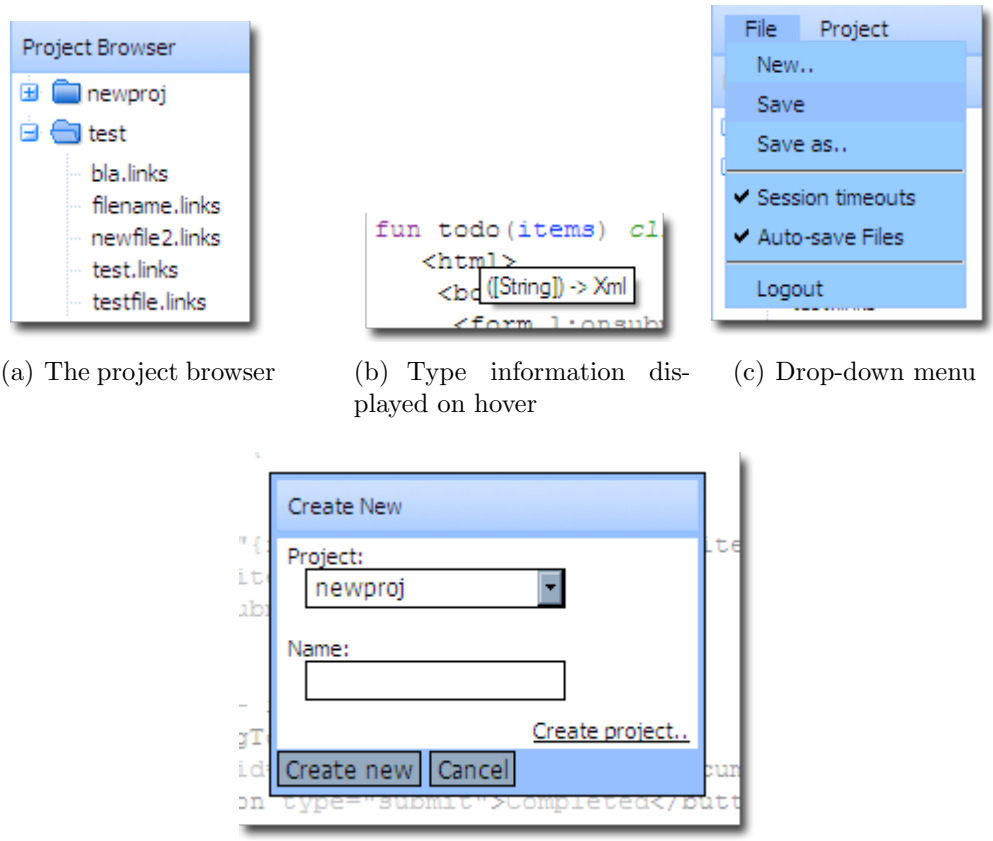
attached to the plus/minus image to allow users to expand/collapse the project listing (using the `display`-technique described previously), as well as to the file names to allow files to be loaded.

These event handlers' functions navigate the document's DOM dynamically, i.e. by looking at the attributes and types of nodes found, in order to determine which elements to display (or not). While the handler functions could make use of the fact that the structure of the each project list is static, I felt that using absolute element relationships to navigate the tree would make the application less extensible — if, in the future, alterations were to be made to the tree structure, the functions would need to be updated accordingly. Dynamic tree traversal allows the structure to change underneath the functions, as long as the element they are looking for still exists.

Early implementations of LODE built the project browser's contents directly on the client-side, requiring separate server calls for each project and file found. This was, quite naturally, found to be too inefficient and was altered such that the user's projects and their contents are returned in a structured manner (using a project *variant*) from the server, allowing the trees to be built on the client without further remote calls.

I initially planned for the project browser to give the user more direct access to file operations on individual files by attaching a context-sensitive right-click menu to the file and project nodes of the tree, but due to an issue with Links' handling of events this has not been implemented. In essence, Links is unable to stop events propagating to the browser: any event occurring in the web application is seen by both the application's handlers and the browsers event handling mechanisms. While this is fine for most events (as they will be ignored by the browser), for key-press and mouse-click events this can cause issues. For applications wishing to attach menus to the right mouse-button, as was the case for LODE, being able to stop propagation is crucial, as web browsers like Firefox have their own menu attached to right-click events.

While the IDE does not support their creation explicitly, mechanisms for organising files in projects into folders do exist in the browser-building functions (`Folder` is one of the variant options), as well as in the database. In the case of the latter, the `files`-table contains a row for identifying a parent, which is simply the identifier for another file in the table with a *folder*-flag set. This means that folders are special-case files, and that arbitrary depths of foldering is supported by the project browser. The decision not to include support for creating folders was partly due to the inability to create context-sensitive right-click menus, as I felt that without it, capturing the exact position the user wanted to insert a folder would lead to non-intuitive pop-up menus or would clutter the interface.



(a) The project browser

(b) Type information displayed on hover

(c) Drop-down menu

(d) Overlay: "Create new" pop-up

Figure 3.3: Various parts of the LODE user interface.

3.6.2.3 The Editor Window

The editor section of the user interface is the centre point and the most heavily used component of the application, allowing the user to edit code and performing operations on what has been entered in order to syntactically highlight it. In modern web applications, the standard way of creating so-called *rich text editors* is by using embedded internal frames, or *iframes*, and setting the document in that frame to be editable (achieved by setting the iframes body's `contentEditable` (Internet Explorer) or `designMode` (Firefox) attributes), and LODE is no different. LODE implements syntax highlighting by using a purpose-modified version of CodeMirror[10], which has been extended to work for Links within LODE and is embedded in the document using an iframe created by instantiating a CodeMirror-object.

The above is a deviation from the specification, which states that the majority of LODE is to be implemented in Links. The reason for this diversion is that

```
fun remove(item, items) {
  switch (items) {
     case []     -> []
     case x::xs -> if (item == x) xs
                   else x::remove(item, xs)
  }
}
```

(a) Highlighted Links code

```
<span class="part keyword">fun </span>
<span class="part variable">remove</span>
<span class="part punctuation">(</span>
<span class="part variabledef">item</span>
<span class="part punctuation">, </span>
<span class="part variabledef">items</span>
<span class="part punctuation">) </span>
<span class="part punctuation">{</span>
<br/>
```

(b) CodeMirror-generated structure for the first line

Figure 3.4: Highlighted Links code and its HTML structure

implementing a full, correct highlighter is likely to be very time-consuming, and while it is an important part of the project, it is not integral that it is implemented in Links. In fact, having an editor written in JavaScript yet using Links functions to manipulate its contents (via the `alien` interface described previously) has been a very good way of exploring how well Links can be integrated with external JavaScript libraries.

CodeMirror is, as is described in section 1.4, a powerful JavaScript-based syntax highlighting editor, making use of continuations to keep track of the state of the document. When a user enters text into the editor iframe, CodeMirror makes use of a lexer to split the text into annotated tokens, which are then iterated through by a parser in order to capture the context of each token, and from there derive how it fits into the language's syntax. A `span`-tag is created for each token, with the `class`-attribute used to give the token's type. The actual highlighting is then achieved using a style sheet, linking token classes to styles. More information on CodeMirror is available at [10].

