Detecting and Parsing Embedded Lightweight Structures

by

Philip Rha

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

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2005

Author ....
Department of Electrical Engineering and Computer Science
May 19, 2005

Certified by..................
Rob Miller

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

Detecting and Parsing Embedded Lightweight Structures

by

Philip Rha

Submitted to the Department of Electrical Engineering and Computer Science
on May 19, 2005, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Computer Science

Abstract

Text documents, web pages, and source code are all documents that contain language structures that can be parsed with corresponding parsers. Some documents, like JSP pages, Java tutorial pages, and Java source code, often have language structures that are nested within another language structure. Although parsers exist exclusively for the outer and inner language structure, neither is suited for parsing the embedded structures in the context of the document. This thesis presents a new technique for selectively applying existing parsers on intelligently transformed document content.

The task of parsing these embedded structures can be broken up into two phases: detection of embedded structures and parsing of those embedded structures. In order to detect embedded structures, we take advantage of the fact that there are natural boundaries in any given language in which these embedded structures can appear. We use these natural boundaries to narrow our search space for embedded structures. We further reduce the search space by using statistical analysis of token frequency for different language types. By combining the use of natural boundaries and the use of token frequency analysis, we can, for any given document, generate a set of regions that have a high probability of being an embedded structure. To parse the embedded structures, the text of the region must often be transformed into a form that is readable by the intended parser. Our approach provides a systematic way to transform the document content into a form that is appropriate for the embedded structure parser using simple replacement rules.

Using our knowledge of natural boundaries and statistical analysis of token frequency, we are able to locate regions of embedded structures. Combined with replacement rules which transform document content into a parsable form, we are successfully able to parse a range of documents with embedded structures using existing parsers.

Thesis Supervisor: Rob Miller
Title: Associate Professor

Acknowledgments

I would like to thank Professor Rob Miller, without whose invaluable advice and guidance this thesis would never have been possible. Thanks also to the members of the User Interface Design Group whose feedback and support were sources of constant improvement.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Text documents, web pages, and source code are all documents that contain language structures that can be parsed with corresponding parsers. Some documents, like JSP pages, code tutorial webpages, and Java source code, often have language structures that are nested within another language structure. [1] These *embedded structure documents* pose an interesting parsing problem. Although parsers exist exclusively for the outer and inner language structure, neither is suited for parsing the *embedded structures*, or nested language structures, in the context of the document. Established techniques of signaling the boundaries of embedded structures using explicit markers [2] provide parsers with entry points for a second grammar, but there has been no established technique to parse embedded structures whose boundaries have not been explicitly marked.

This thesis presents a new technique for selectively applying existing parsers on intelligently transformed document content with embedded structures. The general goal of this body of work is to make syntactic information that is inherent to the embedded structures available for other tools and applications. A lightweight approach to this problem is used to parse embedded structures as they are detected without constructing custom parsers.

1.1 Embedded Structure Documents

A key to detecting and parsing embedded structures lies in the nature of embedded documents themselves. The embedded structure, or *snippet*, is nested within another language type. This encompassing language is known as the *containing structure.* There are many types of embedded language documents, and each type embeds its structures in different ways.

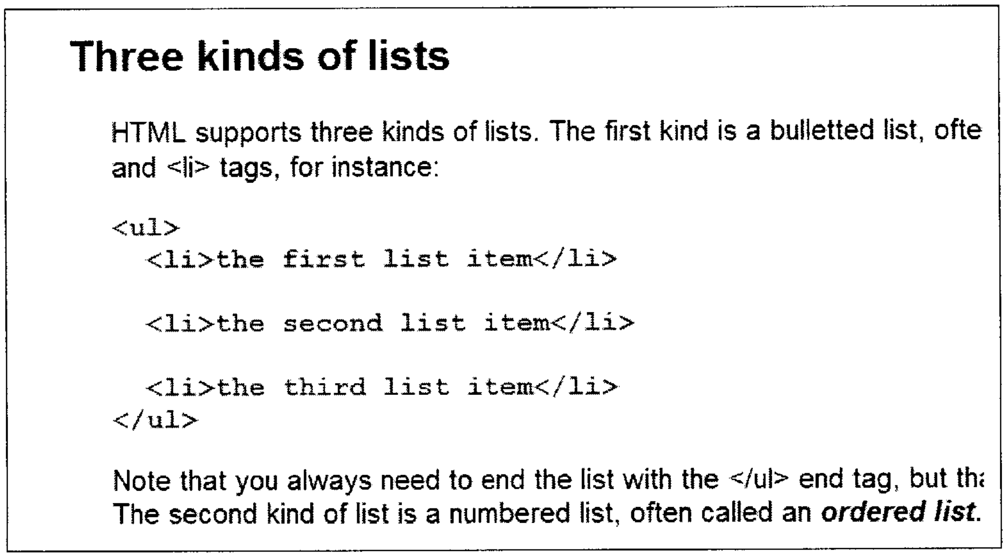
Some examples of embedded structure documents include:

- **Java Documentation.** One feature of Java documentation is the ability of the author to automatically generate web pages for API documentation. HTML formatting tags may be included in these Java documentation comments in order for the author to format the resulting web pages to his or her liking. As a result, many Java files end up having HTML structures embedded within the containing Java code.

```
/**
 * Graphics is the abstract base class for all graphics contexts
 * which allow an application to draw onto components realized on
 * various devices or onto off-screen images.
 * A Graphics object encapsulates the state information needed
 * for the various rendering operations that Java supports.  This
 * state information includes:
 * <ul>
 * <li>The Component to draw on
 * <li>A translation origin for rendering and clipping coordinates
 * <li>The current clip
 * <li>The current color
 * <li>The current font
 * <li>The current logical pixel operation function (XOR or Paint)
 * <li>The current XOR alternation color
 *     (see <a href="#setXORMode">setXORMode</a>)
 * </ul>
        .
        .
        .
 * @author      Sami Shaio
 * @author      Arthur van Hoff
 * @version     %I%, %G%
 * @since       1.0
 */
public abstract class Graphics {
```

Figure 1-1: Example Java documentation comment with embedded HTML structures [3]

- **Web Tutorials.** The Web has become a great resource for code developers to learn from others experience. A large part of this involves educating developers by displaying sample code in web tutorials. This requires that it be embedded within another language, namely HTML. HTML is an interesting language as a container for embedded structures. This is because the primary purpose of HTML is to provide structural information to web browsers for visual rendering of web pages. The text of the HTML document source code can differ greatly in appearance from the text of the rendered web page. This can be seen in the following example, which is a web tutorial for writing HTML. Even though both the containing code and the embedded code are HTML, it is possible to embed HTML within HTML because the embedded structure exists at the rendered level, not at the source code level.



Three kinds of lists

HTML supports three kinds of lists. The first kind is a bulletted list, ofte
and <li> tags, for instance:

```
<ul>
  <li>the first list item</li>

  <li>the second list item</li>

  <li>the third list item</li>
</ul>
```

Note that you always need to end the list with the </ul> end tag, but th
The second kind of list is a numbered list, often called an ***ordered list***.

Figure 1-2: Example web tutorial page with HTML embedded in HTML

- **Web server pages.** Many server pages like Java Server Pages (JSP) or Active Server Pages (ASP) embed other languages in HTML in order to enhance their web pages with dynamic content created. This embedding is invisible to the viewer of the served web page, but developers still must embed their code within the HTML structures of the page in such a way that the server can find and interpret it.

```
<TR>
  <TD>&nbsp;</TD>
  <TH valign="TOP">Month<BR>(1-12)</TH>
  <TH valign="TOP">Day</TH>
  <TH valign="TOP">Year</TH>
  <TH valign="TOP">Time (2400)</TH>
</TR>

<%
  java.util.Calendar cal = new GregorianCalendar();
  int year = cal.get(java.util.Calendar.YEAR);
%>

<TR>
  <TH>Start</TH>
  <TD><INPUT name="startmonth"  size="2"></TD>
  <TD><INPUT name="startday"    size="2"></TD>
  <TD><INPUT name="startyear"   size="4" value="<%= year %>"></TD>
  <TD><INPUT name="starttime"   size="4"></TD>
</TR>
```

Figure 1-3: Example Java Server page with Java embedded in HTML

1.2 Applications

One of the goals of this body of work was to outline a technique for parsing embedded structure, serving as a framework for a variety of applications. Of particular interest are applications that make use of syntax information of parsed embedded structures with unmarked boundaries. Some applications for this work include syntax coloring, indexing and information retrieval, and advanced web navigation.

- **Syntax coloring.** One simple application that uses the syntax information gained from parsing embedded structures is syntax coloring. Syntax coloring is a tool used to help developers write and understand source code. For those developers writing source code with embedded structures, syntax coloring can help preserve consistency of how text editors treat similar language structures.

- **Indexing and information retrieval.** By indexing documents by their embedded structures, users can search specifically for content that is embedded in another language. A scenario that demonstrates the usefulness of this indexing is as follows: A developer wishes to develop a piece of code in Java using the Swing toolkit. In order to maximize efficiency, developer would like to make use of similar work has already be done and is documented on the web. The developer searches for Java swing sample code which yields many discussions, articles, and advertisements on the topic, but the developer must dig through the search results in order to find actual sample code that will help him. With embedded structure indexing, the developer could have searched the same topic but with the stipulation that all the pages returned contain embedded structures of the desired Java type. Furthermore, the developer could make use of the Java syntax and request pages with embedded structures that contain the Java type JButton. Indexing parsed embedded structures enhances the experience of developers searching for specific sample code.

- **Advanced web navigation.** Using web scripting tools, it is possible to make use of syntax information from parsed embedded structures to enable advanced web navigation. One example would be to automatically detect Java types in sample Java code embedded within web tutorial pages. Using a tool that allows dynamic end-user webpage modification [4], a user could script all web pages with sample Java code in it to hyperlink the Java types it encounters to the appropriate API documentation pages. A prototype of this feature has already been implemented, and is outlined in Chapter 6.

The rest of this thesis explains the principles of this technique and how it was applied. Chapter 2 describes similar work related to parsing embedded structures. Chapter 3 talks about the design goals of this system. Chapter 4 discusses the actual embedded structure parsing system, including the type detection, view transformation, and parser application phases. Evaluation of the embedded structure parsing system is described in Chapter 5. A description of an implemented application of embedded structure parsing is discussed in Chapter 6. Finally, Chapter 7 talks about future work that can be done to improve the current system.

Chapter 2

Related Work

2.1 Text Classification

One area of research related to the detection of embedded structures is text classification. The problem of embedded structure detection can be reduced to a text classification problem where the type classification of a region in the document must be different from the classification of the rest of the document. A well-known subset of text classification research is the development of effective spam filtering.[5] The task that spam filters face is to determine whether a given email message is either a legitimate message or a spam message. These two types of messages can be considered two language types. The current standard for spam filters is the Bayesian filter algorithm outlined by Paul Graham.[6] In this spam filter, the message is tokenized and and a probability of the message being spam is calculated based on these probabilities. Due to the unstructured and evolving nature of documents that spam filters must examine, spam filters must use sophisticated techniques such as probabilities for individual tokens, which is natural given the assumption that feature probabilities are independent.

In embedded structure parsing, statistical classification methods are used to detect where embedded structures are located. Our classification types are well-structured languages with an established syntax, so we deemed the overhead of Bayesian filtering as too computationally intense for a tool that must classify the large number of possible regions in which embedded structures can occur.

2.2 LR parsing

YACC, also known as Yet Another Compiler-Compiler, is a parser generation tool that imposes user-specified structure on an input stream. This structure is specified by a collection of grammar rules, which pair input descriptions with code that is called when input text structures that meet those descriptions are encountered. YACC converts this input specification into an actual parser, which works in conjunction with a lexical analyzer to check that the input stream matches the specification.[7] This parser acts as a finite state machine that operates left to right on tokens that are passed to it from the lexical analyzer. The nature of the parser operating incrementally from left to right yields the term LR parsing.[8] YACC generates the code for its parser in the C programming language. Many parser generation tools related to YACC have since been developed, like GNU Bison[9], Berkeley YACC[10], and JavaCC for Java[11].

2.3 GLR parsing

Generalized LR, or GLR, parsing algorithms have certain advantages over standard LR parsers like YACC. Two key advantages of GLR parsing algorithms are the fact that they allow unbounded look-ahead, and that they handle input ambiguities. GLR handles parsing ambiguities by keeping multiple potential parses until the ambiguities can be resolved. It is forking the parsers in order to keep track of each potential parse.

Blender[12], developed in the Harmonia project, is a combined lexer and parser generator that is able to handle ambiguous boundaries for embedded languages and parsing the corresponding structures according to the appropriate structural rules. Blender uses GLR parsing to resolve ambiguities at the boundaries of embedded structures. It does this by providing a framework to write modular, lexical descriptions including rules for embedding structures. These lexical descriptions support multiple grammars that are merged to create a single parser. This parser, provided with the appropriate embedding rules in its lexical description, is now able to parse documents with embedded structures by handling ambiguities between languages the same way it handles other lexical ambiguities.

Similarly, MetaBorg[13], developed using a grammar called syntax definition formalism (SDF), is a method that to embed and assimilate languages to provide scannerless parsing of documents with embedded structures. MetaBorg provides two advantages over Blender: reduction in parse tree size, and a more concise parser grammar in the form of SDF. By adopting a scannerless approach, MetaBorg can use the context of lexical tokens to resolve ambiguities. This reduces the size of the parse tree. Using SDF for a grammar formalism provides support for all context-free grammars, including ambiguous grammars. One feature of SDF that is useful for parsing embedded languages is that instead of forcing the syntax definition into a non-ambiguous state, SDF creates filters to prioritize different parse interpretations. This allows for more flexibility across different types of embedding.

