End-User Programming for the Web

by

Michael Bolin

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 6, 2005

Author . . . .
Department of Electrical Engineering and Computer Science
May 5, 2005

Certified by. . . . .
Robert C. Miller
Assistant Professor
Thesis Supervisor

Accepted by . . . .
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

End-User Programming for the Web

by

Michael Bolin

Submitted to the
Department of Electrical Engineering and Computer Science

May 6, 2005

In partial fulfillment of the requirements for the degree of

Master of Engineering in Computer Science and Engineering

ABSTRACT

On the desktop, an application can specify its user interface down to the last pixel, but on the World Wide Web, a content provider has little control over how the client will view the page once it has been delivered to the browser. This creates an opportunity for end-users who want to automate and customize their web experiences, but the growing complexity of web pages and standards prevents most users from realizing this opportunity. This thesis describes a programming system named Chickenfoot that enables end-users to automate, customize, and integrate web applications without examining their source code. It accomplishes this by embedding a programming environment directly into the Firefox web browser, where end-users can interactively develop programs that manipulate the interfaces of web pages. The design and implementation of the system's language are described, as well as the results of a user study that influenced the design. A range of applications built using Chickenfoot are also presented.

Thesis Supervisor: Robert C. Miller
Title: Assistant Professor

Acknowledgments

I would like to thank Rob Miller for being incredibly generous with his time and constantly supporting of my work. I know of no other thesis adviser who takes such a strong and active interest in his students, and I am extremely fortunate to have him as an adviser.

I would also like to thank all the other members of the LAPIS group, especially Philip Rha, Matthew Webber, and Tom Wilson. They were a constant source of support throughout this process. I would also like to thank David Huynh and Vineet Sinah for their technical advice.

Finally, I am especially thankful of the constant support of my parents, Tom and Linda, and my sister, Katie. Their patience and encouragement throughout this project has been spectacular.

Contents

List of Figures

List of Tables

Chapter 1 Introduction

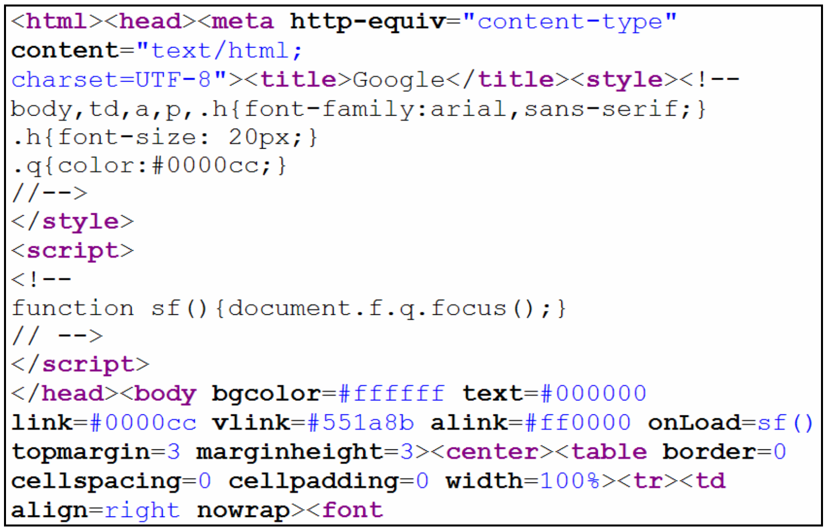
End-user programmers are users who are not formally trained in programming, yet need to program in order to accomplish their daily tasks. Spreadsheets are often touted as the major success story in end-user programming [1] -- millions of users successfully write formulas in Microsoft Excel even though only a fraction of them consider themselves programmers, or even realize that they are programming at all. But when we look to the web browser, which is the most common tool for accessing information on the Web, we find that the existing tools for automating and customizing interactions with the Web are insufficient for developers and end-user programmers alike.

For example, consider a user who has compiled a list of homes that he is interested in through a realty web site, but now he wants to see how far each home is from his workplace. He could visit a site that provides driving directions, such as Google Maps, to plug in the addresses of each house and his workplace to find the distance between them, but this will be tedious if the list of houses is long. Ideally, this service would be provided by the realty web site -- it could provide its own web form for this task as many commercial sites have done by providing a "Store Locator" that finds the nearest Target or Wal-Mart to your home. However, as user queries get more intricate (now the user wants to find the home closest to his workplace that has a Dunkin' Donuts on the way there and a McDonald's on the way back), the likelihood that a web site can support such a query diminishes. Thus, the user needs the ability to write his own scripts that will automate his personal web tasks. To that end, the user needs a tool that simplifies the process of web scripting, so that the development and execution of the script take less time than it would to do the task manually.

Most existing tools for scripting web pages [13, 15] require the user to work with the raw HTML of a page, as shown in Figure 1.1. In the *string model*, a web site is represented as a string of HTML, and users identify parts of the page by matching character patterns in the text. Because the HTML for most web sites is machine-generated rather than written by hand, it is often incomprehensible to an end-user programmer who is trying to script it, so writing scripts in this manner is time-consuming. Further, these scripts have a tenuous dependency on the current text of a web site, which may break if the site changes.

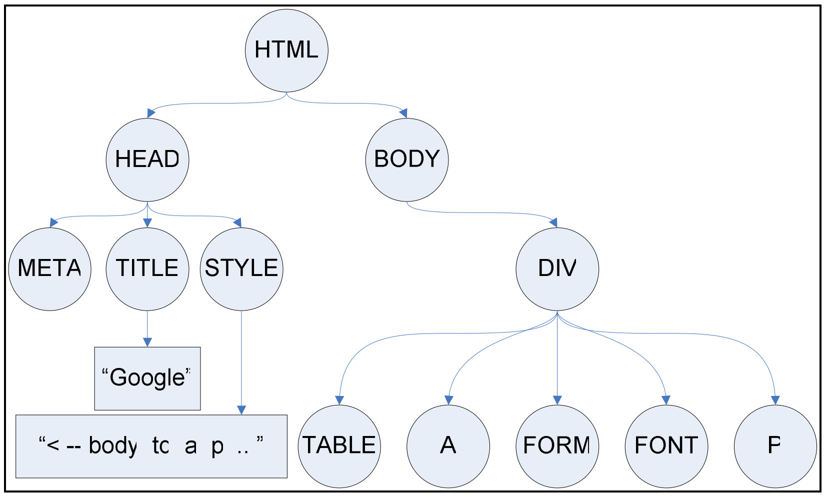
Other tools allow the user to work with the *Document Object Model (DOM)* of a web site, in which the page is represented as a tree of HTML elements, as shown in Figure 1.1. Although the DOM is the standard model for documents on the Web [2], it is not an appropriate model for end-user programmers because it still requires users to be familiar with the underlying HTML of the page.

To address these shortcomings, my thesis presents *Chickenfoot*, an end-user programming system for automating and customizing web applications through a familiar interface – as web pages rendered in a web browser. Chickenfoot enables users to work with the *rendered model* of a web page, as shown in Figure 1.1. The rendered model represents a page as a two-dimensional, typeset document, which aims to be consistent with the user's mental model of the page when viewing it through a web browser.

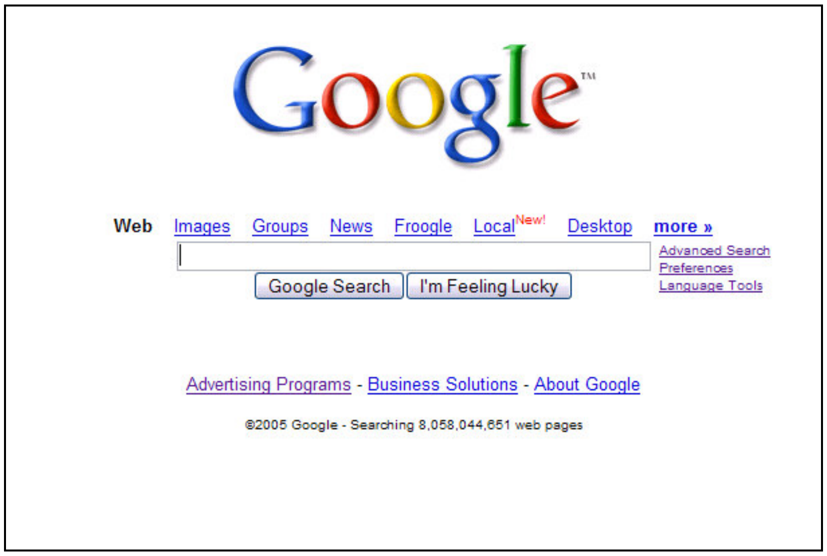


```
<html><head><meta http-equiv="content-type"
content="text/html;
charset=UTF-8"><title>Google</title><style><!--
body,td,a,p,.h{font-family:arial,sans-serif;}
.h{font-size: 20px;}
.q{color:#0000cc;}
//-->
</style>
<script>
<!--
function sf(){document.f.q.focus();}
// -->
</script>
</head><body bgcolor=#ffffff text=#000000
link=#0000cc vlink=#551a8b alink=#ff0000 onLoad=sf()
topmargin=3 marginheight=3><center><table border=0
cellspacing=0 cellpadding=0 width=100%><tr><td
align=right nowrap><font
```

String model
In the string model, a web page is represented as a string of HTML text.



Document Object Model (DOM)
In the Document Object Model, a web page is represented as a hierarchical tree of nodes. This tree is constructed from the string model using an HTML parser.



Rendered model
In the rendered model, a web page is represented a two-dimensional, typeset document. The browser creates this view by rendering the DOM.

Figure 1.1 Three models of a web page (www.google.com).

Instead of using substrings of HTML or nodes in a tree to identify elements in a page, Chickenfoot identifies elements using *keyword patterns*. A keyword pattern is text that appears in the rendered view of a web page that can be heuristically evaluated to identify a component of the page. For example, in the rendered model of google.com shown in Figure 1.1, **Google Search** is a keyword pattern that identifies the left button below the textbox. In this case, the label of the button is the heuristic used to match the keyword pattern with the page component.

Chickenfoot uses keyword patterns in its programming language, *Chickenscratch*. Chickenscratch is an extension of JavaScript [4] that includes commands that make sense for operating on the rendered model of a web page. For example, the Chickenscratch command for following a hyperlink or pressing a button is `click()`, so the code for submitting a search query to Google is `click("Google Search")`.

Writing code that fills out web forms is a common goal for Chickenfoot users, so Chickenscratch has commands to automate form entry: `enter`, `check`, `uncheck`, `pick`, and `click`. These commands take keyword patterns to identify web form elements, such as textfields, checkboxes, dropdown boxes, and buttons. An example of using Chickenscratch to fill out a web form is shown in Figure 1.2.

Web Form	Chickenscratch Code

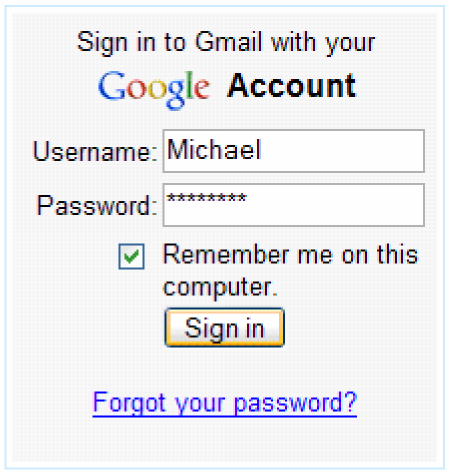
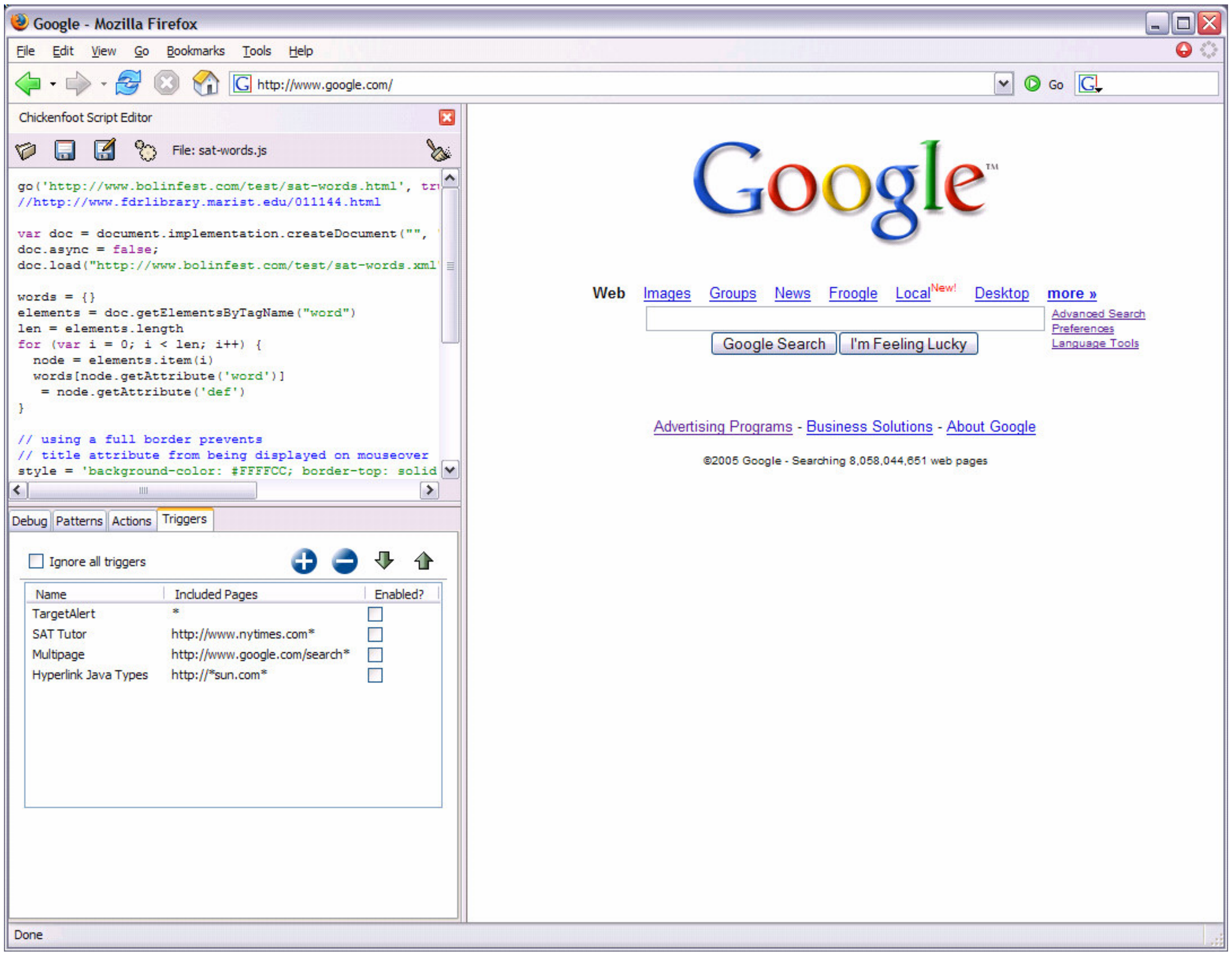
 Sign in to Gmail with your Google Account
Username: Michael
Password: ********
✓ Remember me on this computer.
Sign in
Forgot your password?
	`enter("username", "Michael")`
	`enter("password", "mypasswd")`
	`check("remember")`
	`click("sign in")`

Figure 1.2 Filling out a form using Chickenscratch (www.gmail.com)

Chickenscratch also has commands named `insert` and `remove` that allow users to add and delete content from a page, respectively. This is especially important for users who wish to amend pages with their own content, or to integrate content from multiple web sites.

Users can access other sites by using Chickenscratch commands: `go(url)` will create a rendered model of the `url` by loading it in the browser, and `fetch(url)` will create the model without displaying the page. Once the model has been created, Chickenscratch has a `find` command that takes a pattern and returns any matches that it finds. The pattern may be a keyword pattern or a *text constraint*, which is a pattern that can refer to the implicit structure of a page. An example of a text constraint is `image in first row in second table`. Like keyword patterns, text constraints can be created from the rendered model of a page alone.

To ensure that the rendered model will be available when developing Chickenscratch code, Chickenfoot is implemented as a sidebar inside the popular Firefox web browser, as shown in **Error! Reference source not found.**. From here, users can experiment with a web site by writing and running Chickenscratch code.

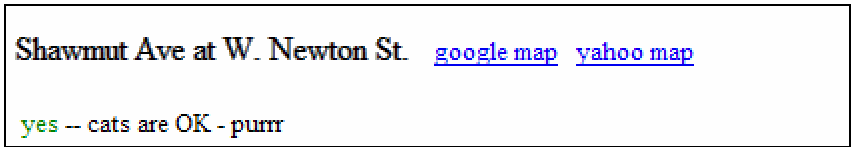

Google - Mozilla Firefox
File Edit View Go Bookmarks Tools Help
http://www.google.com/
Go
Chickenfoot Script Editor
File: sat-words.js
Google
Web Images Groups News Froogle Local New! Desktop more »
Advanced Search
Preferences
Language Tools
Google Search
I'm Feeling Lucky
Advertising Programs - Business Solutions - About Google
©2005 Google - Searching 8,058,044,651 web pages
Debug Patterns Actions Triggers
Ignore all triggers
Done

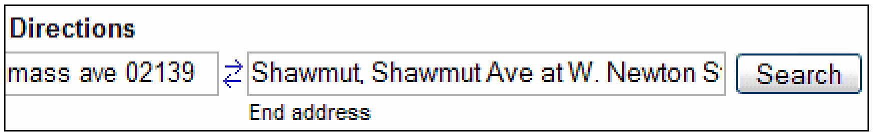
```
go('http://www.bolinfest.com/test/sat-words.html', tr
//http://www.fdrlibrary.marist.edu/011144.html

var doc = document.implementation.createDocument("",
doc.async = false;
doc.load("http://www.bolinfest.com/test/sat-words.xml

words = {}
elements = doc.getElementsByTagName("word")
len = elements.length
for (var i = 0; i < len; i++) {
  node = elements.item(i)
  words[node.getAttribute('word')]
   = node.getAttribute('def')
}

// using a full border prevents
// title attribute from being displayed on mouseover
style = 'background-color: #FFFFCC; border-top: solid
```

Name	Included Pages	Enabled?
TargetAlert	*	☐
SAT Tutor	http://www.nytimes.com*	☐
Multipage	http://www.google.com/search*	☐
Hyperlink Java Types	http://*sun.com*	☐

Figure 1.3 Chickenfoot as a sidebar in the Firefox web browser

Returning to the prospective homeowner mentioned earlier, he could solve his problem by using Chickenfoot to create a script that would get the driving distance from Google Maps and automatically insert it after the address on the realty site:

First, he would use the `find` command to extract a house's address from a web site.	 **Shawmut Ave at W. Newton St.** google map yahoo map yes -- cats are OK - purrr `location = find('text just before "google map"')`

Then, he would use `go` to navigate to Google Maps. There he would use `enter` to fill in the address data, and `click` to submit the query to Google Maps.	 **Directions**
	mass ave 02139 ⇄ Shawmut, Shawmut Ave at W. Newton S Search
	End address
	`go('http://maps.google.com/')`
	`click('Directions')`
	`enter('start address', '77 mass ave 02139')`
	`enter('end address', location + ' boston')`
	`click('search')`

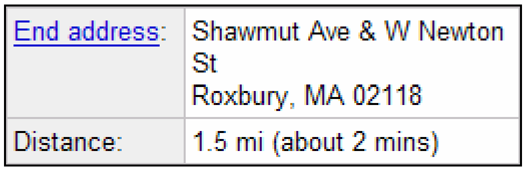
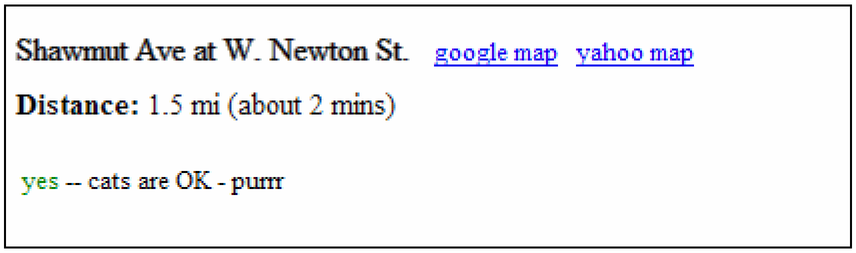
Next, he would use the `find` command to extract the driving distance from the directions page returned by Google Maps.	 End address: Shawmut Ave & W Newton St Roxbury, MA 02118 Distance: 1.5 mi (about 2 mins) `distance = find('text just after distance')`
Finally, he would use `insert` to amend the realty site with the new information.	 **Shawmut Ave at W. Newton St.** google map yahoo map **Distance:** 1.5 mi (about 2 mins) yes -- cats are OK - purrr `insert('point just after "yahoo map"', distance)`

Figure 1.4 Integrating Google Maps with a realty site.

Once the user has written this script, he will want to run it automatically whenever he checks a listing on the realty web site. Chickenfoot provides a *trigger system* that lets a user define a collection of URLs that will trigger a user's script automatically when a URL in the collection is loaded, causing the user's script to be run.

Note that the user is able to create this script without looking at any HTML; all the interactions that he needed to do with the above web pages could be done through the rendered model.

My thesis statement is: **Chickenfoot allows users to customize and automate web pages without viewing their HTML source**. In defending this claim, my thesis makes the following contributions:

- **Chickenfoot**, an end-user programming system for web automation that provides users with access to the rendered model of a web page, which abstracts the underlying HTML from the user.
- **Chickenscratch**, a language for operating on the rendered model.
- The concept of **keyword patterns**, including a web survey justifying their usability as well as an algorithm for matching them with web page components.
- A **development environment for developing JavaScript code** as well as extensions to the Firefox web browser.
- A **trigger system** that can execute Chickenscratch code whenever a user visits a web site so that the user's customizations automatically become part of the page.
- **Improvements to the W3C DOM specification** for updating Ranges in the DOM after mutation.

The rest of this dissertation explains the details of the Chickenfoot system. A survey of related work in other Web automation systems is presented in Chapter 2. The design of Chickenscratch is explained in Chapter 3. Examples of applications that have been built using Chickenfoot are provided in Chapter 4. The design of the development environment, including the trigger system, is explained in Chapter 5. A web survey that motivated the design of keyword patterns is discussed in Chapter 6, and the algorithm used to identify keyword patterns is presented in Chapter 7. The implementation of the Chickenfoot system is covered in Chapter 8. Finally, future extensions to Chickenfoot as well as its contributions are discussed in Chapter 9.

Chapter 2 Related Work

Several systems have addressed specific tasks in web automation and customization, including adding links [4], building custom portals [5], crawling web sites [6], and making multiple alternative queries [7]. Chickenfoot is a more general toolkit for web automation and customization that can address these tasks and others as well. Here I survey some of the major features of existing toolkits and compare how they are supported in Chickenfoot. The survey includes:

- programming languages, WebL [8] and Perl [9] (with Mech [10]),
- macro recorders, WebVCR [11] and LiveAgent [12],
- proxy-based tools, WBI [13] and Screen-Scraper [14],
- browser extensions, Greasemonkey [15] and Chickenfoot [16],
- and an experimental web browser, LAPIS [17].

A summary of the results of this survey is presented in a table at the end of this section.

2.1 Access Points to the Web

When doing a task on the Web, the first step is to access a web page. Though web pages are always accessed by sending a request to a server, the point of access can be significant in determining the page that is returned. The three types of access points that are seen in web automation toolkits are: outside the browser (usually from the command-line), within a proxy, and inside the browser.

2.1.1 Outside the Browser

Most modern scripting languages, Perl, Python, Ruby, etc., have a method for taking a URL, connecting to it, and downloading its content. In these languages, every connection to the Web is an independent request with no sense of state. The main benefit of this method is that programs can be run from the command-line, which is helpful in automating access to the Web.

Unfortunately, a URL accessed in this way often returns different content than it does when accessed through a browser. Web browsers support cookies, session variables, and client-side scripting, all of which affect the way web pages are displayed. Because these scripting languages do not support these advanced features, the content that they download may not be consistent with the content that the user is accustomed to viewing in his web browser. For example, accessing the home page for an e-commerce site that uses cookies to display personalized information will have different content when accessed through a web browser than it does when downloaded by a Perl script run on the command-line.

Also, although most pages can be accessed directly by their URL, some pages are dynamically generated only after a series of navigations, and other pages require a secure connection to be established before the URL can be accessed. These "hard-to-reach" pages [11] cannot be accessed by the independent requests made by scripting languages because they lack the sense of state required to reach them. Because not every URL can be accessed from outside the browser, and even the pages that can be accessed outside the browser may not be consistent with what users expect, accessing pages in this way is insufficient for a web automation system. In addition to the scripting languages listed above, even WebL [8], a programming language designed for the Web, suffers from this problem.

2.1.2 Within a Proxy

The next-best solution is to use a *proxy* that sits between the user's web browser and the Internet. When a user requests a page from his browser, the proxy may intercept the request, or the server's response, and modify it before it returns to the browser. This is a good approach, in that the activity of the proxy is hidden from the end-user and is therefore seamlessly integrated into the user's web experience. Another benefit is that the effects of a proxy can be seen through any browser on the user's desktop, so toolkits that use proxies do not force the user to use a particular browser.

However, there are two major limitations of using a proxy in a web automation toolkit. The first is that a proxy cannot read pages that have been encrypted by the browser, and the second is that the proxy cannot have any effect on a page after it returns it to the browser.

When a client accesses a site over a secure connection, every transaction with the site is encrypted. Because the proxy will only see the page after it has been encrypted by the browser, any toolkit that accesses pages through a proxy will not be able to manipulate such a page. Because security is becoming a greater concern on the Web, the number of sites that use encryption is likely to increase, so this limitation of proxy-based toolkits is significant.

Another growing trend is the heavy use of client-side JavaScript in web pages. Because there is in an inherent latency in accessing information over the Web, some sites embed complex JavaScript in their pages that can run in the client's web browser, after the page has been loaded. Responding to user input with this client-side JavaScript is much faster than responding with a subsequent request to the server, so this technique yields web applications whose performance rivals that of desktop applications. Because this activity happens in the web browser after the page has been loaded, a proxy has no knowledge of these events, so proxy-based toolkits cannot respond to this activity.

All proxy-based toolkits are affected by these *proxy problems*, including WBI [13], LiveAgent [12], and Screen-Scraper [14].

2.1.3 Inside the Browser

The third point of access, which is the one that Chickenfoot uses, is from inside the browser itself. By embedding a web automation tool inside the browser, the tool is guaranteed to be able to access the page as the user sees it, incorporating the effects of stylesheets, session identifiers, etc. Unlike a proxy-based toolkit, it can react to changes in the page that are caused by client-side scripting. It also overcomes the proxy's restriction to insecure pages by letting the browser decrypt encrypted pages before acting upon them. Both Chickenfoot and Greasemonkey [15] are extensions to the Firefox web browser that take this approach.

2.2 Automated Navigation

To access pages that are generated dynamically or that require a login, the user must fill out and submit Web forms, so to provide programmatic access to any page on the Web, it is necessary to automate entering form data on the Web as well.

Perl provides support for automating form interaction through its `WWW::Mechanize` module, often referred to as Mech [10]. Mech allows the user to write Perl code to automate a form in a web page by supplying the names and values of the inputs the user wants to enter. From there, Mech can submit the form data and return the result to the user. The user is restricted to using the same names for input elements as the web site does. This is often undesirable for end-user programmers because such names are often unintuitive to end users; for example, the name of the search box on Google is `q`.

Some toolkits give the user the ability to record macros where the user records the actions taken to require access to a particular page, such as filling out forms and clicking on links. Later, the user can play the macro back to automate access to the same page. LiveAgent [12] takes this approach, recording macros with a proxy that sits between the user's browser and the Web. The proxy augments pages with hidden frames and event handlers to capture the user's input, and uses this information to play back the recording later. Unfortunately, because LiveAgent uses a proxy, it suffers from proxy problems and therefore cannot automate pages that are accessed over a secure connection.

WebVCR [11] is another macro recorder for web navigation that skirts the proxy problems by using a signed Java applet to detect page loads and LiveConnect [18] to instrument the page with event-capturing JavaScript after the page loads. Because WebVCR runs as an applet inside the browser instead of sitting behind a proxy, it can record all types of navigation.