In order for the editor to be able to work with Links code, I created a lexer and a parser for the language, using the JavaScript versions available with the CodeMirror source as guidance. Parsing Links code is non-trivial due to XML being a native type: XML is allowed to exist (almost) anywhere in Links, and Links code is allowed to exist within the XML. To overcome this issue, I included an alternative lexer and parser for basic XML in the Links versions, giving CodeMirror a

notion of language *context* within Links using a stack of parsers. Anytime XML is encountered within Links, the XML context parser is pushed onto the stack, and is popped only when the tag causing the context switch is closed. Similarly, if a single left curly brace ({) is found in the XML parsing context, the Links parser is pushed onto the stack and is popped only when a matching right brace is encountered. Figure 3.4(a) and 3.4(b) show a syntactically highlighted piece of Links code and the structure of the HTML generated for the first line by LODE's CodeMirror implementation.

A number of smaller modifications to CodeMirror have also been made in order to allow LODE to attach and access information about the active file, including its name, which project it belongs to, and whether it has been modified. Earlier versions of LODE inserted and retrieved code to and from the editor manually by converting between XML and string representations of the code. This was found to be cumbersome as it required LODE to traverse the editor's DOM-tree, so further extensions were made to CodeMirror to enable LODE to invoke the functions attached to the CodeMirror object, including functions to get and set the editor's code.

One of the possible extensions to LODE stated in the specification is the ability of the user to undo and redo previous actions. Initially, LODE used an early version of CodeMirror containing no resources for this, meaning that if such functionality was to be included in LODE I would have to extend CodeMirror further. Due to the amount of structural changes to the document that are caused by CodeMirror's highlighting process, the most straight-forward method of implementing undo/redo would be to keep a finite number of snapshots of the document's state. This was deemed too costly time-wise to implement, and was therefore not going to be included. However, versions of CodeMirror released in early 2008 contain undo/redo functionality natively, and although the newer versions required significant work to enable re-integration, I decided to port the Links-specific changes to the new version of CodeMirror as I felt that the functionality would increase the appeal of LODE.

For a statically typed programming language, function types play a very important role in the correctness of a program, which is why I also decided to implement tool-tip type information for Links code as a further extension. For this to be possible, LODE required the following:

1. Access to Links' type checker and the ability to invoke it for a given piece of code;

2. The knowledge of which elements in a document are eligible for type information;

3. The ability to match a given set of types to the elements in a document.

To achieve (1), the Links team extended the library to include the function `getTypes`, which takes a Links program as a string and, assuming the program compiles, returns a list of (*name, position, type*) triplets. By getting LODE's code parser to annotate the elements it creates with their token type, I was able to fulfil (2), and sorting the list of triplets according to position allowed me to match the type information to the elements found in the editor document. In LODE, the process of fetching and attaching type information invoked any time a file is loaded or saved. Elements are designated their matching type using the `title` attribute, which causes web browsers to display the type information when the user's cursor hovers over them, as seen in figure 3.3(b).

At present, type information is not attached to XML items' `l:`-attributes nor to the required top-level `page` declaration due to the way in which the `getTypes`-function is unable to return their position correctly, making it difficult for the editor to find the correct elements in the page. Both of these issues could be remedied by attaching type information during the parsing process as all `l:`-attributes return the type `Event` and the `page` always has the type `bodyp`, but I felt that this would make the distinction between the parsing and type-attaching less clear. I also did not think the loss of type information of these particular items removes anything from the type attachment feature, as not only do the items always carry the same type, they cannot be explicitly called from other parts of the code and therefore not likely to cause mismatches. For similar reasons (task-distinction and obvious types), I also chose not to attach type information to atomic tokens such as numbers and lists.

3.7 File Handling

3.7.1 File Operations

The previous section mentions loading and saving files, but the mechanisms behind this have not yet been explained. In LODE, files and file data are stored in the database, meaning that when a user clicks on a file in the project browser or selects *Save* from the drop-down *File*-menu, database operations are used to fetch and store Links files. LODE's various file operations work as follows:

New File When the user clicks the 'Create New'-button in the *New*-pop-up, the filename and project chose are validated and, if validation passes, the current file is checked for changes. If there are unsaved changes, the user is given the option to save before the new file is created; otherwise the iframe's document is cleared straight away. Similarly, if the user loads a different file or creates another new file without choosing to save the current file first, the document is discarded and removed from the browser. A new file is

not inserted into the database until the user saves it, but a node is inserted into the project browser under the chosen project, with an asterisk next the name signifying the file's as-yet unsaved status. If a user saves the new file after editing, the asterisk is removed. I chose to not add the new file to the database straight away for efficiency reasons as it removes two potential client-server calls — one for adding the file, and one for deleting it should the user wish to discard it.

New Project The *New*-pop-up also allows users to create new projects. This operation differs from that of creating a new file as clicking 'Create New' not only inserts the new project into the project browser, but also creates a new entry in the `project`-table and adds the project-name to the project lists in the new and save-as popups.

Save File (As) In order to save an active file, LODE calls the CodeMirror object's `getCode` function, which returns the code as a string. If a normal save has been requested, LODE retrieves information embedded in the editor frame regarding the file's name, project and status, the latter of which determines whether the file has been altered since the last save. Only if the file has been changed is a database `insert`/`update` (depending on if the file exists in the database) call made. If the save is issued from the *Save As*-pop-up, the editor frame's file information is ignored and the information entered by the user is used instead.

Load File Similarly to file saving, LODE makes use of the CodeMirror object's `setCode` function to insert code from the database, which automatically triggers syntax highlighting. When the user clicks a file-node in the project browser, LODE uses the event to find both the name of the file and its project and performs a database `select` using this information to get the code to insert.

Both the load and save file operations perform some minor cleanup operations on the file contents in order to ensure that line breaks and spaces are represented in ways the editor and database expect, and as mentioned, saving and loading files fetches and attaches type information to the appropriate elements in the editor frame.

When developing the file handling operations, it became clear that as the saving and loading requires database calls they are likely to take at least a few seconds to complete, the results of which are used in between calls causing LODE and the browser to grind to a halt while waiting for the operations to complete. Causing applications to freeze, even temporarily, is never a good thing, so I decided to make use of Links' concurrency features for file handling: instead of performing the operations explicitly, a process is started when LODE loads, and the file handling functions called by the interface simply send messages to this

process containing the appropriate data when a file operation is required. This allows both the browser and LODE to remain responsive, and is therefore more user-friendly. However, to discourage users from "interfering" with the loading and saving of files (for example by inserting new text), LODE uses the overlay-technique used for popups to disallow interaction until the operation is complete.

A final file feature of LODE is the auto-save functionality, which piggy-backs on the session-timeout process, checking for file changes once a minute. If changes are found, the process performs a save transparently to the user, allowing him to continue working on the file. The auto-save feature, like the standard file operations, performs a save by passing a message to the file handling process. Auto-saving is disabled for files not yet in the database (i.e. unsaved new files) and, as mentioned in section 3.6.2.1, users are able to turn the feature on and off using the *File*-menu.

3.7.2 Running and Deploying Projects

The section discussing the file menu (3.6.2.1) describes LODE's separation between running and deploying projects. I decided to make this distinction to enable programmers to develop and test their code in a way that does not interfere with deployed code: running a file from LODE uses a temporary directory on the server, and therefore does not overwrite the files in the user's home directory. This requires files and folders to be written to the appropriate places on the server, but as standard, Links has no functions for performing file I/O. In order for LODE to include this functionality such functions needed to be implemented and added to the Links library's server-side functions.