One difference that the technique outlined in this paper has from Blender and MetaBorg is the fact that it abstracts away the knowledge of how to write parser specifications from the user. In Blender and MetaBorg, the user must construct a custom parser by specifying possible embedded structures in the lexical definitions for the grammars. This requires the construction of a new parser each time a new type of embedding is added. This is because Blender and MetaBorg require that its single parser has full knowledge over all the possible lexical structures across the different languages in order to resolve ambiguities. In contrast, our approach keeps the embedded structure rules separate from the parsers, and composes each parser by transforming its input and mapping its results back to the original document.

Chapter 3

Design Goals

As stated before, the main purpose of this thesis is to outline a new technique for selectively applying existing parsers on intelligently transformed document content with embedded structures. More simply, the system outlined in this paper should be able to detect and parse syntax embedded within another syntax. The input to the system is a document which contains text and certain document metadeta (filename or URL, MIME type). The output of the system is a mapping between syntax concepts and a set of *regions* in the document to which they correspond . In this thesis, a *region* is a representation of a start offset and an end offset of a document. The text in a region can easily be determined using the start and end offsets and the full text of the document.

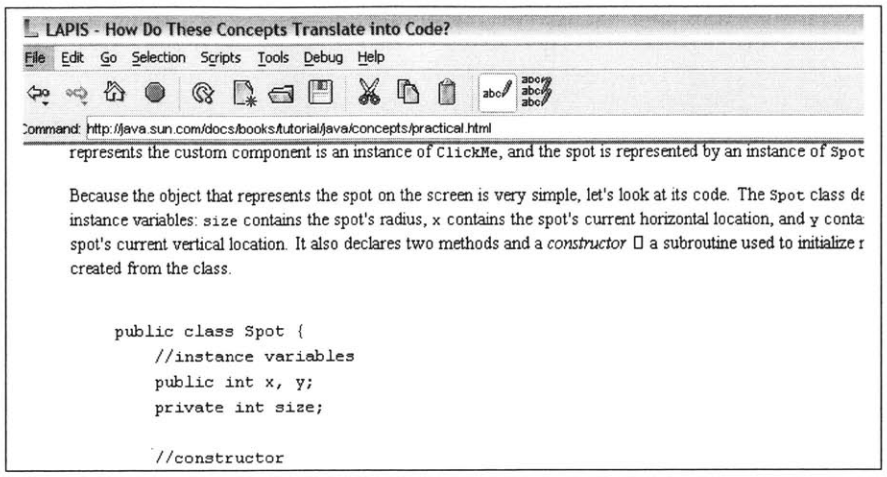
The following scenario sketches an example of the desired functionality of the system:

1. The user loads an HTML document with embedded Java structures into the system. See Figure 3-1.

2. The system creates a mapping between Java syntax concepts and regions in the embedded structures.

3. The user queries the system about Java expressions. See Figure 3-2.

4. The system returns a set of regions in the document corresponding to Java


LAPIS - How Do These Concepts Translate into Code?
File Edit Go Selection Scripts Tools Debug Help
Command: http://java.sun.com/docs/books/tutorial/java/concepts/practical.html

represents the custom component is an instance of `ClickMe`, and the spot is represented by an instance of `Spot`

Because the object that represents the spot on the screen is very simple, let's look at its code. The `Spot` class de
instance variables: `size` contains the spot's radius, `x` contains the spot's current horizontal location, and `y` conta
spot's current vertical location. It also declares two methods and a *constructor* □ a subroutine used to initialize r
created from the class.

```
public class Spot {
    //instance variables
    public int x, y;
    private int size;

    //constructor
```

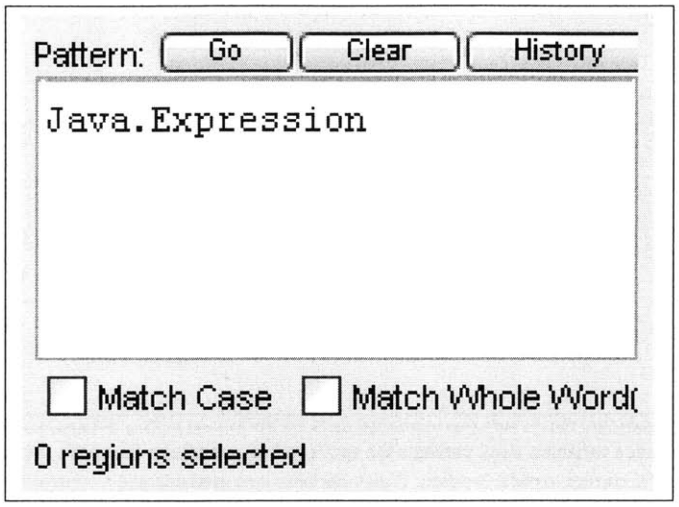
Figure 3-1: Java web tutorial loaded into a web browser

expressions. See Figure 3-3.

In addition to this functionality, there are a few key characteristics of our desired system:

- **Automatic detection of embedded structures.** It is imperative that the system automatically detect the embedded structures and parse them without prompting from the user. This maintains a level of abstraction that removes the notion of embedding and simply presents syntax of embedded structures at the same level as syntax of containing structures. In other words, the implementation of how structures are parsed in the document should be invisible to the user. From the users perspective, there should be no distinction between parsing embedded structures and parsing non-embedded structures. If the user needs to prompt the parsing of embedded structures by selecting them, the user could very well copy and paste the embedded structure into a dedicated editor. Automatic detection offers the user the ability to view parse results in the context of the document without having to do any work.

- **Lightweight detection of embedded structures.** The detection of embedded structures should be lightweight. Instead of maintaining large, complicated,



```
Java.Expression
```

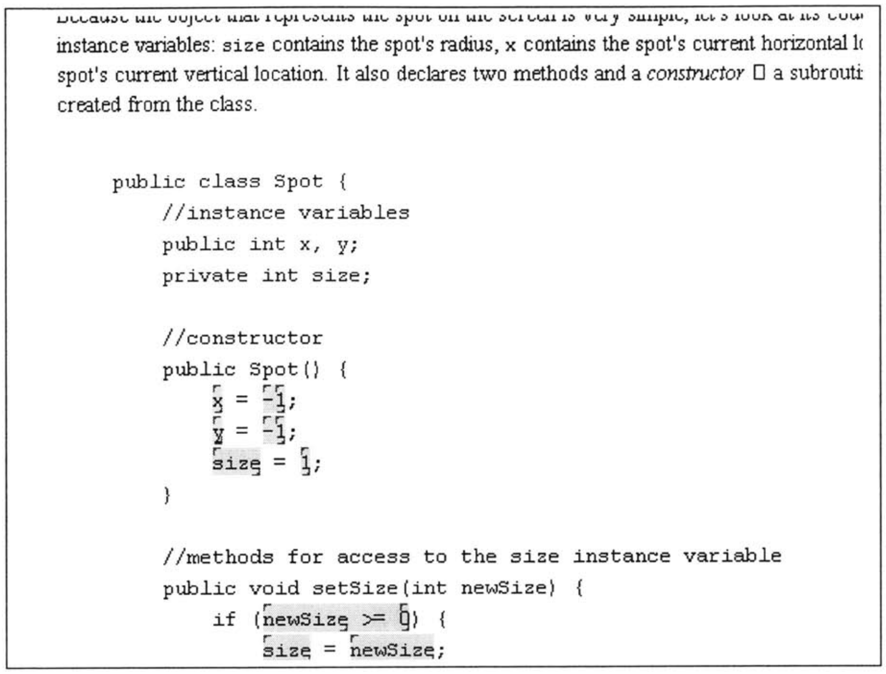
0 regions selected

Figure 3-2: User queries the system for Java expressions

potential parse trees for ambiguities, the system should deterministically locate the embedded structures and parse them as they are encountered.

- **Extensible support for other embedded language documents.** Extensibility is an important consideration for our system. New rules for embedding structures should be easily added to the system. In contrast to Blender or MetaBorg, the user should not have to know anything about writing grammars to support embedded structure parsing.

- **Use of existing parsers.** In order to parse embedded structures, this system should use existing parsers in a modular way. In Blender and MetaBorg, the grammar definitions for each parser was modular, but they were always used to construct a single, custom parser that had to be reconstructed every time a rule was added or changed.



Because the object that represents the spot on the screen is very simple, let's look at its cod
instance variables: `size` contains the spot's radius, `x` contains the spot's current horizontal l
spot's current vertical location. It also declares two methods and a *constructor* □ a subrouti
created from the class.

```
public class Spot {
    //instance variables
    public int x, y;
    private int size;

    //constructor
    public Spot() {
        x = -1;
        y = -1;
        size = 1;
    }

    //methods for access to the size instance variable
    public void setSize(int newSize) {
        if (newSize >= 0) {
            size = newSize;
```

Figure 3-3: Java expressions highlighted by the browser

Chapter 4

System Overview

In this chapter, the overall design and implementation of this embedded structure parsing technique is discussed. LAPIS[14], the developing environment for this embedded structure parsing technique is described. We then discuss the three conceptual phases that make up the process of parsing embedded structures: type detection, view transformation, and parser application. Type detection answers the question of what kind of document the system is trying to parse. It further answers the question of what kind of embedded structure the system is trying to parse, and where those embedded structures might occur. View transformation is responsible for transforming the content of the embedded structure for the appropriate parser. Parser application is the application of existing parsers to this transformed input and mapping the results back to the original context of the document. Finally, we discuss the embedded structure rules file, which is a formal specification which embodies these three concepts of type detection, view transformation, and parser application.

4.1 LAPIS

The main body of this system was developed within the framework of LAPIS, a programmable web browser and text editor. LAPIS maintains a library of parsers which it runs on every document that is loaded. One interesting and useful feature of LAPIS is its use of *text constraints*, a pattern language that allows the user to specify regions of the document with respect to structures detected by its library of parsers. This pattern language provides the user with a level of abstraction from actual regions of a specific document. As a result, text constraints, or TC, can be used as a high-level pattern to describe a region of text in a document.

In LAPIS, patterns are abstractions for text structures in a document. These abstractions can be populated by parsers, regular expressions, manual selections, and TC patterns. The output of each of these patterns is a set of contiguous text regions in the document, called *region sets*. The parsers that LAPIS keeps are thus run on entire documents and return region sets for each pattern they are responsible for populating.

In LAPIS, the notion of a *view* of a document is one in which different aspects of the content of the document can be viewed. For example, an HTML document will have a default view called the *raw view* that is the plain HTML source code. But an HTML document also has a *cooked view*, which presents the content that would be visually shown in a browser, without all the tags and metadata. Views can present different content from the default view, but each of these views must contains some internal mapping that associates each region in the view with a region in the default view. We will discuss later in this chapter how views are used to present only relevant syntax to appropriate parsers in the context of parsing embedded structures.

4.2 Type Detection

Much of the process of how to detect and parse embedded structures is tied to the type of the containing code. One of the goals of our system was to provide automatic detection and parsing of embedded structures. This requires automatic type detection of the document.

There are a number of tests that can be applied to a document to determine its type. In this section, we will discuss a number of these tests and describe a framework in which both whole documents and code fragments can be type tested. The ultimate goal of this type detection is to determine the type of the document and the types of

Type	File extension
Java	*.java
HTML	*.html, *.htm
Java Server Page	*.jsp
XML	*.xml

Table 4.1: Example URL tests across types

all its embedded structures.

4.2.1 URL Testing

One way to test the type of a document is to examine the Uniform Resource Locator, or URL of the document. The URL is a unique identifier for the document, which tells a lot about where the document is from as well as how the viewer of the document should interpret it. The domain name of a web URL can often identify a certain corpus of files. For example, I can expect that a majority Google[15] search results pages are HTML documents with Javascript embedded within it. Upon the loading of a new document, I can check to see if the URL prefix is "http://www.google.com/search?" to see if I can expect the document to be an HTML document with Javascript structures. The file extension of the document is also helpful in determining the documents type. URLs generally end with an extension indicating the type of the document.

4.2.2 MIME type testing

Another way to check the type of a document is to examine the MIME (Multipurpose Internet Mail Extension) property that is associated with the document. This property gives the system a cue as to how to handle the binary data. In other words, the MIME property can directly tell us the type of the document.

4.2.3 Byte sampling

Given a document of unknown MIME type and location, it is impossible to use URL or MIME property testing to determine the type of the document. Instead, the actual content of the document must be tested. One way of doing this is to take the first 100 bytes of the document and examine it for any telltale features. For example, if the system encounters the tag ⟨html⟩ within the first 100 bytes of the document, it can make the assumption that this is an HTML document.

4.2.4 Parser success

One way of typing a region of text is to run parsers over that region of text, to check to see if the parser can find any structural information. For example, if the HTML parser is run over a given section of text, one indication that the text is HTML is to see if the HTML parser can detect any tag structures. Using parsers to test for type, however, can be extremely costly in terms of performance time. To effectively check the type of a given region using this testing method, the parser corresponding for every possible embedded type must be run on the region.

4.2.5 Token analysis

Every language has a different syntax, yielding different keywords and punctuation in some structured order. In turn, that set of keywords and punctuation, also known as tokens, characterize that language type. By analyzing the frequency of these tokens, we could attempt to differentiate between two language types.

Token Sets

For each language, we selected a set of tokens that *characterized* that language. To characterize a language, tokens must occur in that language with high frequency and in other languages with low frequency. This is so that the frequency of a given token set can distinguish two languages. Tokens that would appear frequently across all language types, such as spaces and carriage returns were excluded from all token sets.