2.3 Pattern Language

Once a web automation toolkit has acquired input, its next step is usually to extract content from the page, often referred to as screen-scraping. To do this, the toolkit needs to have a rich pattern language to describe the content to extract.

Probably the most primitive tool for extracting material from an HTML document is regular expressions [20]. Though regular expressions (regexps) are a powerful technique for matching patterns in ordinary text documents, using them for processing HTML is often undesirable because by default regexps have a greedy "leftmost longest match" rule that consumes nested HTML elements, returning one large match instead of the individual matches contained within it

[8]. Further, though precise, regular expressions are often cryptic. For example, suppose a programmer comes across the following regular expression in a script:

```
^(([A-Za-z0-9]+_+)|([A-Za-z0-9]+\-+)|([A-Za-z0-9]+\.+)|([A-Za-z0-9]+\++))*[A-
Za-z0-9]+@((\w+\-+)|(\w+\.))*\w{1,63}\.[a-zA-Z]{2,6}$
```

What is this regex supposed to match? Is the regex correct? It is hard to answer these questions without studying it in detail. This regex is intended to match an email address; however, its author admits that it fails to match email addresses that use IP numbers in the host portion [20]. Thus, despite the power and precision of regular expressions, it is still difficult to get them right. If this is the case for mature software developers, then what hope do end-user programmers have?

More importantly, regexps require the user to become intimately familiar with the HTML of the page from which they wish to abstract information. This forces the client to use the string model of an HTML document rather than the more expressive DOM that is inherent within it.

The commercial Screen-Scraper tool [14] builds on regular expressions by providing a pattern type called an extractor pattern. An *extractor pattern* is "a block of text (usually HTML) with special tokens inserted where data is to be pulled." [21] In Screen-Scraper, an extractor pattern may look like this:

```
<p>This is the <b>~@EXTRACTED_TEXT@~</b> I'm interested in.</p>
```

where `EXTRACTED_TEXT` is a variable that can be used later in the program. In practice, this is no more powerful than using capturing parentheses in regular expressions; however, this may be more readable for a novice programmer.

A popular, more powerful pattern language for HTML and XML documents is XPath [22]. Unlike pure regular expressions, XPath allows users to match nested nodes within a parent node. Also, the syntax of an XPath expression closely resembles the form of the matches to the expression, making it easier for other programmers to understand what the expression is trying to match. For example, `/doc/chapter[5]/section[2]` selects the second `section` of the fifth `chapter` of the `doc`. However, this syntax has the same drawback that regular expression syntax does, in that writing an XPath expression requires the user to become intimately familiar with the HTML of the page. Even so, the fact that the majority of the sample scripts on the Greasemonkey site use XPath expressions [23] is a testament that many script authors are willing to plumb through a site's HTML in order to automate it. Also, as XPath is a W3C standard, Perl, Java, and JavaScript all have libraries that support XPath queries, so XPath expressions may be reused in other programming languages.

WebL [8] is a programming language for the Web that focuses on giving users a higher-level language to describe web page elements. In WebL, the user provides names of HTML elements to create *piece-sets*, where a *piece-set* is a set of *piece* objects, and a *piece* is a contiguous text region in a document. WebL provides various methods to combine *piece-sets* called *operators*, including set operators such as `union` and `intersection`, positional operators such as `before` and `after`, and hierarchical operators such as `in` and `contain`. Although these operators help produce more readable scripts, the language does not eliminate the need to inspect a web page for the names of its HTML elements, as the user must provide those to construct the basic pieces on which the operators work. In this way, WebL provides a pattern language that is similar to XPath, but is more expressive because of the hierarchical operators that it provides.

LAPIS [17] has a pattern language that is a cut above that of the previous toolkits called text constraints. As mentioned in Chapter 1, *text constraints* is a pattern language that can refer to the implicit structure of page. The text constraint `image in first row in second table` is devoid of HTML and regexp syntax, so it is much more appropriate for an end-user programmer. It is also possible for an end-user programmer to create this pattern from the rendered model of a web page, rather than the string model. Finally, this pattern is more likely to succeed even if the web site's HTML changes because it is based on lightweight structure rather than an overfitting regexp pattern.

Unlike any of the previous toolkits, LAPIS also makes it possible to create new patterns by demonstration. To create a pattern by demonstration in LAPIS, a user can highlight a portion of a document using the mouse, and LAPIS will offer various text constraints that match the pattern. This is especially helpful to users who have trouble formulating text constraints.

Indeed, LAPIS text constraint patterns are more accessible to end-user programmers than other pattern libraries are, so Chickenfoot includes the LAPIS pattern library as part of its implementation. However, Chickenfoot also builds upon this library by supporting *keyword patterns*, which are patterns that use the spatial location of text in the rendered web page to find matches. Keyword patterns are discussed in more detail in section 3.1. Though Chickenfoot provides high-level patterns such as keywords and text constraints, it also supports XPath and regular expressions, which users may already be familiar with.

2.4 Modifying Page Content

Of the toolkits described thus far, only WBI, Greasemonkey, and Chickenfoot empower the user to write scripts that change the appearance of a web page in the user's browser. WBI uses a proxy to intercept page requests, letting user-authored Java code mutate either the request or the resulting page before it appears in the user's browser, and Greasemonkey and Chickenfoot can run JavaScript code on a page just after it is loaded in Firefox. Each toolkit lets users manipulate pages with a high-level programming language that ultimately enables the user to seamlessly alter his web browsing experience (though WBI cannot mutate encrypted pages because of its proxy problems).

In addition to manually running scripts, users of all three of these toolkits can write code that will allow their scripts to be triggered automatically upon loading particular web pages. Users can specify whether their script should run on all pages that are loaded in the browser, or only on pages whose URL matches a special pattern. Additionally, both WBI and LiveAgent allow users to schedule scripts or agents to be triggered by time of day rather than by URL.

2.5 Development Environment

One major drawback of most of the aforementioned tools (with the exception of the macro recorders), is that they do not allow scripts to be developed inside the web browser. We consider the ability to experiment with a web site from the script development environment one of the greatest advantages of Chickenfoot – the user does not have to wait to see how it will affect the appearance of the web page because Chickenfoot gives immediate feedback on the rendered page. LAPIS, a predecessor of Chickenfoot, took a similar approach, giving the user an interactive environment in which to experiment with pattern matching and web automation. Unfortunately, the LAPIS web browser does not support web standards like JavaScript, cookies, and secure connections, so it fails to provide the user with a complete web experience.

2.6 Summary

Table 2-1 is a summary of the various web automation toolkits discussed in this section.

		Chickenfoot (2005)	Greasemonkey (2005)	Perl with Mech (2003)	LAPIS (2002)	Screen-Scraper (2002)	WebVCR (2000)	WebL (1998)	WBI (1997)	LiveAgent (1997)
Development	Can develop scripts by experimenting with web page	X			X		X			X
	Can develop scripts by demonstration				X		X			X
	Can develop scripts offline in text editor	X	X	X	X	X		X	X	
Language	Supports keyword patterns	X								
	Supports text constraints in pattern language	X			X					
	Supports regular expressions	X	X	X	X	X		X	X	
	Supports XPath-style expressions	X	X	X		X		X	X	
	Language usually used	Java-Script	Java-Script	Perl	Tcl	N/A	N/A	WebL	Java	N/A
Access	Scripts can run on secure pages	X	X				X			
	Cookies can be used when accessing a page	X	X			X	X		X	X
	Uses a proxy					X			X	X
Features	Scripts can be triggered automatically when a page loads	X	X						X	X
	Scripts can be triggered by time of day								X	X
	Can learn patterns by demonstration				X					

Table 2-1 Comparison of features of existing web automation tools.

Chapter 3 Language Design

Rather than creating an entire language from scratch, I designed Chickenscratch as an extension of the JavaScript programming language [3]. In addition to the technical benefit of being able to take advantage of the existing JavaScript interpreter built into a web browser, this design decision also facilitates the adoption of Chickenscratch by those with web design experience. To that end, Chickenscratch provides a number of JavaScript objects and functions that are familiar to JavaScript web programmers. These are listed in Appendix A.

However, most Chickenfoot users are focused on facilitating the manipulation of web content, which means that users need to be able to programmatically describe elements in the page on a high level. Pure JavaScript provides an interface to a page's DOM, but this is too low-level for end-user programmers. To bridge this gap, Chickenscratch extends JavaScript by adding a pattern matching system for identifying elements in the rendered model. It also provides commands to cut and paste these elements, as well as commands to automate user input to the browser.

3.1 Pattern Matching

Pattern matching is a fundamental operation in Chickenscratch. To operate on a web page component, most commands take a pattern describing that page component.

Chickenscratch supports two kinds of patterns: *keyword patterns* and *text constraint patterns*. A keyword pattern consists of a string of keywords that are searched in the page to locate a page component, followed by the type of the component to be found. For example, `"Search form"` matches a form containing the keyword **Search**, and `"Go button"` matches a button with the word **Go** in its label. The component type is one of a small set of primitive names, including `link`, `button`, `textbox`, `checkbox`, `radiobutton`, `listbox`, and `table`. When a keyword pattern is used by a form manipulation command, the type of page component is implicit and can be omitted. For example, `click("Go")` searches for a hyperlink or button with the keyword "Go" in its label. Case is not significant, so `click("go")` has the same effect.

A text constraint pattern combines a library of primitive patterns (such as `link`, `textbox`, or `paragraph`), literal strings (such as `Go`), and relational operators (e.g., `in`, `contains`, `just` `before`, `just after`, `starts`, `ends`). Text constraint patterns are generally used to identify parts of a page for modification and extraction, although they can also be used for form manipulation.

The `find` command takes a pattern of either kind and searches for it in the current page, e.g.:

```
find("Search form")
find("link in bold")
```

`find` returns a `Match` object which represents the first match to the pattern and provides access to the rest of the matches. Here are some common idioms using `find`:

```
// test whether a pattern matches
if (find(pattern).hasMatch) { ... }

// count number of matches
find(pattern).count

// iterate through all matches
for (m = find(pattern); m.hasMatch; m = m.next) {
    // use m
    ...
}
```

A `Match` object represents a contiguous region of a web page, so it also provides properties for extracting that region. For example, if `m` is a `Match` object, then `m.html` returns the source HTML of the region and `m.text` returns the text of the region without the HTML tags. The complete list of properties for `Match` is listed in Table 3-1.

3.1.1 Other Patterns

`find` actually accepts a number of types, the union of which is called a `Pattern` in Chickenscratch. There are a number of other Chickenscratch commands, such as `insert` and `click`, that also accept a `Pattern` as an argument. Each of the following qualifies as a `Pattern`:

- **Text constraint (TC).** A string whose content is a valid LAPIS pattern. Examples of TCs are `second row in first table` and `3rd Word in Sentence`. See the LAPIS documentation [17] for a complete description of text constraints.
- **Keywords.** A string of keywords that appear in the web page. If a string pattern parses successfully as a TC pattern, then it is interpreted as a TC pattern; otherwise, it is interpreted as a keyword pattern.
- **Match.** `Match` object returned by an earlier call to `find()`. When supplied as an argument to `find()` it will simply return itself; however, it may be useful as an argument to other commands that accept a `Pattern`, such as `replace()`.
- **Node.** A Node in the Document Object Model (DOM) representation of the web page. As Nodes are abstracted by Chickenfoot's rendered model of a page, they are not commonly used by Chickenscratch script authors to define a `Pattern`; however, using a Node as a `Pattern` is supported.
- **Range.** A Range in the DOM. Like Node, it reflects the underlying structure of the page, so its use as a `Pattern` is not favored, but it is supported.

3.1.2 Match as a Search Context

The `find` command is not only a global procedure, but also a method of `Match`. In this way, pattern matching in Chickenscratch can be constrained to a region of a document by finding a `Match` for the desired region and then using its `find` method to restrict the search to the part of the page delimited by the `Match`. This technique could be used to locate rows within a particular table:

```
table = find("third table after first heading")
for (row = table.find("row"); row.hasMatch; row = row.next) {
    ... // use row
}
```

A `Match` can be used as a context for a variety of Chickenscratch commands that take patterns, including the web form commands seen in the examples in Chapter 1. For example, consider a page with multiple fields with the same label:

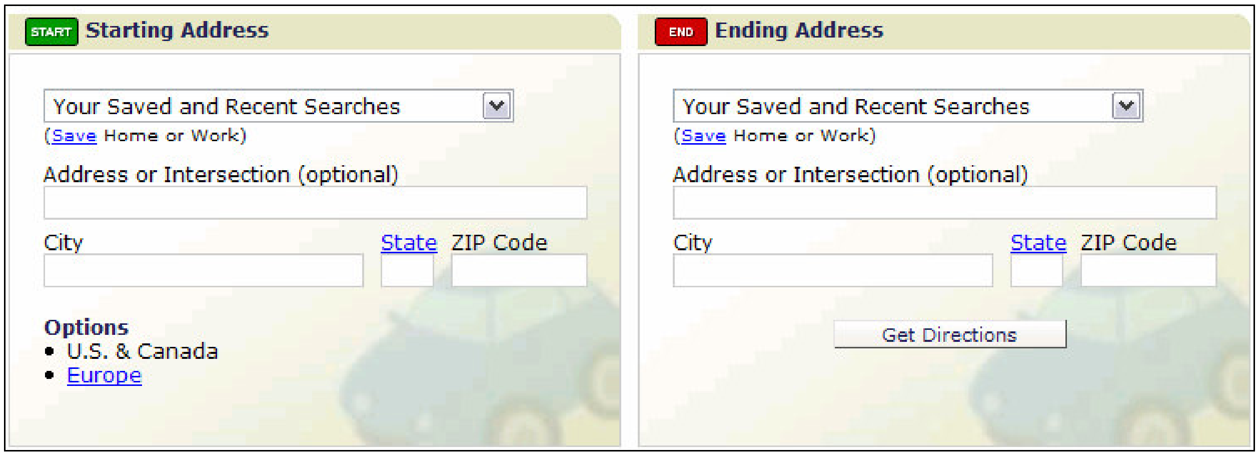


Figure 3.1 Web form for requesting driving directions (www.mapquest.com)

In this case, the command `enter("state", "CA")` would be ambiguous because there are two boxes labeled **State**. This problem can be solved by first matching the appropriate section of a page, and then using it as a context for for subsequent commands:

```
// starting address is context for enter
start = find("starting address table")
start.enter("address", "32 vassar st")
start.enter("zip", 02139")

// ending address is context for enter
end = find("ending address table")
end.enter("address", "1600 amphitheatre parkway")
end.enter("zip", "94043")

// no special context needed because label is unambiguous
click("get directions")
```

Contexts can help users focus on matching patterns in a particular part of a page.

`range`	the DOM Range whose content matched the `Pattern` used to create this object
`next`	(possibly null) reference to the next `Match` in the linked list of matches to the `Pattern`
`hasMatch`	boolean indicating whether this is the empty `Match`
`count`	number of remaining matches in this linked list of matches (including this `Match`)
`index`	0-based index of `Match` within the linked list of matches (undefined for the empty `Match`)
`content`	a DocumentFragment cloned from `range`
`element`	if the content of the `Match` contains exactly a single Element node, then `element` is non-null reference to that Element
`document`	the document that was searched to create this `Match`
`html`	the HTML content of `range`
`text`	the text of `html` that is visible in the rendered web page

Table 3-1 The complete list of properties of the Chickenfoot Match object

3.2 Automation

To be a total web automation system, the user must be able to programmatically fill out web forms and access web pages. This section describes how Chickenscratch is designed to support these operations.

3.2.1 Web Forms

Chickenscratch has a number of commands to automate interactions with a web site. Each command listed in this section can take a `Pattern` to identify the element to be automated.

The `click` command takes a pattern describing a hyperlink or button on the current page and causes the same effect as if the user had clicked on it. For example, these commands click on various parts of the Google home page:

```
click("Advanced Search") // a hyperlink
click("I'm Feeling Lucky") // a button
```

Keyword patterns do not need to match the label of the button or hyperlink exactly, but they do need to be unambiguous. Thus, `click("Lucky")` would suffice to match the I'm Feeling Lucky button, but in this case, `click("Search")` would be ambiguous between the Google Search button and the Advanced Search link, and hence would throw an exception. (Exact matches take precedence over partial matches, however, so if there were a single button labeled "Search," then the `click` command would succeed.) Buttons and links labeled by an image can be matched by keywords mentioned in their ALT text, if any. The keyword matching algorithm is described in more detail in Chapter 7.

The `enter` command enters a value into a textbox. Like `click`, it takes a keyword pattern to identify the textbox, but in this case, the keywords are taken from the textbox's caption or other visible labels near the textbox. Here is a script that logs into Gmail:

```
enter("username", "Michael")
enter("password", "mypasswd")
```

because **Username** and **Password** were the visible labels to the left of the appropriate text boxes.

When the page contains only one textbox in the page, which is often true for search forms, the keyword pattern can be omitted. For example, this sequence does a search on Google:

```
enter("how many bathrooms are there in the white house")
click("Google Search")
```

Checkboxes and radio buttons are controlled by the `check` and `uncheck` commands, which take a keyword pattern that describes the checkbox:

```
check("Yes, I have a password")
uncheck("Remember Me")
```

Finally, the `pick` command makes a selection from a listbox or drop-down box (which are both instantiations of the HTML `<select>` element). The simplest form of `pick` merely identifies the choice by a keyword pattern:

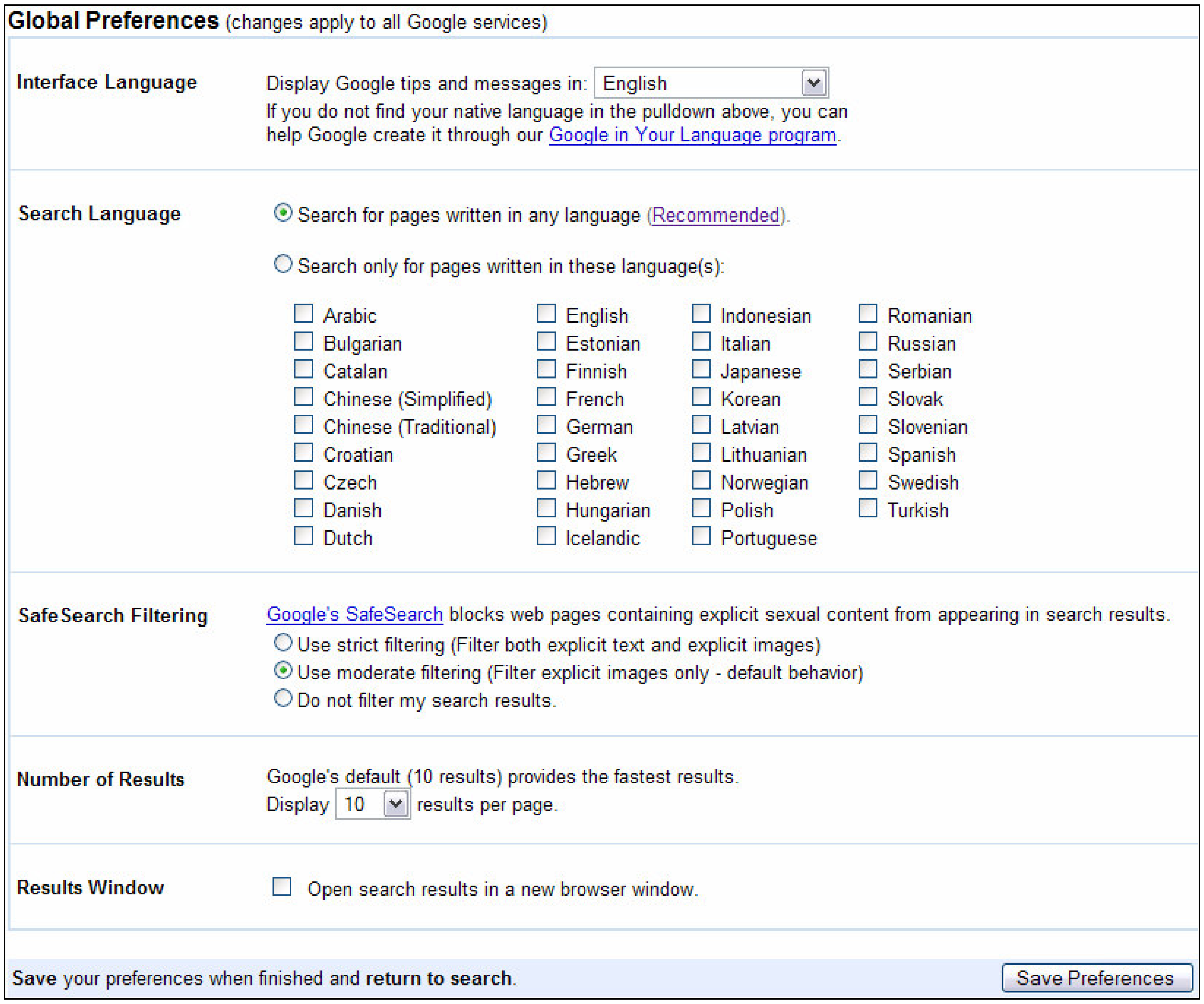
```
pick("California")
```

If only one choice in any listbox or drop-down on the page matches the keywords (the common case), then that choice is made. If the choice is not unique, then `pick` can take two keyword patterns, the first identifying a listbox or dropdown by keywords from its caption, and the second identifying the choice within the listbox:

```
pick("State", "California")
```

All of these commands can be used either as a global procedure or in a context, as they are all methods of `Match`. The following script is an example that exhibits all of the commands in this section to automate the Google preferences page shown in Figure 1.1:

```
go('www.google.com')
click('preferences')
uncheck('search for pages in any language')
check('english')
pick('results per page', '20')
click('save preferences')
```



Global Preferences (changes apply to all Google services)

Interface Language

Display Google tips and messages in: English

If you do not find your native language in the pulldown above, you can help Google create it through our Google in Your Language program.

Search Language

[✓ Search for pages written in any language (Recommended).]

Search only for pages written in these language(s):

☐ Arabic	☐ English	☐ Indonesian	☐ Romanian
☐ Bulgarian	☐ Estonian	☐ Italian	☐ Russian
☐ Catalan	☐ Finnish	☐ Japanese	☐ Serbian
☐ Chinese (Simplified)	☐ French	☐ Korean	☐ Slovak
☐ Chinese (Traditional)	☐ German	☐ Latvian	☐ Slovenian
☐ Croatian	☐ Greek	☐ Lithuanian	☐ Spanish
☐ Czech	☐ Hebrew	☐ Norwegian	☐ Swedish
☐ Danish	☐ Hungarian	☐ Polish	☐ Turkish
☐ Dutch	☐ Icelandic	☐ Portuguese	

SafeSearch Filtering

Google's SafeSearch blocks web pages containing explicit sexual content from appearing in search results.

Use strict filtering (Filter both explicit text and explicit images)

[✓ Use moderate filtering (Filter explicit images only - default behavior)]

Do not filter my search results.

Number of Results

Google's default (10 results) provides the fastest results.

Display 10 results per page.

Results Window

☐ Open search results in a new browser window.

Save your preferences when finished and **return to search**.

Figure 3.2 Google Preferences Page (www.google.com/preferences)

3.2.2 Navigation and Page Loading

Chickenscratch provides a `go` command to navigate to a URL in the current window:

```
go(String url [, Boolean force_reload])
```

The second argument to `go` is an optional reload flag; if true, it indicates that the browser should navigate to the URL even if it is already the current URL being displayed in the browser (effectively forcing a refresh). The reload flag is false by default.

If the `url` input to `go` is not recognized as a well-formed URL, then `http://` is prepended to the `url` before it attempts to navigate to the new page, so either of these commands can be used to load the Google home page in the browser:

```
go('http://www.google.com/')
go('www.google.com')
```

It is also possible to load a page without displaying it in the browser by using the `fetch` command:

```
fetch(String url) // returns an object that delegates calls
                  // to the page's DOM, once it has been loaded
```

Pages accessed by `go` and `fetch` are loaded asynchronously, which means that calls to `go` and `fetch` will return right away; however, any methods invoked on a page before it finishes loading will cause Chickenfoot to hang until the page is loaded.

To avoid locking up Chickenfoot, Chickenscratch has a `ready` command that can test if a page is loaded without invoking one of its methods. `ready` can take one page, or an array of pages, and it will return the first one that is finished loading, or `null` if all of the pages are still downloading:

```
ivy = fetch('fas.harvard.edu') // load the a page in the background
sleep(10)                      // wait for 10 seconds
if (!(doc = ready(ivy)) {
    alert('this site is too slow!')  // complain if it is slow to load
}
```

Other times, the user will want to start downloading a number of pages and process them as they come in. In this case, the user will want to be notified whenever a page is finished downloading. For this, Chickenscratch has a `wait` command that takes a page, or an array of pages, and returns the first page that finishes loading, removing it from the array (if it exists):

```
urls = [url1, url2, url3, ..., urlN]
for (var i = 0; i < urls.length; i++) urls[i] = fetch(urls[i])
while (doc = wait(urls)) {
    ... // process doc
}
```

Chickenscratch also supports the following commands that allow programmatic access to the browser buttons of same name:

```
back()
forward()
reload()
```

3.3 Page Modification

End-users must to be able to insert and remove content in order to to customize a web site. This includes moving content within the page, taking content from other pages, or creating fresh content. Chickenscratch users can do all of this in the context of the rendered model.

3.3.1 Insertions and Deletions

Chickenfoot offers three primitive commands for changing the content of web pages: `insert`, `remove`, and `replace`.

The `insert` command takes two arguments: a location on a page and a fragment of web page content that should be inserted at that location. In its simplest form, the location is a text constraint pattern, and the web page content is simply a string of HTML:

```
	insert("just before textbox", "<b>Search: </b>")
```

The location can also be derived from a `Match` object, but it must represent a single point in the page, not a range of content. The `before` and `after` commands can be used to reduce one of these objects to a point:

```
	t = find("textbox")
	insert(after(t), "<b>Search: </b>")
```

The page content to be inserted can also be a `Match` object, allowing content to be extracted from another page and inserted in this one:

```
	map = googlemaps.find("image")
	insert("just after Directions", map)
```

The `remove` command removes page content identified by its argument, which can be a text constraint pattern or `Match` object. For example:

```
	remove("Sponsored Links cell")
```

The `replace` command replaces one chunk of page content with another. It is often used to wrap page content around an existing element:

```
	discount = find("10% off")
	replace(discount, "<b>***" + discount + "***</b>")
```

The exact definitions for these functions are as follows:

```
insert(Position position, Chunk chunk) // returns a Match
remove(Pattern pattern)                // returns a Position
replace(Pattern, Chunk chunk)          // returns a Match
```

Like `Pattern`, a `Chunk` is a union of types rather than its own type. Each of the following qualifies as a `Chunk` in Chickenfoot:

- **String.** The text of the string will be interpreted as HTML if there is HTML markup present; otherwise, it will be interpreted as plaintext.
- **Match.** Same as in section 3.1.1.
- **Node.** Same as in section 3.1.1.
- **Range.** Same as in section 3.1.1.
- **Link or Button.** These are special `Chunks` that are defined in the next section.

A `Position` is a `Pattern` that identifies a single point in the web page. Not every `Pattern` identifies a single point in a web page; on the contrary, a `Pattern` often refers to a nonempty region of a web page rather than an individual point. However, `before` and `after` can always be used to produce a `Position` from a `Pattern`:

```
before(Pattern pattern) // returns a Position at the start of the pattern
after(Pattern pattern)  // returns a Position at the end of the pattern
```

If the `Pattern` passed to `insert` is not a `Position`, then `insert` will throw an `Error`.

The `Match` returned by `insert` is a reference to the content that was actually inserted into the page that the client can use it as a point of reference for future insertions or deletions. Similarly, `remove` returns a `Position` where the deletion took place so the user can have a reference to it if he did not have one already.

The API for insert and remove make it trivial to implement `replace`:

```
replace(pattern, chunk) ::= insert(remove(pattern), chunk)
```

Nevertheless, `replace` is included as part of Chickenscratch to improve the readability of scripts.

Finally, although `delete` would be a better name for the command that serves as the complement of `insert`, `delete` is a JavaScript keyword, so it would not be possible to define it as a function in Chickenscratch.