When a user logs in, a temporary directory is created using the `mkdir` function, with their session ID as the directory name. Any time the user clicks *Run*, LODE uses the `writeToFile` function repeatedly to write out all the files in the active file's project to the database to this temporary directory, creating the project directory if necessary. The same function is used for *Deploy* menu-option for writing out all the files to the user's home directory. Upon a user logging out, the `rmdirForce` function is used to delete the temporary folder created for the session, including any contained files.

The running and deploying of files and projects is enabled only for files that exist in the database, meaning that newly created files are only runnable once they have been explicitly saved. If an active file has unsaved changes when running or deploying, those changes do not exist in the database and are therefore not included. This is unlike stand-alone IDEs like Eclipse, which require the user to save or disregard changes before compiling, but I felt that mandating that the file the user wishes to run is the most up-to-date one is rather pointless — it seems

more user-friendly to trust the user is aware of the active file's state.

The original design for running files wrote only the active file to the temporary directory, meaning that any inter-dependencies between files were ignored which, although Links does not at current have mechanisms for importing functions from other files, could mean that the file would not run properly. The original design for the deployment of projects was also flawed, as it deployed files by copying the contents of the current session's temporary directory. This is obviously wrong, as it relies on the assumption that the user has chosen to test any changes during the *current* session, which isn't necessary for changes to exist in the database. In both cases, writing out the entirety of the project from the database into the appropriate directory (temporary or otherwise) remedies this as it ensures that not only are all files included in the run directory, but also that they are all up to date.

4. Evaluation

The previous two chapters detail both the specification and implementation of LODE, but make no judgements on the end-product and no attempt to link the two. This chapter discusses and evaluates the abilities of LODE, comparing what has been achieved to what I originally set out to do, as well as the experience of programming in Links and any major issues found. It also discusses ways in which LODE can be improved, both in terms of design and in extended functionality. Despite this project's end-product being a user-focused application, no human-computer interaction (HCI) evaluation is made. The reason for this is that I feel that it is beyond the main purposes of the project and that, since the interface is both simple and inspired by an established IDE, an analysis would be excessive. However, should additional features be added in the future requiring the interface to change, an HCI evaluation would be appropriate.

4.1 The Links IDE

4.1.1 Specification Comparison

The specification used for LODE distinguishes between core and extended features, stating that what is included in the former is to be the main focal point of development, with the contents of the latter being looked at only if time permits. From this perspective, the LODE project is successful: all features of the core list have been included and expanded upon, as have two of the extended features.

4.1.1.1 The UI

The user interface ties together the three interface aspects of the specification ("IDE UI", "Syntax Highlighting", and "Project-style Code Organisation"), as well as the two extensions implemented ("Tool-tips" and "Undo/Redo"), and is very much inspired by the *Eclipse* and *WWWorkspace*-projects, with a clean separation of components. Focused efforts during implementation have led to it being simple and uncluttered, with menu-driven access to the IDE's file operations (with the exception of file loading) and settings. Whenever a user's pointer hovers over an area or element attached to some action (for example a file in the project browser), the mouse pointer changes shape, which helps to identify areas with which the user can interact.

The project browser mimics the behaviour and design of its inspirators to the greatest extent possible, although the lack of a right-click menu detracts somewhat from its usability. There is a clear distinction between different projects and files, and the tree-like organisation of files makes navigating between different resources both fast and simple. In addition to the lack of a menu, the project browser does not allow users to click-and-drag files to reorganise project structure, requiring instead that the *Save as*-menu option be used. In fact, the interface and LODE as a whole lacks some standard file operations, such as *close*, *delete*, *rename* and *discard*, affecting both files and projects.

The main point of the IDE, the editor itself, highlights code accurately both for imported code (whether through pasting or from the IDE's load operation) and for user-typed code. CodeMirror's non-regular expression take on web-based highlighting means that the structure of Links code is more easily captured, allowing a notion of language context without which switching between XML and Links code would be much more difficult. The parser's capturing of information also allows extra information to be attached to particular token classes, which is imperative in enabling type information display as it allows LODE to easily distinguish between variables and non-variables. The initial plans of implementing a syntax highlighter using Links would most likely not have included full parser capabilities, and would therefore have made both of the above a great deal more work.

Attaching type information upon file operations works well, especially when the autosave feature is turned on as it allows LODE to continuously fill in the active file's type information transparently to the user. The use of a separate process to handle file I/O allows LODE to make use of the save and load operations without causing itself and the web browser to freeze.

One feature available in standard IDEs but not included in LODE is that of menu buttons, giving users direct access to various menu options. While this is perhaps not an imperative part of an interface it is useful in applications with a large amount of menu options to choose from, as it enables easy access. For an application like LODE with a relatively low range of options to choose from, I felt buttons would only detract from the simplicity of the interface. If the IDE is extended with further features, this might be a useful addition to the interface.

LODE does not include access to the undo/redo functions from its interface — they are only available as key-bindings (*Ctrl+z* and *Ctrl+y* respectively) from inside the editor window. The reason for this omission is the rather late addition of the updated CodeMirror, at which point I felt altering the interface would be too time-consuming. Although the key-bindings are the same as in most common graphical editing environments (with the exception of Emacs and its ilk), it may not be obvious to users what keys are to be used.

4.1.1.2 User Handling

LODE allows a number of different users to be logged in at the same time, each with their own unique session and seeing their own projects and files displayed in the project browser. The distinction between different users maps directly to the core specification's "User Accounts", and furthers the notion of a "personal" portable development environment. Each user is required to have their own home directory (relative to LODE) on the server to be able to deploy their projects, but beyond that need only exist in the `user`-table in the database to be able to use the application.

The extended session-handling capabilities of LODE enabling session timeouts are useful in cases where users cannot rely on other mechanisms to ensure access to their general session is kept secure. However, while LODE allows the timeout capabilities to be disabled, users are unable to configure the length of time after which their sessions will time out, giving them less control over the feature than is perhaps desired. The way the trigger functions are attached are also not ideal, as they require a separate call to the database to explicitly update the access times. A better way to do this would be to have the triggers attached to the `files` table, and use the user ID to update the `session`-table's access times.

4.1.1.3 Running Links Projects

The "Code-and-Run" part of the specification has been covered in LODE, with users able to run any active file as long as it exists in the database. The application also distinguishes between "test" and "live" files, allowing users to test any changes made to a project before deploying the active project to their home directory. This distinction is a useful addition to LODE, as it maps the behaviour of the application to how I perceive programmers to work, and integrates nicely with the interface. Running a file opens a new browser window pointing at it's temporary location, deploying a project has no visible effect on the user's browser behaviour (disregarding the pop-up announcing the deployment). The latter of the behaviours should perhaps be reconsidered, or at least configurable, as the user is likely to want to ensure that the deployed live version of his application does indeed work correctly.

Filename	*Size*	*Time taken*
codemirror.js	5kB	620ms
editor.js	38kB	1025ms
event.js	83kB	2238ms
jslib.js	63kB	951ms
json.js	10kB	661ms
Mochi.js	34kB	1003ms
parselinks.js	19kB	507ms
regex.js	4kB	285ms
select.js	18kB	393ms
stringstream.js	4kB	165ms
tokenizelinks.js	12kB	95ms
undo.js	13kB	185ms
util.js	3kB	191ms
yahoo.js	30kB	921ms
proj.css	7kB	627ms
highlight.css	2kB	172ms
main.links	220kB	1604ms

Table 4.1: Sizes and loading times of LODE's files, divided by filetype.