Characteristic tokens for Java, HTML, and XML, and C were chosen by examining the reserved keyword set as defined in the associated parser. The characteristic tokens for English were chosen by examining the most frequent tokens used in the English language based on the root of the word.The token sets that were used are shown in Table 4.2.5.

Using these token sets, we can generate statistics of relative frequency for each type of language over a corpus of representative files. The corpora chosen must be large to account for any outliers of unusual token frequency. For our type detection system, we chose corpora of at least 500 files, resulting in over 1 million tokens. Once the corpora has been chosen, we can find the frequency of tokens in the characteristic token set for each language. For example, we can find the average percentage of all words and punctuation that are characteristic HTML tokens for each document in a set of 1000 HTML documents. The mean along with the standard deviation of token frequency can be used as a profile for HTML files. Assuming a normal distribution, we can use this profile to calculate the probability that an unknown document is HTML. Generalizing this approach across all types, we have an effective statistical method for typing unmarked regions of text.

Detecting embedded structures

Parsing embedded language documents requires the system to have parsers that are able to parse both the embedded structure and the containing structure. Furthermore, the system must be able to recognize where the embedded structures occur in the document, so that it is able to invoke the appropriate parser.

Embedded structure boundaries

Detecting the boundaries where embedded structures begin and end is a key to actually parsing them. Many embedded language documents make this task very simple by using markers to explicitly define where the embedded structures are. For example, in HTML files with Javascript, the HTML tag ⟨script language=javascript⟩ tag is always used to mark where the embedded Javascript begins, and the corresponding

HTML	lt a abbrev acronym applet area author b banner base basefont bgsound big blink blockquote bq body br caption center cite code col del dir div dl dt dd em embed fig fn font form frame frameset h1 h2 h3 h4 h5 h6 head hr html i iframe img ins kbd lang lh li link map marquee menu meta multicol nobr noframes note ol p param plaintext pre q range samp script select small spacer strike strong sub sup tab table tbody td textarea textflow tfoot th thead title tr tt u ul var href src quot gt ⟨ ⟩ ” / = &
Java	abstract boolean break byte case catch char class const continue default do double else extends false final finally float for goto if implements import instanceof int interface long native new null package private protected public return short static super switch synchronized this throw throws transient true try void volatile while) (. ; / = * { }
English	the is was be are were been being am of and a an in inside to have has had having he him his it its I me my they them their not no for you your she her with on that this these those do did does done doing we us our by at but from as which or will said say says saying would what there if can all who whose so go went gone goes more other another one see saw seen seeing know knew known knows knowing ’ , . ”
C	continue volatile register unsigned typedef default double sizeof switch return extern struct static signed while break union const float short else case long enum auto void char goto for int if do -) (. ; / = * [ ] &
XML	cdata nmtoken nmtokens id idref idrefs entity entities xml ⟨ ⟩ / = ”

Table 4.2: Token sets for various language types

Type	Mean	Standard deviation
Java	0.509	0.180
HTML	0.639	0.222
English	0.512	0.072
XML	0.548	0.187
C	0.639	0.161

Table 4.3: Token frequency statistics over a representative corpus

⟨/script⟩ tag is always used to mark where the structure ends. This marker not only indicates where the embedded structure occurs, but also indicates that the structure is Javascript. A similar example is the ⟨%= code ⟩ tag in Java Server Pages. The tag not only marks where the Java code occurs, but also indicates that the embedded structure is indeed Java, specifically, a Java expression.

While our system can use these explicitly marked boundaries to locate embedded structures within documents that make use of them, there are still many documents that do not use such markings. For example, many web tutorials do not use explicit tags to demarcate where sample code will be displayed. For such documents, our system must use other cues to detect where the embedded structures are.

One method of detection is to inspect the natural regions where embedded structures can occur. Virtually every language has a set of regions in which it is acceptable to embed other language structures. This effectively reduces our search space from an arbitrary number of start and end points to a fixed set of regions. Once we have the set of regions to inspect, we must detect the type of each region in order to determine whether or not it is an embedded structure.

Applying type detection to embedded structures

Type detection of embedded structures differs from type detection of documents mainly in that the system can no longer take advantage of the information gleaned from document properties. The associated URL and MIME property of a document

Type	Natural Boundary
Java	Comments, Strings
C	Comments, Strings
HTML	Elements
English	Lines, Paragraphs

Table 4.4: Marked boundaries for embedded structures

Document	Outer Type	Inner Type	Boundary ({start, end})
Java	Java	HTML	{/**, *}
JSP	HTML	Java	{⟨%, %⟩} {⟨%=, %⟩}
HTML	HTML	Javascript	{⟨script language=javascript⟩, ⟨/script⟩}

Table 4.5: Natural boundaries for embedded structures

provide cues to type the containing code, rather than the embedded structures. Because parser success and analysis of token frequency depend only on the content of the regions text, they are essential for determining the type of the embedded structure.

4.3 View Transformation

Once the embedded structures have been detected, the appropriate parser can be invoked to parse its syntax. This is done by producing a new view of the document which only presents the embedded structures. The system uses the results of the embedded structure detection to transform the original view of the document to a view containing only the structures that have the embedded syntax. This embedded structure in the context of the view transformation phase is known as the *extraction region*. This is due to the fact that the region must be extracted from the original content.

Once we have created a new view with just the extraction region, we must still apply transformations before passing it to the parser. The embedded structure, by the very nature of having been embedded in another language, is often in a state where it cannot be sent directly as input to a parser. One example is the leading * that begins each line within a Java documentation comment. Although the embedded structure detection algorithm can locate the Java comment in which this embedded structure is located, it cannot remove the * characters that occur within this comment structure. In order to parse the content within each comment as HTML or English, these * characters must be removed. Our approach to this problem utilizes *rule-based view transformations*, applying simple replacement rules to original view content to produce a new view with parsable content.

One of the design goals of our system is to be easily extendible by the user. This means that users should be able to easily add support for parsing other embedded language documents. Each type of embedded language document may need to be transformed in different ways. Because these transformations must be specified by the user, we need a simple and rigorous way of describing these view transformations.

View transformations support three types of simple transformations: insertions, deletions, and replacements. Insertions add regions to the new view that the old view did not have. Deletions remove regions from the new view that that the old view did have. Replacements replace regions from the old view with new regions in the new view. We can map the set of possible insertions and deletions to the set of possible replacements by thinking of insertions as the replacement of a zero-length region with a nonzero-length region, and by thinking of deletions as the replacement of a nonzero-length region with a zero-length region. By doing this, we can break down each view transformation into a set of simple replacement rules.

Our system applies transformations by using fixed rules that substitute regions of the document with replacement strings. For example, in the case of the Java documentation comment the required transformation for parsing embedded structures could be reduced to the following set of replacement rules:

1. /** symbol beginning a comment $\rightarrow$ empty string

2. */ symbol ending a comment $\rightarrow$ empty string

3. * symbol beginning a line in a comment → empty string

Although the regions are described informally in the above example, our system requires a formal description for regions in the document. Fortunately, LAPIS provides just such a description with its text constraints pattern language. Using TC patterns, the user can describe any set of regions in the document in a systematic way. TC patterns have been shown to be easy to learn, and users of LAPIS, in which this system was developed, should be comfortable using this pattern language. Using TC patterns, we can rewrite our replacement rules as such:

1. /** starting Java.Comment → empty string

2. */ ending Java.Comment → empty string

3. * starting Line in Java.Comment → empty string

One large concern with applying multiple transformation rules to a single view is how collisions are handled. In the Java comment example, the system must apply three transformations to the original view. To illustrate the problem that collisions pose, consider the following extraction region:

```
/**
 * This is a comment.
 */
```

The last * symbol in the extraction matches Rule 3, but it also matches the first part of Rule 2. Depending on which of these rules are applied to the conflicting symbol, the transformed view will either look like this:

```
  This is a comment.
```

or this:

```
  This is a comment.
   /
```

One possible way of resolving this problem is to apply all of the rules sequentially, requiring the user to specify the order in which they should be applied. This unnecessarily burdens the user with additional constraints on the replacement rule specification. Our approach eliminates the need for rule ordering by resolving conflicts using the basis of precedence and size. In general, if the region matching Rule A begins before the region matching Rule B, then Rule A is applied. If the regions matching Rules A and B both begin at the same point, the rule matching the larger region is applied. This eliminates the need for rule ordering and the view transformation algorithm can incrementally scan the region for replacement matches and apply rules as they are needed as opposed to applying individual rules sequentially.

4.4 Parsing Application

Once the embedded structures have been located in the type detection phase and the appropriate view has been generated in the view transformation phase, those embedded structures are ready to be parsed. In order to apply the parser to these embedded structures, the system must first know which parser to invoke. This can be determined in the type detection phase. Once the appropriate parser has been selected, the parser can then parse the structure and return mappings between syntax patterns and the regions of the view to which they match.

The regions that this parser returns are all defined in terms of offsets of the content presented in the transformed view. For the system to have useful data it can share with other applications, it must map these offsets to correspond to the original document content. This is done by using the internal map stored in the constructed view that relates offsets from one view to the other.

4.5 Rules File

In the actual implementation of this system, most of the knowledge for parsing embedded structures is placed in a rules file. This is done primarily for the purpose of extensibility and modularity. The user can extend the system to support new types of embedded documents by adding a new rule set to the rules file. Also, this design enabled modularity between rule generation and the actual embedded structure parsing. The rules file can be explicitly written by the user, or it could be automatically generated by some other program.

The rules file was written as an XML to reinforce the fact that it can be automatically or manually generated. Rules files that followed the specified XML schema could be read by the system to actually implement the embedded structure parsing. The rules file encapsulated the following ideas: natural boundaries type tests, token sets, type parsers, and view transformation replacement rules. These concepts make up the two elements that can appear in the rules file: type elements and transformer elements.

Each type element is a representation for a particular language type. The type element contains a single attribute *name* which takes a string as a value. The type element contains a number of children. These include type testing elements, base elements, and parser elements.

There are currently three type testing elements: urltest, mimetest, mimetest. The urltest element represents a type predicate that examines the URL of a given document. Each urltest element has a *pattern* attribute that has a string as a value. If the ending of the URL of the document matches *pattern*, then the document is recognized as the urltest parent type.

```
<urltest pattern=ENDING/>
```

The mimetest element represents a type predicate that checks the MIME property of a given document. Each mimetest element has a single attribute *mime* that has a string as a value. If the MIME property of the document matches *mime*, then the document is recognized as the mimetest parent type.

```
<mimetest mime=MIME/>
```

The stattest element represents a type predicate that checks the token frequency of a specified token set on a given document against the provided mean and standard deviation for the token frequency across an entire corpus. Each stattest element has two attributes, *mean* and *stddev*, which both have string doubles as values. Each stattest also has a child CDATA element containing the characteristic token set. If a document or region is determined by these statistics to be the stattest parent type with the highest probability, the region is marked as that language type.

```
<stattest mean="DOUBLE" stddev="DOUBLE">
     <![CDATA[TOKENS]]>
</stattest>
```

The base element is a description of the language type as a base or container for embedded structures. Each base element contains as children a regions element, a view element, and an optional snippet element. The regions element is a description of the marked or natural boundaries for embedded structures. Each regions element contains a single attribute, *TC*, which has a TC description as a value. The view element contains two attributes: a *transformer* attribute that references the name of a view transformer in the rules file, and a *input* attribute which takes in either "raw" or "cooked". The two string values indicate whether the view transformer should operate at the source code ("raw") level, or at the rendered "cooked" level. The snippet element indicates that there is a preferred type of embedded structure, and the *type* attribute is the name string of that preferred type.

```
<base>
<regions TC="TC"/>
    <view transformer=NAME input={"raw", "cooked"}/>
    [<snippet type=TYPE/>]
</base>
```

The parser element references the parser responsible for parsing the given type. Each parser element has a *name* attribute whose value is the name of the parser.

```
<parser name=NAME>
```

Here is an example of a rules file specifying the Java type:

```
<type name="java">
    <urltest pattern="*.java"/>
    <stattest mean="0.5086" stddev="0.1803">
        <![CDATA[abstract boolean break . . . ]]>
    </stattest>
    <base>
        <regions TC="Java.Comment just before Java.Method
                     or Java.Class"/>
        <view transformer="javadocView" input="raw"/>
        <snippet type="html"/>
    </base>
    <parser name="JavaParser"/>
</type>
```

Figure 4-1: Rules file specification of Java type

The transformer element represents the way a given view is to be transformed. Each transformer element contains a single *name* attribute that has a string value. Transformer elements contain rule elements which have a *TC* attribute and a *replacement* attribute. The *TC* attribute is a TC description of the regions that are being replaced and *replacement* is the string that replaces each of the regions.

```
<rule TC="TC" replacement=STRING>
```

Here is an example of a rules file specifying the Java Documentation view transformation:

```
<transformer name="javadocView">
    <rule TC="'/**' starting Java.Comment" replacement=""/>
    <rule TC="'*/' ending Java.Comment" replacement=""/>
    <rule TC="'*' starting line in Java.Comment" replacement=" "/>
</transformer>
```