3.3.2 Special Chunks: Link and Button

When a Chickenscratch script needs to present a user interface, it can create links and buttons and insert them directly into a web page. Input buttons are created by the `Button` constructor, which takes a label for the button and an `Action` to execute when it is clicked:

```
      showAll = function() { ... }
      button = new Button ("Show All", showAll)
      insert(position, button)
```

An `Action` is either a JavaScript Function to be executed with no arguments, or a string whose content is a valid Chickenscratch script to be evaluated. It is important to realize that this is not the same as doing this:

```
      insert(position, '<input onclick="showAll" value="Show All">')
```

The difference is that the JavaScript code launched by the `onclick` attribute will be run in the browser's security model, which does not have full access to the browser, the user's filesystem, or the network. By contrast, an `Action` passed to the Button constructor will be run at a *privileged level*, giving the script a level of access comparable to that of any desktop application.

There is a `Link` constructor that is analogous to `Button` that takes a chunk of HTML to display inside the hyperlink:

```
      surprise = function() { ... }
```

```
    new Link("<b>What do I do?</b>", surprise)
    insert(position, surprise)
```

and there is also an `onClick` function to associate an `Action` with any `Pattern` on the page:

```
    onClick("table", "alert('you clicked on the table!')")
```

Chapter 4 Applications

This chapter describes a few of the applications that have been built using Chickenfoot.

4.1 Adding File Type Icons to Links

When a hyperlink points at a resource other than a web page (such as a PDF document, a ZIP archive, or a Word document), it is often helpful for the link to be visually distinguished – first, because the user may be actively scanning the page for one of these resources, and second, because they may want to avoid them while casually surfing. Only a few web sites provide a visual cue to the file type of a hyperlink. TargetAlert is a Firefox browser extension that I developed that adds file type icons to hyperlinks on any web site.

TargetAlert was originally written in 217 lines of Javascript and XUL. I rewrote it in 29 lines of Chickenfoot. The essence of the script is the following loop:

```
for (link = find('link'); link.hasMatch; link = link.next) {
  href = link.element.getAttribute('href')
  if (m = href.match(/\.(\w+)$/)) {
    extension = m[1]
    src = 'moz-icon://.' + extension + '?size=16';
    insert(after(link), ' <img src="' + src + '"> ')
  }
}
```

The script works by finding every hyperlink in the page and inspecting the URL of its destination. It uses a simple regular expression to extract the file extension, indicating the type of file that the URL points to. In creating the file type icon, the script exploits a feature of Firefox that works only on Windows: URLs of the form `moz-icon://.ext?size=16` return the icon associated with file extension `.ext` in the Windows registry. (Firefox normally uses these URLs to display local directories in the browser.) Using the `moz-icon` protocol, it is simple to get the icon for each file type, so the script uses this trick to insert an image that displays the appropriate icon after each link. The result of amending these links with images is shown in Figure 4.1.

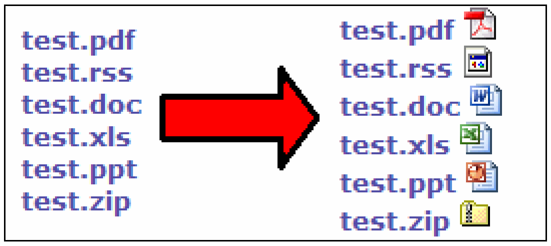


Figure 4.1 TargetAlert

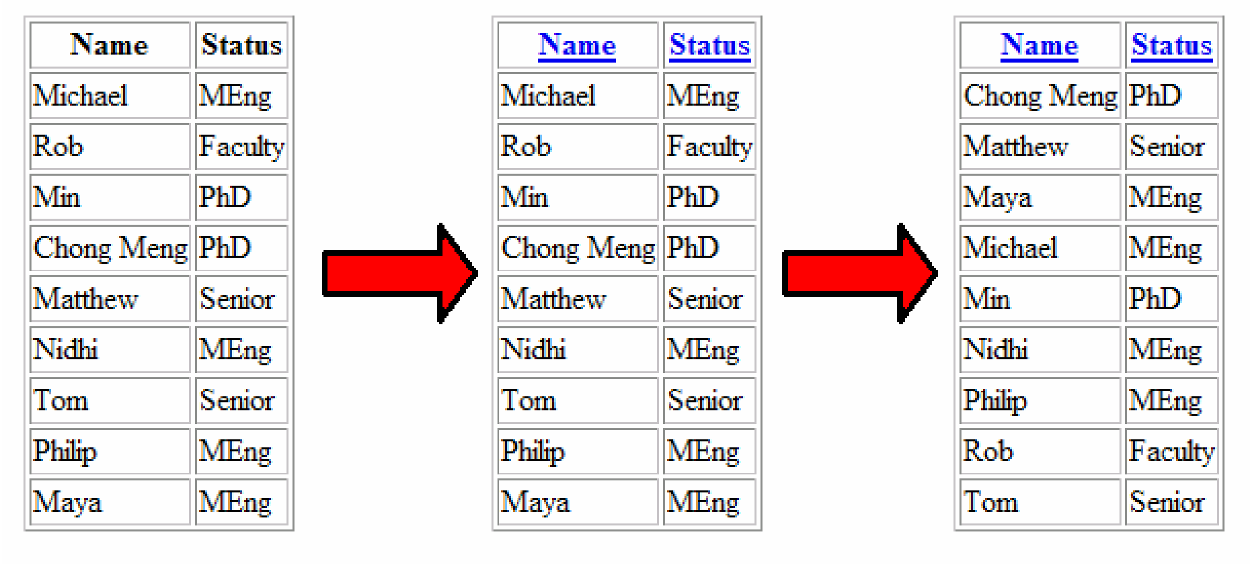
4.2 Sorting Tables

Another feature that some web sites have, but many lack, is the ability to sort a table of data by clicking one of its column headers. A Chickenfoot script can add this functionality automatically to most tables by replacing every table header cell it finds with a link that sorts the table by that column.

Most of the script is concerned with managing the sort, but here is the part that replaces headers with links:

```
for (var table = find('table'); table.hasMatch; table = table.next) {
  var heading = table.find('first row')
  for (var h = heading.find('text in cell'); h.hasMatch; h = h.next) {
    var sorter = makeRowSorter(table.index, h.index)
    replace(h, new Link(h.text, sorter))
  }
}
```

The `makeRowSorter` function returns a function that sorts the specified table by the specified column number. It does this by copying every cell in the column to be sorted into a temporary array, and then uses JavaScript's built-in quicksort function to sort the array. Because the order of the cells in the temporary array reflects the order that the rows should have when sorted, it uses a map from the sorted cells to their rows to create a new array that contains the rows in sorted order. The last step is to iterate over this sorted array of rows and replace the *i*th row in the table with the *i*th element of the array. The results of this script can be seen in Figure 4.2.



Name	Status
Michael	MEng
Rob	Faculty
Min	PhD
Chong Meng	PhD
Matthew	Senior
Nidhi	MEng
Tom	Senior
Philip	MEng
Maya	MEng

Name	Status
Michael	MEng
Rob	Faculty
Min	PhD
Chong Meng	PhD
Matthew	Senior
Nidhi	MEng
Tom	Senior
Philip	MEng
Maya	MEng

Name	Status
Chong Meng	PhD
Matthew	Senior
Maya	MEng
Michael	MEng
Min	PhD
Nidhi	MEng
Philip	MEng
Rob	Faculty
Tom	Senior

Figure 4.2 Table sorting demo: First the script adds headers to the column and then the user can click on a header to sort the column. Here, the user clicked on the header of the first column.

4.3 *Concatenating a Sequence of Pages*

Search results and long articles are often split into multiple web pages, mainly for faster downloading. This can inhibit fluid browsing, however, because the entire content is not accessible to scrolling or to the browser's internal Find command. Some articles offer a link to the complete content, intended for printing, but this page may lack other useful navigation.

Matthew Webber [24] has written a Chickenfoot script that detects a multi-page sequence by searching for its table of contents (generally a set of numbered page links, with Next and Previous). When a table of contents is found, the script automatically adds a **Show All** link to it (Figure 4.3). Clicking this link causes the script to start retrieving additional pages from the sequence, appending them to the current page. In order to avoid repeating common elements from subsequent pages (such as banners, sidebars, and other decoration), the script uses a conservative heuristic to localize the content, based on searching for an HTML element that includes both the table of contents and the vertical midpoint of the page. The content element from each subsequent page is inserted after the content element of the current page.

```
function showAll() {
  var mostRecentNode = getPageContent()
  insert(after(mostRecentNode), "NEXT INSERT")
  for (var m1 = find("numberedlink in (first multipage in [body])");
           m1.hasMatch;
           m1 = m1.next) {
      openTab()
      go(m1.element.getAttribute("href"))
      importNode = getPageContent().cloneNode(true)
      closeTab()
      insert(before("NEXT INSERT"), importNode)
      mostRecentNode = importNode
    }
}
```

The `showAll` function gets executed when the user clicks on the **Show All** link. It locates the table of contents and the links within it using the LAPIS patterns NumberedLink and Multipage, respectively. These patterns were created by Webber. Once `showAll` has the table of contents, it iterates over each link in the table of contents, makes a connection to it in a new tab window in Firefox, gets its content, and inserts it to the original page.

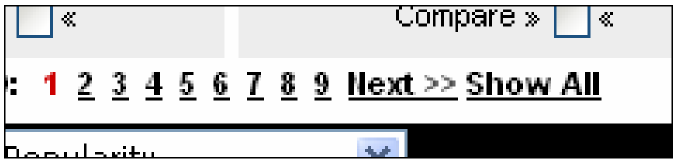


Figure 4.3 A "Show All" link embedded after a series of sequential links. Note that this link has the same style of the surrounding links, so it appears like a natural part of the page. Clicking this link will cause the browser to start downloading the other links shown here and concatenating their content to the current web page.

4.4 Coloring Java Syntax and Linking to Documentation

The text constraint patterns used by Chickenfoot can draw upon the rich library of patterns and parsers implemented in LAPIS. Philip Rha's recent work [25] in using LAPIS to detect snippets of other languages in documents with mixed syntax has made possible it to use LAPIS's Java parser to find and parse Java syntax even if it is embedded in a web page. This Chickenfoot script uses this parser for coloring embedded Java syntax:

```
for (c = find('Java.Comment'); c.hasMatch; c = c.next) {
  replace(c, '<span style="color:green">' + c + '</span>')
}
```

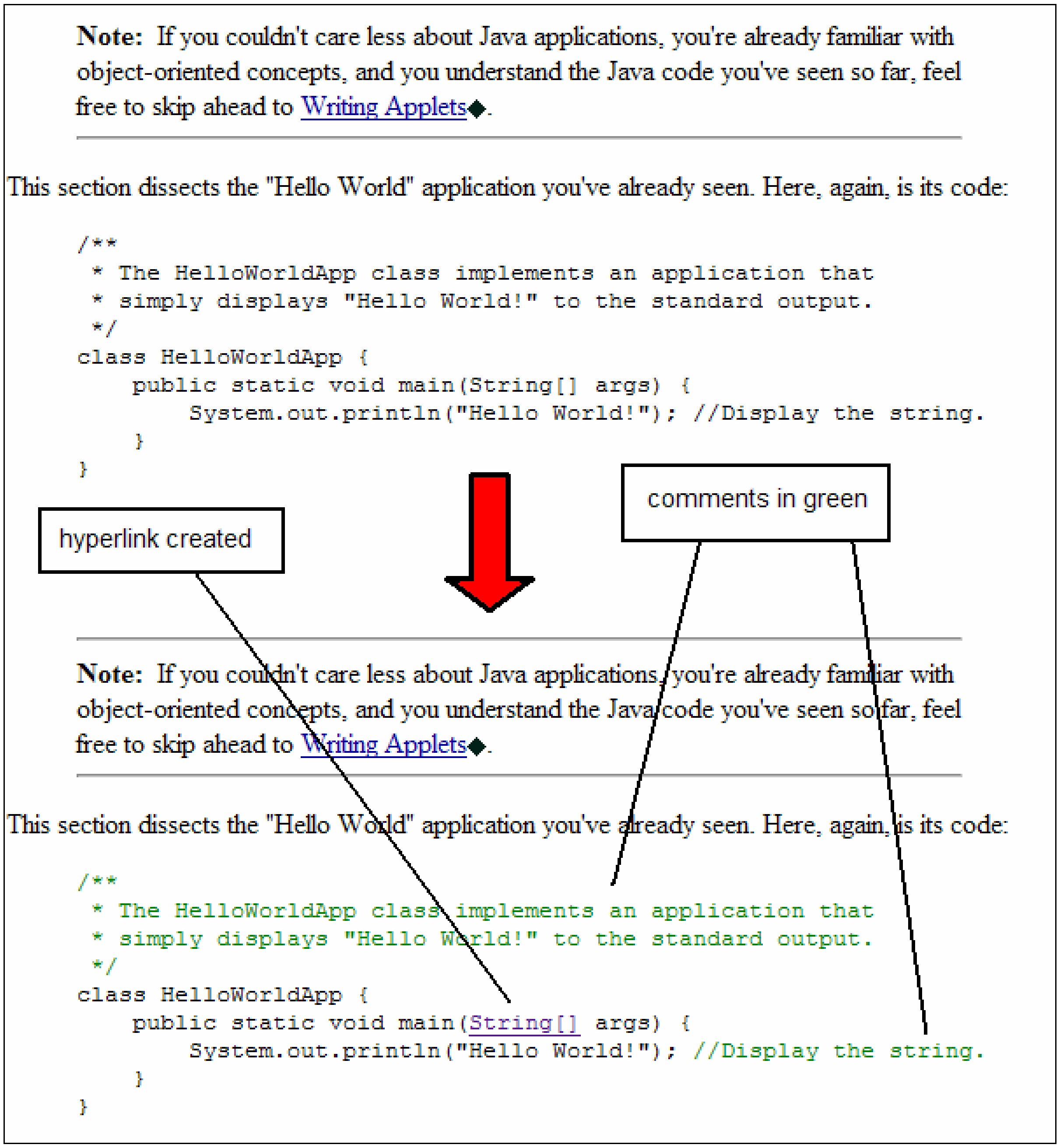
The script also links each occurrence of a class name to its Javadoc documentation:

```
for (c = find('Java.Type'); c.hasMatch; c = c.next) {
  if (c.text in classURL) {
    replace(c, '<a href="' + classURL[c.text] + '">' + c.text + '</a>')
  }
}
```

The effects of these scripts can be seen in Figure 4.4. Also, the `classURL` mapping in the script above maps a Java class name, such as `String`, to its Javadoc URL. This mapping is extracted from a Javadoc web site using Chickenfoot:

```
go("java.sun.com/j2se/1.5.0/docs/api")
click("No Frames")
click("All Classes")
for (link = find('link'); link.hasMatch; link = link.next) {
  classURL[link.next] = link.element.href
}
```

These scripts mutate the page by simply wrapping each match to the Java parser with the appropriate style or hyperlink.


Note: If you couldn't care less about Java applications, you're already familiar with object-oriented concepts, and you understand the Java code you've seen so far, feel free to skip ahead to Writing Applets◆.

This section dissects the "Hello World" application you've already seen. Here, again, is its code:

```
/**
 * The HelloWorldApp class implements an application that
 * simply displays "Hello World!" to the standard output.
 */
class HelloWorldApp {
    public static void main(String[] args) {
        System.out.println("Hello World!"); //Display the string.
    }
}
```

Note: If you couldn't care less about Java applications, you're already familiar with object-oriented concepts, and you understand the Java code you've seen so far, feel free to skip ahead to Writing Applets◆.

This section dissects the "Hello World" application you've already seen. Here, again, is its code:

```
/**
 * The HelloWorldApp class implements an application that
 * simply displays "Hello World!" to the standard output.
 */
class HelloWorldApp {
    public static void main(String[] args) {
        System.out.println("Hello World!"); //Display the string.
    }
}
```

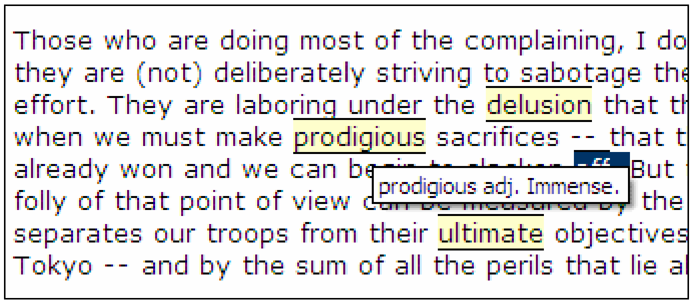
Figure 4.4 LAPIS Java-snippet parser used in Chickenfoot to hyperlink to Javadoc API and syntax-highlight Java comments

4.5 Highlighting Vocabulary Words

Students studying for college placement exams, such as the SAT, often work hard to expand their vocabulary. One way to make this learning deeper is to highlight vocabulary words while the student is reading, so that the context of use reinforces the word’s meaning. One of my Chickenfoot scripts takes a list of vocabulary words and definitions (posted on the web) and automatically highlights matching words in any page that the user browses. The script uses a title attribute to pop up the word’s definition as a tooltip if the mouse hovers over it as shown in Figure 4.5.

```
for (word = find('word'); word.hasMatch; word = word.next) {
  if (word.text in vocab) {
    html = '<span style="background-color: yellow" title="'
          + vocab[word.text] + '">'
          + word + '</span>'
    replace(word, html)
  }
}
```

Like the Java Syntax Coloring script, the Vocab Word script finds matches to a pattern in a web page and uses CSS styles to draw attention to the matches.


Those who are doing most of the complaining, I do
they are (not) deliberately striving to sabotage th
effort. They are laboring under the delusion that th
when we must make prodigious sacrifices -- that t
already won and we can be But
folly of that point of view c the
separates our troops from their ultimate objectives
Tokyo -- and by the sum of all the perils that lie a

prodigious adj. Immense.

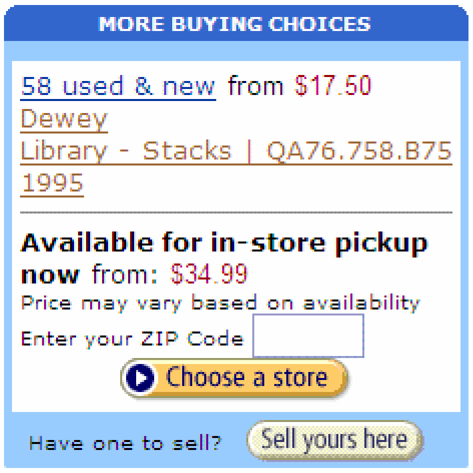
Figure 4.5 User viewing definition of **prodigious** as a tooltip after running Vocabuarly script.

4.6 Integrating a Bookstore and a Library

The last example is a short script that augments book pages found in Amazon with a link that points to the book's location in the MIT library:

```
isbn = find('number just after isbn')
with (fetch('libraries.mit.edu/')) {
  pick('Keywords’);
  enter(isbn)
  click('Search')
  link = find('link just after Location')
}
// back to Amazon
if (link.hasMatch) {
  insert(before('first rule after "Buying Choices"'), link.html)
}
```

The script extracts the ISBN number from the book's page on Amazon using `find`. Then it fetches the MIT library page and fills its search form using `pick` and `enter`. `click` is used to submit the search request, and when the search results page loads, the script uses `find` to extract a hyperlink to the book's availability and uses `insert` to slip the link into the Amazon page. The final product of this script is shown in Figure 4.6.



MORE BUYING CHOICES

58 used & new from $17.50
Dewey
Library - Stacks | QA76.758.B75
1995

Available for in-store pickup now from: $34.99
Price may vary based on availability
Enter your ZIP Code

Figure 4.6 Book availability in MIT Library inserted among Amazon purchasing options.

Chapter 5 User Interface Design

Embedding the Chickenfoot development environment inside a popular, modern web browser is a key element of its design. If Chickenfoot were a standalone application, then it would be difficult for end-user programmers to write scripts because the site to be scripted may not be in view. Further, it would reduce the spontaneity of Web scripting because the user may be loath to start another application when he is in the middle of doing something in his browser – if the user encounters a problem from within the browser that could be solved by end-user scripting, then he should be able to solve the problem from the browser. Creating a special web browser to contain the development environment, as LAPIS and Haystack [26] do, also suffers from the "reduced spontaneity" problem. What's worse with these instrumented browsers is that users expect the same level of support for their bookmarks, plugins, etc., as they have in their preferred browser; however, such support is often deficient because it is not a priority for the developers.

5.1 Layout Decisions

Embedding a development environment into a web browser is a challenge because it needs to have enough screen real estate to be a useful tool without taking up so much space that it interferes with the user's browsing. Chickenfoot is implemented as a sidebar, just as History and Bookmarks are in most web browsers, so it takes up no more space than other common sidebars. This also means that the development environment can be hidden when it is not needed, but that it can be opened quickly, encouraging spontaneous scripting.

As the user's main goal will be script development, the editor for writing the script is the top half of the sidebar. Tools to help with script development are in the bottom half of the sidebar. Each tool is used independently, so they are grouped together in a tabbed pane so that only one tool is visible at a time. This ensures that each tool has as much screen space as possible, and that the editor is always in view when a tool is being used.

The interface is implemented in XUL [27], as that is the standard windowing toolkit for Firefox. Using XUL ensures internal consistency with the rest of the Firefox UI.

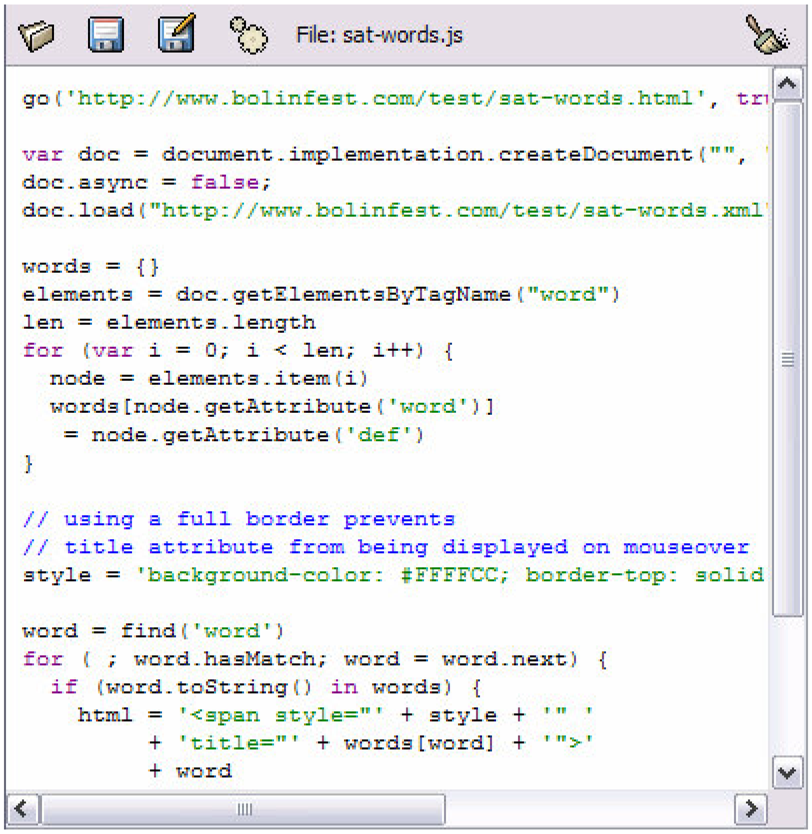
5.2 Panel Design

The sidebar is divided into two panels. The Editor panel appears on top and contains a toolbar and the script editor. The Tools panel appears on bottom and contains four panes, each of which contains a tool for script development. This section describes each of these components, with the exception of the Triggers pane which is described in the next section.

5.2.1 Editor Panel

As shown in Figure 5.1, the top of the interface contains a toolbar with iconified buttons that run standard file input-output commands: Open, Save, and Save As. There is also a Run button that executes the current script and a Clear button to clear the editor. The Clear button is placed away from the other buttons to reduce the chance that it is clicked by accident. A toolbar was chosen instead of a menubar because it would be the second menubar in the interface, far from the top of the browser window with its own File menu, which would be inconsistent with the way menubars are used in other desktop applications.

The script editor appears below the toolbar. Because there is no room for standard Edit menu commands in the toolbar, they are available in a context menu when the user right-clicks in the editor. The standard keyboard shortcuts for the Edit commands work in the editor as well, so these commands should be learnable even if they are not visible. The editor also supports syntax highlighting to help reduce syntax errors.

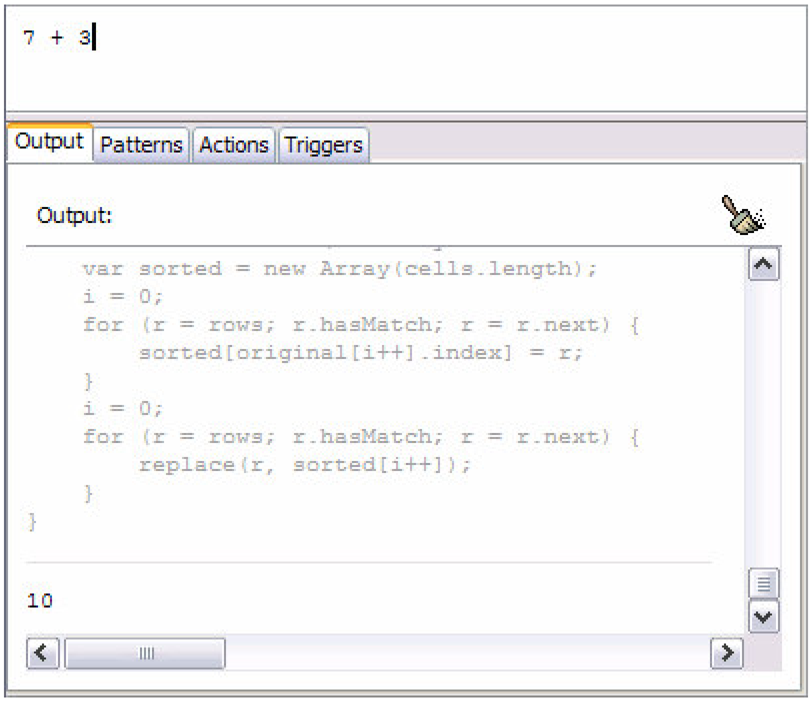


```
go('http://www.bolinfest.com/test/sat-words.html', tr

var doc = document.implementation.createDocument("", '
doc.async = false;
doc.load("http://www.bolinfest.com/test/sat-words.xml'

words = {}
elements = doc.getElementsByTagName("word")
len = elements.length
for (var i = 0; i < len; i++) {
  node = elements.item(i)
  words[node.getAttribute('word')]
   = node.getAttribute('def')
}

// using a full border prevents
// title attribute from being displayed on mouseover
style = 'background-color: #FFFFCC; border-top: solid

word = find('word')
for ( ; word.hasMatch; word = word.next) {
  if (word.toString() in words) {
    html = '<span style="' + style + '" '
         + 'title="' + words[word] + '">'
         + word
```

Figure 5.1 Editor Panel

5.2.2 Tools Panel

The first pane in the tools panel is the Output pane, which is analogous to standard out and standard error in other programming systems. The user can write to the Output pane using the Chickenscratch command `output()`, which takes a variable number of arguments and prints each argument to the Output pane, in order. If a Chickenfoot script throws an error, then the error will also be printed in the Output pane. Values written to Output during the current execution of the script appear in black whereas values from previous runs appear in gray. This makes it easier to distinguish new output from old output.