4.1.2 Performance

4.1.2.1 Loading LODE

Although not an explicit requirement on the project, LODE does suffer from some issues with regards to performance, specifically when it comes to loading the application after logging in. The times used in this section are averaged over a sample of five, and are measured manually using a stopwatch (in the case of LODE as a whole) or using Firebug (for individual files) from a computer accessing the application from over the Internet using a cable modem connection, with none of the files cached. Manual timing of loading LODE is made necessary by the client-server nature of web applications, which makes timing programmatically difficult.

The time taken from the point at which a user clicks "Log in" on the welcome page to LODE's interface being fully loaded in the user's browser is 42.2 seconds. This can be compared with the loading time of the *Eclipse* IDE on the same computer running no other applications, 31.4 seconds, and the load time of the online IDE *tide4javascript*, 15.6 seconds.

Table 4.1 shows the sizes and loading times of the files required to be able to initialise LODE, organised by filetype and sorted by filename. From the table,

Filename	*Filesize*	*JavaScript size*	*Adjusted time taken*
todo.links	0.72kB	43kB	3.2s
progress.links	0.77kB	42kB	3.7s
draggable.links	1.8kB	47kB	3.8s
mandelcolor.links	4.4kB	49kB	4.0s
crop.links	5.8kB	59kB	5.3s
wine.links	22.1kB	169kB	15.9s
main.links	67.9kB	220kB	37.1s

Table 4.2: Sizes and loading times of various Links files.

we can see that although a sizeable number of files need to be downloaded for the application to run, the average total time required is about 11.6 seconds. The table makes no distinction between the files needed for Links applications with client-side functionality and files needed for the adapted version of CodeMirror. In terms of time required, the split between the two is 7.3 seconds for the former, and 4.3 seconds for the latter. The total time for peripheral files is in stark contrast to the 42.2 seconds taken to actually load LODE, meaning that the majority of the time is not spent on the client, but elsewhere.

To investigate what might be causing the delay, I have measured the amount of time needed to load the Links examples containing client-side code. The result of the tests can be seen in table 4.2, which shows not only the file-size of the Links file, but the size of the JavaScript-code generated for the client as well. The table is sorted by filesize, and any time taken up by AJAX requests to the server (as measured by Firebug) has been subtracted from the total time. From the table, we can see that files that contain only a small amount of code (e.g. `todo.links`) still generate a sizeable amount of JavaScript-code, and hence take some time to load. We can also see the (quite natural) progression of file sizes: larger Links files lead to more JavaScript code, which in turn leads to longer loading times.

The last three rows of table 4.2 are the most interesting to compare. `crop.links` generates a 59kB of JavaScript code, and takes 5.3 seconds to load, while `wine.links` is about three times as large and takes three times as long to load. `main.links`, however, is "only" 50kB larger than `wine.links`, yet still takes more than twice as long to load, making the increased delay unlikely to be caused simply by the large amount of code generated by Links. In fact, if one makes the (unfounded) assumption that it is possible to correlate loading times to JavaScript code size exactly, then judging by `wine.links` and `crop.links` each kB of code would take 0.09 seconds to download. Applied to `main.links`, this would yield a projected loading time of 19.8s, which, when added to the time taken to download all the peripheral files needed by LODE, still leaves 5.7 seconds unaccounted for.

From the above, we can gather that there are a number of causes for LODE's

load time being as slow as it is. First of all, the JavaScript code generated by Links is rather sizeable, accounting for part of the delay. Secondly, LODE requires a number of additional files to function, all of which need to be loaded during initialisation, increasing the total loading time. Thirdly, when rendering the interface, LODE makes a number of calls to the server to get additional data about the session, each of which takes further time. Finally, the time unaccounted for is likely to be caused by the actual rendering of the page, which requires a number of database lookups on the server and is, on the whole, quite complex.

4.1.2.2 Interface Events

It is difficult to achieve accurate timing measurements for the interactive features of LODE's interface, as the time between a user event taking place and the interface responding is always very small (and depends on the nature of the event). However, what can be said of the interface's mouse-click events (i.e. displaying a menu or expanding a project in the project browser) is that when compared to similar actions in the pure JavaScript IDEs *tide4javascript*, they feel somewhat "sluggish" - there is a small delay between the event happening and the effects being shown. In some cases, for example when the user clicks *New* in the *File*-menu, this can be attributed to the application performing a number of different tasks to support the event (in this case, LODE checks to ensure that the list of projects is up-to-date, building it from the project browser's contents if not); in others, the JavaScript code generated by Links allows the web browser to update the page contents only at certain points which cannot be controlled by the Links developer.

During the development of LODE I have taken care to avoid both of these issues, putting as much of the time-consuming logic in separate threads, but it has not always been possible to avoid completely. This is especially true in cases were non-standard DOM navigation is required. For example, attempting to find an element of a particular type, with a particular attribute, with a particular relationship to some other element, requires iterating through a node's relatives in some fashion.

4.1.3 Security

For any web application, security is one of the most important areas to get right, but unfortunately very easy to get wrong — once an application is made available to the Internet, it is susceptible to any number of malicious attempts to bring it, or the server it runs on, under an attacker's control. The purposes of attacking a web application vary, ranging from subverting the server to use it as

a host for some other purpose (spamming, distributed denial-of-service attack) to simply wanting to wreak havoc. To gain control, attackers either use known vulnerabilities in the software running on the server or, more commonly, exploit mistakes made by the application developers. From a developer's point of view, the former is difficult to guard against, whereas the latter, as long as security is kept in mind throughout the development process, can be avoided completely.

Throughout the development of LODE, I have attempted to ensure that all aspects of the application are guarded against attempts of subversion by identifying potential risk-areas and working to plug any security holes found. To this end, the following security measures are included:

Password Hashing To safeguard against sensitive information being stolen from the database, the passwords are never stored in plain-text; an MD5-hash is stored instead. The password information entered by the user during the log-in phase is itself hashed and matched against the given username's hashed password in the database.

Unique Sessions Hashing is again used during log-in when generating the user's session identifier. This is important since the identifier is stored in plaintext in a cookie on the client-side, and would therefore be vulnerable to session spoofing (i.e. altering the information stored in the cookie to access another user's session).

Session Timeouts As has been argued previously, although session timeouts are perhaps beyond the scope of LODE, they have been included as an optional feature, requiring users to re-enter their password if no save or load operations are made in a 20-minute period. Again, the password is hashed before comparison.

Session Validity Tying in the the previous point, LODE carries a notion of valid session, meaning that users are unable to access the main page without being logged into an active LODE session.

Input Validation In LODE, any action requiring user input, for example the creation of a new project, is seen as a potential hazard and is therefore sanity-checked before any action is taken. This is especially necessary for cases where user input is used to perform file operations, as allowing arbitrary filenames could lead to a malicious user overwriting or creating files elsewhere on the server. Input validation ensures that only alphanumerics, underscores and hyphens are used, and that all filenames end in the extension *.links*.

