Detecting and Parsing Embedded Lightweight Structures

M.Eng Thesis Proposal
Philip Rha
May 13, 2004
1. Introduction

Text documents, web pages, and source code all contain language structures that may be parsed by custom parsers. However, documents may contain embedded language structures that require multiple parsers operating on the entire document. There are many documents that contain such embedded structures, yet many cannot be fully parsed by the existing parsers available to us due to the fact that parsers are often very strict about the input that they can effectively parse. To concretize the problems that parsing documents with embedded structures poses, let us look at some example documents.

• Java tutorial pages are instructional web pages written in HTML to help teach concepts of Java programming. The majority of the text is written in HTML, but there are frequently sections of Java code included in the tutorial to serve as an example. The snippets of Java code in these tutorial pages have inherent structure in them that can be parsed by the Java parser. These fragments of code when parsed alone by the Java parser reveal structures like methods, variables, fields, comments, and classes. However, none of this information can be found when the Java tutorial webpage is loaded because the structure is buried within HTML structures, which the Java parser will fail on.

Figure 1: Screenshot of a Java tutorial page of rendered HTML with embedded Java structures

• Certain comments in Java source files often contain HTML code, as the Javadoc tool that generates API documentation creates HTML pages describing the classes and methods declared within the files. For example, a Javadoc comment below describing a Graphics object contains a number of HTML tags.
Figure 2: Example Javadoc comment using embedded HTML structures

The HTML parser when run on just the comment could detect structures like start tags, end tags, elements, and text. However, when this comment appears within a Java file, we lack the means to get to this information. This is because those HTML structures are embedded within the Java structure of a Javadoc comment.

Javadoc comments and Java tutorial pages are two examples of documents with embedded structures that we cannot currently detect or parse. Even if we had access to both a Java parser and an HTML parser, the Java parser would not successfully run on the HTML document and the HTML parser would not be able to detect the embedded Java structures. But because we have access to both these parsers, we would still be reluctant to create a new parser to specially parse these documents with embedded structures. Creating custom parsers for each type of embedded document is cumbersome and has a very limited scope.

For my thesis, I propose to use the internal structures in the document to define regions of the document that existing parsers can be run on, effectively creating a custom parser. This will be done within the framework of LAPIS, a programmable web browser and text editor. LAPIS allows users to select regions of text or region sets by applying different patterns to the text. LAPIS utilizes lightweight structure or the ability to detect text structure automatically using patterns and parsers. [1] There are already a number of built-in patterns and parsers in LAPIS to detect different structures, whether these structures are Java code, HTML code, or English grammar. These built-in patterns can define the region set on which a specific parser is to act. This paper will outline a design that will take a user-generated example of an embedded structure within a document and
create these region sets over which the appropriate parser will run. The parser will be chosen by testing the type of the document and following a user-generated rule system. Through an effective document-typing system and an intelligent inference tool, users should be able to parse embedded structures using a handful of well-chosen examples.

2. Design Goals

To guide the design discussion, we should examine more closely how LAPIS should deal with documents containing embedded structures.

2.1 Desired Functionality

The functionality that we are hoping to achieve is to have LAPIS open a document and automatically detect what kinds of embedded structures are contained in the document. LAPIS can then infer which parsers it should apply to these structures, and should furthermore know which regions to which it should apply those parsers. LAPIS would then apply the appropriate parsers to each of the regions of embedded code, populating the pattern library with its mappings.

2.2 Parsing Problems

As previously mentioned, there are a number of different examples of documents with embedded structures. The examples of Java tutorial webpages and Java files with Javadoc comments are only two kinds of these embedded structures. In order to build a useful parser composition tool, it is important that this tool handle all types of parser composition problems. This section describes some of the documents containing embedded structures that this project hopes to construct composite parsers for.

- Online tutorial pages for programming languages are generally good examples for documents with embedded structures. This is because just the fact that these pages are online usually means that they are written in HTML. Any code fragments used in the tutorial will serve as an embedded structure in the HTML.

- Java tutorial pages for writing Javadoc comments
  Embedded structures can be nested; an HTML page could have fragments of Java code to be displayed that contains Javadoc comments containing HTML code. The HTML that appears in the Javadoc comment is well-formed when it is rendered by the browser (see Figure 3), but is not well-formed HTML in the source code (see Figure 4).
Examples of Doc Comments

```java
/**
 * 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
 * ```

Figure 3: Rendered Javadoc tutorial webpage, displaying HTML in a Javadoc comment

Figure 4: Javadoc tutorial source code written in HTML, displaying HTML within Javadoc comments

- Text documents broken up into a page structure
  A sentence interrupted by a page break should still be able to be parsed by a sentence parser so that the page break is ignored.

- Web pages with Javascript code
  Javascript code is embedded in HTML documents, preceded by the special HTML tag `<SCRIPT LANGUAGE="JavaScript">` and followed by the HTML tag `</SCRIPT>`.

- JavaServer Pages (JSP)
  JSP pages include custom Java code as well as XML and HTML tags.
• Active Server Pages (ASP)
  ASP pages have support for Visual Basic, JavaScript, and C# as embedded languages.

• Structured Query Language (SQL)
  SQL queries can be embedded in various languages like Java and Perl.

These documents guide the design of the parser composition system as they introduce the variety of problems that parsing encounters. For example, the nested nature of the embedded structures in the last example indicates that an effective parser composition system should be iterative. Because these documents drive the design of the system, these example problems also serve to evaluate the system.