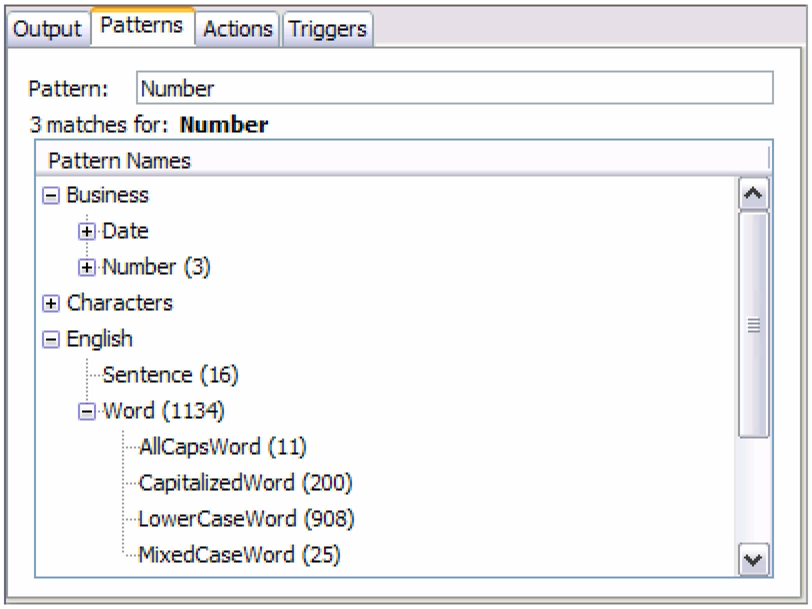


```
7 + 3
```

```
    var sorted = new Array(cells.length);
    i = 0;
    for (r = rows; r.hasMatch; r = r.next) {
        sorted[original[i++].index] = r;
    }
    i = 0;
    for (r = rows; r.hasMatch; r = r.next) {
        replace(r, sorted[i++]);
    }
}

10
```

Figure 5.2 Output Pane

The next two panes, Patterns and Actions, are tools that aid in development, but also attempt to increase the learnability of the system. The Patterns pane presents the user with matches to a predefined list of LAPIS patterns that Chickenfoot has found in the page. This introduces the user to patterns that he may not have realized were supported in Chickenfoot, such as `EmailAddress`. The Patterns pane also lets the user type in a pattern and see if it matches anything in the page.

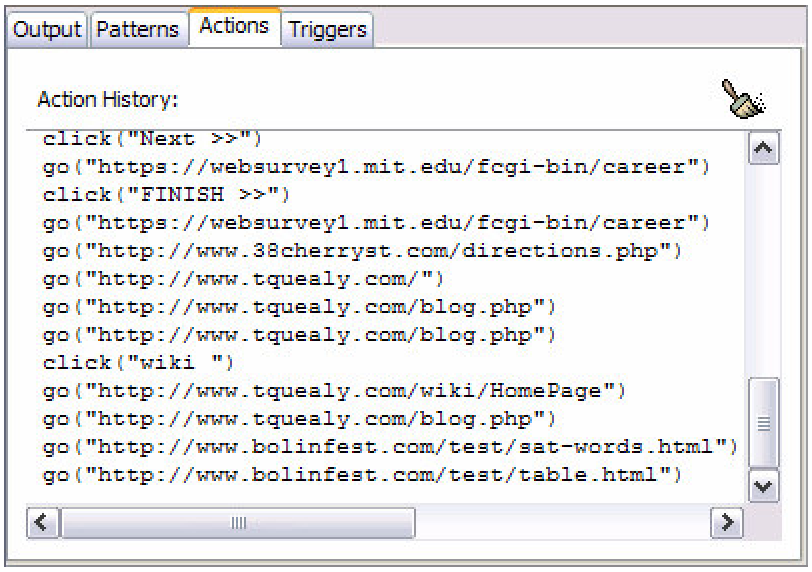


Pattern: Number
3 matches for: **Number**

Pattern Names
- Business
  - Date
  - Number (3)
- Characters
- English
  - Sentence (16)
  - Word (1134)
    - AllCapsWord (11)
    - CapitalizedWord (200)
    - LowerCaseWord (908)
    - MixedCaseWord (25)

Figure 5.3 Patterns Pane

The Actions pane keeps a log of the user's actions in the browser: clicking on links, checking radio buttons, etc. This log is formatted as a list of Chickenfoot commands. The goal is that a user can watch what appears in the Actions pane as he browses to learn what Chickenfoot code he should write to automate what he just did. In this way, Chickenfoot can act as a macro recorder, but unlike existing recorders, it indiscriminately records all actions instead of requiring the user to start and stop the recorder. This lets users can go back and retrieve a copy the transcript later, even though they may not have realized that such a log would be valuable at the time it was recorded.



Action History:

```
click("Next >>")
go("https://websurvey1.mit.edu/fcgi-bin/career")
click("FINISH >>")
go("https://websurvey1.mit.edu/fcgi-bin/career")
go("http://www.38cherryst.com/directions.php")
go("http://www.tquealy.com/")
go("http://www.tquealy.com/blog.php")
go("http://www.tquealy.com/blog.php")
click("wiki ")
go("http://www.tquealy.com/wiki/HomePage")
go("http://www.tquealy.com/blog.php")
go("http://www.bolinfest.com/test/sat-words.html")
go("http://www.bolinfest.com/test/table.html")
```

Figure 5.4 Actions Pane

Although the current implementation of Chickenfoot is not as reliable as LiveAgent or WebVCR in recording every user action, the recorded transcript is more accessible in Chickenfoot than it is in these tools, making it easier for end-users to edit and understand. Improving Chickenfoot's recording capability is future work.

5.3 Trigger Design

For a user to seamlessly integrate Chickenfoot automations and customizations into his browser, he should be able to trigger Chickenfoot scripts by his ordinary browsing habits. Chickenfoot is designed so that it can run a script automatically when a user navigates to a URL, even if the Chickenfoot sidebar is not currently visible.

5.3.1 Defining Triggers

A user can define a collection of URLs that can trigger a script. Because a URL may trigger multiple scripts, the user must also impose a total order on the triggers so that Chickenfoot can run them sequentially. The alternative would be for Chickenfoot to try to run all scripts that matched a trigger in parallel; however, this would likely lead to concurrency issues.

Because Chickenfoot is designed for end-user programmers, asking users to provide a regular expression to determine which URLs should trigger a script is too technical. Instead, Chickenfoot uses the simple pattern matching scheme for URLs used by the Adblock Firefox extension [28]. This scheme asks the user for a URL that may contain asterisks as wildcards, and uses it to produce a regular expression for matching URLs. To convert the URL to a regexp, it escapes all of the special regular expression characters with backslashes (such as periods and question marks) and replaces asterisks with the dot-star repeat operator. It also adds appropriate start and end anchors, and makes the regexp case-insensitive. For example, if the user provides:

```
http://*.sun.com/*
```

then the regular expression produced to match this pattern will be:

```
/^http:\/\/.*\.sun\.com\/.*$/i
```

This regexp will match these URLs:

```
http://www.sun.com/
http://java.sun.com/
http://java.sun.com/tutorial/index.html
```

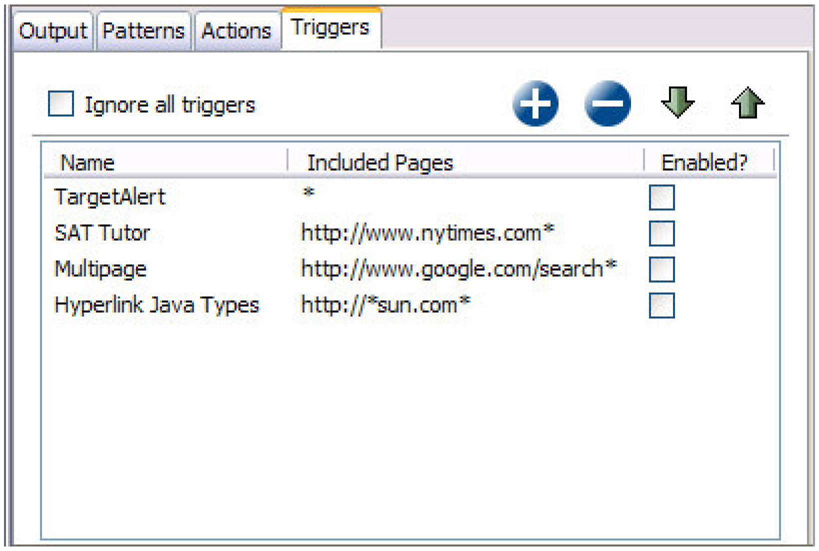
But not these:

```
http://www.sunsets.com/
http://java.sun.net/
```

This scheme aims to be simple and to meet user expectations. Greasemonkey also uses this scheme to define URL triggers.

5.3.2 Triggers Pane

The Triggers pane is shown in Figure 5.5. It has a list of the triggers that the user has configured. Each item in the list shows the trigger's name, its URL pattern, and whether it is currently enabled. From this pane, the user can add or remove triggers, temporarily disable or enable triggers, or edit the name of a trigger. There is a separate checkbox for globally disabling all of the triggers if the user wants to disable Chickenfoot temporarily without losing his current settings in the **Enabled?** column.



Name	Included Pages	Enabled?
TargetAlert	*	☐
SAT Tutor	http://www.nytimes.com*	☐
Multipage	http://www.google.com/search*	☐
Hyperlink Java Types	http://*sun.com*	☐

Figure 5.5 Trigger pane

Chapter 6 Keyword Pattern Survey

One of the novel aspects of Chickenfoot is the use of keyword patterns to identify page elements, such as `"Search button"` and `"address textbox."` A similar technique is used by Google to associate search terms with pictures on the Web, and the success of Google Image Search is testament to the viability of this approach. However, image elements often have obvious labels, in the form of ALT or TITLE attributes, making it easier to deduce names for these images.

I was interested in testing this approach for naming web froms because I wanted Chickenfoot users to be able to write scripts that could uniquely identify form elements without having to look up the web site's name for the element. Another possibility I considered was trying to automatically produce logical names for form elements and inserting them into the web page, near the element, providing users with names for elements that they could find in the rendered model. Unfortunately, this seemed even harder than resolving keyword patterns because the space of names to consider is so large. However, the converse is a more tractable problem because on any given web page, the number of input elements is relatively small, making the problem of resolving a user-provided name to a web form much more tractable.

To explore the usability of this technique when applied to web forms, I conducted a small study to learn what kinds of keyword patterns users would generate for one kind of page component (textboxes), and whether users could comprehend a keyword pattern by locating the textbox it was meant to identify. The results collected in this survey were used as training data to motivate the algorithm used to resolve keyword patterns in Chickenfoot. The algorithm's procedure and its performance on the training data is explained in the next chapter.

6.1 Method

The study was administered anonymously over the Web. It consisted of three parts, always in the same sequence. Part 1 explored freeform generation of names: given no constraints, what names would users generate? Each task in Part 1 showed a screenshot of a web page with one textbox outlined in red, and asked the user to supply a name that "uniquely identified" the marked textbox. Users were explicitly told that spaces in names were acceptable. Part 2 tested comprehension of names that we generated from visible labels. Each task in Part 2 presented a name and a screenshot of a web page, and asked the user to click on the textbox identified by the given name. Part 3 repeated Part 1 (using fresh web pages), but also required the name to be composed only of "words you see in the picture" or "numbers" (so that ambiguous names could be made unique by counting, e.g. "2nd Month").

The whole study used 20 web pages: 6 pages in Part 1, 8 in Part 2, and 6 in Part 3. The web pages were taken from popular sites, such as the Wall Street Journal, the Weather Channel, Google, AOL, MapQuest, and Amazon. Pages were selected to reflect the diversity of textbox labeling seen across the Web, including simple captions (Figure 6.1a), wordy captions (Figure 6.1b), captions displayed as default values for the textbox (Figure 6.1c), and missing captions (Figure 6.1d). Several of the pages also posed ambiguity problems, such as multiple textboxes with similar or identical captions.

Subjects were unpaid volunteers recruited from the university campus by mailing lists. Forty subjects participated (20 females, 20 males), including both programmers and nonprogrammers (24 reported their programming experience as "some" or "lots," 15 as "little" or "none," meaning at most one programming class). All but one subject were experienced web users, reporting that they used the Web at least several times a week.

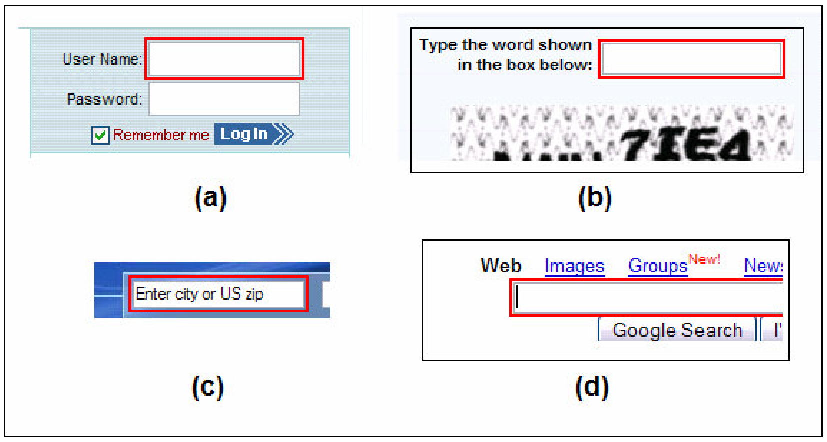


Figure 6.1 Examples of textboxes used in the web survey

6.2 Results

We analyzed Part 1 by classifying each name generated by a user into one of four categories:

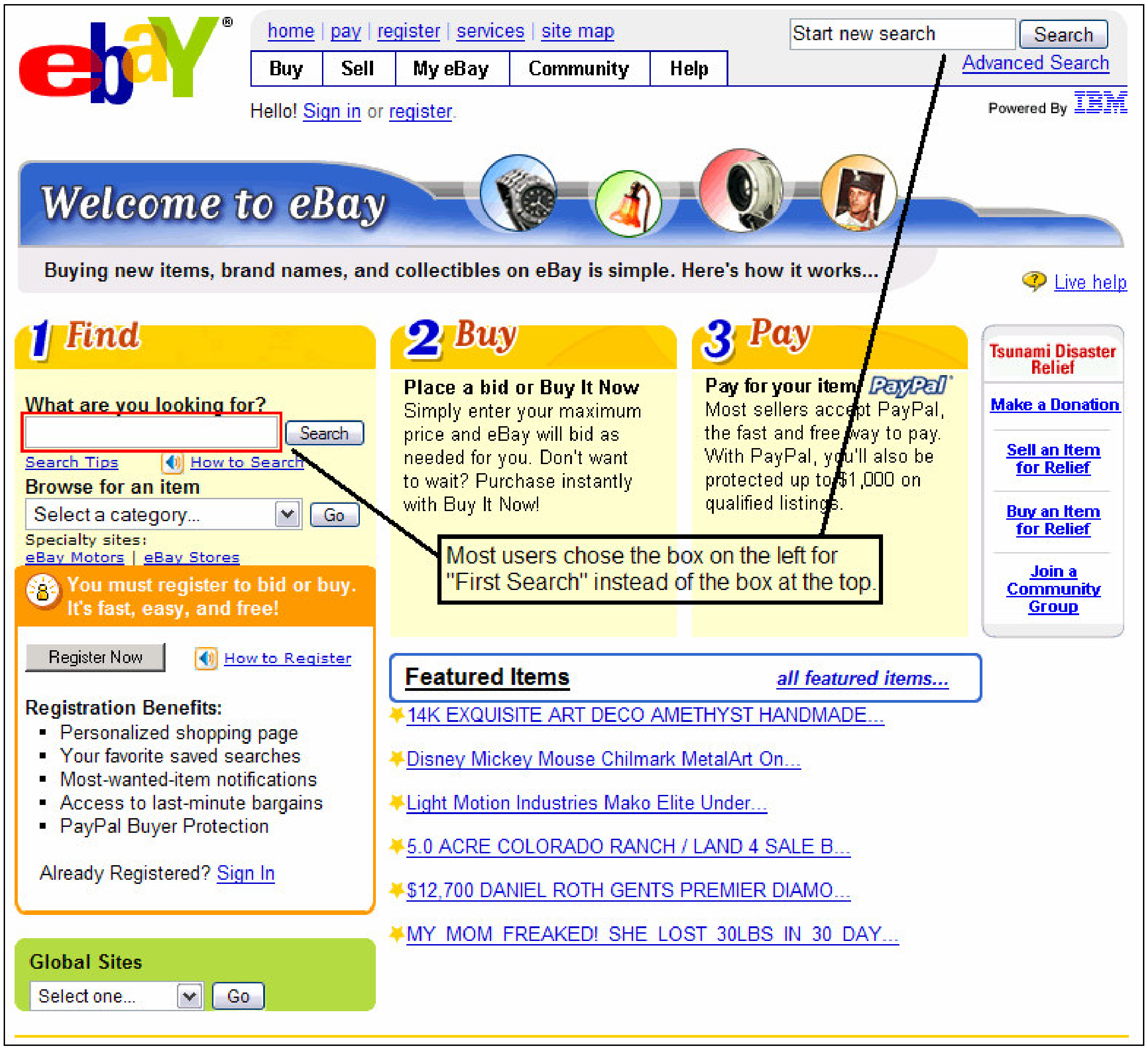
- *Visible* if the name used only words that were visible somewhere on the web page (e.g., "User name" for Figure 6.1a)
- *Semantic* if at least one word in the name was not found on the page, but was semantically relevant to the domain (e.g., "login name");
- *Layout* if the name referred to the textbox's position on the page rather than its semantics (e.g., "top box right hand side")
- *Example* if the user used an example of a possible value for the textbox (e.g. "johnsmith056").

About a third of the names included words describing the type of the page object, such as "field," "box," "entry," and "selection;" we ignored these when classifying a name. The prevalence of "field" and "box" was a significant result in motivating the design of keyword patterns because it suggested that these words should be *ignored* for a keyword pattern to `enter()` where the system already knows that it is looking for a textbox, but that these words should be treated as *special identifiers* for a keyword pattern to `find()` because it indicates the type of element that the user is trying to match rather than keywords to match in the page. Indeed, supporting keyword patterns that end with `link`, `button`, `textbox`, etc. was a direct result of this survey.

Two users consistently used *example* names throughout Part 1; no other users did. (It is possible these users misunderstood the directions, but since the study was conducted anonymously over the Web, it was hard to ask them.) Similarly, one user used *layout* names consistently in Part 1, and no others did.

The remaining 37 users generated either *visible* or *semantic* names. When the textbox had an explicit, concise caption, *visible* names dominated strongly (e.g., 31 out of 40 names for Figure 6.1a were visible). When the textbox had a wordy caption, users tended to seek a more concise name (so only 6 out of 40 names for Figure 6.1b were visible). Even when a caption was missing, however, the words on the page exerted some effect on users' naming (so 12 out of 40 names for Figure 6.1d were visible). This was a promising result in that when forced to name an unabeled element, one quarter of users still opted to use the page author's words to come up with a name rather than their own.

Part 2 found that users could flawlessly find the textbox associated with a visible name when the name was unambiguous. When a name was potentially ambiguous, users tended to resolve the ambiguity by choosing the first likely match found in a visual scan of the page. When the ambiguity was caused by both visible matching and semantic matching, however, users tended to prefer the visible match: given "City" as the target name for go.com, 36 out of 40 users chose one of the two textboxes explicitly labeled "City;" the remaining 4 users chose the "Zip code" textbox, a semantic match that appears higher on the page. The user's visual scan also did not always proceed from top to bottom; given "First Search" as the target name for eBay.com (shown in Figure 6.2), most users picked the search box in the middle of the page, rather than the search box tucked away in the upper right corner. This suggested that supporting ordinals such as "first" and "second" to distinguish elements with similar labels would be difficult – other resolution techniques would need to be employed.



Hello! Sign in or register.

Buying new items, brand names, and collectibles on eBay is simple. Here's how it works...

What are you looking for?

Browse for an item

Specialty sites:

Place a bid or Buy It Now
Simply enter your maximum price and eBay will bid as needed for you. Don't want to wait? Purchase instantly with Buy It Now!

Pay for your item PayPal
Most sellers accept PayPal, the fast and free way to pay. With PayPal, you'll also be protected up to $1,000 on qualified listings.

Most users chose the box on the left for "First Search" instead of the box at the top.

You must register to bid or buy. It's fast, easy, and free!

Registration Benefits:
- Personalized shopping page
- Your favorite saved searches
- Most-wanted-item notifications
- Access to last-minute bargains
- PayPal Buyer Protection

Already Registered? Sign In

14K EXQUISITE ART DECO AMETHYST HANDMADE...
Disney Mickey Mouse Chilmark MetalArt On...
Light Motion Industries Mako Elite Under...
5.0 ACRE COLORADO RANCH / LAND 4 SALE B...
$12,700 DANIEL ROTH GENTS PREMIER DIAMO...
MY MOM FREAKED! SHE LOST 30LBS IN 30 DAY...

Figure 6.2 Most users selected the left box rather than the top one for "First Search" in Part 2 (ebay.com)

Part 3's names were almost all visible (235 names out of 240), since the directions requested only words from the page. Even in visible naming, however, users rarely reproduced a caption exactly; they would change capitalization, transpose words (writing "web search" when the caption read "Search the Web"), and mistype words. Some Part 3 answers also included the type of the page object ("box", "entry", "field"). When asked to name a textbox which had an ambiguous caption (e.g. "Search" on a page with more than one search form), most users noticed the ambiguity and tried to resolve it with one of two approaches: either counting occurrences ("search 2") or referring to other nearby captions, such as section headings ("search products").

Chapter 7 Keyword Pattern Algorithm

We now describe the heuristic algorithm that resolves a keyword pattern to a web page component. This algorithm is used for identifying the following components: textfields, textareas, lists, drop-down boxes, push buttons, checkboxes, radio buttons, and hyperlinks. Given a keyword pattern and a web page, the output of the algorithm is one of the following:

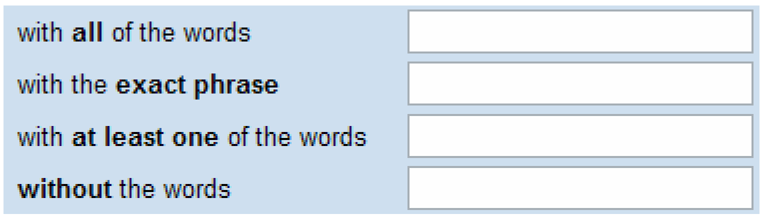
- the component on the page that best matches the pattern
- *ambiguous match* if two or more components are considered equally good matches
- *no match* if no suitable match can be found.

The outline of the algorithm is as follows: first, the visible text in the page is carved up into groups called *text blobs*. Each blob is then compared to the keyword pattern, and blobs that include the pattern (within a tolerance) are kept in a list. Once this list has been created, the algorithm compares each blob with each component of interest in the page, e.g., if it is trying to match the keyword component with an element that lets users enter text, then textfields, password fields, and multiline textareas are the components of interest. In this comparison, each blob-component pair is given a score. If this set of pairs is nonempty, then the component of the pair with the highest score is returned by the algorithm.

7.1 Finding Text Blobs

The first step in resolving keyword patterns is to carve the visible text of an HTML document into text blobs that the keyword pattern will be matched against. A *text blob* is a visible string of content in the web page delimited by the opening and closing tags of a *partitioning* HTML element. A partitioning element is an ancestor of a group of text nodes that appear together in the rendered page. The heuristic used to decide whether an element is a partitioning element is its tag name. For example, a `<p>` tag blocks off a paragraph in HTML, so all of the text nodes under that node are likely to be part of the same blob. However, a `<b>` tag is generally used to add boldface for emphasis within a section of text, so it is not considered a partitioning element. The HTML tags that represent partitioning elements in this algorithm are listed in Appendix B.

Consider the snippet of a web page shown in Figure 7.1 and its corresponding DOM in Figure 7.2. Although the text "with **all** of the words" appears to be one unit in the rendered model, it is actually made up of three text nodes in the DOM. Because a user may use "with all words" as a keyword pattern for the first textfield, it is important for the algorithm to compare the pattern to the concatenation of these three nodes rather than to each one individually.


with **all** of the words

with the **exact phrase**

with **at least one** of the words

without the words

Figure 7.1 Rendered model of web page with multiple textfields (www.google.com/advanced _search)

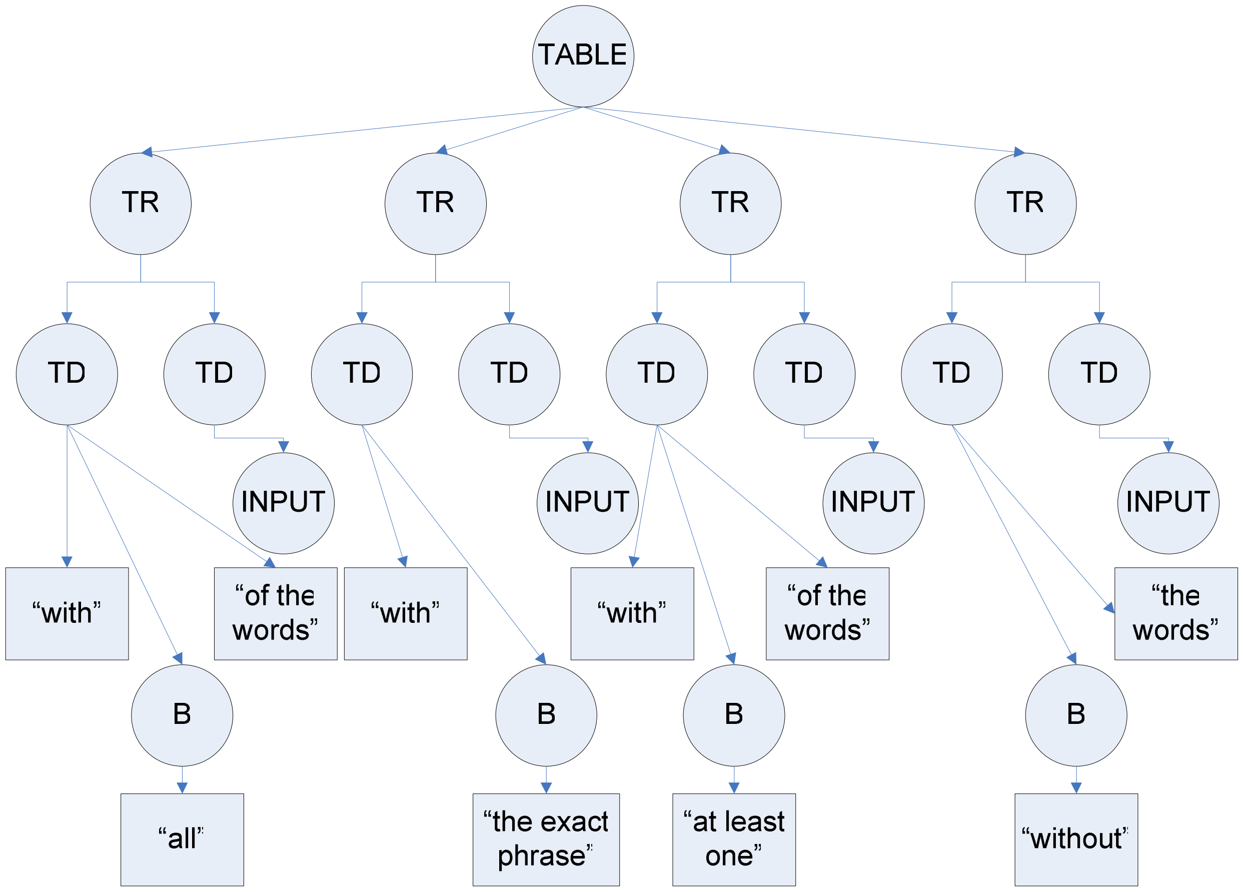


Figure 7.2 DOM of the web page shown in Figure 7.2 (www.google.com/advanced _search)

To determine which text nodes constitute a blob, a preorder traversal is done over the DOM tree. Every time a partitioning element is reached, a new blob is created and every text node under that element is added to the blob. If the partitioning element has a partitioning element as a child, then text nodes that appear under its child belong to its child's text blob, not its own.

Once all the blobs have been found, the text of the nodes within the blob is concatenated into a string (again, by a preorder traversal of the nodes in the blob) and an entry pairing the string with its partitioning element is added to a map. For the DOM in Figure 7.2, this map would contain the following four entries:

Blob content	Partitioning element
"with all of the words"	first TD under first TR
"with the exact phrase"	first TD under second TR
"with at least one of the words"	first TD under third TR
"without the words"	first TD under fourth TR

Figure 7.3 Map of text blob content to the partitioning element that contains it

Certain partitioning elements, such as `SCRIPT` and `STYLE`, must be treated as special cases because the text nodes that they contain are not visible in the web page, so those text nodes should not be considered when doing keyword matches, and therefore do not become part of a blob.