Non-configurable Running/Deployment When developing in LODE, users have no control over where the files they work with live on the server, meaning that those in charge of the web server and the LODE deployment

remain in complete control over where files are written to when the *Run* or *Deploy*-buttons are clicked.

One issue with some of the security measures mentioned above is that the hashing functions implemented make use of OCaml functionality (see appendix B.1), meaning that a server call is required to be able to hash data on the client-side, which seems rather counter-productive as although the calls are serialised, the data is still sent unhashed. This is a general problem with LODE: communication between the client- and server-sides is not encrypted, relying on data obfuscation through serialisation, which isn't particularly secure. Although the data sent is unlikely to be critical, it might make sense to employ some type of general encryption mechanism, e.g. by configuring SSL for the web server.

Although LODE's implementation has focused on ensuring security in all its operations, the server used for the application deployment needs to add a small number of extra security mechanisms to prevent malicious attempts. First and foremost, LODE requires a number of additional Links library functions to be able to perform necessary file operations, meaning that the Links version on the server running LODE needs be extended. However, exposing these extensions to the users of the application would open up an enormous security hole, as they would allow files to be created and deleted anywhere on the server where the web server process has write access. This in turn could lead to users' files being deleted, or the disk being filled with useless files, causing the server to crash. This security hole is the reason I chose not to include user-registration functionality — since LODE cannot control how deployments are set up, I felt allowing just anyone to use the application could be potentially dangerous, and that control of user creation should therefore be left to the server administrators. To plug this hole, the web server needs to be set up in such a way that users are able to access only a subset of Links' library (for example the official one). Secondly, the web server should be set up to not display the contents of the user directories to stop users from being able to access each others' files.

4.1.4 Further Work

As can be seen in section 4.1.1, LODE fulfils all of the core requirements mentioned in the specification, as well as two of the additional ones. However, the specification was intentionally left rather bare-boned as I initially did not know how the limitations of Links and the time constraints would affect the development process, meaning that there is wide scope for further improvement of LODE. The main focus of this section is on adding on features to the IDE that would make it a more effective programming tool.

The evaluation of the interface mentions that the application is, at current, miss-

ing a number of useful file operations, mainly the ability to remove files and projects; close active files; and move files between projects. Other useful operation additions would be an *Edit*-menu, including access to the Undo/Redo-functionality, as well as search, replace, and goto line functions. The extended features mentioned in the specification would also be of benefit, especially completions of code entered (whether it be brackets or statements/declarations), as would more visual bracket matching (the editor does currently match brackets when they are typed for parsing purposes, but does not indicate this visually). In general, any operations that aid in navigating and manipulating a file would be of use.

The editor window could also be worked on further. Although it has been useful in exploring the language's capabilities for JavaScript integration, an interesting extension would be to port CodeMirror to native Links code. Very much a non-trivial task, this would require the developer to make clever use of Links' data structures to be able to keep state at each line, but would be a very interesting exploration of the language in terms of interactivity and more advanced linking to JavaScript and its objects. Changing the way the editor structures its elements would also be of use, as it would allow a more precise mapping of sections of an active file to its semantics. This could in turn enable more efficient file saving operations to be made in which only parts of the file that have been altered are sent to the server, as well as more aesthetic functionality like collapsing and expanding function declarations and simpler refactoring operations. The editor could also be extended to cope with multiple files being edited at once, removing the need to close the current file when a new one is created or loaded.

The type information is currently retrieved on file operations, but works only if the file is syntactically correct and type-safe — an error is returned stating the source of the incorrectness, but this is not yet used anywhere in LODE. Attaching this information to the line or element causing the issue (perhaps using an *Eclipse*-like red squiggly line) would be very useful for the programmer, as it would avoid having to run the project to ensure files are correctly typed and written. It might also be interesting to attempt to integrate a Links console into the application, allowing Links code to be executed in a similar manner to the command-line interface available on the server. This could be achieved by either linking in to the server-side CLI, or by implementing something similar on the client.

As it stands, once a project has been deployed, the files exist in the user's home directory in the deployed state, but have possibly been altered in the database. A useful feature would be the ability to revert back to the deployed version, as a kind of very crude version control system.

Finally, further investigation into the efficiency of LODE would be useful: whether the user interactions can be optimised further; and whether the loading time of LODE can be reduced in some way (e.g. by removing database calls).

This report has also not undertaken any type of scalability testing: I do not know how heavy use or large files affect the usability of the application. The latter would be interesting as LODE uses tail-recursive functions in a number of places in its client-side code, including the sorting and attaching of file information. While tail-recursion itself should not be an issue, the Links team tells me that the JavaScript representation of a Links list is an array rather than a linked list, which makes taking the tail of a list slow. Large files could require a large amount of recursive calls, leading to performance issues.

4.2 The Links Programming Language

Before working on this project, I had no experience of Links, and only limited experience in functional programming, making the initial development quite slow, as functional programming is quite different from the imperative programming style I was used to. Once that hurdle was passed, Links programming felt straightforward and, in most cases, quick, as functions can be kept small while still containing a lot of functionality. The syntax of the language is clear and precise, and the language features (native XML, concurrency, etc) are both interesting and useful. However, one problem I found initially was that there is a lack of documentation for some of the features or examples provided, or that the documentation is not precise enough to be able to create working code. This meant that in order to implement a certain feature, I needed to piece together information from both the documentation and the examples provided with Links source. This in turn means that getting up to speed with the language, above what can be considered "basic", is perhaps more difficult than is necessary.

As Links is still in development, there are times when particular functionality is desired but doesn't yet exist, requiring the developer to either find a work-around using existing functionality, or to make additions to the language as described in section 3.4. While the additions are often straightforward, it does require knowledge of both OCaml and JavaScript, which is quite contrary to the objective of Links — instead of having to learn three languages (client, server, database), development in Links can require four (Links, JavaScript, OCaml and SQL). As the language grows, the need for this will diminish, but unless access to all JavaScript library functions is made available through Links, I don't think it will never go away completely, lessening the appeal of the language somewhat.

As mentioned in the evaluation (section 4.1.2.1), code requiring any type of client-side functionality can make loading a Links web page a rather lengthy affair, as it requires a large amount of JavaScript code to be included in the web page. For example, looking at table 4.2 the file `todo.links` contains almost no Links code and is less than one kilobyte in size, yet when requested by a browser an

additional 42kB of code is generated. While this is less of a problem once a website is deployed, it can be troublesome during development as it increases the time needed to test code, which in turn increases the total development time. A possible solution to this problem would be to only include client-side code that is actually used by the application, rather than all of the built-in library functions as is currently the case.

The development of LODE has meant integrating a large piece of external JavaScript code into a Links application, calling its functions from Links functions. The `alien` interface makes this a surprisingly straight-forward task: as long as the function we wish to call is accessible from the document, Links can call it. Accessing objects is slightly more difficult, as they need some sort of reference point to be accessible. In the case of LODE, I attached the CodeMirror-object to the iframe, and called its functions from there.

Database functionality in Links is, as described in section 2.2, limited to only the basic `select`, `insert`, `update` and `delete` operations. Anything more advanced (e.g. the session handling functionality employed in LODE) requires making use of the functionality available in the DBMS chosen, which again goes against the tier-unifying goal of Links. Even standard database operations such as creating or dropping tables are not supported in Links. It would be very useful from a developer point of view if the database functionality were extended to allow more database features to be accessed from within Links, for example some of the built-in functions (my initial thoughts when implementing the MD5 hashing function was to make use of MySQL's version, but I found I could not access it by a simple `select` operation). Perhaps down the line, the Links compiler could be extended to not only allow the creation of tables, but views, triggers and stored procedures too. The ability to attach a Links function (e.g. annotated with `database` instead of `client`/`server`) to database events would make advanced database operations accessible in a novel and useful way.