Figure 4-2: Rules file specification for Java Documentation view transformation

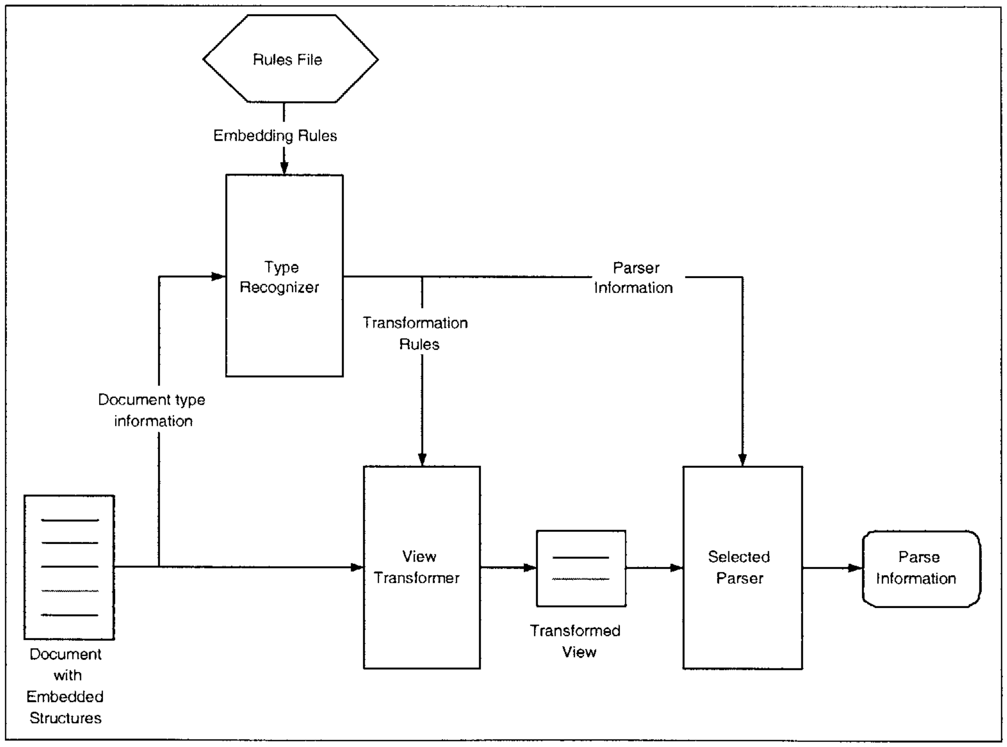


Figure 4-3: System Overview

Chapter 5

Evaluation and Results

In order to evaluate the system, we must see how well the system did in parsing embedded structures. The regions that the system output as embedded structures would have to be compared against the actual embedded structures in the document. In order to make such a comparison, we made manual selections of embedded structures and compared those regions to the ones produced by each parser. For each comparison, we came up with a measure of the precision and recall of the parsing.

One of the goals of this system was to support the parsing of a variety of embedded structures. In order to evaluate how well our system met this goal, we used documents of different types and from a range of sources. Documents of four different types were used: Java source files (HTML in Java), JavaServer Pages (Java in HTML), Java tutorial webpages (Java in HTML), and HTML tutorial webpages (HTML in HTML). Five corpora were selected for each corpus, but one Java tutorial corpus was excluded later due to rendering problems in LAPIS unrelated to this thesis. Each corpus is a collection of related documents of the same document type. The Java source files were chosen from projects in SourceForge, an online repository for open-source projects. The JavaServer pages were collected from versions of online instructional textbooks that have source code available for download. Both sets of web tutorial pages were collected from sites that ranked high on Google results for queries of "Java tutorial" and "HTML tutorial", respectively. From each corpus, 10 documents were randomly selected to compose a *test bed* for our system. For our purposes, a test bed is a collection of documents with which we can evaluate the performance of our system. Due to the random nature of document selection from each corpus to the test bed, a number of the documents in this test bed did not actually contain embedded structures.

Once a test bed has been constructed from randomly selected documents across a range of document types and corpora sources, the embedded structures of each document were manually selected and recorded. These manual selections were compared against the selections that our system made when the documents were loaded. Upon loading, our system automatically went through the process of type detection, view transformation, and parser application. The region set returned by the appropriate parser indicated where our system was able to parse the embedded structures.

Every region that was marked by the parser to be an embedded structure is considered a positive decision and every region of the document that was not marked by the parser is considered a negative decision. Precision is defined as the percentage of positive decisions made that were correct. Recall is defined as the percentage of correct regions that were decided positively. Both of these measures are important in evaluating the correctness of the system. It is trivially easy to maximize precision by ensuring that the system had zero output regions. Likewise, it is trivially easy to maximize recall by ensuring that the system output every region in the document. The ideal system will output all of the correct regions and only the correct regions, maximizing both precision and recall. The $F1$ measure is an even combination of precision and recall and is defined by the following formula:

$$F1 = \frac{2 * precision * recall}{precision + recall} \tag{5.1}$$

While precision, recall, and $F1$ measure are derived from a set of individual decisions, the decisions we are examining are sets of regions. It is difficult to come up with quantitative comparisons of entire region sets as opposed to individual data points, so we take advantage of the region algebra that LAPIS provides to generate region representations of positive and negative decisions. We can then use the size of the region as data points with which to calculate the precision and recall of the system.

Corpus Label	Name	Location
Java tutorial A	Java Sun tutorial	http://java.sun.com
Java tutorial B	Thinking in Java	http://mindview.net
Java tutorial C	Cafe au Lait Java	http://www.ibiblio.org/javafaq
Java tutorial D	Java for Students	http://www.javaforstudents.co.uk
HTML tutorial A	W3C tutorial	http://www.w3.org/MarkUp
HTML tutorial B	HTMLSource tutorial	http://www.yourhtmlsource.com
HTML tutorial C	Dave's HTML Code Guide	http://www.davesite.com
HTML tutorial D	WEBalley tutorial	http://www.weballey.net
HTML tutorial E	PageResource tutorial	http://www.pageresource.com
JSP A	JSP Cookbook	http://www.oreilly.com
JSP B	PSK JSP files	http://www.bolinfest.com
JSP C	Web Development with JavaServer Pages	http://www.manning.com
JSP D	Beginning JavaServer Pages	http://www.wrox.com
JSP E	Head First Servlets and JSP	http://www.oreilly.com
Java source A	Java 1.4.2 SDK	http://java.sun.com
Java source B	Azureus	http://sourceforge.net
Java source C	hipergate CRM	http://sourceforge.net
Java source D	File indexer	http://sourceforge.net
Java source E	HTML Unit	http://sourceforge.net

Table 5.1: Corpora for evaluation test bed

Let X represent the region set generated by our system parsing the embedded structures. Let Y represent the region set matching the actual embedded structures. We can find the true positives by calculating the region set Z which is the set of all intersections between all of the regions in X with all of the regions in Y. To do this, we examine each of the regions in X to see if they intersect with any of the regions in Y. If there is an intersection, that intersection is added to Z. The magnitude of a region set R is defined to be the following:

$$|R| = \sum_r (r.endr.start) \tag{5.2}$$

By that definition, the formulas for precision and recall are as follows:

$$precision = \frac{|Z|}{|X|} \tag{5.3}$$

$$recall = \frac{|Z|}{|Y|} \tag{5.4}$$

The $F1$ score, is then, as follows:

$$F1 = \frac{2Z^2}{ZY + ZX} \tag{5.5}$$

In examining the results, we found that of the 190 documents, our system achieved an F1 score of over 0.90 on only 95 of them. Of the 190 documents, 59 contained no embedded structures and our system correctly did not label any structures for 46 of them. Of particular importance is the performance of our system on documents that did contain embedded structures in them. Of the 131 documents that contained embedded structures, our system achieved an F1 score of 0.9 for only 49 of them.

While there are many interpretations for this data, it is important to point out some trends in the data. Of the 104 documents with embedded structures that rated an F1 score of below 0.9, only 8 were JSP files. This reinforces the difficulty of parsing documents with unmarked boundaries. Examining some of the charts of the data, there appears to be a correlation between the performance of the system and the corpus it is using as input. This is to be expected, as a corpora uses a similar method to embed structures in its documents. One encouraging thing to note is that the system did perform consistently well on at least 2 of the 5 corpora for each of the document types.

Analysis of the types of errors encountered could shed some insight on how to improve the effectiveness of the embedded structure parsing. By looking at errors of our system on specific documents, several classes of errors can be distinguished to offer possible explanations for detection and parsing failures. These error classes include errors in MIME type labeling by LAPIS and false negative HTML embedded structures.

For the D corpus in the Java tutorial test bed, it seemed that LAPIS misrecognized the documents as a different MIME type it should have been. The document was a web tutorial, so the MIME type should have been "text/html" to indicate that this was contained HTML code, but LAPIS had labeled the document as "text/plain". This caused problems with the view transformation phase of the document. The view transformation was specified to operate at the rendered or cooked level, but was unable to do so because a rendered view of the document was never created. Excluding the results of the D corpus, the detection and parsing of embedded Java structures (Java tutorial and JSP test beds) achieved an F1 score over 0.9 for 33 out of the 45 documents containing embedded structures.

Most of the failures on documents with HTML embedded structures were failures of the type detection phase. For all of corpora A and C of the HTML tutorial test bed and half of corpora B and D in the Java source test bed, our system did not recognize any HTML structures. This was in large part due the failure of the statistical analysis type detection phase that to positively identify regions with embedded HTML structures. The documents for which our system failed all had large natural bounding regions containing a small number of HTML structures and a large number of English structures. This can be a problem with HTML in general. HTML is in itself an embedded language; there are typically English language structures embedded within HTML tags. Because the amount of English language structures within these HTML tags can be arbitrarily large, the effect of the simple statistical analysis on tokens is diminished. More flexibility is needed in the statistical type detection tests to ensure that potential embedded HTML structures are not discarded as plain English structures.

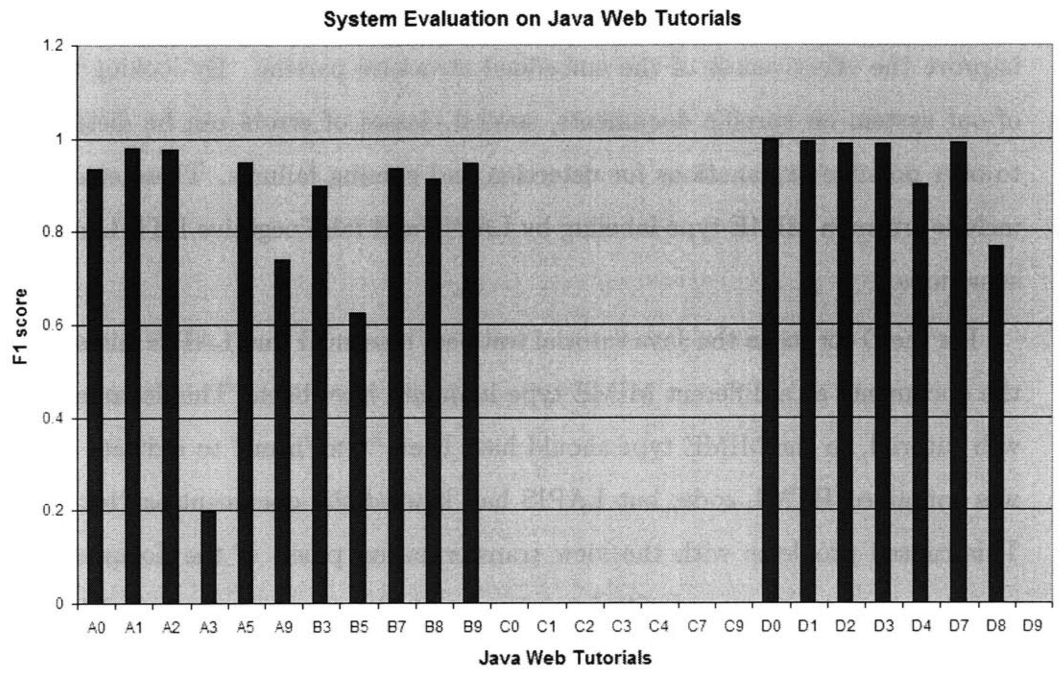


Figure 5-1: Performance results for Java structures embedded in HTML (Java Tutorial Pages)

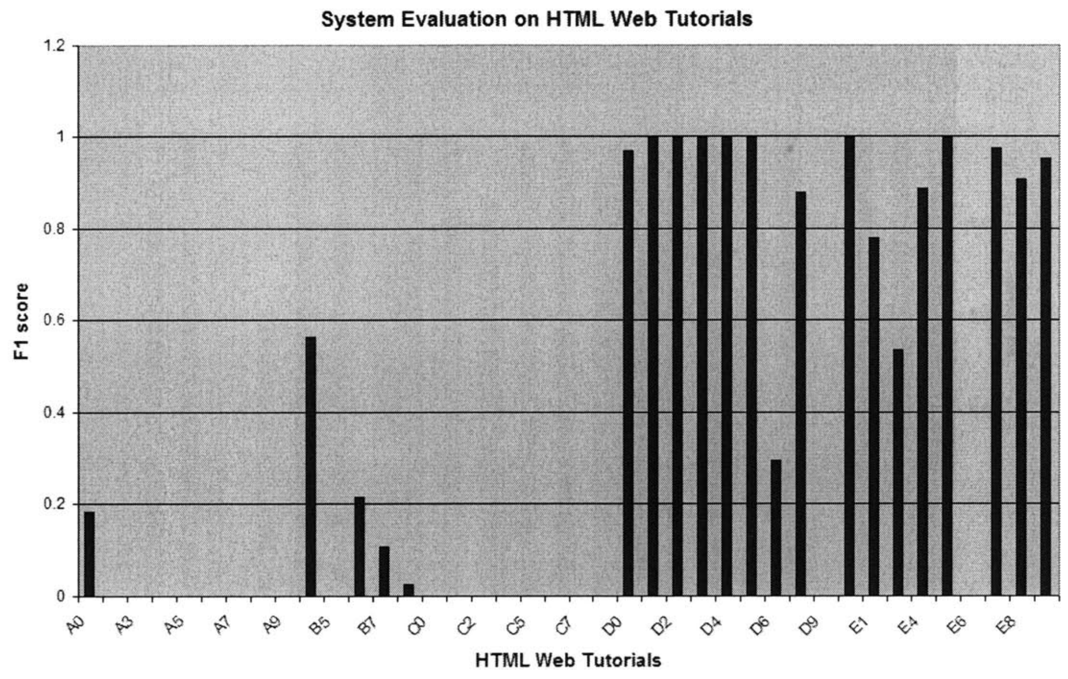


Figure 5-2: Performance results for HTML structures embedded in HTML (HTML Tutorial Pages)