7.2 Determining Candidate Text Blobs

Once the mapping from blobs to partitioning elements has been built, an E-score is calculated for each blob. An *E-score* (for edit distance score) is a number from 0 to 1 that reflects how strongly the blob contains the keyword pattern. If the E-score exceeds a certain threshold, then the blob is added to a list of candidate blobs.

When the pattern is a substring of the blob, then the minimum value of the E-score is 0.95. If the pattern and the blob must are the same string (when normalized), then the E-score is its maximum value, 1.0. This is because exact matches should be ranked higher than substring matches. For example, if a web page has one link labeled "MIT" and another labeled "MIT Libraries," then the keyword pattern should identify the first link instead of considering the two links equally. If it did not, and the algorithm returned *ambiguous* match, then this would inconsistent with user expectations because identifying an element by its exact name should be unambiguous.

Keyword patterns that are not substring matches should not be disregarded altogether. Users may omit or transpose words in their keyword patterns, such as using "all words" as a keyword pattern to match the first blob in Figure 7.1. However, not all non-substring matches should be ranked equally: patterns that do a better job of matching a blob should have higher E-scores.

To calculate an E-score for a blob B, I augment a conventional edit distance algorithm [29] as follows. First, both the blob and the pattern are normalized by eliminating capitalization and punctuation, and by replacing a sequence of whitespace characters with one space. Then the blob is searched for an approximate occurrence of the pattern P, using the edit distance algorithm to tolerate typos and omitted words. If the edit distance D is zero, that is, if the pattern is a substring of the blob, then the value of the E-score is:

$$0.95 + 0.05 * (|P| / |B|),$$

but if D is nonzero, then the value of the E-score is:

$$0.95 - (D / |P|).$$

When D is nonzero, that means that the pattern requires D insertions or deletions to be a substring of B. If the entire pattern would have to be edited to match B, then D is the length of the pattern, in which case the E-score is 0, indicating that B does not match the pattern, as desired. But when D is less than the length of the pattern, then it is scaled so that blobs that require fewer edits have larger E-scores than those that require more edits. Note that an E-score for D equal to zero will always be higher than an E-score for a nonzero D. This scheme favors substring matches, as desired.

7.3 Determining Candidate Matches

Once the list of candidate blobs is constructed, the next step is to find components that are likely to be associated with the blob. This is done by taking the bounding box in the rendered web page (b-box) of the root node for each blob and comparing it to the b-box of each component of the type for which a match is sought. For example, when keyword pattern matching is used in the context of the `enter` command, only the bounding boxes of textfields are considered.

When the b-box of the blob is to the left of the b-box of the component and the top and bottom of the blob b-box are within the top and bottom of the component b-box (plus some small threshold), then the blob b-box is considered a *left match* for the component. A *right match* is determined in a similar manner.

If the top of the b-box of the component is within some threshold distance (which is currently 1.5 times the height of the component b-box) from the bottom of the blob b-box and the left edge of either b-box is contained within the width of the other component, then the blob b-box is considered a *above match* for the component.

Examples of these different types of matches are shown in Figure 7.4.

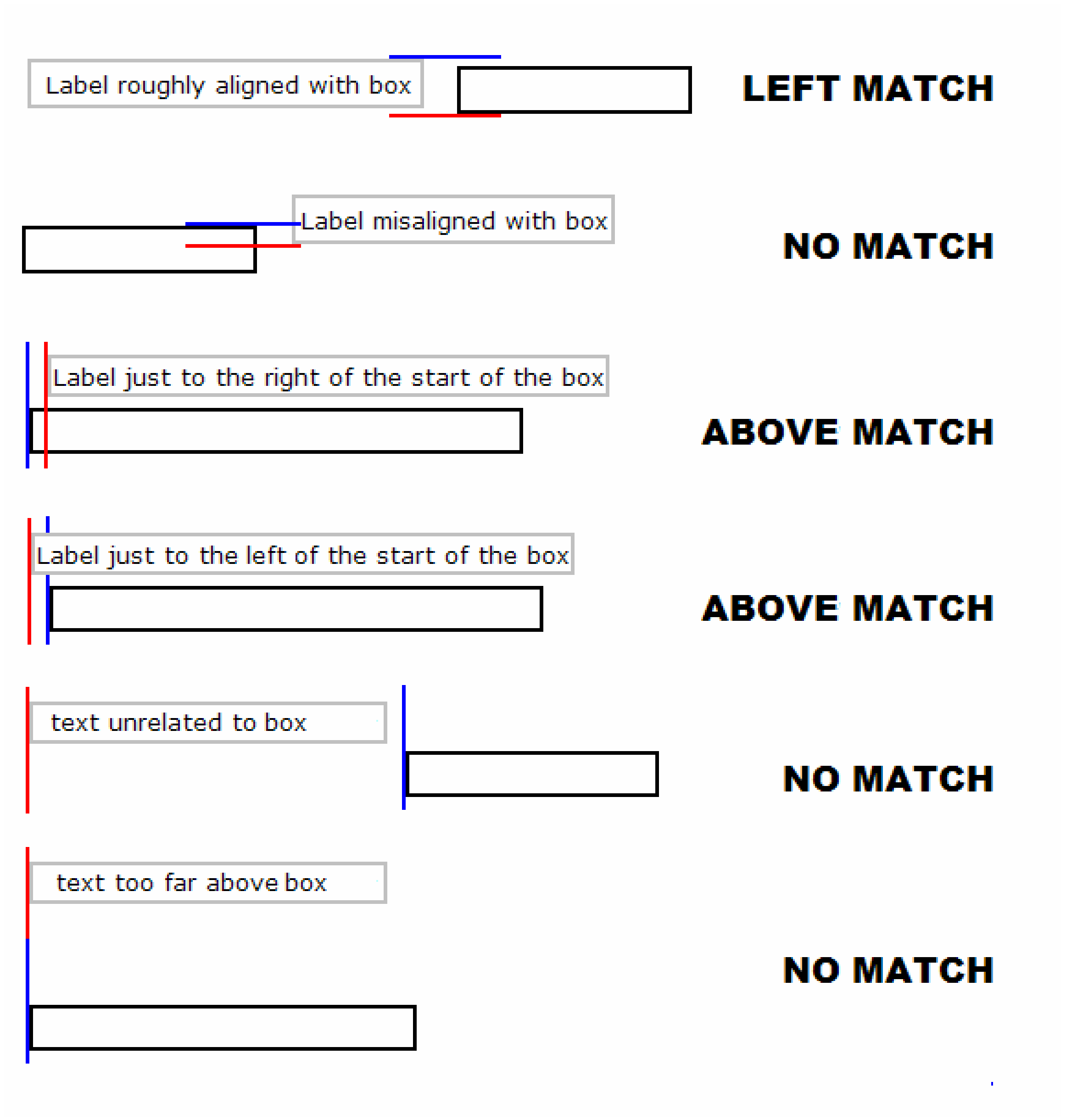

Label roughly aligned with box
LEFT MATCH
Label misaligned with box
NO MATCH
Label just to the right of the start of the box
ABOVE MATCH
Label just to the left of the start of the box
ABOVE MATCH
text unrelated to box
NO MATCH
text too far above box
NO MATCH

Figure 7.4 Comparing text blobs with textfields for matches. Bounding boxes of text blobs appear in gray and textboxes appear in black. Some edges of the boxes are extended to show how the boxes line up.

7.4 Evaluating Candidate Matches

The context of keyword pattern matching determines the strength of a match. For example, when using a keyword pattern to identify a textfield, a left match is considered stronger than a right match because text boxes are usually labeled to their left. For a checkbox, however, a right match is considered stronger than a left match because checkboxes are usually labeled to their right. To accommodate the differences between contexts, each blob-component pair is given a strength index as a function of the following heuristics within each context:

Type. As described above, the type of match that is considered more likely is determined by the context of keyword pattern that is trying to be matched. These differences are summarized in Table 7-1.

Type being matched	**Special conditions**
Textfields, textareas, lists, and drop-down boxes	The raw strength index of a right match is scaled down by 60%.
Checkboxes and radio buttons	The raw strength index of a top match or a left match is scaled down by 60%.
Hyperlinks	If the text blob's element is not a descendant of the link, then its strength index is 0.
Push buttons	If the text blob's element is not a the button itself, then its strength index is 0.

Table 7-1 Special conditions for each type of component matched by this algorithm.

E-score. A higher E-score always contributes positively to the strength index; however, the degree to which it contributes is determined by the context.

Pixel Distance. The distance in pixels between a text blob and its component is most helpful in deciding which component is the best match for a blob when a blob has multiple matches of the same type (e.g., right, left, above) by taking a "nearest neighbors" approach. Indeed, many pages have textfields justified to the right of a page with their labels justified to the left, so this creates a large distance between the label and the textfield that should not reduce the strength of their association. However, consider a page that has a two-column web form:

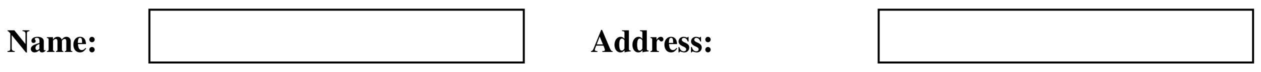


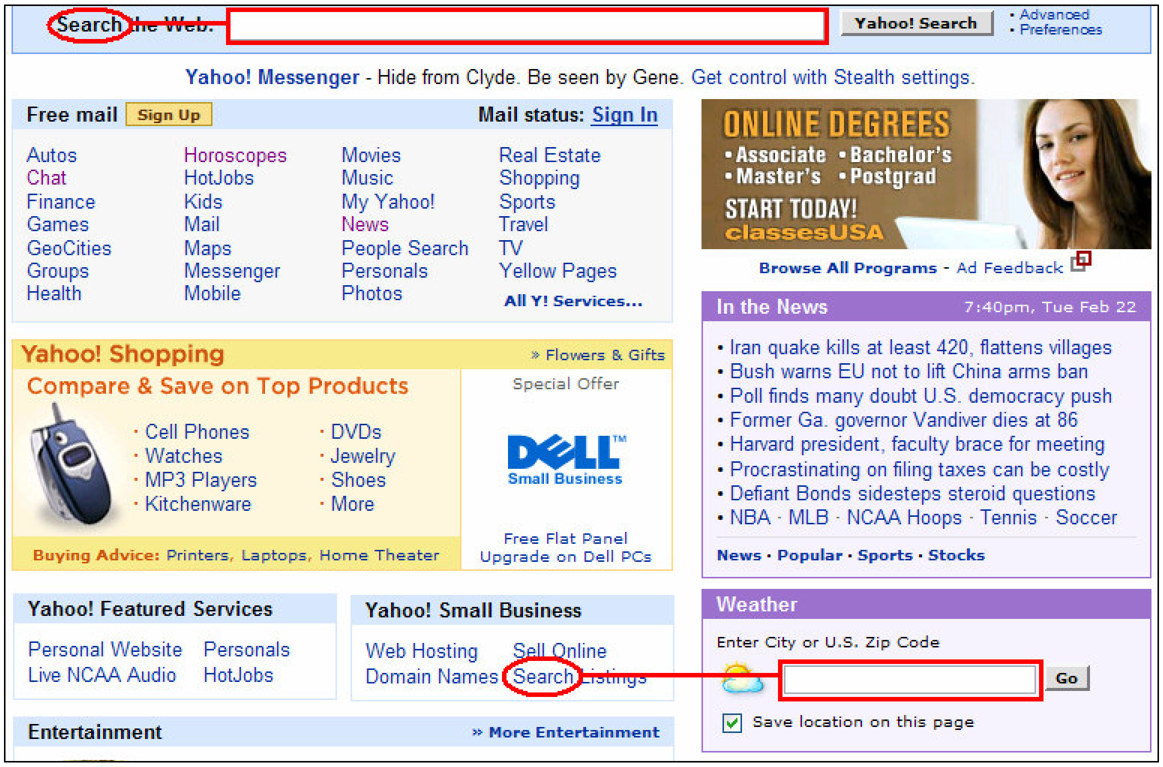
Figure 7.5 Labels that are candidate matches for multiple textfields

In this case, the keyword pattern `"name"` would be a left match for both textfields because it lies to the left of each of them. Because the pixel distance between **Name** and the left box is less than the distance between **Name** and the right box, the match between **Name** and the left box should have a higher strength index. This is also an example of why textfields favor right matches – if it favored left and right matches evenly, then the field associated with **Address** would be ambiguous.

Path Length. The length of the path in the DOM tree from the blob element to the component element is called its *path length*. Blobs and components that are siblings have stronger associations than those that are not. For example, in Figure 7.2, the path length between a text blob and the `INPUT` it is supposed to label is 3, but its path length to any other `INPUT` is 5.

Using path length can help eliminate false positives that occur when text in one section of a page coincidentally lines up with a component in another section. Web portals and pages with sidebars are prone to this type of error, as shown in Figure 7.6. In the Figure, the text "Search" appears at least twice in the page. The first instance lines up with the main Yahoo! search box, and the second instance lines up with the Weather search box. Using the pixel distance between the blob and the textfield to determine the better match is error-prone when the pattern happens to appear at the edge of its block. Although `Search the Web` is be closer than `Search Listings` in this example, this is a coincidence and is not a reliable heuristic.

Sections of a page often map to subtrees in the DOM. Because this type of mismatch error is caused by matching across sections of a page, path length is a good heuristic to combat this problem because it will rank pairs of nodes that are closely related in the DOM higher than those with a more distant relationship.


Search the Web: Yahoo! Search · Advanced · Preferences

Yahoo! Messenger - Hide from Clyde. Be seen by Gene. Get control with Stealth settings.

Free mail Sign Up Mail status: Sign In

Autos	Horoscopes	Movies	Real Estate
Chat	HotJobs	Music	Shopping
Finance	Kids	My Yahoo!	Sports
Games	Mail	News	Travel
GeoCities	Maps	People Search	TV
Groups	Messenger	Personals	Yellow Pages
Health	Mobile	Photos	All Y! Services...

ONLINE DEGREES •Associate •Bachelor's •Master's •Postgrad START TODAY! classesUSA

Browse All Programs - Ad Feedback

Yahoo! Shopping » Flowers & Gifts

Compare & Save on Top Products

· Cell Phones · DVDs
· Watches · Jewelry
· MP3 Players · Shoes
· Kitchenware · More

Buying Advice: Printers, Laptops, Home Theater

Special Offer DELL Small Business Free Flat Panel Upgrade on Dell PCs

In the News 7:40pm, Tue Feb 22

- Iran quake kills at least 420, flattens villages
- Bush warns EU not to lift China arms ban
- Poll finds many doubt U.S. democracy push
- Former Ga. governor Vandiver dies at 86
- Harvard president, faculty brace for meeting
- Procrastinating on filing taxes can be costly
- Defiant Bonds sidesteps steroid questions
- NBA · MLB · NCAA Hoops · Tennis · Soccer

News · Popular · Sports · Stocks

Yahoo! Featured Services

Personal Website Personals
Live NCAA Audio HotJobs

Yahoo! Small Business

Web Hosting Sell Online
Domain Names Search Listings

Weather

Enter City or U.S. Zip Code Go

✓ Save location on this page

Entertainment » More Entertainment

Figure 7.6 Screenshot of the Yahoo! home page with amiguous textfield match for "Search" (yahoo.com)

The result of calculating the strength indices from these heuristics is a list of (*component, strength*) pairs ranked by strength. The algorithm returns the component of the top pair, unless the top two pairs have the same strength, in which case it returns *ambiguous match*. If the list of pairs is empty, then it returns *no match*. Some contexts will only use a subset of these heuristics to calculate the strength of a pair, and then will only use the remaining heuristics in the event of an *ambiguous match*.

7.5 Evaluation

I evaluated this algorithm on the 240 names (40 for each of 6 pages) generated by Part 3 of the study. Its performance is shown in Figure 7.7, and the screenshots, along with some of the responses that users provided, appear in Figures 7.8-7.13. For each name, Chickenfoot either found the right textbox (*Match*), reported an ambiguous match (*Ambiguous*), or returned the wrong textbox (*Mismatch*). Precision is high for 5 of the 6 pages. Performance is poor on the MIT page (shown in Figure 7.11) because it involved an ambiguous caption, and my heuristic algorithm does not yet recognize the disambiguation strategies used for this caption (counting and section headings). Also the 100% success rate on Vivisimo (shown in Figure 7.12) may be misleading because that page only had one textfield; however, my algorithm disregards labels that it believes are unrelated, even if there is only one textfield on the page, so the result is not insignificant.

This evaluation is only preliminary; a proper evaluation should use a larger selection of web sites. Nevertheless, it suggests that keyword patterns can be automatically resolved with high precision.

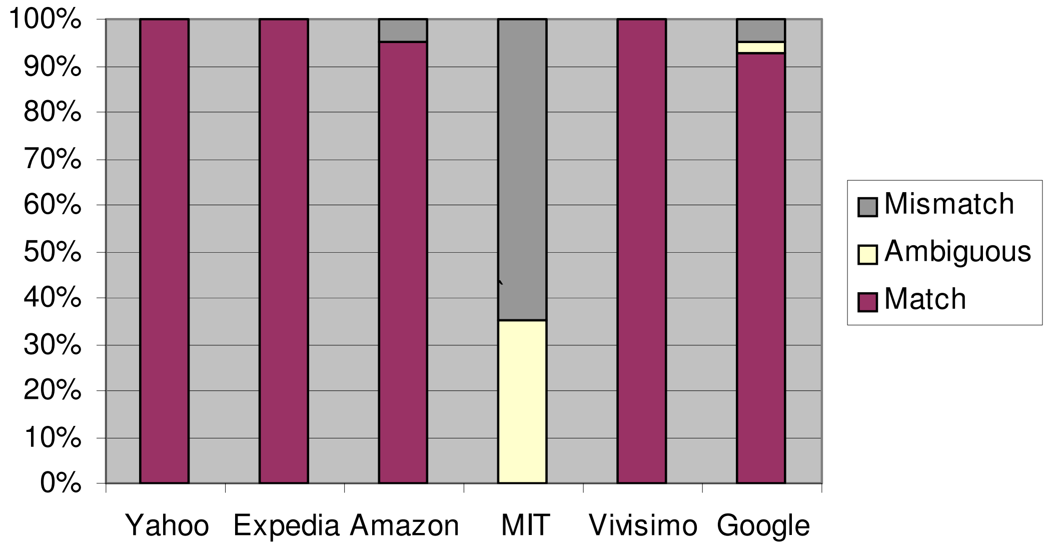
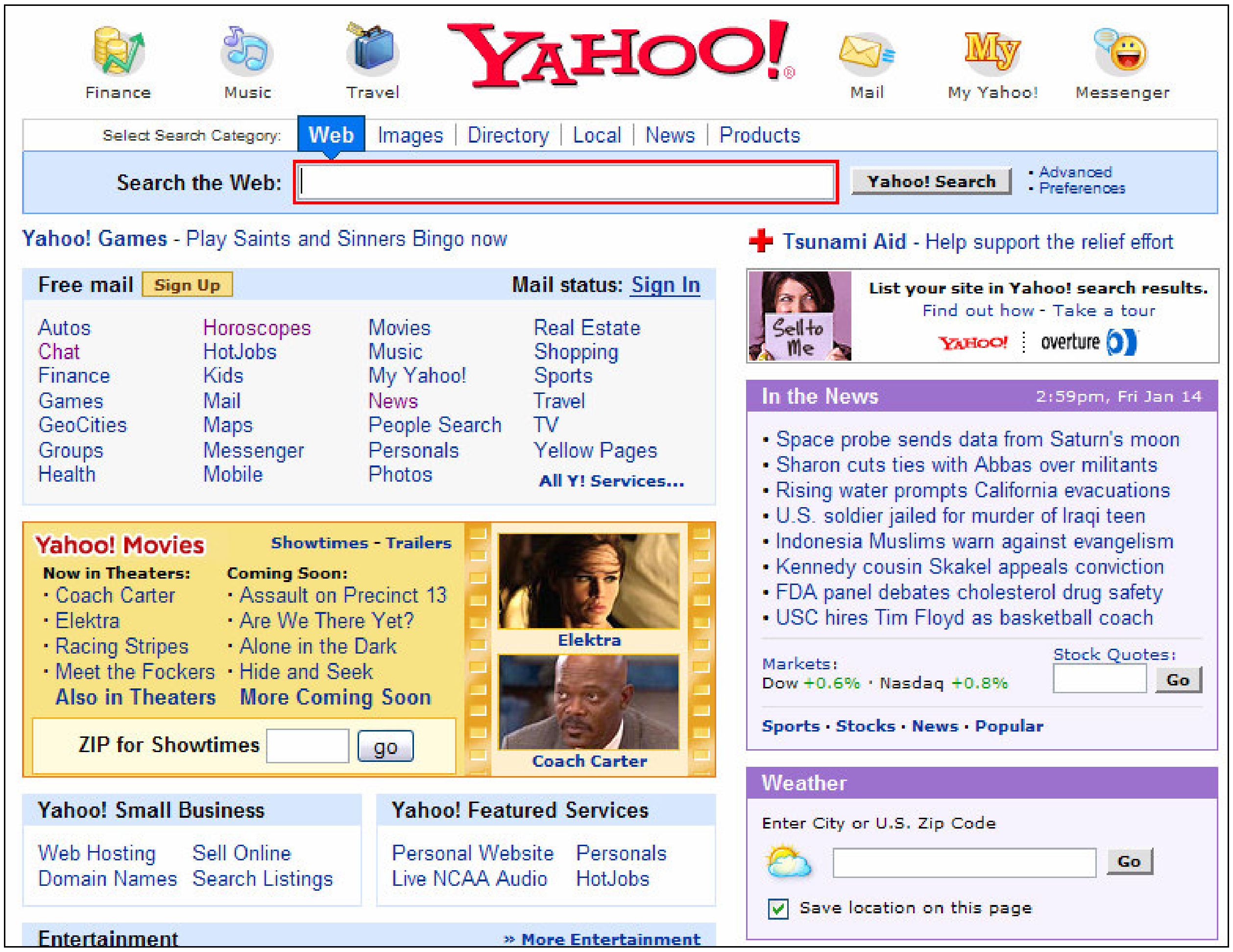


Figure 7.7 Results of algorithm for resolving keyword patterns on Part 3 survey data


Finance Music Travel YAHOO! Mail My Yahoo! Messenger

Select Search Category: Web | Images | Directory | Local | News | Products

Search the Web: [ ] Yahoo! Search · Advanced · Preferences

Yahoo! Games - Play Saints and Sinners Bingo now

Tsunami Aid - Help support the relief effort

Free mail Sign Up Mail status: Sign In

Autos	Horoscopes	Movies	Real Estate
Chat	HotJobs	Music	Shopping
Finance	Kids	My Yahoo!	Sports
Games	Mail	News	Travel
GeoCities	Maps	People Search	TV
Groups	Messenger	Personals	Yellow Pages
Health	Mobile	Photos	All Y! Services...

Sell to Me

List your site in Yahoo! search results.
Find out how - Take a tour
YAHOO! overture

In the News 2:59pm, Fri Jan 14

- Space probe sends data from Saturn's moon
- Sharon cuts ties with Abbas over militants
- Rising water prompts California evacuations
- U.S. soldier jailed for murder of Iraqi teen
- Indonesia Muslims warn against evangelism
- Kennedy cousin Skakel appeals conviction
- FDA panel debates cholesterol drug safety
- USC hires Tim Floyd as basketball coach

Markets: Dow +0.6% · Nasdaq +0.8%
Stock Quotes: [ ] Go

Sports · Stocks · News · Popular

Yahoo! Movies Showtimes - Trailers

Now in Theaters:	Coming Soon:
· Coach Carter	· Assault on Precinct 13
· Elektra	· Are We There Yet?
· Racing Stripes	· Alone in the Dark
· Meet the Fockers	· Hide and Seek
Also in Theaters	More Coming Soon

ZIP for Showtimes [ ] go

Elektra

Coach Carter

Weather

Enter City or U.S. Zip Code

[ ] Go

[✓] Save location on this page

Yahoo! Small Business

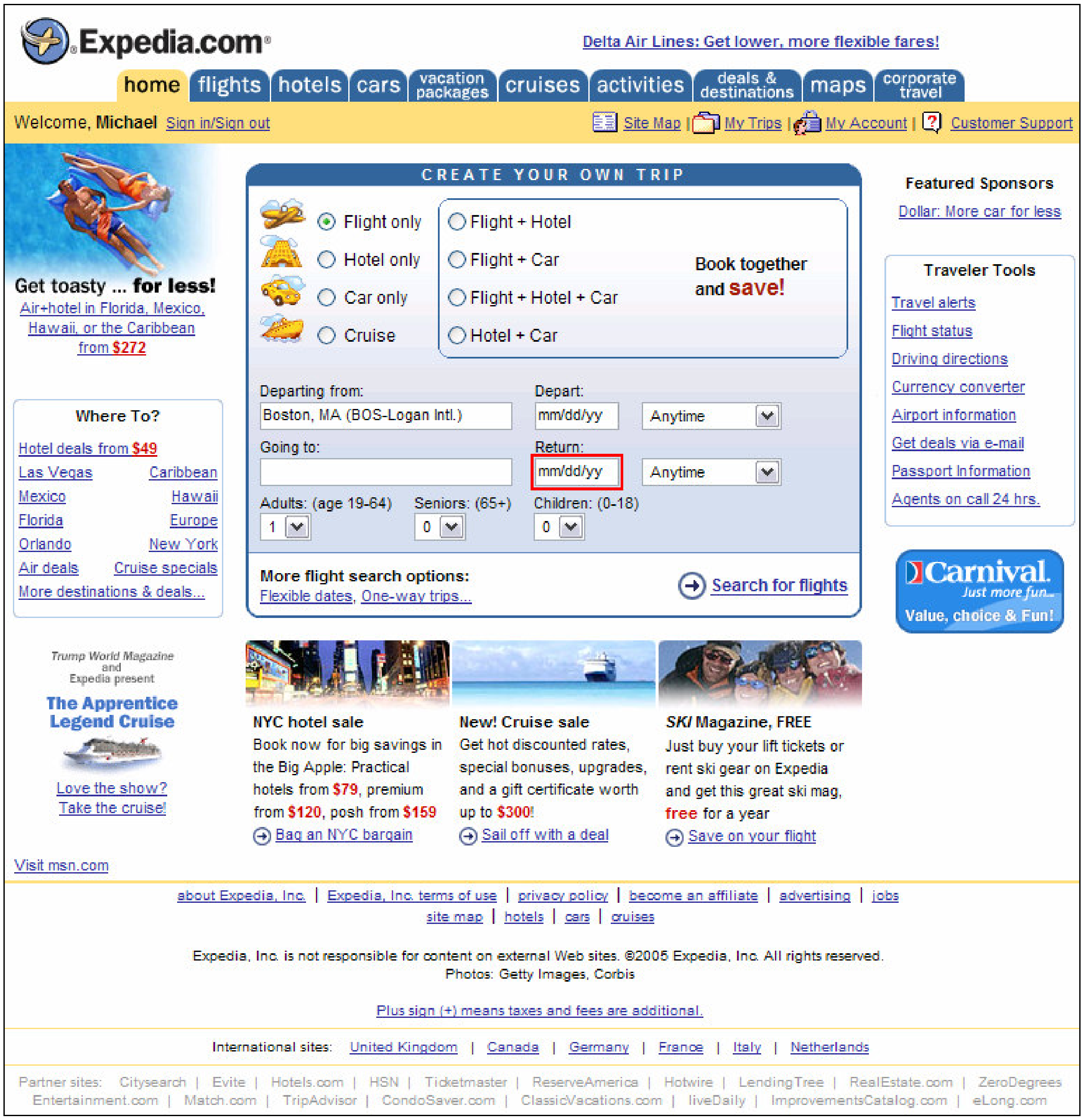
Web Hosting Sell Online
Domain Names Search Listings

Yahoo! Featured Services

Personal Website Personals
Live NCAA Audio HotJobs

Entertainment » More Entertainment

Figure 7.8 Yahoo! home page (www.yahoo.com) shown in Part 3 of the user study. Users' names for the search box included: "Search," "Search the Web," "First Search," and "Seach [sic] the Web Text Box."