In Links, strings are treated as lists of characters, making any and all list operations available to them. While this is often a good thing when it comes to generalising functions, it can cause issues when more advanced string operations are required, for example splits/joins and pattern replacement. For the development of LODE, any required string functionality has been implemented as library extensions, but I feel that it would be beneficial for the language as a whole if what in most languages would be seen as fairly standard string functions were included in the Links standard library.

Finally, one of the most serious issues found during the development of the current version of LODE is that debugging Links applications is both slow and difficult. The client side code created by the Links compiler is heavy-weight and almost impossible to decipher (the fact that the resulting JavaScript is written in a continuation-passing style makes it very difficult to read), and while functional

programming in theory makes debugging easier, as the result of a function call is always going to be the same if passed the same value, Links error messages can sometimes be cryptic or just plain wrong. There are also issues with the way Firefox handles JavaScript in some instances, leading to portions of code sometimes simply crashing the browser without giving any indication of what went wrong, and other times working without issues. These issues are difficult to work around, so while it does in no way void the introduction of bugs, the best approach is to write code in very small increments, testing each alteration as it is added to ensure that it does not result in an error. Thankfully, newer versions of Firefox (Firefox 3 beta) appear to handle Links-generated JavaScript much better — LODE has yet to cause Firefox 3 to crash.

During the development of LODE, a number of bugs in Links were found, and have since been fixed by the Links team. Appendix B.2 lists a number of these.

5. Conclusion

From a development perspective, this project has been both interesting, formative and successful. Not only have the core goals of the project been achieved in terms of functionality, but the process of achieving these goals has taught me a great deal about functional programming, web programming, and Links in particular. The end-product of the project is web-based development environment, allowing users to log on, create and edit Links projects and files in an interactive setting, before testing and deploying them on the web server. Although it is not without problems, in particular in terms of the security and performance aspects mentioned in the evaluation, and despite it lacking some of the standard IDE functionality, it is a fully usable development tool. LODE is accessible through a web-browser, removing the need for each user to install Links on their own web server, in addition to aiding the development process through the use of syntax highlighting and type information.

Part of the purpose of this project was to create one of the first major applications in the Links programming language, exploring both the language itself and any problems developing a non-toy project. The implementation of LODE has highlighted that while developing an application in Links is very much feasible, it is not always as easy as would perhaps be hoped due to the lack of support for some fairly standard functions (JavaScript or otherwise) in the language. The lack of functionality requires the developer to either work around the problem using the available functions or, more often, extend the library to implement what is missing. For client-based functions, this is not too problematic thanks to the `alien`-interface, but having to implement OCaml functions to add some server-side functionality is beyond what should be expected of a web developer.

The introduction to this report mentions the HTML `<blink>`-tag, and it does so for a reason. Prior to working with Links, I had very little relevant experience with the development of web applications, having been introduced to HTML in the aforementioned tag's heyday in the 1990s and have since only had a fleeting look at creating web sites in more modern web languages such as PHP and Ruby-on-Rails. Because of this, I feel this project has given me a great deal of useful exposure to the tools and methods that make up modern web development, having taught me JavaScript and the DOM, as well as CSS and general markup. I have also learnt much about functional programming, both in Links and in OCaml, neither of which I had come into contact with before.

The development of LODE has required me to take several non-standard approaches to problem-solving, drawing from a number of different resources to enable the implementation of some of the desired features where the standard

Links functionality has not been sufficient. However, the tier-unifying concepts of the language are more than just interesting, as they have the potential make development much less painful than it is currently. Links is still far from being a finished language, and while it still has a long way to go, I feel certain that as the language matures there will be enough support for whatever a developer might want to include in his application, making excursions outside Links unnecessary.

6. Acknowledgements

A number of people have helped make this project successful, and I would like to express my gratitude for their contributions. In no particular order: Will Ryan, whose work on *WWWorkspace* was the inspiration for this project; Marijn Haverbeke, for not only creating an impressive open-source syntax highlighting editor, but providing in-depth documentation about it too; Phil Wadler, for helping me clearly define the direction of the project, and for ensuring my feet were kept solidly grounded in the early stages of the project; and Sam Lindley, for providing all-round support with the project, regardless of whether the problems were practical or theoretical, and for spending time with me pouring over bugs in both Links and LODE.

Bibliography

[1] Akhmechet, S, *Functional Programming For The Rest of Us*, http://www.defmacro.org/ramblings/fp.html (retrieved 24/02/08)

[2] Biblio Tech, *Technology review: Databases*, http://www.biblio-tech.com/html/databases.html (retrieved 29/01/08)

[3] Collin-Sussman, B; Fitzpatrick, B. W; Pilato, C. M, *Version Control with Subversion*, http://svnbook.red-bean.com/ (retrieved 11/02/08)

[4] Cooper, E; Lindley, S; Wadler, P; Yallop, J, *Links: Web-programming Without Tiers*, 5th International Symposium, FMCO'06 (Revised Lectures), p. 266-269

[5] Cooper, E; Lindley, S; Wadler, P; Yallop, J, *An idiom's guide to formlets*, http://groups.inf.ed.ac.uk/links/papers/formlets-draft2007.pdf (retrieved 20/01/08)

[6] The Eclipse Foundation, *Eclipse.org home*, http://www.eclipse.org (retrieved 20/01/08)

[7] ECMA, *ECMAScript*, http://www.ecmascript.org/ (retrieved 07/02/08)

[8] Open-source Erlang, *Erlang*, http://www.erlang.org (retrieved 24/02/08)

[9] Garrett, J. J, *Ajax: A New Approach to Web Applications*, http://www.adaptivepath.com/ideas/essays/archives/000385.php (retrieved 11/02/08)

[10] Haverbeke, M, *CodeMirror: In-browser Code Editing*, http://marijn.haverbeke.nl/codemirror/ (retrieved 20/01/08)

[11] Hurni, M; M.A.d.S, F, *CodePress: Real Time Syntax Highlighting Editor written in JavaScript*, http://codepress.org (retrieved 18/01/08)

[12] IDC, *Rich Internet Applications*, whitepaper, 2003

[13] INRIA, *Objective Caml*, http://caml.inria.fr/ocaml/index.en.html (retrieved 09/02/08)

[14] Keith, J, *DOM Scripting: Web Design with JavaScript and the Document Object Model*, 1ed, Boston: APress 2005

[15] Kocucl, A, *rainbow9 - open source web development kit*, http://www.rainbow9.org (retrieved 18/01/08)

[16] Links team, *Links Syntax*, http://groups.inf.ed.ac.uk/links/quick-help.html (retrieved 20/01/08)

[17] Mayer-Schnberger, V, *The Cookie Concept*, http://www.cookiecentral.com/c_concept.htm (retrieved 24/02/08)

[18] mozilla developer center, *A re-introduction to JavaScript*, http://developer.mozilla.org/en/docs/A_re-introduction_to_JavaScript (retrieved 07/02/08)