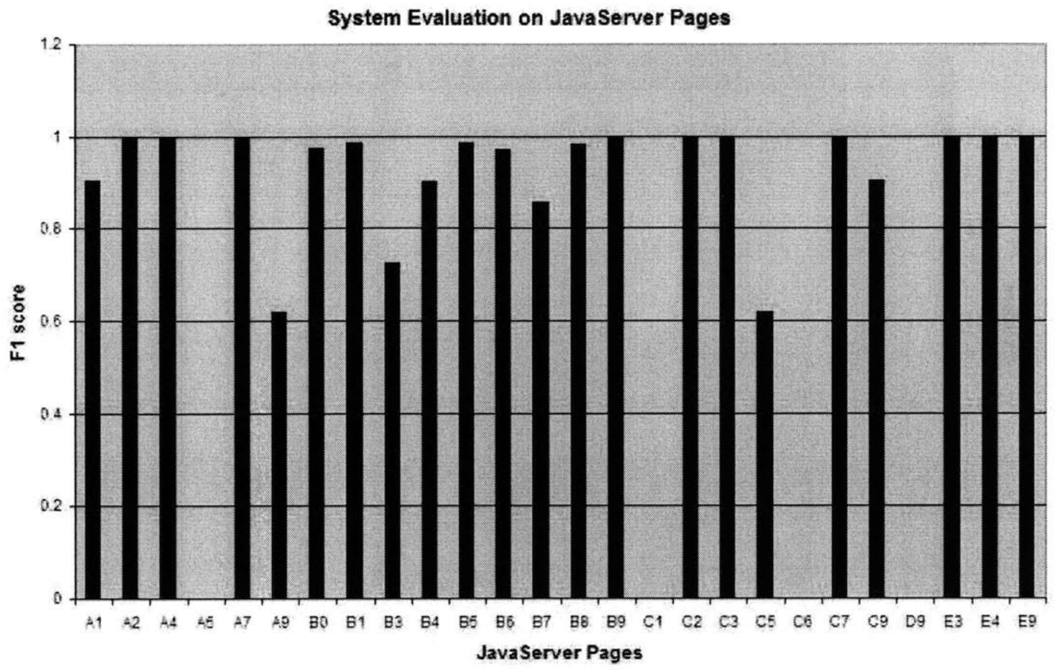


Figure 5-3: Performance results for Java structures embedded in HTML (JavaServer Pages)

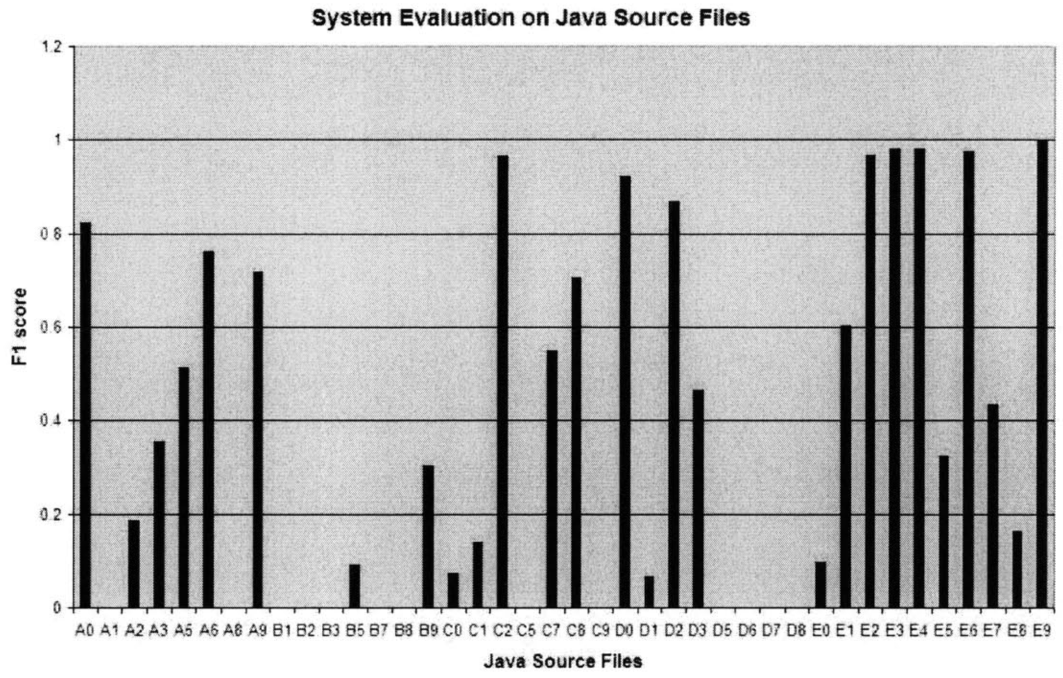


Figure 5-4: Performance results for HTML structures embedded in Java (Java Source Files)

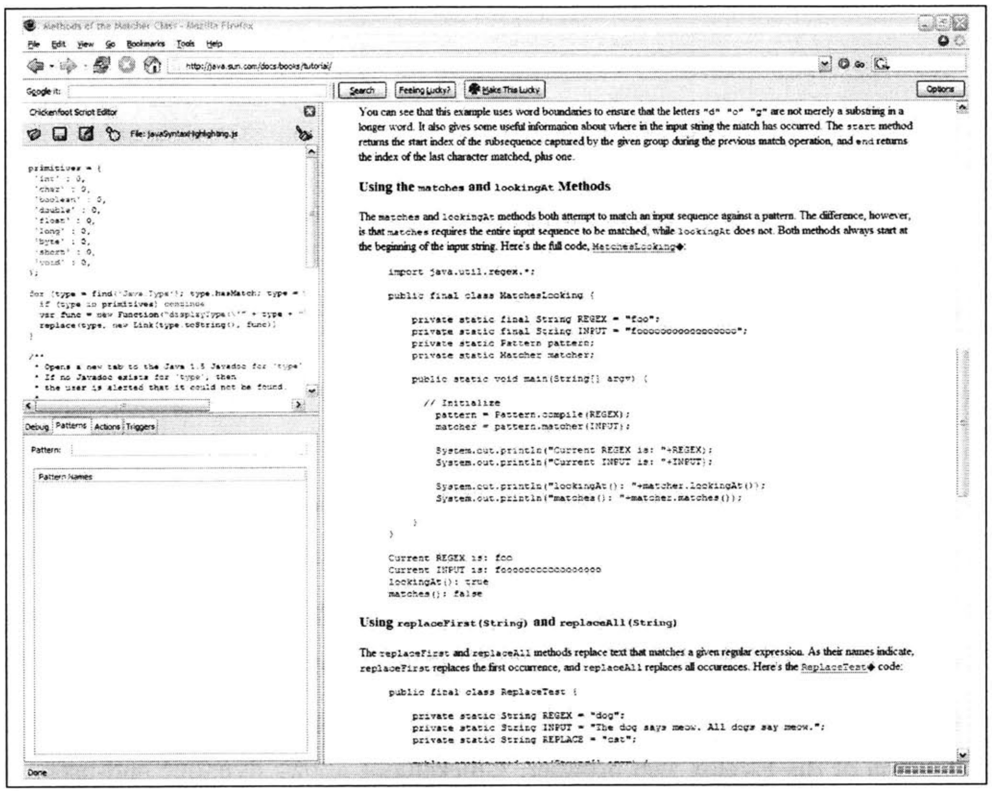
Chapter 6

Applications

One of the applications for embedded structure detection and parsing outlined in Chapter 1 is to enable advanced web navigation that makes use of syntax information from parsed embedded structures. An end-user programming tool could utilize the knowledge of embedded code in web pages to guide webpage modification. One such end-user programming tool is Chickenfoot, a scripting environment built as a plug-in extension of the Firefox web browser. Chickenfoot allows dynamic end-user webpage modification, enabling advanced web navigation by injecting hyperlinks and scripting capabilities into web pages at the rendered level.

Chickenfoot uses the TC pattern language from LAPIS to specify regions of the documents loaded into Firefox. As our system populates LAPIS patterns with embedded structure syntax, Chickenfoot inherits knowledge of embedded structures by its use of LAPIS patterns. Chickenfoot can then detect and parse embedded structures in web pages loaded in Firefox. With the syntax information gained from this detection and parsing, Chickenfoot can direct its webpage modification capabilities to transforming these embedded structures. A concrete example of this would be the use of Chickenfoot to detect Java types in a Java web tutorial and hyperlink them to the appropriate documentation in the Java API. The following scenario illustrates Chickenfoot implementing this advanced web navigation enabled by the embedded structure syntax:

1. The user loads a Java web tutorial page into a Firefox web browser enabled with Chickenfoot.


Methods of the Matcher Class - Mozilla Firefox
File Edit View Go Bookmarks Tools Help
http://java.sun.com/docs/books/tutorial/
Google it:
Search
Feeling Lucky?
Make This Lucky
Options
Chickenfoot Script Editor
File: javaSyntaxHighlighting.js
Debug
Patterns
Actions
Triggers
Pattern:
Pattern Names
Done

```
primitives = {
  'int' : 0,
  'char' : 0,
  'boolean' : 0,
  'double' : 0,
  'float' : 0,
  'long' : 0,
  'byte' : 0,
  'short' : 0,
  'void' : 0,
};

for (type = find('Java Type'); type.hasMatch; type = 
  if (type in primitives) continue
  var func = new Function("displayType(\'" + type + "
  replace(type, new Link(type.toString(), func))
}

/**
 * Opens a new tab to the Java 1.5 Javadoc for 'type'
 * If no Javadoc exists for 'type', then
 * the user is alerted that it could not be found.
```

You can see that this example uses word boundaries to ensure that the letters `"d" "o" "g"` are not merely a substring in a longer word. It also gives some useful information about where in the input string the match has occurred. The `start` method returns the start index of the subsequence captured by the given group during the previous match operation, and `end` returns the index of the last character matched, plus one.

Using the `matches` and `lookingAt` Methods

The `matches` and `lookingAt` methods both attempt to match an input sequence against a pattern. The difference, however, is that `matches` requires the entire input sequence to be matched, while `lookingAt` does not. Both methods always start at the beginning of the input string. Here's the full code, `MatchesLooking`◆:

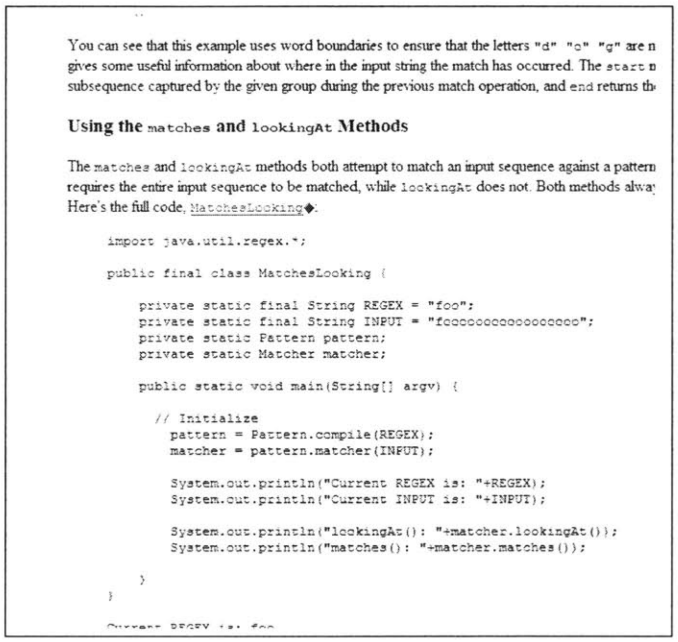
```
import java.util.regex.*;

public final class MatchesLooking {

    private static final String REGEX = "foo";
    private static final String INPUT = "fooooooooooooooooo";
    private static Pattern pattern;
    private static Matcher matcher;

    public static void main(String[] argv) {

      // Initialize
        pattern = Pattern.compile(REGEX);
        matcher = pattern.matcher(INPUT);

        System.out.println("Current REGEX is: "+REGEX);
        System.out.println("Current INPUT is: "+INPUT);

        System.out.println("lookingAt(): "+matcher.lookingAt());
        System.out.println("matches(): "+matcher.matches());

    }
}

Current REGEX is: foo
Current INPUT is: fooooooooooooooooo
lookingAt(): true
matches(): false
```

Using `replaceFirst(String)` and `replaceAll(String)`

The `replaceFirst` and `replaceAll` methods replace text that matches a given regular expression. As their names indicate, `replaceFirst` replaces the first occurrence, and `replaceAll` replaces all occurences. Here's the `ReplaceTest`◆ code:

```
public final class ReplaceTest {

    private static String REGEX = "dog";
    private static String INPUT = "The dog says meow. All dogs say meow.";
    private static String REPLACE = "cat";
```

Figure 6-1: Java web tutorial in Firefox with Chickenfoot sidebar



You can see that this example uses word boundaries to ensure that the letters "d" "o" "g" are n
gives some useful information about where in the input string the match has occurred. The start n
subsequence captured by the given group during the previous match operation, and end returns th