Expedia.com

Delta Air Lines: Get lower, more flexible fares!

home | flights | hotels | cars | vacation packages | cruises | activities | deals & destinations | maps | corporate travel

Welcome, Michael Sign in/Sign out

Site Map | My Trips | My Account | Customer Support

Get toasty ... for less!
Air+hotel in Florida, Mexico, Hawaii, or the Caribbean from $272

Where To?
Hotel deals from $49

Las Vegas	Caribbean
Mexico	Hawaii
Florida	Europe
Orlando	New York
Air deals	Cruise specials

More destinations & deals...

CREATE YOUR OWN TRIP

[✓ Flight only] | Hotel only | Car only | Cruise

Flight + Hotel | Flight + Car | Flight + Hotel + Car | Hotel + Car

Book together and save!

Departing from:	Depart:	
Boston, MA (BOS-Logan Intl.)	mm/dd/yy	Anytime
Going to:	Return:	
	mm/dd/yy	Anytime

Adults: (age 19-64) 1 | Seniors: (65+) 0 | Children: (0-18) 0

More flight search options:
Flexible dates, One-way trips...

Search for flights

Featured Sponsors
Dollar: More car for less

Traveler Tools
Travel alerts
Flight status
Driving directions
Currency converter
Airport information
Get deals via e-mail
Passport Information
Agents on call 24 hrs.

Carnival. Just more fun... Value, choice & Fun!

Trump World Magazine and Expedia present
The Apprentice Legend Cruise
Love the show? Take the cruise!

NYC hotel sale
Book now for big savings in the Big Apple: Practical hotels from $79, premium from $120, posh from $159
Bag an NYC bargain

New! Cruise sale
Get hot discounted rates, special bonuses, upgrades, and a gift certificate worth up to $300!
Sail off with a deal

SKI Magazine, FREE
Just buy your lift tickets or rent ski gear on Expedia and get this great ski mag, free for a year
Save on your flight

Visit msn.com

about Expedia, Inc. | Expedia, Inc. terms of use | privacy policy | become an affiliate | advertising | jobs
site map | hotels | cars | cruises

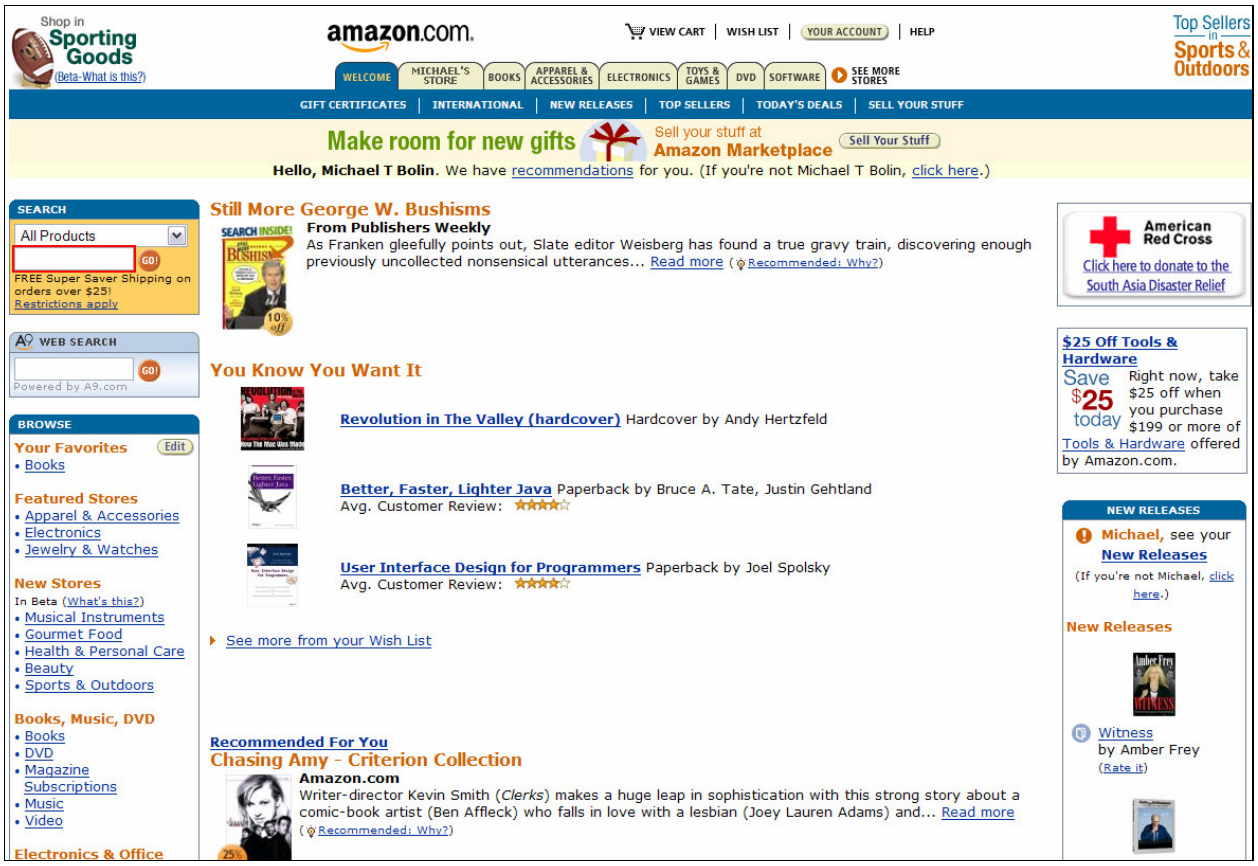
Expedia, Inc. is not responsible for content on external Web sites. ©2005 Expedia, Inc. All rights reserved.
Photos: Getty Images, Corbis

Plus sign (+) means taxes and fees are additional.

International sites: United Kingdom | Canada | Germany | France | Italy | Netherlands

Partner sites: Citysearch | Evite | Hotels.com | HSN | Ticketmaster | ReserveAmerica | Hotwire | LendingTree | RealEstate.com | ZeroDegrees
Entertainment.com | Match.com | TripAdvisor | CondoSaver.com | ClassicVacations.com | liveDaily | ImprovementsCatalog.com | eLong.com

Figure 7.9 Expedia home page (www.expedia.com) shown in Part 3 of the user study. Users' names for the return date field included: "return," "return date," "trip return," and "Return mm/dd/yy."



Shop in Sporting Goods (Beta-What is this?)

amazon.com. VIEW CART | WISH LIST | YOUR ACCOUNT | HELP

WELCOME | MICHAEL'S STORE | BOOKS | APPAREL & ACCESSORIES | ELECTRONICS | TOYS & GAMES | DVD | SOFTWARE | SEE MORE STORES

Top Sellers in Sports & Outdoors

GIFT CERTIFICATES | INTERNATIONAL | NEW RELEASES | TOP SELLERS | TODAY'S DEALS | SELL YOUR STUFF

Make room for new gifts — Sell your stuff at Amazon Marketplace — Sell Your Stuff

Hello, **Michael T Bolin**. We have recommendations for you. (If you're not Michael T Bolin, click here.)

SEARCH

All Products

GO!

FREE Super Saver Shipping on orders over $25! Restrictions apply

WEB SEARCH

GO!

Powered by A9.com

BROWSE

Your Favorites — Edit
- Books

Featured Stores
- Apparel & Accessories
- Electronics
- Jewelry & Watches

New Stores

In Beta (What's this?)
- Musical Instruments
- Gourmet Food
- Health & Personal Care
- Beauty
- Sports & Outdoors

Books, Music, DVD
- Books
- DVD
- Magazine Subscriptions
- Music
- Video

Electronics & Office

Still More George W. Bushisms

From Publishers Weekly

As Franken gleefully points out, Slate editor Weisberg has found a true gravy train, discovering enough previously uncollected nonsensical utterances... Read more (Recommended: Why?)

You Know You Want It

Revolution in The Valley (hardcover) Hardcover by Andy Hertzfeld

Better, Faster, Lighter Java Paperback by Bruce A. Tate, Justin Gehtland
Avg. Customer Review: ★★★★☆

User Interface Design for Programmers Paperback by Joel Spolsky
Avg. Customer Review: ★★★★☆

See more from your Wish List

Recommended For You

Chasing Amy - Criterion Collection

Amazon.com

Writer-director Kevin Smith (*Clerks*) makes a huge leap in sophistication with this strong story about a comic-book artist (Ben Affleck) who falls in love with a lesbian (Joey Lauren Adams) and... Read more (Recommended: Why?)

American Red Cross — Click here to donate to the South Asia Disaster Relief

$25 Off Tools & Hardware

Save $25 today — Right now, take $25 off when you purchase $199 or more of Tools & Hardware offered by Amazon.com.

NEW RELEASES

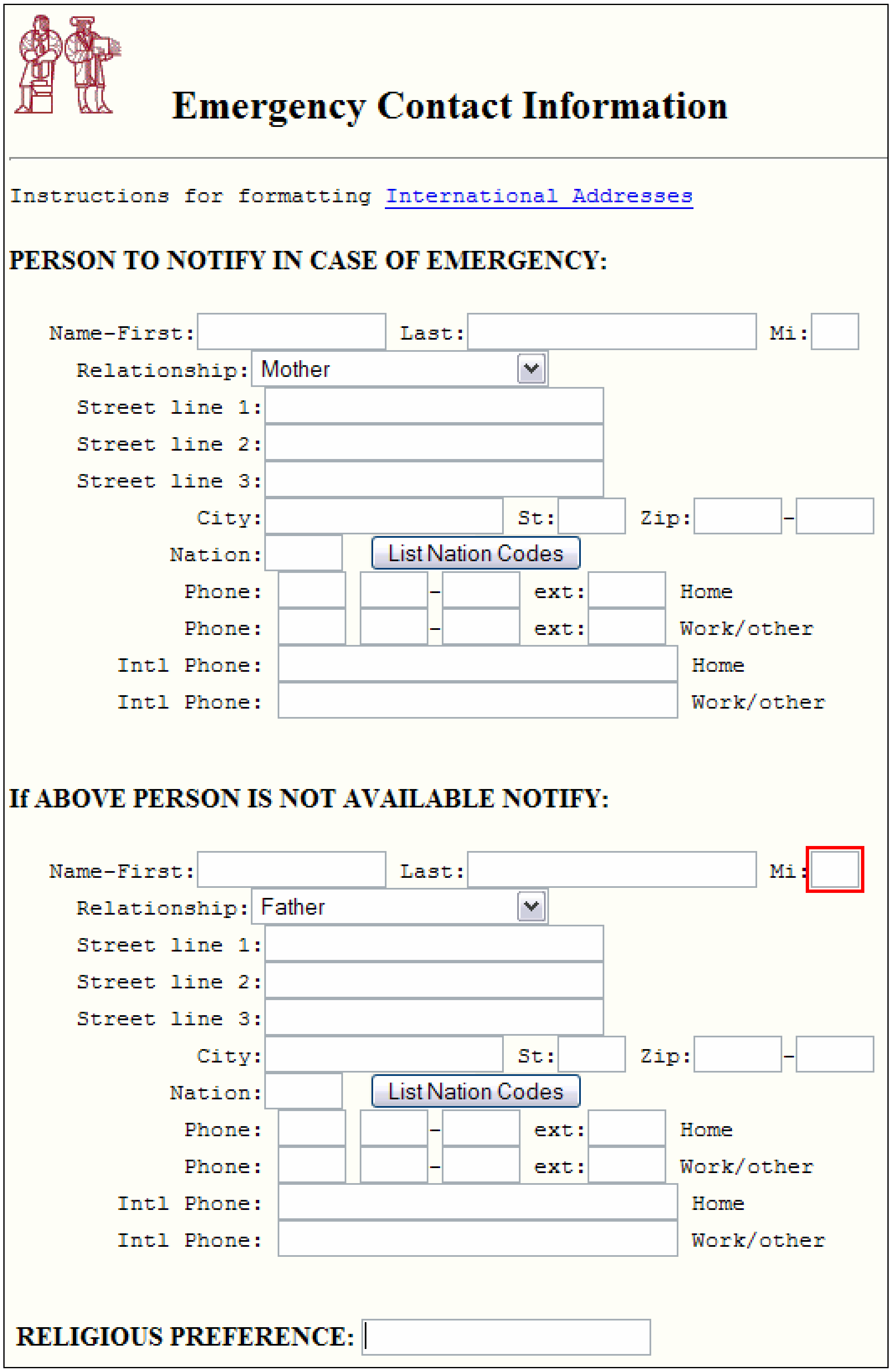
Michael, see your **New Releases**

(If you're not Michael, click here.)

New Releases

Witness
by Amber Frey
(Rate it)

Figure 7.10 Amazon home page (www.amazon.com) shown in Part 3 of the user study. Users' names for the search box included: "Search Amazon," "ProductSearch," "Search1," and "search 2."



Emergency Contact Information

Instructions for formatting International Addresses

PERSON TO NOTIFY IN CASE OF EMERGENCY:

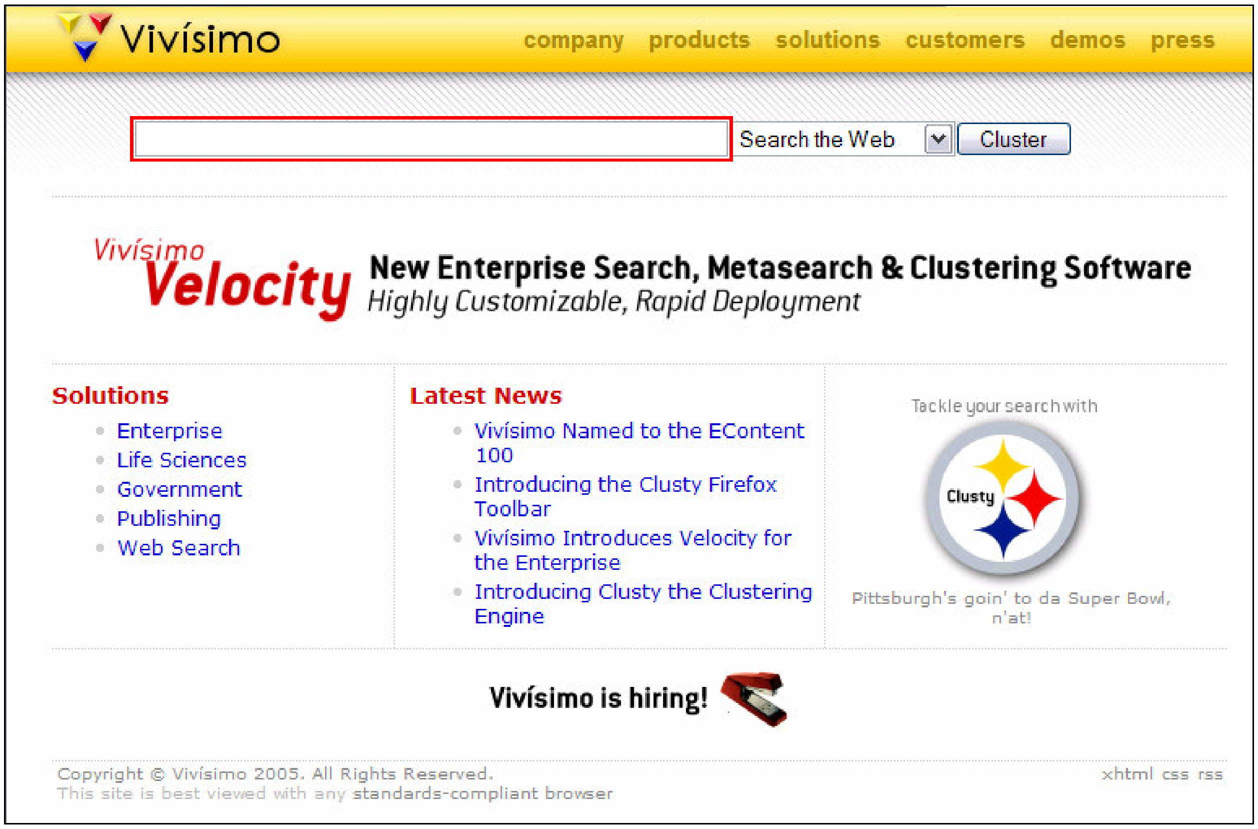
Name-First: Last: Mi:
Relationship: Mother
Street line 1:
Street line 2:
Street line 3:
City: St: Zip: -
Nation: List Nation Codes
Phone: - ext: Home
Phone: - ext: Work/other
Intl Phone: Home
Intl Phone: Work/other

If ABOVE PERSON IS NOT AVAILABLE NOTIFY:

Name-First: Last: Mi:
Relationship: Father
Street line 1:
Street line 2:
Street line 3:
City: St: Zip: -
Nation: List Nation Codes
Phone: - ext: Home
Phone: - ext: Work/other
Intl Phone: Home
Intl Phone: Work/other

RELIGIOUS PREFERENCE:

Figure 7.11 MIT emergency contact information page shown in Part 3 of the user study. Users' names for the search box included: "MI2," "Notify Mi Two," "Emergency2Mi," "backup contact mi," "Mi," "above person not available Mi," "Name MI," and "Contact 2 Mi."


Vivísimo

company products solutions customers demos press

Search the Web | Cluster

Vivísimo Velocity

New Enterprise Search, Metasearch & Clustering Software

Highly Customizable, Rapid Deployment

Solutions

- Enterprise
- Life Sciences
- Government
- Publishing
- Web Search

Latest News

- Vivísimo Named to the EContent 100
- Introducing the Clusty Firefox Toolbar
- Vivísimo Introduces Velocity for the Enterprise
- Introducing Clusty the Clustering Engine

Tackle your search with

Clusty

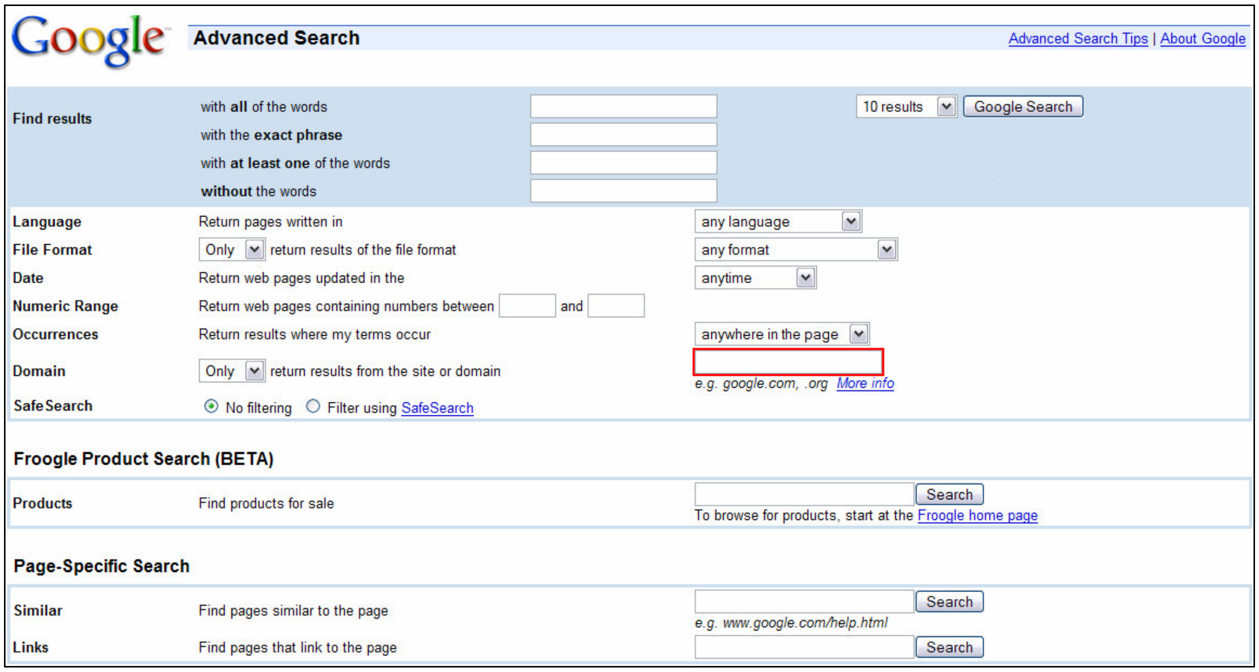
Pittsburgh's goin' to da Super Bowl, n'at!

Vivísimo is hiring!

Copyright © Vivísimo 2005. All Rights Reserved.
This site is best viewed with any standards-compliant browser

xhtml css rss

Figure 7.12 Vivisimo home page (www.vivisimo.com) shown in Part 3 of the user study. Users' names for the search box included: "search," "web search," "Cluster," and "vivisimo search."


Google **Advanced Search**

Advanced Search Tips | About Google

Find results	with **all** of the words	10 results · Google Search
	with the **exact phrase**	
	with **at least one** of the words	
	without the words	
Language	Return pages written in	any language
File Format	Only · return results of the file format	any format
Date	Return web pages updated in the	anytime
Numeric Range	Return web pages containing numbers between ___ and ___	
Occurrences	Return results where my terms occur	anywhere in the page
Domain	Only · return results from the site or domain	e.g. google.com, .org More info
SafeSearch	● No filtering ○ Filter using SafeSearch	

Froogle Product Search (BETA)

Products	Find products for sale	Search · To browse for products, start at the Froogle home page

Page-Specific Search

Similar	Find pages similar to the page	Search · e.g. www.google.com/help.html
Links	Find pages that link to the page	Search

Figure 7.13 Google advancd search page (www.google.com/advanced_search) shown in Part 3 of the user study. Users' names for the domain box included: "site," "Domain," "only from," "Advanced Search Domain," "domain/site," "GOOGLE," and "return results from the site or domain."

Chapter 8 Implementation

In this chapter, I explain how Chickenfoot is implemented. I begin by describing Chickenfoot's internal representation of a web page, as this is the fundamental data structure that the rest of the system is built upon. Then I explain how Chickenfoot commands and objects operate on this model of the page. Subsequently, I show how this model is maintained in response to changes in the underlying page. Next, I reveal how Chickenfoot scripts are interpreted so that its functions can be called using a standard JavaScript interpreter. Finally, I explain how page loading is monitored so that the `go` command and URL triggers work as desired.

8.1 Chickenfoot Model for a Web Page

Chickenfoot has two copies of the DOM of a web page, one in Java and one in C++. This section explains the motivation for these two copies, and how the Java DOM is created from the C++ DOM.

8.1.1 Motivation

Chickenfoot leverages the LAPIS pattern library to match web page elements against text constraint patterns. Because LAPIS is a large codebase written in Java, and Firefox is an even larger codebase written in C++ and JavaScript, it is impractical to translate either codebase into the other's language, so a Firefox-LAPIS bridge is employed so that Chickenfoot can use LAPIS.

Fortunately, Firefox is bundled with a bridge called LiveConnect [18] that enables JavaScript to communicate with Java objects. Because Chickenfoot is written mostly in JavaScript, LiveConnect makes it fairly easy to call into LAPIS; however, the cost of calling Java from JavaScript is relatively expensive (see the benchmarking data in Table 8-1). To compensate, Chickenfoot is designed so that it makes a few calls to Java with large inputs rather than many calls with small inputs.

Java calling Java within its own Java Virtual Machine	0.03 us/call
JavaScript calling JavaScript within Firefox	15-20.00 us/call
JavaScript calling Java via LiveConnect in Firefox	950-1000.00 us/call

Table 8-1 Benchmarking data for overhead of method calls in different languages. These benchmarks were made on a 1.7GHz laptop running WinXP, Firefox 1.0, and Java 1.5.

Firefox's LiveConnect technology was a major factor when deciding which web browser to use for Chickenfoot. Microsoft Internet Explorer (IE) is a far more popular browser than Firefox, so Chickenfoot may have wider appeal if it were embedded in IE; however, it was difficult to write a Browser Helper Object for IE that could make calls to Java code. Also, Firefox has better support for Web standards, such as DOM 2 [2], that are used heavily in the Chickenfoot code base. Finally, Firefox is available on Windows, Mac OSX, and Linux, whereas IE is only available on Windows (and to some extent, on Mac OSX).

8.1.2 Building a Bridge between Firefox and LAPIS

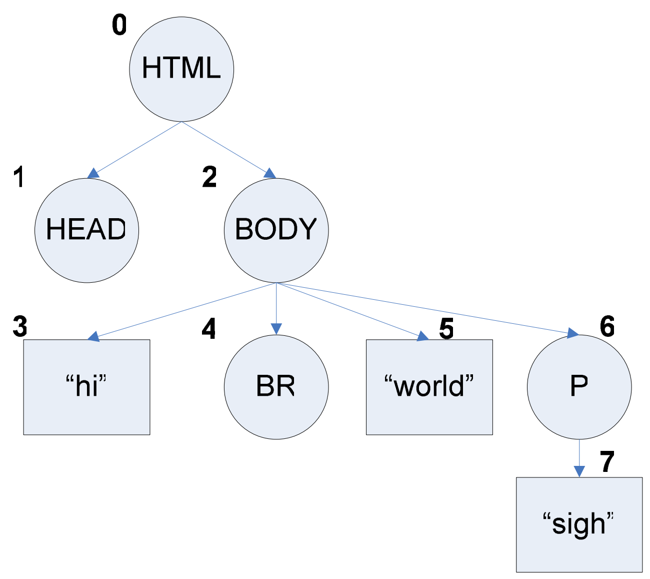
For Chickenfoot to use LAPIS to find pattern matches in a DOM, it must create a mapping between the DOM in Firefox and the string model in LAPIS. This section explains the construction of this mapping, and a flow chart of how the mapping is created appears at the end of the section in Figure 8.3.

When Firefox loads a web page, it parses the page’s HTML and automatically creates a DOM representation of it. Once this DOM representation has been created, the HTML source that was used to create it is discarded in lieu of the DOM because the well-formed, structured tree representation is more convenient for Firefox to work with than a raw string of HTML.

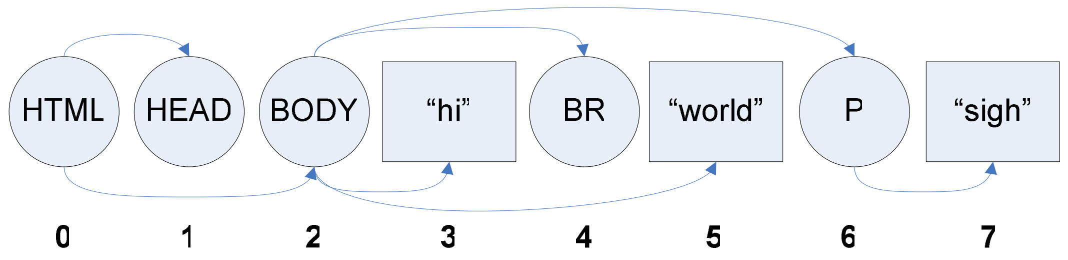
In order to be sure that LAPIS is matching against the same HTML that Firefox sees, Chickenfoot creates a string of well-formed XHTML from the Firefox DOM, and sends it via LiveConnect so LAPIS can create an equivalent HTML document. As Chickenfoot does a preorder traversal to create this string of XHTML, it assigns each node an id number, starting at zero, and places each node in an array at the index that corresponds to this id. Because the id number is set as a property of the node, it is possible to get the id of a node in constant time. Similarly, because the array is indexed by node id, looking up a node by its id is also a constant time operation. The process of creating the XHTML from the HTML is illustrated in Figure 8.1.

```
<HTML>hi<BR\>world<P>bye</HTML>
```

(1) Sloppy HTML received from server. `<BR\>` has an erroneous slash and `HEAD` and `BODY` are missing.



(2) DOM created by Firefox HTML parser. Note that the parser adds missing **HEAD** and **BODY** nodes.



(3) DOM nodes arranged in an array after a preorder traversal. A node's index in array serves as its id.

```
<HTML><HEAD></HEAD><BODY>hi<BR></BR>world<P>sigh</P></BODY></HTML>
```

(4) XHTML produced by the traversal. This XHTML is sent to Java via LiveConnect.