[19] mozilla developer center, *JavaScript - MDC*, http://developer.mozilla.org/en/docs/JavaScript (retrieved 07/02/08)

[20] MySQL AB, *About MySQL*, http://www.mysql.com/company/ (retrieved 07/02/08)

[21] MySQL AB, *MySQL AB :: The world's most popular open source database*, http://www.mysql.com (retrieved 20/01/08)

[22] NetBeans, *Welcome to NetBeans*, http://www.netbeans.org (retrieved 20/01/08)

[23] Netscape,*Open Directory - Computers: Programming: Internet: Server Side Scripting*, http://www.dmoz.org/Computers/Programming/Internet/Server_Side_Scripting/ (retrieved 24/02/08)

[24] Parakey, Inc., *Firebug*, http://www.getfirebug.com/ (retrieved 11/02/08)

[25] PostgreSQL Global Development Group, *PostgreSQL: The world's most advanced open source database*, http://www.postgresql.org (retrieved 24/02/08)

[26] Ryan, W, *Web-Based Java Integrated Development Environment*, http://www.willryan.co.uk/Dissertation.pdf (retrieved 07/02/08)

[27] Schuringa, J, *Tide4JavaScript*, http://www.tide4javascript.com (retrieved 18/01/08)

[28] SiteHeart Inc., *CodeIDE*, http://www.codeide.com (retrieved 18/01/08)

[29] Smith, J. B, *Practical OCaml*, 1ed, Boston: APress 2006

[30] SQLite, *SQLite Home Page*, http://www.sqlite.org (retrieved 24/02/08)

[31] University of Illinois, *CGI: Common Gateway Interface*, http://hoohoo.ncsa.uiuc.edu/cgi/intro.html (retrieved 05/02/08)

[32] World Wide Web Consortium, *W3C Document Object Model*, http://www.w3.org/DOM/ (retrieved 05/02/08)

[33] World Wide Web Consortium, *Cascading Style Sheets*, http://www.w3c.org/Style/CSS (retrieved 05/02/08)

[34] World Wide Web Consortium, *CGI - Common Gateway Interface*, http://www.w3c.org/CGI (retrieved 05/02/08)

[35] Wadler, P, *Links: Linking Theory to Practice for the Web - Case for Support*, http://groups.inf.ed.ac.uk/links/epsrc05/case.pdf (retrieved 20/01/08)

Appendix A. Specification

A.1 Requirements

From an end-user point of view, part of the pain the LODE project is trying to ease is that of being unable to develop applications in Links without explicit access to a Links-enabled web server. It is therefore necessary that LODE is made as accessible as possible: a user should be able to use the application in a standard web browser; load times should be minimised; and any work-intensive components should be kept on the server rather than the client. As LODE is meant to be a way in which to develop Links code "on the go", files created by a user should be kept on the server and should be runnable from within the application. A further goal of the project is to make development in Links easier for users unfamiliar with the language, meaning that the development environment should include mechanisms for helping its users to write correct code.

From a developer point-of-view, the purpose of creating LODE is to explore application development in the Links programming language, and the project should therefore mainly be developed in Links. Any deviations from this requirement need to be justified and documented. As Links is still in development, it is possible that functionality needed to implement some part of LODE does not yet exist. Should such a situation arise, attempts should be made to extend Links as required, either by the developer or by aid of the Links team.

The time available for the design and implementation of this project is rather limited. This means that the initial development should focus on getting a number of "core" features in place, with the possibility of adding functionality at a later stage, which in turn requires the code to be structured, clear, and well-documented.

No explicit requirements are placed on the responsiveness and general speed of LODE, but as the tool is meant to be employed by real developers used to stand-alone applications, usability requires load and interactivity times to be "reasonable".

A.2 Functionality

In order to achieve an end-product that adheres to the client-based requirements of the previous section, a number of features need to be implemented. This section lists those features, together with a short justification for their inclusion.

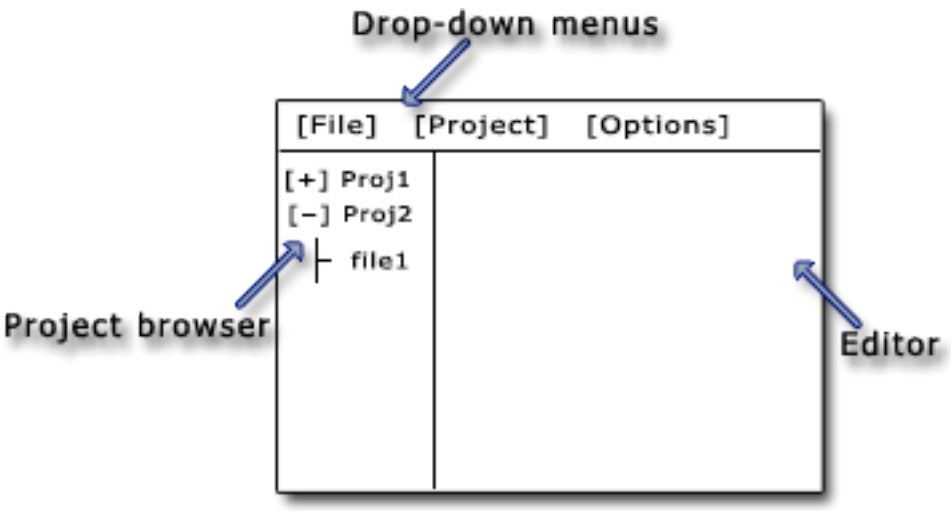


Figure A.1: A mock-up of the user interface.

The list is separated into "Core" and "Extended" functionality, the former being features that need to be included to some degree; and the latter features that would be useful and that could be included, time permitting.

A.2.1 Core Functionality

- **IDE UI**
  In order for the IDE to be as easy to use as possible, it is important that its behaviour and structure/layout is similar to that of a stand-alone IDE. Despite the limitations placed upon the application due to the chosen medium, the user should feel at home using it. Figure A.1 shows a mock-up of the user-interface.
- **Syntax Highlighting**
  With syntax being one of the most important aspects of any programming language, a clear and concise colouring of code is an obvious part of aiding developers. Syntax highlighting not only makes code more readable, but also helps new users gain familiarity with its structure. The syntax highlighting system should, in as precise a manner as possible, highlight different sections of text in accordance with the language definitions.
- **Code-and-Run**
  Allowing users of the IDE to develop and run applications in Links without requiring them to set up a web server supporting Links in order to do so furthers the project's goal of lowering the barriers of entry.
- **Project-style Code Organisation**

Organising a user's code into a tree-like structure allows for easier separation of files, and having a logical distinction between different projects more closely mirrors the way most programmers work. This also ties in to the user interface requirement, as both *Eclipse* and *NetBeans* organise project files this way.

- **User Accounts**
  While the project need not explicitly support the *creation* of user accounts, it needs to carry the notion of separate user workspaces, allowing users to work only on their own projects.