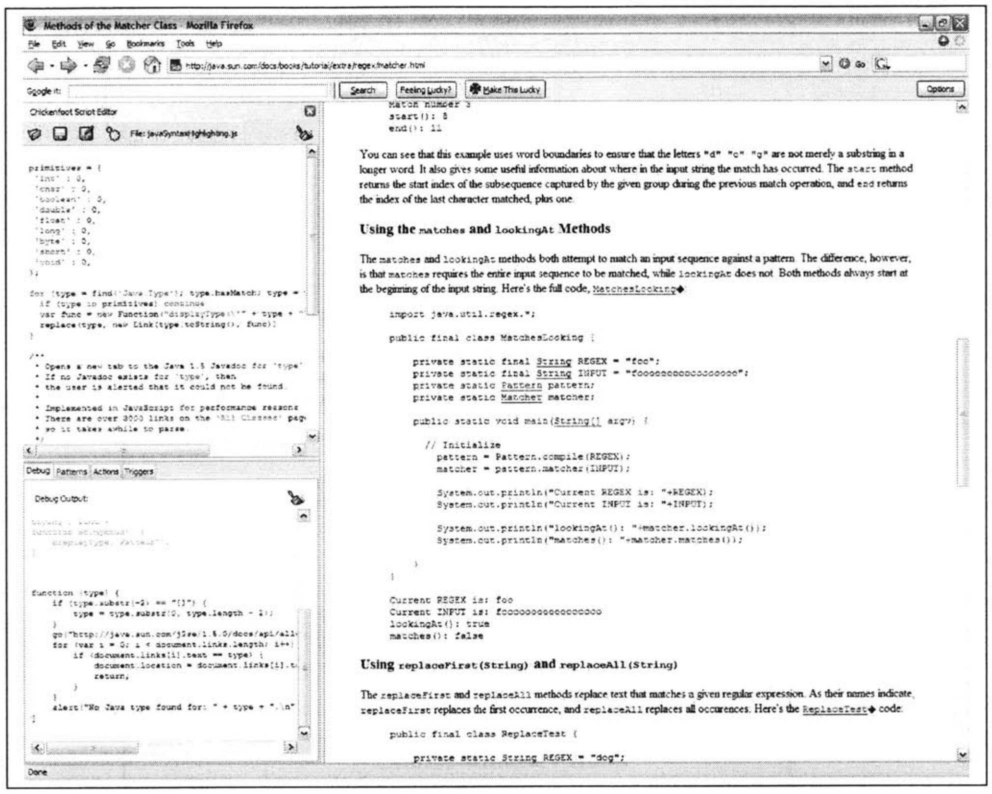
Using the `matches` and `lookingAt` Methods

The matches and lookingAt methods both attempt to match an input sequence against a pattern
requires the entire input sequence to be matched, while lookingAt does not. Both methods alwa
Here's the full code, MatchesLooking◆:

```
import java.util.regex.*;

public final class MatchesLooking {

    private static final String REGEX = "foo";
    private static final String INPUT = "fooooooooooooooooo";
    private static Pattern pattern;
    private static Matcher matcher;

    public static void main(String[] argv) {

      // Initialize
        pattern = Pattern.compile(REGEX);
        matcher = pattern.matcher(INPUT);

        System.out.println("Current REGEX is: "+REGEX);
        System.out.println("Current INPUT is: "+INPUT);

        System.out.println("lookingAt(): "+matcher.lookingAt());
        System.out.println("matches(): "+matcher.matches());

    }
}
```

Figure 6-2: Close-up view of embedded Java structures in Java web tutorial

2. The user loads a JavaScript script to replace Java types with hyperlinks to their API


Methods of the Matcher Class - Mozilla Firefox
File Edit View Go Bookmarks Tools Help
http://java.sun.com/docs/books/tutorial/extra/regex/matcher.html
Go
Google it:
Search
Feeling Lucky?
Make This Lucky
Options
Chickenfoot Script Editor
File: javaSyntaxHighlighting.js
Debug
Patterns
Actions
Triggers
Debug Output:
Done

```
primitives = {
  'int' : 0,
  'char' : 0,
  'boolean' : 0,
  'double' : 0,
  'float' : 0,
  'long' : 0,
  'byte' : 0,
  'short' : 0,
  'void' : 0,
};

for (type = find('Java Type'); type.hasMatch; type = 
  if (type in primitives) continue
  var func = new Function("displayType(\"" + type + "
  replace(type, new Link(type.toString(), func))
}

/**
 * Opens a new tab to the Java 1.5 Javadoc for 'type'
 * If no Javadoc exists for 'type', then
 * the user is alerted that it could not be found.
 *
 * Implemented in JavaScript for performance reasons
 * There are over 3000 links on the 'All Classes' pag
 * so it takes awhile to parse.
 */
```

```
[illegible]

function (type) {
    if (type.substr(-2) == "[]") {
        type = type.substr(0, type.length - 2);
    }
    go("http://java.sun.com/j2se/1.5.0/docs/api/all
    for (var i = 0; i < document.links.length; i++)
        if (document.links[i].text == type) {
            document.location = document.links[i].t
            return;
        }
    }
    alert("No Java type found for: " + type + ".\n"
}
```

```
Match number 3
start(): 8
end(): 11
```

You can see that this example uses word boundaries to ensure that the letters "d" "o" "g" are not merely a substring in a longer word. It also gives some useful information about where in the input string the match has occurred. The start method returns the start index of the subsequence captured by the given group during the previous match operation, and end returns the index of the last character matched, plus one.

Using the matches and lookingAt Methods

The matches and lookingAt methods both attempt to match an input sequence against a pattern. The difference, however, is that matches requires the entire input sequence to be matched, while lookingAt does not. Both methods always start at the beginning of the input string. Here's the full code, MatchesLooking◆:

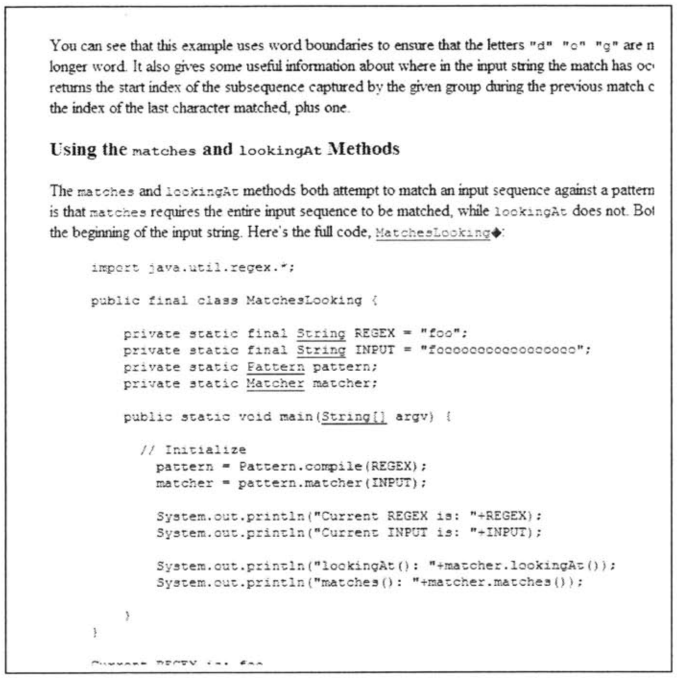
```
import java.util.regex.*;

public final class MatchesLooking {

    private static final String REGEX = "foo";
    private static final String INPUT = "fooooooooooooooooo";
    private static Pattern pattern;
    private static Matcher matcher;

    public static void main(String[] argv) {

      // Initialize
        pattern = Pattern.compile(REGEX);
        matcher = pattern.matcher(INPUT);

        System.out.println("Current REGEX is: "+REGEX);
        System.out.println("Current INPUT is: "+INPUT);

        System.out.println("lookingAt(): "+matcher.lookingAt());
        System.out.println("matches(): "+matcher.matches());

    }
}

Current REGEX is: foo
Current INPUT is: fooooooooooooooooo
lookingAt(): true
matches(): false
```

Using replaceFirst(String) and replaceAll(String)

The replaceFirst and replaceAll methods replace text that matches a given regular expression. As their names indicate, replaceFirst replaces the first occurrence, and replaceAll replaces all occurences. Here's the ReplaceTest◆ code:

```
public final class ReplaceTest {

    private static String REGEX = "dog";
```

Figure 6-3: Java web tutorial modified by Chickenfoot



You can see that this example uses word boundaries to ensure that the letters "d" "o" "g" are n
longer word. It also gives some useful information about where in the input string the match has oc
returns the start index of the subsequence captured by the given group during the previous match c
the index of the last character matched, plus one.

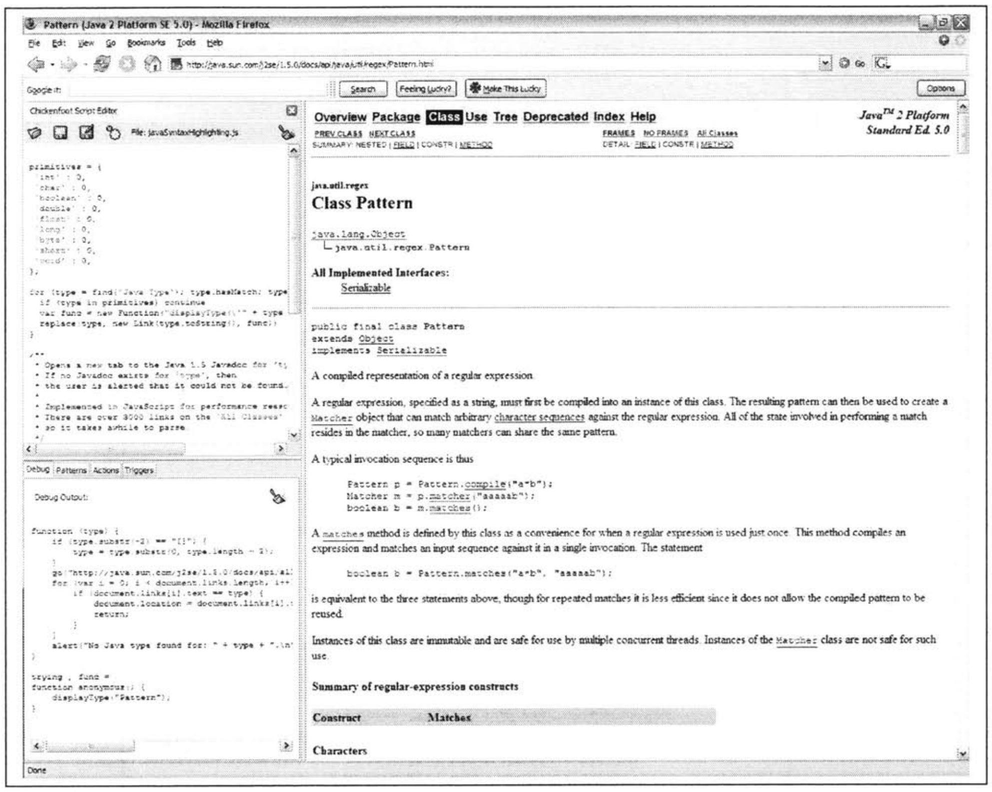
Using the `matches` and `lookingAt` Methods

The `matches` and `lookingAt` methods both attempt to match an input sequence against a pattern
is that `matches` requires the entire input sequence to be matched, while `lookingAt` does not. Bo
the beginning of the input string. Here's the full code, MatchesLooking◆:

```
import java.util.regex.*;

public final class MatchesLooking {

    private static final String REGEX = "foo";
    private static final String INPUT = "foooooooooooooooooo";
    private static Pattern pattern;
    private static Matcher matcher;

    public static void main(String[] argv) {

      // Initialize
        pattern = Pattern.compile(REGEX);
        matcher = pattern.matcher(INPUT);

        System.out.println("Current REGEX is: "+REGEX);
        System.out.println("Current INPUT is: "+INPUT);

        System.out.println("lookingAt(): "+matcher.lookingAt());
        System.out.println("matches(): "+matcher.matches());

    }
}
```

Figure 6-4: Java types replaced with hyperlinks

3. The user navigates to the Java API for one of the Java types with a single click.


Pattern (Java 2 Platform SE 5.0) - Mozilla Firefox
File Edit View Go Bookmarks Tools Help
http://java.sun.com/j2se/1.5.0/docs/api/java/util/regex/Pattern.html
Go
Google it:
Search
Feeling Lucky?
Make This Lucky
Options
Chickenfoot Script Editor
File: javaSyntaxHighlighting.js
Debug
Patterns
Actions
Triggers
Debug Output:
Done
Overview Package Class Use Tree Deprecated Index Help
PREV CLASS NEXT CLASS
FRAMES NO FRAMES All Classes
SUMMARY: NESTED | FIELD | CONSTR | METHOD
DETAIL: FIELD | CONSTR | METHOD
Java™ 2 Platform Standard Ed. 5.0

```
primitives = {
  'int' : 0,
  'char' : 0,
  'boolean' : 0,
  'double' : 0,
  'float' : 0,
  'long' : 0,
  'byte' : 0,
  'short' : 0,
  'void' : 0,
};

for (type = find('Java Type'); type.hasMatch; type
  if (type in primitives) continue
  var func = new Function("displayType(\'" + type
  replace(type, new Link(type.toString(), func))
}

/**
 * Opens a new tab to the Java 1.5 Javadoc for 't
 * If no Javadoc exists for 'type', then
 * the user is alerted that it could not be found.
 *
 * Implemented in JavaScript for performance reasc
 * There are over 3000 links on the 'All Classes'
 * so it takes awhile to parse
 */
```

```
function (type) {
    if (type.substr(-2) == "[]") {
        type = type.substr(0, type.length - 2);
    }
    go("http://java.sun.com/j2se/1.5.0/docs/api/al
    for (var i = 0; i < document.links.length; i++
        if (document.links[i].text == type) {
            document.location = document.links[i].
            return;
        }
    }
    alert("No Java type found for: " + type + ".\n"
}

trying , func =
function anonymous() {
    displayType("Pattern");
}
```

java.util.regex

Class Pattern

```
java.lang.Object
  └ java.util.regex.Pattern
```

All Implemented Interfaces:
Serializable

```
public final class Pattern
extends Object
implements Serializable
```

A compiled representation of a regular expression.