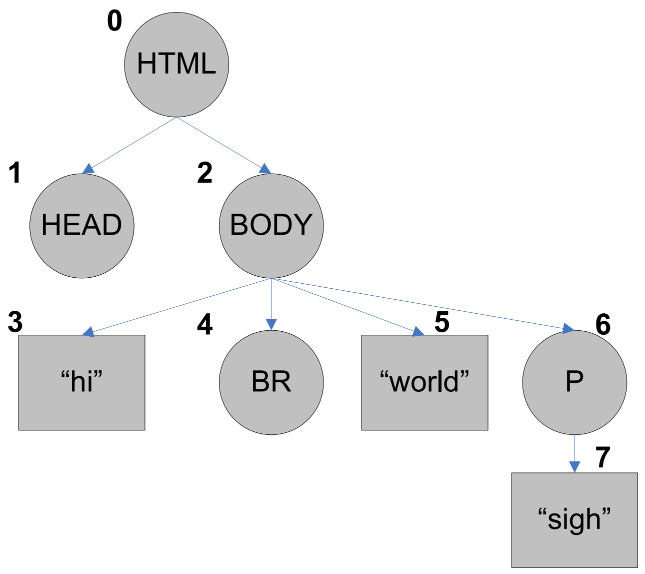
Figure 8.1 Creating the XHTML in Firefox

Because the XHTML produced by the traversal is guaranteed to be well-formed, LAPIS can use its XML parser to recreate the DOM in Java from the XHTML string. In Java, Chickenfoot uses the Xerces XML parser to parse the XHTML, but then it wraps each node in the Xerces DOM with a class called `AnnotatedNode`, and stores the tree of `AnnotatedNodes` in a class called `MozillaDocumentParseTree`. By wrapping a Xerces DOM node, an `AnnotatedNode` can store additional information about the node, such as the index of the node in a preorder traversal of the tree – this will be important later when mapping from a `MozillaDocumentParseTree` to a Firefox DOM.

Once the `MozillaDocumentParseTree` has been created, it needs to be flattened into a string of HTML because a document in LAPIS must be created from a string. Because LAPIS will delimit pattern matches by offsets into this string, a mapping between positions in the DOM and character offsets in the string is created as the DOM is flattened into HTML so that LAPIS pattern matches can be mapped back into positions in the DOM when necessary. The process of converting the XHTML to HTML in Java is shown in Figure 8.2.

```
<HTML><HEAD></HEAD><BODY>hi<BR></BR>world<P>sigh</P></BODY></HTML>
```

(1) XHTML received from Firefox to be parsed by Xerces.



(2) `MozillaDocumentParseTree` that wraps the DOM created by the Xerces XML parser.

```
<HTML><HEAD></HEAD><BODY>hi<BR>world<P>sigh</P></BODY></HTML>
```

(3) HTML produced by traversing the `MozillaDocumentParseTree`. This HTML is sent to LAPIS.

Figure 8.2 Recreating the DOM and generating the HTML in Java

Unfortunately, a LAPIS document cannot be created directly from the XHTML rather than the HTML because an XHTML document is not necessarily a valid HTML document. For example, the XHTML in Figure 8.2 has an opening and closing tag for `<BR>`, but according to the HTML 4.01 specification, closing tags are forbidden on `<BR>` as well as some other elements. Because closing tags are required on all elements in XHTML [30] the XHTML must be converted to HTML before being passed to LAPIS.

Another alternative would be for Chickenfoot to create a valid HTML string from the DOM in JavaScript rather than an XHTML string; however, this would take the XML parser out of the loop on the Java side, and so no Java DOM would be created. Without a Java DOM, it would be difficult to map from LAPIS coordinates to positions in the Firefox DOM, so the additional mapping from XHTML to HTML must remain part of the system.

It would seem that much of this complexity could be avoided if Chickenfoot simply sent LAPIS the URL of the page that is currently being displayed in Firefox (which would surely be less expensive to send over the JavaScript-Java boundary) and had LAPIS fetch the page and parse it itself. Unfortunately, because LAPIS does not have the same cookies, authentication, etc. that Firefox has, LAPIS might not get the same HTML as Firefox when it tries to access the same URL, so this intricate mapping must be created between Firefox and LAPIS to ensure that the two representations of the page are consistent.

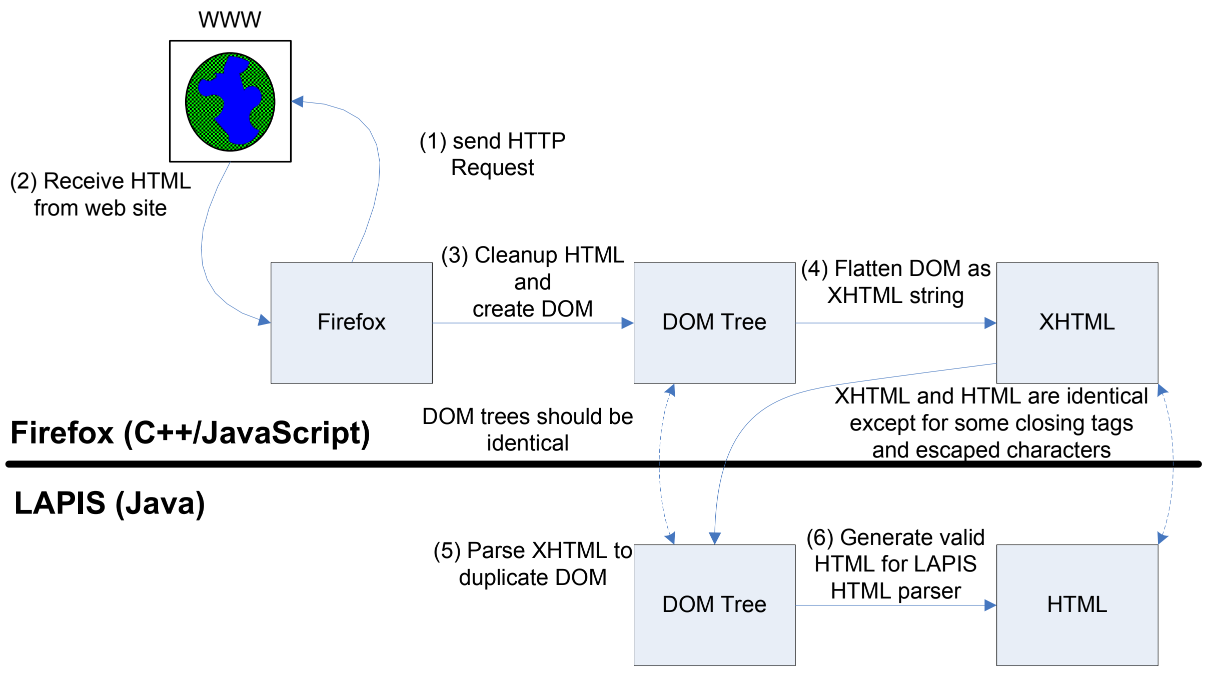

WWW
(1) send HTTP Request
(2) Receive HTML from web site
Firefox
(3) Cleanup HTML and create DOM
DOM Tree
(4) Flatten DOM as XHTML string
XHTML
Firefox (C++/JavaScript)
DOM trees should be identical
XHTML and HTML are identical except for some closing tags and escaped characters
LAPIS (Java)
(5) Parse XHTML to duplicate DOM
DOM Tree
(6) Generate valid HTML for LAPIS HTML parser
HTML

Figure 8.3 Construction of Mapping between Firefox and LAPIS

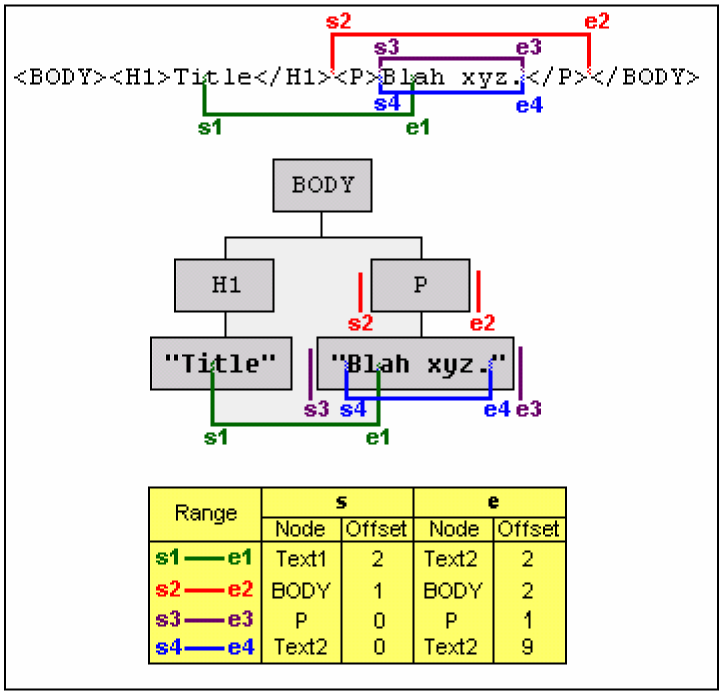
8.2 How Chickenfoot Operates on this Model

As described in Chapter 3, Chickenscratch provides commands that let the user talk about objects in a web page in terms of the rendered model. This section explains how objects in this high-level model are translated to those in the lower-level DOM.

8.2.1 How the find() Command Works

The `find` command needs to be able to take a text constraint pattern and create a list of `Match` objects that correspond to the matches that LAPIS finds in the document's string model. To do this, the first step is to run LAPIS on the HTML that was generated from the DOM in the previous section. The matches found by LAPIS are represented as a *RegionSet*, which is a set of Region objects where each *Region* is a substring of the HTML that contains the content of the pattern match. Each Region is represented by the start and end indices of the substring in the HTML.

The next step is to map each Region into an equivalent section of the DOM called a Range. As described by the W3C specification, a DOM Range "identifies a range of content in a Document. . .It is contiguous in the sense that it can be characterized as selecting all of the content between a pair of boundary-points." [31] As shown in Figure 8.4, each boundary-point is defined by a node and an offset, so two nodes and two offsets (which are nonnegative integers) completely determine a Range. For a boundary-point whose node is a text node, its offset is a 0-based index into the node's text string. For all other boundary-points, its offset is the node's position among its siblings.



Range	s		e	
	Node	Offset	Node	Offset
s1 — e1	Text1	2	Text2	2
s2 — e2	BODY	1	BODY	2
s3 — e3	P	0	P	1
s4 — e4	Text2	0	Text2	9

Figure 8.4 Illustration of a Range [32]

Therefore, to convert a Region to a Range, the start and end indices of the Region need to be converted into boundary-points. This is done by finding the most specific node that contains the index. Consider the HTML created from the DOM in the previous section:

```
<HTML><HEAD></HEAD><BODY>hi<BR>world<P>sigh</P></BODY></HTML>
```

The Regions that correspond to the nodes in the DOM are as follows:

Id	Node Value	Region
0	<HTML>	[0, 61]
1	<HEAD>	[6, 19)
2	<BODY>	[19, 54]
3	"hi"	[25, 27)
4	<BR>	[27, 31]
5	"world"	[31, 36]
6	<P>	[36, 47]
7	"sigh"	[39, 43]

Table 8-2 Regions for DOM nodes

To find the boundary point for index 41, consider each node and see if 41 is within its Region. It turns out that the Regions for nodes 0, 2, 6, and 7 all contain index 41; however, node 7 has the smallest Region containing 41, so it will be the node for 41's boundary-point. Because Regions for nodes are strictly nested, there must always be a smallest Region that contains an index.

Id	Node Value	Region
0	<html>	[0, 61]
1	<head>	[6, 19]
2	<body>	[19, 54]
3	"hi"	[25, 27]
4	<br>	[27, 31]
5	"world"	[31, 36]
6	<p>	[36, 47]
7	"sigh"	[39, 43]

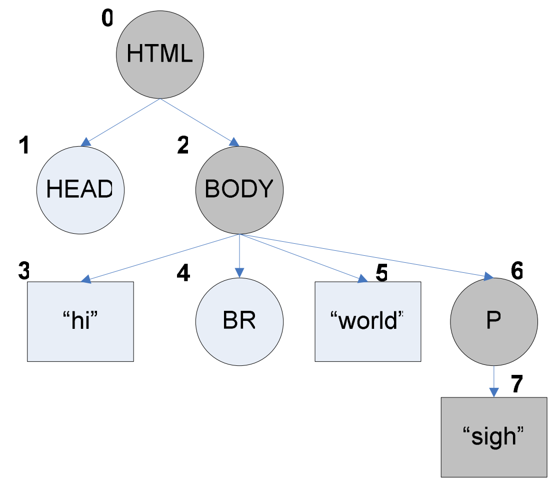


Figure 8.5 Finding the node for the boundary-point for index 41

Since the start index for node 7 is 39, then the offset of 41 within node 7 is (41 – 39) or 2, so the offset for 41's boundary-point is 2.

When each node knows the indices of the Region that covers it, an index can be converted into a boundary point in $O(b \log_b N)$ time where N is the number of nodes and b is the branching factor of the DOM tree. As the start and end indices are included in each `AnnotatedNode` as the `MozillaDocumentParseTree` is built, converting indices to boundary-points does indeed run in $O(b \log_b N)$ time.

Things become slightly trickier when a coordinate coincides with the beginning or end of a node. For example, index 43 (the `<` in `</P>`) could be considered the boundary point at node 7 with offset 4, or the boundary point at node 6 with offset 1. Chickenfoot prefers mapping Regions to elements rather than substrings, so the Region [39, 43] would be mapped to the boundary-point at node 6 with offsets 0 and 1 rather than the boundary-point at node 7 with offsets 0 and 4.

Elements are preferred to substrings because moving nodes in the DOM is simpler than moving text, as shown in section 8.3.1.

Because the Firefox DOM and the Java DOM are identical trees, the index of a node in a preorder traversal in one tree can identify the corresponding node in the other. Thus, a Range can be sent across the JavaScript-Java boundary as two pairs of integers where each pair represents a boundary-point such that the first integer is the index of the node and the second integer is the offset. For example, the Range covering the children of `<BODY>` could be expressed as ((2,0),(2,4)).This establishes a consistent mapping between LAPIS Regions and DOM Ranges.

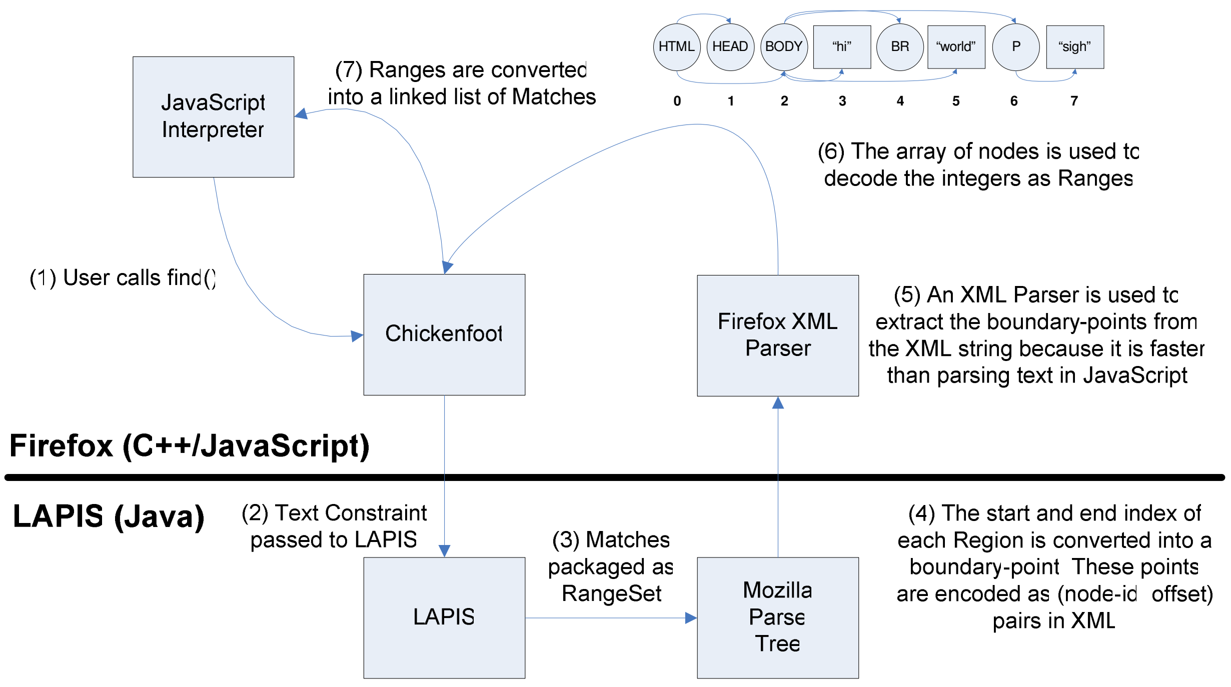
Once `find` has a Range that corresponds to each match found by LAPIS, the final step is to create a `Match` object for each Range. The fields of a new `Match` object are a populated by a Range by the rules in Table 8-3.

`next`	a reference to the empty `Match`
`hasMatch`	true
`count`	1
`index`	0
`range`	a reference to the Range
`content`	a DocumentFragment cloned from the Range
`element`	if the Range delimits a single Element node, then `element` is non-null reference to that Element; otherwise, it is null
`document`	the HTMLDocument that `range` belongs to
`html`	the HTML produced by a preorder traversal of the Range
`text`	`range.toString()`

Table 8-3 Rules for creating a Match from a Range

As `find` converts each Range to a `Match`, it builds up a linked list of `Match`es and returns a reference to the first `Match` in the sequence.

The implementation of `find` is outlined in a flow chart in Figure 8.6.



HTML
HEAD
BODY
"hi"
BR
"world"
P
"sigh"
0 1 2 3 4 5 6 7
JavaScript Interpreter
(7) Ranges are converted into a linked list of Matches
(6) The array of nodes is used to decode the integers as Ranges
(1) User calls find()
Chickenfoot
Firefox XML Parser
(5) An XML Parser is used to extract the boundary-points from the XML string because it is faster than parsing text in JavaScript
Firefox (C++/JavaScript)
LAPIS (Java)
(2) Text Constraint passed to LAPIS
(3) Matches packaged as RangeSet
(4) The start and end index of each Region is converted into a boundary-point. These points are encoded as (node-id, offset) pairs in XML
LAPIS
Mozilla Parse Tree

Figure 8.6 Implementation of find

8.2.2 How the insert() and remove() Commands Work

`insert` and `remove` work by translating their arguments into nodes and then using standard methods for adding and removing nodes to mutate the DOM. Recall that the definitions of the functions are as follows:

```
insert(Position position, Chunk chunk) // returns a Chunk
remove(Pattern pattern)                // returns a Position
```

`insert` converts a `Position` into a *collapsed Range*, which is a Range whose start and end boundary-points are the same. It then converts a `Chunk` into a node, and uses the `insertNode` method of the Range to insert the node into the DOM.

Because a `Position` is a `Pattern`, converting a `Position` into a Range can be broken into cases. Recall that a `Pattern` is one of five types, two of which are strings, so the four cases are as follows:

- **String.** Use the string as an argument to `find` to get a `Match`. This reduces the **String** case to the **Match** case.
- **Match.** Return its `range`. Note that this may be null.
- **Node.** Create a new Range whose start and end container is the node's parent node and whose start and end offsets delimit the node.
- **Range.** Already a Range.

If the Range produced by this conversion is null or is not a collapsed Range, then `insert` will throw an error, as specified.

The process of converting a `Chunk` into a node may also be broken into cases:

- **String.** Use the current document to create a Range and pass the string to the Range's `createContextualFragment()` method which will produce a DocumentFragment which is a subtype of node.
- **Match.** If the `Match`'s `range` is available, this reduces to the **Range** case. If `range` is null, then the `Match`'s `html` is used, and this reduces to the **String** case.
- **Node.** Already a node.
- **Range.** Use the Range's `cloneContents()` method to get the Range as a DocumentFragment, which is a subtype of node.
- **Link or Button.** Already an element node.

`remove` is implemented by converting a `Pattern` into a Range and then invoking the `deleteContents()` method of that Range. A side-effect of invoking `deleteContents()` is that it collapses the Range, so once the Range has been collapsed, it is a valid `Position` that can be returned by `remove`. As the algorithm for converting a `Pattern` into a Range is defined above, implementing `remove` is trivial.

8.3 Updates to the Model

Because Chickenfoot uses data structures that are built on top of the DOM and its derivative Ranges, changes to either of these objects must trigger updates to Chickenfoot's data structures to reflect the objects' new state. How Chickenfoot is notified of these changes and how it deals with them is explained in this section.

8.3.1 Updates to the DOM

There are three types of changes that can be made to the DOM: a node can be inserted, removed, or mutated. Fortunately, the DOM allows clients to add themselves as listeners for each of these events, which Chickenfoot does.

When a node is added to the DOM, the node and its descendants are traversed to create a string of XHTML in the same way an HTML document is flattened. During the traversal, the nodes are assigned ids (starting with the last available id for the DOM) and are added to the id-indexed array of nodes for the document. Once this is complete, the XHTML is sent over the JavaScript-Java boundary, along with the id of the parent of the node that was inserted and the node's index within its siblings. This information is sufficient to update the Java DOM.

In Java, the XHTML is parsed as before, and it also numbers these new nodes starting with the last available id. It looks up the parent node by id number, and inserts the subtree that it has created from the XHTML at the specified index among its children. This ensures that the Java DOM is still identical to the Firefox DOM, and that the id numbers used in both DOMs are consistent.

When a node is removed from the DOM, only the id of the node that was removed needs to be sent to Java. When the Java DOM receives this id, it simply removes it from its DOM, as well.

There are two types of node mutations that can occur in the DOM: changing the text of a text node and changing the attribute of an element. To notify the Java DOM of a change to a text node, only the id of the node and the new text of the node need to be sent. The message for an attribute update is also simple, requiring the id of the node whose attribute was changed, and two strings, one for the key and one for the value of the attribute. These updates are simple to perform on the Java DOM.

After any of these three mutations to the Java DOM, Chickenfoot must also regenerate the HTML from the DOM so that it can create a new document in LAPIS. Fortunately, LAPIS has a mechanism for creating mappings from old versions of documents to new ones, so this is used so that matches in the old DOM can be translated to matches in the updated DOM.

8.3.2 Updates to Ranges

A Range over a DOM might not contain the same content after the DOM has been mutated. Although the DOM 2 Range Specification defines a policy for Range modification under DOM mutation, Chickenfoot adds two improvements to the standard policy. As shown in Table 3-1, a `Match` can be completely determined by its `range`, so enusring that Ranges match the same content after mutations to the DOM is sufficient for ensuring that `Match`es match the same content, as well.

The DOM 2 Range specification states the following:

> **2.12 Range modification under document mutation**
> *There are two general principles which apply to Ranges under document mutation: The first is that all Ranges in a document will remain valid after any mutation operation and the second is that, as much as possible, all Ranges will select the same portion of the document after any mutation operation.*

It then goes on to define the following policy for updating Ranges under insertions:

> **2.12.1 Insertions**
> *An insertion occurs at a single point, the insertion point, in the document. For any Range in the document tree, consider each boundary-point. The only case in which the boundary-point will be changed after the insertion is when the boundary-point and the insertion point have the same container and the offset of the insertion point is strictly less than the offset of the Range's boundary-point. In that case the offset of the Range's boundary-point will be increased so that it is between the same nodes or characters as it was before the insertion.*

The above policy for insertion is not as good as it should be in terms of maintaining the second general principle listed in **2.12**. For example, consider the following HTML:

```
<P><I>thing1</I><I>thing2</I></P>
```

and Ranges R1 and R2 that match the `<I>` elements:

R1 = [(`<P>` , 0) , (`<P>` , 1)], so R1 corresponds to `<I>thing1</I>`
R2 = [(`<P>` , 1) , (`<P>` , 2)], so R2 corresponds to `<I>thing2</I>`

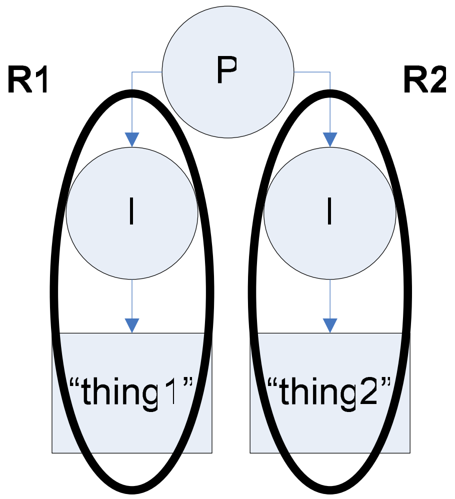


Figure 8.7 DOM with Range content outlined with ovals

Consider what happens when `<b>bold</b>` is inserted before `<i>thing2</i>`. Because "the only case in which the boundary-point will be changed after the insertion is when the boundary-point and the insertion point have the same *container* and the *offset* of the insertion point is strictly less than the *offset* of the Range's boundary-point," the only boundary point that gets changed is the endpoint of R2. Therefore, after the insertion:

R1 = [(`<p>` , 0) , (`<p>` , 1)], so R1 corresponds to `<i>thing1</i>`
R2 = [(`<p>` , 1) , (`<p>` , 3)], so R2 corresponds to `<b>bold1</b><i>thing2</i>`

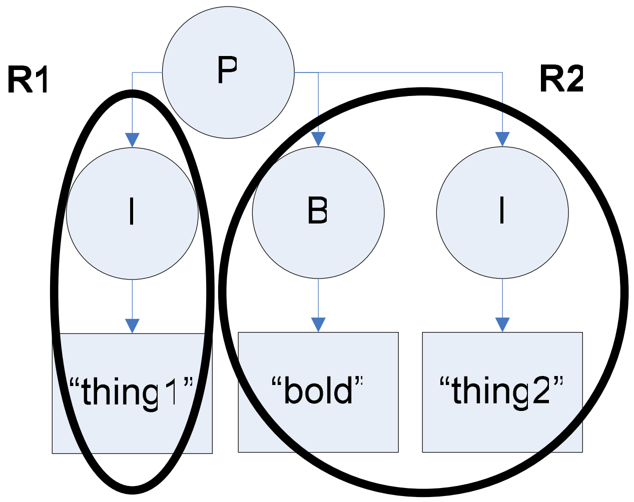


Figure 8.8 DOM with Range content outlined with ovals after insertion

Clearly, R2 does not select the same portion of the document after the mutation operation. This would be a problem for a `Match` that had R2 as its Range because its `toString()` method would return a different value after the new node was inserted even though the new node should have no effect on the `Match`. If the Range update policy were such that R2 would be [(`<p>` , 2) , (`<p>` , 3)] after the mutation, then the `Match` would be consistent. Therefore, Chickenfoot uses the following policy to update Ranges, in addition to the DOM 2 policy:

For any Range in the document tree, consider each boundary-point. The boundary-point will also be changed after the insertion when the boundary-point is a startpoint of a Range, the startContainer is not a text node, and the insertion point is equal to the boundary-point. In that case the offset of the Range's boundary-point will be increased so that it is at the start of the same node as it was before the insertion.