A.2.2 Extended functionality

- **Code Completion**
  Linking in to the syntax highlighting above, a nice feature of any IDE is the ability to "understand" what the user is typing and from snippets of text suggest and/or insert snippets of syntactically correct code. This could include things like automatically matching brackets, completing common statements (e.g. conditionals and loops), and auto-completing function names.
- **Refactoring**
  This refers to the ability to change the names, scope, definitions etc of functions, files and variables, and having the changes affect the project/file as a whole.
- **Customisation**
  Allowing users to control certain aspects of the development environment not only increases ease-of-use, but has the potential to increase efficiency by making the IDE more finely tuned to how a particular developer prefers to work.
- **Tool-tips**
  A useful feature of an editing environment is the displaying of additional information when a user hovers over a specific section. For an IDE, this could mean displaying the signature and comments of a function, or the type of a variable, the latter of which is especially useful in strictly typed languages.
- **Undo/Redo**
  Users should be able to step backwards and forwards in time, removing or re-adding code inserted into the document.

Appendix B. Additional Notes

B.1 Links Functions Added

The functions in this section are organised by their implementation language. Each function is listed with its name and signature, as well as a high-level description of its implementation. It should be noted that although it is not included in the list below as it has since been included in the main Links source code, a server-side random number generator was also implemented in OCaml for this project.

B.1.1 OCaml Functions

A number of the following functions return a variant of the form

`[| Ok:() | Error: String |]`

This is to circumvent Link's lack of exceptions: the *unit*-type `Ok` is returned if no errors occur; the *String*-type `Error` is returned with an error if one occurs.

- `escapeString: (String) -> String`
  Wrapper function for the *String* library function `escaped`.
- `getDirContents: (String) -> [String]`
  Recursively retrieve the contents of a directory using the OCaml *Unix* library functions. Not used in the final version of LODE; if it is to be included in the future, its implementation needs to follow the variant method of exception handling described above.
- `isDir: (String) -> Bool`
  Assert whether or not the string given points to a directory on the server. Makes use of the *Unix* library, and returns false on all errors.
- `md5: (String) -> String`
  An MD5 hashing function. Uses the OCaml *Digest* library functions.
- `mkdir: (String) -> [| Ok:() | Error: String |]`
  Wrapper function for the OCaml *Unix* library function of the same name. Its argument is the absolute path of the directory to create.
- `moveFile: (String, String) -> [| Ok:() | Error: String |]`
  Move a file existing in the first argument to the location in the second. Requires both arguments to be absolute paths, and if the file being moved

is a directory it is required to be empty. Makes use of the *Unix* library function `rename`.

- `replace: (String, String, String) -> String`
  Using the regular expression functions available in the *Str* library, this replaces sections of a string matching a particular pattern with another string. The arguments are (respectively) (*pattern, source, replacement*).
- `rmdir: (String) -> [| Ok:() | Error: String |]`
  Wrapper function for the OCaml *Unix* library function of the same name. Its argument is the absolute path of the directory to delete. The directory must be empty for this function to return without error.
- `rmdirForce: (String) -> [| Ok:() | Error: String |]`
  More advanced version of the above, ensuring that the argument is a legal directory before recursively deleting all contents (including other directories).
- `split: (String, String) -> [String]`
  Using the *Str* library's regular expression functions, this function splits the second argument using the regular expression in the first.
- `writeToFile: (String, String) -> [| Ok:() | Error: String |]`
  Writes the contents of the second argument to the absolute file location given as the first argument. The implementation uses OCaml's standard `output_string` function to perform the write.

B.1.2 JavaScript Functions

A number of the functions below have been included as part of the LODE-specific Links library despite perhaps being better suited to be called using the `alien` interface. The reason for their being in the library is the lack of documentation for `alien` - I simply did not know this interface was available in the earlier stages of development. Conversely, the JavaScript functions that are called included using the interface are not included in the list below as I feel they would not belong there.

- `adoptNode: (DomNode, DomNode) -> DomNode`
  Given a DOM node referring to an element in a foreign document and a `document` node, this function calls the JavaScript `adoptNode` function from that `document`, allowing the foreign element to be inserted[1]. The DOM

[1] `adoptNode` changes an elements `ownerDocument`, which is necessary as a document is only allowed to insert elements it is itself the owner of.

node returned is a reference to the new node.

- `domGetNodeValueFromRef: (DomNode) -> String`
  Given a DOM node, this function returns the value of the node's *value*-attribute.
- `domSetNodeValueFromRef: (DomNode, String) -> ()`
  Sets the *value*-attribute of the given DOM node to the value of the second argument.
- `getContentDocument: (DomNode) -> DomNode`
  Given a DOM node, this function returns a reference to the `document` containing it.
- `getElementsByName: (String) -> [DomNode]`
  Wrapper for the JavaScript function of the same name. This function returns all the elements in the active document whose *name*-attribute match the given name.
- `getElementsByTagName: (String) -> [DomNode]`
  Wrapper for the JavaScript function of the same name. As the function name implies, this function returns all elements in the active document matching the name given in the first argument.
- `getElementsByTagNameFromRef: (String, DomNode) -> [DomNode]`
  Given a DOM `document` node as the second argument, this function returns all the elements of that document matching the name given in the first argument.
- `getModifiers: (Event) -> [Char]`
  Given an event, this function returns a list of the character codes of the event modifiers, i.e. the *Alt, Ctrl* and *Shift* keys. If the event is not a mouse, click or key event, an error is thrown.

B.2 Links Bugs Found

During the development of LODE, a number of bugs in Links have been found. Most of them have been minor, and a majority have since been fixed. This section gives examples of different bugs found, but is in no way exhaustive.

`getCharCode` is a Links function that returns the character codes of any keyboard- and mice-buttons pressed during a key or mouse event. The original signature of the function was `(Event) -> Char`, returning the character representation of the char code found. This doesn't make much sense, as

the point of the function is to get an integer representation of the character. In newer versions of Links, `getCharCode` has the signature `(Event) -> Int`.

Marshalling bugs were found when sending file contents from the server to the client, causing certain characters to not be displayed properly or not be included at all. After investigation by the Links team, this was found to be caused by differences in encodings on the client and server sides, and the parser used for serialised objects not working well with newline characters (`\n`).

Attaching event handlers to elements created by Links did not work correctly when using `appendChildren` to attach the elements to some other node in the document. The Links team found the cause of this to be an implementation bug whereby `appendChildren` (and `insertBefore`) did not activate event handlers attached.

Stopping event propagation is currently not working properly in Links. As mentioned in chapter 3, for applications to be able to override the web browser's default actions on events, it needs to stop it from propagating to the browser's event handlers. In Links, this isn't possible, meaning that things like right-click menus cannot be displayed without also displaying the browser's menu.

Processes looping infinitely caused some issues during development. If a process contains nested conditional statements, it is possible that the JavaScript code generated causes the process to go into an infinite loop. The Links team has established the cause of this (lexical scoping differing in JavaScript and Links), but the defect has yet to be fixed.

B.3 Installing LODE

In order for LODE to work on a web server, the following is required:

A web-server capable of running Links , i.e. with CGI capabilities and OCaml 3.08.4 or later (but not 3.10+);

A MySQL database with appropriate OCaml-bindings;

The LODE-code , available in the LODE-branch of the Links repository. The branch contains the source code for both the application itself and all additions made to the Links library, as well as a schema for the MySQL tables and functions. The code and the schema exist in the `html/proj`-directory or the branch.

Once Links has been built, `main.links` needs to be edited to include the path names to appropriate directories on the server to write users' project files to. These directories must be given write-permissions for the web server process (for Apache, this is the `www-data` user and group on Linux) for the run/deploy functionality to work. When adding users, the passwords must be hashed using MD5, and each user needs to have a sub-directory in the run directory specified in `main.links`.