A regular expression, specified as a string, must first be compiled into an instance of this class. The resulting pattern can then be used to create a `Matcher` object that can match arbitrary character sequences against the regular expression. All of the state involved in performing a match resides in the matcher, so many matchers can share the same pattern.

A typical invocation sequence is thus

```
    Pattern p = Pattern.compile("a*b");
    Matcher m = p.matcher("aaaaab");
    boolean b = m.matches();
```

A `matches` method is defined by this class as a convenience for when a regular expression is used just once. This method compiles an expression and matches an input sequence against it in a single invocation. The statement

```
    boolean b = Pattern.matches("a*b", "aaaaab");
```

is equivalent to the three statements above, though for repeated matches it is less efficient since it does not allow the compiled pattern to be reused.

Instances of this class are immutable and are safe for use by multiple concurrent threads. Instances of the `Matcher` class are not safe for such use.

Summary of regular-expression constructs

Construct	Matches

Characters

Figure 6-5: Java API page resulting from navigation from modified tutorial page

```
primitives = {'int' : 0,
              'char' : 0,
              'boolean' : 0,
              'double' : 0,
              'float' : 0,
              'long' : 0,
              'byte' : 0,
              'short' : 0,
              'void' : 0,
};

for (type = find('Java.Type'); type.hasMatch; type = type.next) {
    if (type in primitives) continue
    var func = new Function("displayType(\'" + type + "\')")
    replace(type, new Link(type.toString(), func))
}

/**
 * Opens a new tab to the Java 1.5 Javadoc for 'type'
 * If no Javadoc exists for 'type', then
 * the user is alerted that it could not be found.
 */

displayType = function(type) {
    if (type.substr(-2) == '[]') type = type.substr(0, type.length - 2)
go('http://java.sun.com/j2se/1.5.0/docs/api/allclasses-frame.html')
    for (var i = 0; i < document.links.length; i++) {
        if (document.links[i].text == type) {
            document.location = document.links[i].toString()
            return;
        }
    }
    alert("No Java type found for: " + type + ".\n" +
          "Perhaps " + type + " is part of a nonstandard Java library.")
}
```

Figure 6-6: JavaScript code that modifies the web page using the embedded Java type

Chapter 7

Conclusion

7.1 Contributions

In this thesis, we described a technique that automatically detects and parses embedded language structures. We introduced the concepts of statistical token frequency analysis for type detection, systematic rule-based view transformations, and modular parser applications. This approach provides an extensible, lightweight approach to the problem of parsing the documents with embedded structures.

We have demonstrated that our system can support a range of embedded structure documents, even though the embedded structures are not explicitly marked by bounding markers. Advanced web navigation using Chickenfoot demonstrated our technique in action. Finally, we identified various applications that use embedded structure syntax for their framework.

7.2 Future Work

7.2.1 Type support

One limitation of the current system to parse embedded structures is that it was developed using only the parsers already available in LAPIS. This limited the language types that it could support, and thus limited the types of embedded language documents that it could fully parse. Currently, the effective parsers in LAPIS are for Java, HTML, and XML. One domain for future work would be to plug in more language parsers into LAPIS that could then be used to parse embedded structures.

7.2.2 Rule generation

The knowledge of parsing embedded structures is encapsulated in the rules file, which is specified by the user. The rules file dictates the type detection, view transformation, and parser application for each class of embedded structure in the document. In order to add support for parsing a new class for embedded structure, the user must modify this rules file appropriately. Required knowledge of the rules schema accepted by the system acts as an impetus for the user to add these new classes. It would be nice to have a user interface in which the user could easily add new language types and rules for embedding to the rules file.

Another concept that is tied to rule generation is the fact that currently the notion of natural boundaries must be understood in order to create a new rule. In order for the system to parse embedded structures in HTML, someone must write in the rules file that the natural boundaries for embedded structures in HTML are HTML elements. One possible avenue for future work to avoid this burden on the user is to use programming by demonstration for rule generation. One can imagine a user viewing a document with multiple embedded structures within it. In order to generate a rule for parsing these structures, the user would merely have to select those embedded structures and specify the language type of those embedded structures. The system would use inference algorithms to detect the probable natural boundaries for this document and associate them with the type of the given document. This effectively takes out the requirement for knowledge of the rules schema, and simplifies rule generation into programming by demonstration.

7.2.3 Error Tolerance in Parsers

One of the challenges of parsing embedded structures is resolving complications that arise from the actual embedding of parsable code. In embedded language documents such as web tutorials, where the syntax of the embedded structures is flexible to errors, this challenge is further complicated by trying to parse embedded structures of unparsable code. To concretize this point, take the example of a Java web tutorial that replaces some of its code with an ellipsis. The ellipsis indicates to the viewer of the web tutorial that it is simply serving as a replacement of Java code that is not important to view. From the perspective of the web tutorials viewer, the embedded structure with the ellipsis makes perfect sense as Java code. However, from the perspective of the Java parser, the ellipsis is unrecognizable syntax and thus the embedded structure is not marked as Java code.

One way to deal with permissible errors in embedded structures, such as ellipsis, is to build in error tolerance in the parser. Building up error tolerance in parsers must be done carefully so that it does not incorrectly parse nonsensical structures as admissible syntax. A rudimentary form of error tolerance was implemented in the Java parser to handle parsing failures on ellipsis tokens. This error tolerance could be extended to support other types of errors such as misspellings, pseudocode, and incomplete code.

Appendix A

Full Listing of Results

id	url	precision	recall	F1
A0	http://java.sun.com/docs/books/tutorial/getStarted/application/classdef.html	1	0.879	0.936
A1	http://java.sun.com/docs/books/tutorial/essential/threads/clock.html	1	0.957	0.978
A2	http://java.sun.com/docs/books/tutorial/essential/system/iostreams.html	0.957	0.998	0.977
A3	http://java.sun.com/docs/books/tutorial/pos/converting/incompatibleChangesAWT.html	0.180	0.217	0.197
A4	http://java.sun.com/docs/books/tutorial/essential/exceptions/definition.html	1	1	1
A5	http://java.sun.com/docs/books/tutorial/jdbc/basics/retrieving.html	0.908	0.991	0.947
A6	http://java.sun.com/docs/books/tutorial/reflect/class/index.html	1	1	1
A7	http://java.sun.com/docs/books/tutorial/jar/sign/intro.html	0	1	0
A8	http://java.sun.com/docs/books/tutorial/javabeans/properties/index.html	0	1	0
A9	http://java.sun.com/docs/books/tutorial/getStarted/cupojava/mac.html	0.639	0.877	0.739
B0	file:///c:/JT2/TIJ3fc.htm	0	1	0
B1	file:///c:/JT2/TIJ322.htm	0	1	0
B2	file:///c:/JT2/TIJ321.htm	0	1	0
B3	file:///c:/JT2/TIJ314.htm	0.935	0.86	0.898
B4	file:///c:/JT2/TIJ3.htm	0	1	0
B5	file:///c:/JT2/TIJ303.htm	0.475	0.915	0.625
B6	file:///c:/JT2/TIJ3fc.htm	0	1	0
B7	file:///c:/JT2/TIJ319.htm	0.934	0.932	0.933
B8	file:///c:/JT2/TIJ315.htm	0.935	0.890	0.912
B9	file:///c:/JT2/TIJ311.htm	0.960	0.93	0.944
C0	http://www.ibiblio.org/javafaq/course/week6/17.html	1	0	0
C1	http://www.ibiblio.org/javafaq/course/week13/18.html	1	0	0
C2	http://www.ibiblio.org/javafaq/course/week10/29.html	1	0	0
C3	http://www.ibiblio.org/javafaq/course/week4/59.html	1	0	0
C4	http://www.ibiblio.org/javafaq/course/week3/16.html	1	0	0
C5	http://www.ibiblio.org/javafaq/course/week8/26.html	1	1	1
C6	http://www.ibiblio.org/javafaq/course/week9/11.html	1	1	1
C7	http://www.ibiblio.org/javafaq/course/week2/30.html	1	0	0
C8	http://www.ibiblio.org/javafaq/course/week13/42.html	1	1	1
C9	http://www.ibiblio.org/javafaq/course/week12/22.html	1	0	0
D0	http://www.javaforstudents.co.uk/truefalse.html	0.993	0.999	0.996
D1	http://www.javaforstudents.co.uk/loopy.html	0.988	0.999	0.993
D2	http://www.javaforstudents.co.uk/methods.html	0.978	0.999	0.988
D3	http://www.javaforstudents.co.uk/methods.html	0.978	0.99	0.988
D4	http://www.javaforstudents.co.uk/compile.html	0.820	1	0.901
D5	http://www.javaforstudents.co.uk/start.html	1	1	1
D6	http://www.javaforstudents.co.uk/bitty.html	0	1	0
D7	http://www.javaforstudents.co.uk/variables.html	0.986	0.997	0
D8	http://www.javaforstudents.co.uk/numbers.html	0.623	0.997	0.767
D9	http://www.javaforstudents.co.uk/switch.html	0	1	0

Table A.1: Java tutorial evaluation results

id	url	precision	recall	F1
A0	http://www.w3.org/TR/REC-html40/struct/links.html	0.101	1	0.184
A1	http://www.w3.org/TR/REC-html40/interact/forms.html#h-1	1	0	0
A2	http://www.w3.org/TR/REC-html40/about.html#h	1	1	1
A3	http://www.w3.org/TR/REC-html40/types.html#h-	1	0	0
A4	http://www.w3.org/TR/REC-html40/intro/intro.html#h	1	0	0
A5	http://www.w3.org/TR/REC-html40/appendix/notes.html#h-	1	0	0
A6	http://www.w3.org/TR/REC-html40/interact/forms.html#h-1	1	0	0
A7	http://www.w3.org/TR/REC-html40/appendix/notes.html#h	1	0	0
A8	http://www.w3.org/TR/REC-html40/appendix/notes.html#h	1	0	0
A9	http://www.w3.org/TR/REC-html40/interact/forms.html#h-1	1	0	0
B0	http://www.yourhtmlsource.com/accessibility/redesigning.html	0	1	0
B1	http://www.yourhtmlsource.com/accessibility/	1	1	1
B2	http://www.yourhtmlsource.com/forms/basicforms.html	0.393	0.992	0.563
B3	http://www.yourhtmlsource.com/promotion/linkrequests.html	0	1	0
B4	http://www.yourhtmlsource.com/accessibility/10badthings.html	0	1	0
B5	http://www.yourhtmlsource.com/promotion/	1	0	0
B6	http://www.yourhtmlsource.com/stylesheets/csslinks.html		1	0.214
B7	http://www.yourhtmlsource.com/javascript/scriptingframes.html	0.057	0.949	0.109
B8	http://www.yourhtmlsource.com/myfirstsite/basicwebdesign.html	0	1	0
B9	http://www.yourhtmlsource.com/sitemanagement/ssiecho.html	0.017	0.044	0.025
C0	http://www.davesite.com/webstation/html/chap07.shtml	1	0	0
C1	http://www.davesite.com/webstation/html/chap17.shtml	1	0	0
C2	http://www.davesite.com/webstation/html/chap03.shtml	1	0	0
C3	http://www.davesite.com/webstation/html/chapX1.shtml	1	0	0
C4	http://www.davesite.com/webstation/html/domain.shtml	1	1	1
C5	http://www.davesite.com/webstation/html/chap11.shtml	1	0	0
C6	http://www.davesite.com/webstation/html/chap04.shtml	1	0	0
C7	http://www.davesite.com/webstation/html/chap10.shtml	1	0	0
C8	http://www.davesite.com/webstation/html/chap16.shtml	1	1	1
C9	http://www.davesite.com/webstation/html/chap14_3.shtml	1	0	0
D0	http://www.weballey.net/tables/index.html	0.949	0.993	0.970
D1	http://www.weballey.net/tables/expanding.html	1	1	1
D2	http://www.weballey.net/tables/sizing.html	1	1	1
D3	http://www.weballey.net/tables/colors.html	0.994	1	0.997
D4	http://www.weballey.net/tables/borders.html	1	1	1
D5	http://www.weballey.net/tables/alignment.html	1	0.997	0.998
D6	http://www.weballey.net/tables/nesting.html	0.173	1	0.295
D7	http://www.weballey.net/tables/navbar.html	0.794	0.984	0.879
D8	http://www.weballey.net/tables/quickrecap.html	1	1	1
D9	http://www.weballey.net/tables/tags.html	1	0	0
E0	http://www.pageresource.com/html/metref.htm	1	0.996	0.998
E1	http://www.pageresource.com/dhtml/csstut8.htm	0.687	0.905	0.781
E2	http://www.pageresource.com/dhtml/csstut1.htm	1	1	1
E3	http://www.pageresource.com/html/embed.htm	0.657	0.453	0.537
E4	http://www.pageresource.com/html/formhelp.htm	0.956	0.829	0.888
E5	http://www.pageresource.com/html/linking.htm	1	1	1
E6	http://www.pageresource.com/putweb/ftptut2.htm	1	0	0
E7	http://www.pageresource.com/html/frame1.htm	0.980	0.973	0.977
E8	http://www.pageresource.com/html/hr2.htm	0.873	0.947	0.909
E9	http://www.pageresource.com/html/bgcolor.htm	0.946	0.959	0.952