But what about the case where the container of the boundary-point lies inside a text node? In this case, the update is more complicated because it involves creating new nodes. Consider the following HTML:

```
<body>My cat's breath smells like cat food.</body>
```

and Ranges R1 and R2 that match the string `cat`:

R1 = [(#text1 , 3) , (#text1 , 6)], so R1 corresponds to the first appearance of `cat`
R2 = [(#text1 , 28) , (#text1 , 31)], so R2 corresponds to the second appearance of `cat`

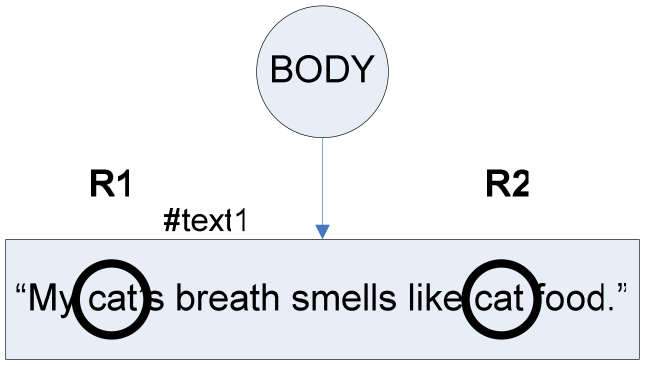


Figure 8.9 DOM with Range content outlined with ovals

Doing an insert of a new text node containing the string `monster` at [(#text1 , 6)] produces the following:

R1 = [(#text1 , 3) , (#text1 , 6)], so R1 corresponds to the first appearance of `cat`
R2 = [(#text1 , 6) , (#text1 , 6)], so R2 corresponds to a point between #text1 and #text2

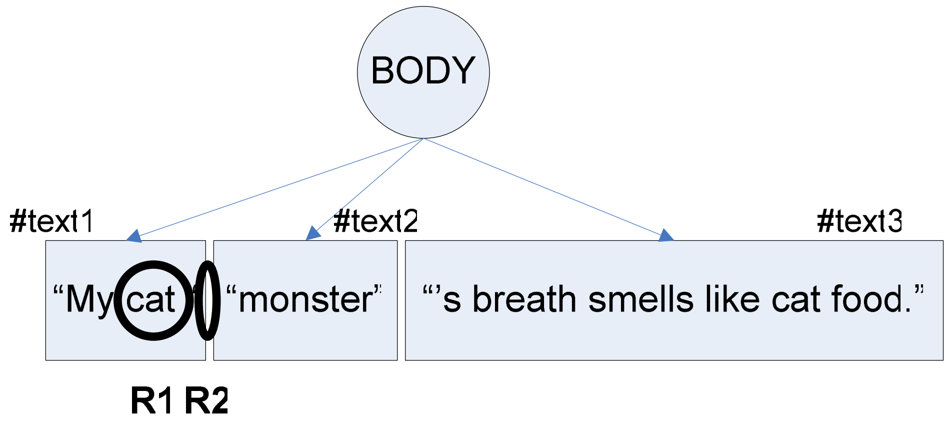


Figure 8.10 with Range content outlined with ovals after insertion

Now there are three text nodes that are consecutive children of `<body>`:

#text1 is `My cat`
#text2 is `monster`
#text3 is `’s breath smells like cat food.`

Note that #text1 has not been replaced with a new node with different text. Instead, the value of #text1 has been changed to the new text. Whether #text1 should be mutated or replaced is not specified by the DOM specification; however, Firefox implements the specification by mutating the first node.

In this case, we want R2 to point to a node that did not exist before the insertion was made. Note that in this case, R2 satisfies the original requirements for changing its boundary point, in that the offset of the insertion point is strictly less than the startpoint for R2; however, the update for R2 still fails to change R2 in such a way that it still selects the same portion of the document. Again, since the original selection for R2 is now at [(#text3 , 22), (#text3, 25)], this suggests that the rules for updating Ranges after mutation can be improved upon even further to accommodate nodes that are created as a result of insert(). To this end, Chickenfoot uses the following policy for updating Range boundary-points upon insertion, in addition to the DOM 2 policy:

If an insertion point lies within a text node, consider all Ranges in the document tree that have a boundary-point whose container is equal to the text node. If the boundary-point is before or equal to the insertion point, then both the container for the start or end (depending on the type of boundary point) will be changed to the new text node that was created, and the offset will be changed to (oldOffset – oldNode.nodeValue.length).

To implement these policies, Chickenfoot needs to know about every Range that has been created in the DOM, and to update each one whenever a node is removed or inserted. Section 8.3.1 already explained how these mutation events can be captured, but getting a reference to all of the Ranges is still a problem.

Because the only way to create a Range in the DOM is to call its `createRange` method, Chickenfoot replaces this method with one that delegates to the original method to create the

Range, but then keeps a reference to it before it is returned to the client. This collection of references is stored so that a Range can be looked up in constant time when the node for either of its boundary-points is mutated. When a mutation occurs, potentially affected Ranges are inspected, and the Chickenfoot policies for Range modification under document mutation are applied, if necessary.

8.4 How Chickenfoot Scripts are Interpreted

Firefox renders the DOM to produce the graphical view of the web page that the end user sees. Any change to the underlying DOM is reflected immediately in the rendered view of the web page, so Chickenfoot scripts effectively work by manipulating the DOM.

In Firefox (and in most browsers), users can manipulate the DOM in JavaScript through an object named `document`. Chickenfoot builds upon this DOM access by creating an extended JavaScript environment called an *evaluation context* in which Chickenfoot scripts are evaluated. In this way, a Chickenfoot script can have access to all of the objects and functions that a JavaScript programmer is accustomed to having, in addition to the higher-level objects and functions that Chickenfoot provides. This is accomplished by taking the text of a Chickenfoot script as a string and evaluating it inside of the evaluation context by using the `with` and `eval` functions in JavaScript:

```
// the variable, script, passed to this function
// is the source code of the user's script as a string
function interpret(script) {

  // This is the evaluation context for the script.
  // Familiar objects, such as document, are defined here,
  // in addition to Chickenfoot commands,
  // such as find() and click().
  evaluationContext = {
    document getter : function() { return getDoc(); },
    find            : function(pattern) { ... },
    click           : function(pattern) { ... },
    ...
  };

  // this evaluates the script in the evaluation context
  with (evaluationContext) {
    eval(script);
  }

}
```

Though the use of `with` in JavaScript is frowned upon because it is often misused [33], it is used appropriately in Chickenfoot. Misuse occurs when the client tries to assign a value to a non-existent field to the object added to the scope chain by `with`:

```
var x = { a : 2 }

with (x) {
  this.a = 3
  this.b = 4 // b is not defined in x, so this has no effect
}

alert(x.a + x.b) // displays NaN because x.b is undefined
```

In the code above, `x.a` is changed from 2 to 3, but `b` is not added as a field of `x`, as the user might expect. Because `b` was not defined in `x` before the `with` statement, its assignment inside the `with` statement has no effect. In Chickenfoot, this is not a concern because the `with` statement is not used to assign values to `evaluationContent`; instead, it is used to run user code in an environment where Chickenscratch commands are in scope, which it does as desired.

8.5 Monitoring Page Loads

It is important to keep track of when web pages are loading in Firefox so that Chickenfoot does not try to operate on partially-loaded pages. This section explains how page loads are monitored, and how Chickenfoot uses this information to suspend an operation until a page has finished loading.

8.5.1 Listening for Load Events

Firefox provides a ProgressListener interface that is notified upon updates to the progress bar: this includes the beginning and end of a page load, including some intermediary updates about what percentage of the page has been downloaded thus far. There is also a LoadListener interface that is notified when a page is completely loaded into the browser. Unfortunately, the ProgressListener reaching 100% does not precisely coincide with a load event because Firefox takes some additional time to finish processing the HTML after it has received all of the bytes from the server.

A web page in Firefox is displayed inside a *tab*, and Chickenfoot keeps track of the loading state of each tab in the browser. First, it registers with Firefox as ProgressListener and a LoadListener. When it receives a STATE_START event from the ProgressListener, it checks the event to see which tab triggered it, and marks that tab as *loading*. When it receives a LOAD event from the LoadListener, it also checks the event to see which tab triggered it, and marks that tab as *loaded*.

8.5.2 Waiting Until a Load is Complete

In the current implementation of Chickenfoot, all accesses to a web page go through the `document` object. In the evaluation context described in section 8.4, `document` is a reference to a function that returns an object that wraps the DOM rather than returning a reference to the DOM itself. By making `document` a function, Chickenfoot can suspend itself until the DOM has finished loading. This is completely abstracted from the user.

When `document` is called, it asks the browser which tab is currently in focus. Once it finds that tab, it asks it which state it is in. If it is in the *loaded* state, then it returns the tab's DOM immediately. Otherwise, the tab is in the *loading* state, so the tab is polled in 100ms intervals until it is *loaded*, at which point the DOM is returned.

Using this approach, a Chickenfoot script can fire commands that start loading pages, and the script can continue executing in parallel with the loads until it reaches a point where it tries to access a page that is in the middle of loading. When that happens, the script will suspend execution until the load is complete, and will resume when the page is loaded. This allows end-users to write scripts without having to worry about this synchronization.

8.5.3 Using Page Loads for URL Triggers

Because Chickenfoot is receiving LOAD events, it uses these events to fire URL triggers. Upon receiving a LOAD event, Chickenfoot looks through the list of URL triggers that the user has defined. For each one, it tries to match the URL of the page that has been loaded against the trigger's URL pattern. If the loaded page matches the regexp for the URL pattern, then it runs the script associated with the trigger.

Chapter 9 Conclusion

Chickenfoot is a web automation toolkit designed to make it easy for end-users to develop scripts to change their web experience: its command language abstracts the underlying representation of a web page so that users can operate on it at high level, its development environment is conducive to experimentation so scripts can be prototyped quickly, and its trigger system makes it possible to seamlessly integrate user customizations into everyday browsing.

9.1 Contributions

In this thesis, I have introduced a system, Chickenfoot, that empowers end-user programmers to automate and customize web pages without viewing their HTML source. I have done this by integrating programming and pattern languages that are focused on describing commands and objects relevant to interactions with web pages.

As part of Chickenfoot's language, Chickenscratch, I have introduced keyword patterns, which are patterns that use the spatial location of keywords in a web page to identify page elements. The web survey data that I provide supports the usability of this technique. I have also presented an algorithm for resolving keyword patterns, and have demonstrated its success on a modest amount of training data.

By embedding Chickenfoot in the Firefox web browser, I have created a development environment that encourages experimentation and spontaneity in web scripting. My environment provides tools that help the user create and debug Chickenscratch code. As a side-effect, I have also created an application for developing JavaScript programs and extensions to Firefox.

I have introduced the rendered model of a web page, which builds upon the Document Object Model that most web browsers use. In creating this model, I offer improvements to the W3C DOM specification for updating Ranges of a DOM under mutation.

Finally, by enabling users to store scripts as triggers, I have given end-user programmers the ability to automate and customize their web experience.

9.2 Future Work

Though the core of the Chickenfoot system has been implemented, there are still many extensions to the system that we would like to implement.

9.2.1 Packaging Scripts

Package Chickenfoot script as a standalone Firefox extension.
Currently, if a user wishes to run a Chickenfoot script that someone else has created, then he must get its source code, install Chickenfoot, and run the code. It would be much easier if the script author could package his script as one file that a user could install and run without downloading Chickenfoot. For Firefox, the logical thing to do is to package the Chickenfoot script as its own Firefox extension. As a Firefox extension is bundled as a single XPI file (pronounced "zippy"), and can be installed by clicking on a hyperlink on a web page, packaging Chickenfoot scripts as XPIs would make it simpler for script authors to deploy their creations to users.

A tool was recently created to package Greasemonkey scripts as a XPIs,[34] so it should not be difficult to create a analogous tool for Chickenfoot. This packaging tool would take a script and a trigger (which is effectively a URL pattern), and in turn create a Firefox extension that ran the script whenever the user visited a page that matched the trigger pattern. The extension would expose the trigger as a setting that could be overridden by the end-user.

Because packaging the script as an extension would not expose the code to the user, Chickenfoot should also have the ability to read XPI files, so script authors could share code in this way, as well. This would also make it easier to explore existing Firefox extensions that were created with tools other than Chickenfoot.

Package Chickenfoot script so it can run as a function on a mobile device.
It is difficult to browse the Web on mobile devices because of the small screen and the absence of a full keyboard. Despite this, many users want to be able to browse the Web from their cell phone so they can get driving directions and other information from the Web that is particularly useful when they are away from their computers. Alex Faaborg, a student at the MIT Media Lab, notes the following when discussing his own macro recorder for the Web:

> I believe that one of the reasons Web browsing doesn't work well on mobile devices is that users don't want to *browse* on mobile devices; they want to quickly and easily retrieve a specific piece of information, or complete a specific action. One of the benefits of [my] application is that it reduces complex processes on the Web to their minimum input and output. Users could record a process on the web, and then copy that process onto their mobile device. . . The process of logging in and retrieving this information using a full sized computer display can be reduced to a single click on a cell phone. [35]

It should be possible to package Chickenfoot scripts in the same way, minimizing processes on the Web to their minimum input and output, so that users can easily do the tasks that are most important to them on their mobile device, even if general web browsing is difficult.

9.2.2 By Demonstration

Make actions logged in the Actions pane on par with those recorded in WebVCR.
The user actions logged in the Actions pane are minimal: only page navigations are recorded in the current implementation of Chickenfoot. This is unfortunate because a better logging system would make it possible for users to demonstrate their activity in the browser and copy the content of the Actions pane rather than trying to compose a script on their own and making errors. This could help novice users learn the Chickenfoot language, as well.

By taking the approach used in WebVCR, listeners could be added to web forms when a page is loaded so that all user input could be logged. For example, after a user does a search from the Google home page, the content of the Actions pane would be:

```
go('http://www.google.com/')
enter('search terms')
click('Google Search')
```

Duplicating the work of WebVCR to add the listeners is not difficult; however, translating the actions into appropriate Chickenfoot code is a challenge. Currently, when a user writes a Chickenfoot script, he provides a keyword pattern that Chickenfoot resolves to a web component. But to do the converse, that is, to take a web component and create a keyword pattern for it, is its own research problem. For example, consider the log of a user doing a web search on Yahoo! instead of Google:

```
go('http://www.yahoo.com/')
enter(???, 'search terms')   // what should replace ???
click('Search the Web')
```

Determining how to name the textfield is an open question. "`Search the Web`" would probably be the most intuitive name for the user, but "`Images`" also matches the correct textfield. Coming up with candidate keyword patterns for a component and choosing the best one is an open problem for Chickenfoot.

Enable user to discover patterns by selecting content in a web page.
Some patterns are difficult or tedious for a user to define in Chickenfoot. For example, a reasonable attempt at a LAPIS pattern for a Google search result is: `paragraph just before hostname`. But coming up with this pattern may require the user to do some prolonged experimentation in the Patterns pane; and further, this pattern does not even reliably match Google search results. Rather than go through the process of trial-and-error, it would be quicker for a user to select an example of a Google search result in the page and have LAPIS suggest patterns to the user. LAPIS already supports pattern-suggestion in the standalone version, so it is simply a matter of porting this functionality over to Chickenfoot.

Build a programming-by-demonstration (PBD) system on top of Chickenfoot.
As Chickenfoot is targeted at end-user programmers, it would be even easier for novice programmers to learn if they did not have to write any code at all! This could be done by adding functionality to Chickenfoot to make it behave like a macro recorder, such as WebVCR or

LiveAgent, where the user explicitly records the task they would like to automate. However, a more interesting system would be one that could data mine the history of a user's web activity for usage patterns, and then suggest and write scripts that would automate actions that the user appeared to do often.

Automatic discovery of web services.
Web pages are subject to change in ways that break Chickenfoot scripts. On the other hand, web services are stable APIs that are unlikely to change. Sites such as Amazon and Google provide web services that allow users to make programmatic queries that are equivalent to the ones that they do by filling out web forms on the same site.

It would be helpful if Chickenfoot could examine the history of queries that a user made to a web site, and their results, and then compare them to the results of doing the same queries through the site's web services. If it found a match, then it could create a Chickenfoot function that abstracted the SOAP calls to the web service. This would give the user reliable programmatic access to the information that he wants.

Such a system could be implemented by using UDDI [36] to find out if a site provides web services. Once it discovered the web services, it could use a brute-force approach, trying the user's inputs on each service the site provides and comparing the web service output to the content the user sees in a web page when he does the same query. This approach may not be efficient, but it suggests that a solution is possible.

9.2.3 User Interface

Provide a graphical view of a Chickenfoot script.
For commands that deal with automating web forms, it would be helpful for users to be able to see the component that the command would affect. For example, displaying the button or hyperlink that would be matched by the Pattern argument to a `click` command would help users predict the effects of running their script. Also, in the event that a web page changed so that the Pattern now matched a different link or button, the user would be more likely to notice the change before running the script because he would see that the image of the component had changed.

Provide a Chickenfoot interpreter or multiple editors.
When users are developing a script in Chickenfoot, there is often a portion of the script that the user has completed as well as a portion of the script that the user is experimenting with. Unfortunately, there is only one editor, so both portions of the script are in the same buffer and the user is frequently commenting and uncommenting portions of the script. The user should be able to have separate space for code that is working and code that is under development.

One approach is the one employed by StarLogo [37], which is a programming environment for creating simulations that is aimed at middle school students. Its editor is split into a top and bottom pane where the top pane is a StarLogo interpreter and the bottom pane contains functions that the user has created. This allows the user to experiment in the top pane and move completed code into the bottom pane.

Another solution would be to have a tabbed pane of Chickenfoot editors instead of only one editor. This way, the user could have more control over how he broke up his code. One drawback of this design, however, is that the user would not be able to view code from more than one editor simultaneously.

In either case, sufficient prototyping and user testing should be done to determine how the interface could provide better support for script development.

Support Bookmark Triggers.
One of the major uses of Chickenfoot is to automate navigation, especially to "hard to reach" pages. As users use bookmarks to automatically take them to a page, they should be able to access Chickenfoot scripts from the Bookmarks menu that do that, as well. Bookmarks that are Chickenfoot scripts should appear the same as other bookmarks in the browser to provide seamless integration.

9.2.4 Robustness

Attempt to identify when a script breaks as a result of a change in a web site.
A Chickenfoot script may suddenly stop working if a web site changes such that a Chickenfoot pattern no longer becomes valid. In this case, it would be ideal if Chickenfoot could recognize the error and alert the user to fix it (or even fix the script itself) rather than failing silently or ploughing along ignoring the error, ultimately returning the wrong result.

One solution would be to store the XPath of each component identified by a pattern in a Chickenfoot script. When the script is run subsequently, Chickenfoot could calculate the tree edit distance between the XPath of the component currently identified by the script and the XPath of the component identified by the script the last time that the script was run. If the tree edit distance exceeds a certain threshold, then the script should abort and warn the user that it believes that the pattern used to identify the component may no longer be valid.

9.2.5 Extensions to Pattern Language

Support CSS patterns.
Because Chickenfoot users should be able to talk about the page on the rendered level, they should also be able to use colors and other CSS styles to identify parts of a web page. For example, "green text with black background" should be a valid pattern for matching text. As the DOM has references to style sheets and style data for its elements, it should be possible to programatically resolve CSS patterns with the elements the user is trying to identify.

Support `above` and `below` as relational operators.
Users should be able to use `above` and `below` as operators in the pattern language. For example, a user should be able to use `image above textbox` to identify the logo on the Google home page. Even though `above` and `below` are not constraints that can be applied to the string model as other TCs can, they should be adopted as part of the TC pattern language so their usage is consistent with other relational operators, such as `just before` and `containing`.

Let users define patterns.

In LAPIS, users are able to define their own patterns in terms of the existing pattern language. This functionality should be made available through Chickenfoot's interface. Care must be taken so that users who share scripts that use patterns they have defined will be sure to deploy their user-defined patterns with the script.

9.2.6 Extensions to Command Language

Provide a simpler idiom for iterating over Match objects.
The following syntax for iterating over `Match`es would be simpler for users to read and to write:

```
for (m in find(pattern)) {
  ... // use m
}
```

Unfortunately, JavaScript interpreters only support this syntax for enumerating keys in an associative array. Because only strings can be keys, it is impossible to use this syntax for enumerating over any other type. This presents a problem for Chickenfoot because the desired syntax shown above is intended to enumerate `Match` objects, not string objects. However, this problem can be solved by rewriting every instance of this:

```
for (A in B) {
  ...
}
```

as this:

```
var b = B
if (b instanceof Match) {
  for (A = b; A.hasMatch; A = A.next) {
    ...
  }
} else {
  for (A in b) {
    ...
  }
}
```

before passing the script to the JavaScript interpreter. This rewriting would most likely have to be performed on the abstract syntax tree (AST) of a script, which unfortunately is not exposed by the Firefox JavaScript interpreter. Rhino [38] is an open-source JavaScript-1.5-compliant interpreter written in Java that provides access to the AST, so hopefully it can be incorporated into Chickenfoot to perform this translation.

Enable users to tag semantic web data and refer to ontologies in Chickenfoot scripts.
If a web page contains semantic web data, then that data should be accessible and scriptable with Chickenfoot. In general, semantic web data will be a more reliable wrapper for information in a page than a Chickenfoot pattern will, so giving Chickenfoot users access to this wrapped data will enable them to write more reliable scripts. Rather than creating a new system for detecting and processing semantic web data in a page, Chickenfoot should be integrated with an existing system, most likely Piggy-Bank [39].

Reduce the amount of quoting used in Chickenfoot scripts.
Chickenfoot scripts often contain many quote characters because keyword and TC patterns appear frequently. Users also construct HTML content from strings, which must be quoted, and the HTML content often contains quotes, as well. This can make it difficult to read and write Chickenfoot scripts. Ideally, quotes would only have to be used to delimit a string when string boundaries are ambiguous.

Add support for other forms of input and output.
Users may want to read or write data from files or databases as part of a Chickenfoot script. Though this is possible to do by scripting XPCOM objects provided by Firefox, the interfaces for these objects are not appropriate for end-user programmers, so they should be wrapped by appropriate, built-in Chickenfoot commands.

9.2.7 Evaluation

Have a "bake-off" between Greasemonkey and Chickenfoot.
Chickenfoot has not been tested in a formal user study to see how it compares to other web automation tools. Because Greasemonkey is also a Firefox extension that enables users to automate and customize the Web by writing JavaScript code, it is an appropriate tool to use for comparison against Chickenfoot. Running a "bake-off" study in which users are given the same set of tasks, some using Chickenfoot and some using Greasemonkey, would provide an estimate on how much of an advantage the Chickenfoot language and development environment provide, if any.

Use data from the Wayback Machine to test how robust Chickenfoot scripts are over time.
The Internet Archive Wayback Machine [40] is a collection of 40 billion web pages archived from 1996 onward. It allows users to view a web site at different points in time over the course of its history. Chickenfoot scripts could be written to automate early versions of a web site, and then could be tested to see if they still worked as the web site changed. This could also be done by scripts developed in other toolkits to serve as a basis for comparison of the robustness of Chickenfoot scripts.

Appendix A Chickenscratch Reference

The following predefined objects and functions are available in Chickenscratch.

Standard JavaScript functions:

```
back()
forward()
```

Standard JavaScript objects:

```
window
document
location
frames
history
screen
status
top
navigator
```

Chickenscratch pattern functions:

```
find
before
after
```

Chickenscratch web form functions:

```
click
enter
check
uncheck
pick
```

Chickenscratch navigation functions:

```
go
fetch
openTab
selectTab
closeTab
```

Chickenscratch page mutation functions:

```
insert
remove
replace
```

Chickenscratch editor functions:

```
output
clear
```

Miscellaneous functions:

```
sleep
```

Chickenfoot objects:

```
Match
```

Chickenfoot types:

```
Pattern
Position
```

Appendix B Partitioning HTML Tags

This is the list of HTML tags that are considered **partitioning** (Section 7.1) in the keyword pattern algorithm:

```
<A>                         <FORM>                      <OPTION>
<ABBR>                      <FRAME>                     <P>
<ACRONYM>                   <FRAMESET>                  <PARAM>
<ADDRESS>                   <H1>                        <PRE>
<APPLET>                    <H2>                        <Q>
<AREA>                      <H3>                        <S>
<B>                         <H4>                        <SAMP>
<BASE>                      <H5>                        <SCRIPT>
<BASEFONT>                  <H6>                        <SELECT>
<BDO>                       <HEAD>                      <SMALL>
<BIG>                       <HR>                        <SPAN>
<BLOCKQUOTE>                <HTML>                      <STRIKE>
<BODY>                      <I>                         <STRONG>
<BR>                        <IFRAME>                    <STYLE>
<BUTTON>                    <IMG>
<CAPTION>                   <INPUT>                     <SUB>
<CENTER>                    <INS>                       <SUP>
<CITE>                      <ISINDEX>                   <TABLE>
<CODE>                      <KBD>                       <TBODY>
<COL>                       <LABEL>                     <TD>
<COLGROUP>                  <LEGEND>                    <TEXTAREA>
<DD>                        <LI>                        <TFOOT>
<DEL>                       <LINK>                      <TH>
<DFN>                       <MAP>                       <THEAD>
<DIR>                       <MENU>                      <TITLE>
<DIV>                       <META>                      <TR>
<DL>                        <NOFRAMES>                  <TT>
<DT>                        <NOSCRIPT>                  <U>
<EM>                        <OBJECT>                    <UL>
<FIELDSET>                  <OL>                        <VAR>
<FONT>                      <OPTGROUP>
```

Bibliography

[1] Burnett, M., Cook, C., Pendse, O., Rothermel, G., Summet, J., and Wallace, C. "End-user software engineering with assertions in the spreadsheet paradigm." Proc. *ICSE*, 2003.

[2] W3C. "Document Object Model (DOM)." www.w3.org/DOM/.

[3] JavaScript 1.5. www.mozilla.org/js/js15.html.

[4] Microsoft. "Smart Tags and Smart Documents." msdn.microsoft.com/office/understanding/smarttags/default.aspx

[5] Sugiura, A. and Koseki, Y. "Internet Scrapbook: Automating web browsing tasks by demonstration." *Proc. UIST '98.*

[6] Miller, R.C. and Bharat, K. "SPHINX: a Framework for Creating Personal, Site-Specific Web Crawlers." *Proc. WWW7*, 1998.

[7] Fujima, J., Lunzer, A., Hornbaek, K., Tanaka, Y. "Clip, connect, clone: combining application elements to build custom interfaces for information access." *Proc UIST 2004*.

[8] Kistler, T. and Marais, H. "WebL – a programming language for the Web." *Proc. WWW7*, 1998.

[9] Perl. www.perl.com/

[10] Mech. search.cpan.org/~petdance/WWW-Mechanize-1.12/lib/WWW/Mechanize.pm

[11] Anupa, V., Freire, J., Kumar, B., and Lieuwen, D. "Automating web navigation with the WebVCR." *Proc. WWW9*, 2000.

[12] Krulwich, B. "Automating the Internet: Agents as User Surrogates." *IEEE Internet Computing*, v1 n4 (July/August 1997).

[13] Barret, R., Maglio P., and Kellem, D. "How to Personalize the Web." *CHI*, 1997.

[14] Ekiwi, LLC. screen-scraper: solutions for web data extraction. www.screen-scraper.com/

[15] Boodman, A. "Greasemonkey." http://greasemonkey.mozdev.org/.

[16] Chickenfoot. www.bolinfest.com/chickenfoot/.

[17] Miller, R.C. and Myers, B.A. "Integrating a Command Shell into a Web Browser." *Proc. USENIX*, 2000.

[18] LiveConnect. http://www.mozilla.org/js/liveconnect/.

[19] Friedl, J. Mastering *Regular Expressions*. O'Reilly, 2002.

[20] Regular Expression Library. http://www.regexlib.com/Search.aspx?k=email

[21] Screen-Scraper extractor patterns. www.screen-scraper.com/support/docs/ using_extractor_patterns.php

[22] W3C. "XML Path language (XPath) Version 1.0," 1999.

[23] GreaseMonkeyUserScripts. http://dunck.us/collab/GreaseMonkeyUserScripts/

[24] Webber, Matthew. "Automatic Web Page Concatenation." MIT AUP, 2005.

[25] Rha, Philip. "Detecting and Parsing Embedded Lightweight Structure." MIT MEng Thesis, 2005.

[26] Karger, et al. Haystack project. http://haystack.csail.mit.edu/

[27] Mozilla. "XML User Interface Language (XUL) Project." http://www.mozilla.org/projects/xul/

[28] Aasted, H. and Palant, W. Adblock. http://adblock.mozdev.org/

[29] Navarro, Gonzalo. "A guided tour to approximate string matching." *CSUR*, v33 n1 (March 2001).

[30] W3C. "XHTML 2.0," 2004.

[31] Document Object Model Range. http://www.w3.org/TR/DOM-Level-2-Traversal-Range/ranges.html

[32] Document Object Model Range diagram. http://www.w3.org/TR/DOM-Level-2-Traversal-Range/ranges.html#Level-2-Range-Position

[33] Flanagan, D. *JavaScript: The Definitive Guide*. O'Reilly, 2001.

[34] Holovaty, Adrian. "Greasemonkey compiler." http://www.holovaty.com/blog/archive/2005/04/24/2227

[35] Faaborg, Alex. *6.831 Final Report*, 2004.

[36] UDDI. http://www.uddi.org/

[37] StarLogo. http://education.mit.edu/starlogo/

[38] Mozilla. "Rhino: JavaScript for Java." http://www.mozilla.org/rhino/.

[39] Hyunh, David. "Piggy-Bank." http://simile.mit.edu/piggy-bank/index.html

[40] Wayback Machine. http://www.waybackmachine.org/.