Table A.2: HTML tutorial evaluation results

id	url	precision	recall	F1
A0	file:///c:/jsp/jspcookbook/chap23/cookieChap23.jsp	1	1	1
A1	file:///c:/jsp/jspcookbook/chap10/cookieSet.jsp	1	0.826	0.904
A2	file:///c:/jsp/jspcookbook/chap27/amazon.jsp	1	1	1
A3	file:///c:/jsp/jspcookbook/chap7/beanSet.jsp	1	1	1
A4	file:///c:/jsp/jspcookbook/chap27/google.jsp	1	1	1
A5	file:///c:/jsp/jspcookbook/chap6/header.jsp	1	0	0
A6	file:///c:/jsp/jspcookbook/chap17/qtmusic.jsp	1	1	1
A7	file:///c:/jsp/jspcookbook/chap22/logoTest.jsp	1	1	1
A8	file:///c:/jsp/jspcookbook/chap6/solutions.jsp	1	1	1
A9	file:///c:/jsp/jspcookbook/chap1/firstJsp.jsp	1	0.447	0.618
B0	file:///c:/jsp/mbolin/psk/white_pages.jsp	1	0.946	0.972
B1	file:///c:/jsp/mbolin/rush/add_event.jsp	1	0.	0.984
B2	file:///c:/jsp/mbolin/toys/index.jsp	1	1	1
B3	file:///c:/jsp/mbolin/rush/rushing_team.jsp	0.95	0.588	0.726
B4	file:///c:/jsp/mbolin/default.jsp	1	0.822	0.902
B5	file:///c:/jsp/mbolin/crap_calendar.jsp	1	0.	0.984
B6	file:///c:/jsp/mbolin/rush/picturebook.jsp	1	0.942	0.970
B7	file:///c:/jsp/mbolin/laundry/index.jsp	0.987	0.754	0.855
B8	file:///c:/jsp/mbolin/admin/process_create.jsp	1	0.965	0.982
B9	file:///c:/jsp/mbolin/rush/process_rushee_decision.jsp	1	0.992	0.996
C0	file:///c:/jsp/manning/wdjsp/webdev/byexample/viewhtml.jsp	1	1	1
C1	file:///c:/jsp/manning/wdjsp/webdev/commontasks/blue-cookie.jsp	1	0	0
C2	file:///c:/jsp/manning/wdjsp/webdev/commontasks/uptime.jsp	1	1	1
C3	file:///c:/jsp/manning/wdjsp/webdev/advtags/forTag.jsp	1	1	1
C4	file:///c:/jsp/manning/wdjsp/webdev/byexample/quote.jsp	1	1	1
C5	file:///c:/jsp/manning/wdjsp/webdev/byexample/whois.jsp	0.498	0.815	0.618
C6	file:///c:/jsp/manning/wdjsp/webdev/scripting/fact-comment.jsp	1	0	0
C7	file:///c:/jsp/manning/wdjsp/webdev/byexample/viewsource.jsp	1	0.997	0.998
C8	file:///c:/jsp/manning/wdjsp/webdev/commontasks/thanks.jsp	1	1	1
C9	file:///c:/jsp/manning/wdjsp/webdev/databases/CachedResults.jsp	1	0.824	0.903
D0	file:///c:/jsp/wrox/ch18-spring-exercise/web/WEB-INF/jsp/form.jsp	1	1	1
D1	file:///c:/jsp/wrox/ch20-tiles-exercises/web/tiles/it/body.jsp	1	1	1
D2	file:///c:/jsp/wrox/ch18-spring-exercise/web/index.jsp	1	1	1
D3	file:///c:/jsp/wrox/web/roster.jsp	1	1	1
D4	file:///c:/jsp/wrox/web/input.jsp	1	1	1
D5	file:///c:/jsp/wrox/web/example1/name-list.jsp	1	1	1
D6	file:///c:/jsp/wrox/ch20-tiles-exercises/web/grandchild-index.jsp	1	1	1
D7	file:///c:/jsp/wrox/web/footy.jsp	1	1	1
D8	file:///c:/jsp/wrox/HelloJSF/greeting.jsp	1	1	1
D9	file:///c:/jsp/wrox/ch18-webwork/webwork-skeleton/template/vxml/filled-header.jsp	1	0	0
E0	file:///c:/jsp/jsp3/ora/ch9/error6.jsp	1	1	1
E1	file:///c:/jsp/jsp3/ora/ch17/userinfovalid.jsp	1	1	1
E2	file:///c:/jsp/jsp3/ora/ch15/phone.jsp	1	1	1
E3	file:///c:/jsp/jsp3/ora/ch12/validate.jsp	1	1	1
E4	file:///c:/jsp/jsp3/ora/ch10/product.jsp	1	1	1
E5	file:///c:/jsp/jsp3/ora/ch17/page3.jsp	1	1	1
E6	file:///c:/jsp/jsp3/ora/ch11/even_and_odd3.jsp	1	1	1
E7	file:///c:/jsp/jsp3/ora/ch11/message.jsp	1	1	1
E8	file:///c:/jsp/jsp3/ora/ch21/convert.jsp	1	1	1
E9	file:///c:/jsp/jsp3/ora/ch19/login.jsp	1	1	1

Table A.3: JSP evaluation results

id	url	precision	recall	F1
A0	file:///c:/JD1/java/awt/event/ContainerAdapter.java	0.697	1	0.822
A1	file:///c:/JD1/org/apache/xpath/axes/ChildIterator.java	1	0	0
A2	file:///c:/JD1/java/util/prefs/WindowsPreferences.java	0.102	0.994	0.185
A3	file:///c:/JD1/java/security/interfaces/RSAMultiPrimePrivateCrtKey.java	0.216	0.992	0.356
A4	file:///c:/JD1/java/nio/DirectFloatBufferS.java	1	1	1
A5	file:///c:/JD1/org/omg/CosNaming/NamingContextExtHolder.java	0.346	0.994	0.514
A6	file:///c:/JD1/java/util/regex/Matcher.java	0.613	1	0.760
A7	file:///c:/JD1/java/nio/ByteBufferAsFloatBufferRL.java	1	1	1
A8	file:///c:/JD1/java/io/Reader.java	1	0	0
A9	file:///c:/JD1/java/security/interfaces/DSAParams.java	0.560	1	0.718
B0	file:///c:/org/bouncycastle/crypto/generators/DHKeyPairGenerator.java	1	1	1
B1	file:///c:/org/gudy/azureus2/pluginsimpl/local/torrent/TorrentAttributeNetworksImpl.java	1	0	0
B2	file:///c:/com/aelitis/azureus/core/networkmanager/impl/RateHandler.java	1	0	0
B3	file:///c:/org/gudy/azureus2/ui/swt/update/UpdateWindow.java	1	0	0
B4	file:///c:/org/gudy/azureus2/ui/swt/views/tableitems/peers/ConnectedTimeItem.java	1	1	1
B5	file:///c:/org/pf/text/StringUtil.java	0.992	0.048	0.092
B6	file:///c:/com/aelitis/azureus/core/diskmanager/cache/CacheFile.java	1	1	1
B7	file:///c:/org/gudy/azureus2/ui/console/commands/Alias.java	0	0	NaN
B8	file:///c:/org/gudy/azureus2/pluginsimpl/local/messaging/MessageManagerImpl.java	1	0	0
B9	file:///c:/org/bouncycastle/asn1/x509/V3TBSCertificateGenerator.java	0.179	1	0.304
C0	file:///c:/hipergate-build/java/org/htmlparser/AbstractNode.java	0	0.039	0.075
C1	file:///c:/hipergate-build/java/org/w3c/tidy/EntityTable.java	0.076	0.997	0.141
C2	file:///c:/hipergate-build/java/com/lowagie/text/html/HtmlEncoder.java	0.998	0.932	0.964
C3	file:///c:/hipergate-build/java/com/lowagie/text/rtf/document/RtfCodePage.java	1	1	1
C4	file:///c:/hipergate-build/java/com/knowgate/http/portlets/HipergatePortletContext.java	1	1	1
C5	file:///c:/hipergate-build/java/org/htmlparser/util/FeedbackManager.java	1	0	0
C6	file:///c:/hipergate-build/java/com/lowagie/text/rtf/style/RtfFontList.java	1	1	1
C7	file:///c:/hipergate-build/java/com/knowgate/workareas/FileSystemWorkArea.java	0.379	0.986	0.548
C8	file:///c:/hipergate-build/java/com/lowagie/text/pdf/PdfWriter.java	0.942	0.562	0.704
C9	file:///c:/hipergate-build/java/com/lowagie/text/rtf/document/RtfPageSetting.java	0	0	NaN
D0	file:///c:/fileIndexer/src/com/warehouse/fileIndexer/exclusion/Exclusion.java	0.944	0.898	0.921
D1	file:///c:/fileIndexer/src/com/warehouse/fileIndexer/datasource/KTDataSource.java	0.033	0.997	0.065
D2	file:///c:/fileIndexer/src/com/warehouse/fileIndexer/parser/Cleaner.java	0.881	0.854	0.86
D3	file:///c:/fileIndexer/src/com/warehouse/fileIndexer/datasource/DataSourceException.java	0.550	0.401	0.464
D4	file:///c:/fileIndexer/src/com/warehouse/fileIndexer/FileIndexer.java	1	1	1
D5	file:///c:/fileIndexer/src/com/warehouse/fileIndexer/datasource/DataSource.java	1	0	0
D6	file:///c:/fileIndexer/src/com/warehouse/fileIndexer/parser/DefaultParser.java	1	0	0
D7	file:///c:/fileIndexer/src/com/warehouse/fileIndexer/permission/PermissionManager.java	1	0	0
D8	file:///C:/fileIndexer/src/com/warehouse/fileIndexer/exclusion/FileExclusion.java	1	0	0
D9	file:///c:/fileIndexer/src/com/warehouse/fileIndexer/ParserTypes.java	1	1	1
E0	file:///c:/src/java/com/gargoyleSW/htmlunit/html/HtmlTableHeaderCell.java	1	0.050	0.095
E1	file:///c:/src/java/com/gargoyleSW/htmlunit/html/HtmlParagraph.java	1	0.430	0.602
E2	file:///c:/src/test/java/com/gargoyleSW/htmlunit/javascript/host/NodeImplTest.java	0.963	0.970	0.967
E3	file:///c:/src/java/com/gargoyleSW/htmlunit/html/HtmlFrame.java	0.9	0.969	0.980
E4	file:///c:/src/test/java/com/gargoyleSW/htmlunit/html/HtmlAnchorTest.java	0.986	0.973	0.980
E5	file:///c:/src/test/java/com/gargoyleSW/htmlunit/WebTestCase.java	0.991	0.193	0.323
E6	file:///c:/src/test/java/com/gargoyleSW/htmlunit/HTMLParserTest.java	0.	0.984	0.976
E7	file:///c:/src/java/com/gargoyleSW/htmlunit/html/HtmlListItem.java	1	0.277	0.434
E8	file:///c:/src/java/com/gargoyleSW/htmlunit/html/HtmlTextArea.java	0.994	0.089	0.164
E9	file:///c:/src/test/java/com/gargoyleSW/htmlunit/ScriptFilterTest.java	1	1	1

Table A.4: Java source evaluation results

Bibliography

[1] De Lara, E., et al. A Characterization of Compound Documents on the Web. *Rice Computer Science technical report*, TR99-351. Nov 29, 1999.

[2] Gray, D., et al. Modern Languages and Microsofts Component Object Model. *Communications of the ACM.* May 1998 - Vol 41.

[3] Ragett, D. "Getting Started with HTML." http://www.w3.org/MarkUp/Guide/ Feb 13, 2002

[4] Bolin, M. "End-User Programming for the Web." http://groups.csail.mit.edu/uid/projects/chickenfoot/ May 2005.

[5] Graham, P. "A Plan for Spam." http://www.paulgraham.com/spam.html Aug 2002.

[6] Graham, P. "Better Bayesian Filtering." http://www.paulgraham.com/better.html Jan 2003.

[7] Johnson, S. "Yacc: Yet Another Compiler-Compiler." http://dinosaur.compilertools.net/yacc/index.html

[8] http://en.wikipedia.org/wiki/LR

[9] Donnelly, C. and Stallman, R. "Bison; The YACC-compatible Parser Generator." http://dinosaur.compilertools.net/bison/index.html Nov 1995.

[10] http://sourceforge.net/projects/byacc/

[11] https://javacc.dev.java.net/

[12] Begel, A. and Graham, S. L. Language analysis and tools for input stream ambiguities. *Proceedings of the Fourth Workshop on Language Descriptions, Tools and Applications (LDTA '04)*, Electronic Notes in Theoretical Computer Science, Barcelona, Spain, Apr 2004.

[13] Bravenboer, M. and Visser E. Concrete Syntax for Objects. *Proceedings of the 19th Annual ACM Conference on Object-Oriented Programmin, Systems, Languages, and Applications (OOPSLA '04)*, Vancouver, Canada, Oct 2004.

[14] Miller, R. Lightweight Structure in Text. PhD thesis. Computer Science Dept., School of Computer Science, Carnegie Mellon University, May 2002.

[15] http://www.google.com/

[illegible]