3. System Overview

Given the goals of detecting and parsing the embedded structures within documents, a reasonable initial system design is as follows. The system is broken up into three phases: type detection, parser composition, and rule construction.

3.1 Type Detection

The type detection phase occurs when a document is newly loaded into LAPIS. LAPIS first detects the type of the document and the corresponding primary parser which operates on the document. It also detects the embedded structures within the document and outlines which parsers act on which regions of the document. It does this by running a series of tests on the document to check what kind of document it is. The tests can include checks for patterns in the filename or URL of the document, as well as patterns in the content of the document. For example, the JavaTutorialTest could test whether the URL had a basename of http://java.sun.com/docs/books/tutorial/java/ and if the page was an HTML file. If the document passes JavaTutorialTest, LAPIS will know that the document is a Java tutorial webpage.

It should be possible to reuse and compose type tests to reduce the number of structures the system must build to detect a new document type. Each type test can be composed of a battery of tests testing the filename of the document, the content of the document, or the source of the document. Type tests should be able to be reused in other type tests. For example, the JavaTutorialTest should be able to run the HTMLTest on the document to make sure that the document is an HTML document.

In order to generate these tests, LAPIS can create a rules file which dictates what tests to use to detect specific types of documents. This rules file will be specified in XML so that it can be dynamically generated.
In Figure 5, the type JavaTutorial has a single test: checking the URL of the document to see if it matches the pattern http://java.sun.com/docs/tutorial/*.html.

3.2 Parser Composition

Once the document type has been determined, LAPIS can check its rules file for instructions on how to parse each document type. The format is as follows:

```xml
<typeset>
  <type name = "JavaTutorial">
    <url pattern = "http://java.sun.com/docs/tutorial/*.html"/>
    <parser primary = "HTMLParser"/>
    <structure name = "javaCode">
      <parser secondary = "JavaParser"/>
      <TC region = "Text after '<pre>' and before '</pre>'"/>
    </structure>
  </type>
  <type name = "Java">
    <url pattern = "*.java"/>
    <structure name = "javadoc">
      <parser name = "HTMLParser"/>
      <TC region = "Java.Comment starting with '/**'"/>
    </structure>
  </type>
</typeset>
```

The URL pattern is used to help detect the type name. Once the type is determined, the primary parser is run over the entire document. For every structure specified in the type, the specified parser is run over the region specified in the TC pattern language. As stated before, the parsers used in LAPIS take in entire documents as input. To correctly implement parser composition, the parsers must be changed to operate only over regions of the document.
3.3 Rule Construction

Constructing the rules file will be done through user input. The user can create a new rule for a given document type by selecting an option on the menu bar. The user will then be prompted for the name of a secondary parser to be run on the document type. After the secondary parser is chosen, the user will select a region in the document for which that parser is valid. LAPIS will then use this selection as an example in its inference algorithm to construct a list of hypothesized patterns that map to region sets including the selected region. LAPIS can filter these hypothesized patterns by removing the patterns that have regions over which the secondary parser fails. The pattern guesses that LAPIS makes can be improved by additional selections by the user. Once the user has chosen one of the guesses that LAPIS has made, it will write the TC pattern to the rules file so that it can be used in the parser composition phase.
4. Evaluation

Before we can begin building a parser composition system, we must first come up with a way of evaluating this system. The system is broken up into three units: type detection, parser composition, and rule construction. It is important to evaluate each unit separately in addition to testing the system as a whole. The type detection phase can easily be tested by loading documents and seeing if the correct document types are found. All the types of documents containing embedded structures listed in Section 2.2 (Parsing Problems) should be considered for these tests.

The documents listed in Section 2.2 should also be included in the parser composition testing. Given the correct regions, the secondary parser should be able to parse each embedded structure as if it were in its own document and not embedded within another structure.

The rule construction phase involves the user generating examples for LAPIS’s inference algorithm. This is to identify a pattern that finds the embedded structures using only a few examples. This can be evaluated by conducting user studies on users isolating regions of embedded structures using different examples. These results of the user studies can be measured qualitatively by response surveys, amount of time required to isolate the regions, and the number of positive and negative examples required to isolate the regions.

5. Related Work

One example of parser composition is the ability of the XML parser to parse XHTML pages. XHTML pages are compound documents that contain multiple XML namespaces.[2] These compound documents can have nested namespaces, with an arbitrary nesting depth. Each namespace represents a different type of structure in the document. Figure 8 shows an example of an XHTML page with HTML and SVG structures in them.
One problem with XML parser composition is that it forces the compound document authors to use XML. Each author must adhere to the XML format and clearly mark the namespace of each embedded structure. The design outlined in this thesis allows the user to detect embedded structures in documents that are not clearly marked in a standardized format.

There is a substantial body of work about constructing compound documents, a problem similar to the parser composition problem that we are trying to solve. Microsoft’s Object Linking and Embedding (OLE) facilitates the construction of compound documents, using several Component Object Model (COM) interfaces to embed applications within one another.[3] Examples of this include the ability to embed Microsoft Excel spreadsheets within Microsoft Word documents. This is because software components in this model are essentially abstracted from each other so that they are object-oriented.

A similar object standard is used in Apple’s OpenDoc, an object-oriented development platform. A document in OpenDoc acts as a container holding different software components. Each document is thus a compound document consisting of different components, which all have its own functionality user interface. The user experience with OpenDoc is determined in part by the ease with which nested components may be activated. The transparency with which nested structures are treated is also critical to the user experience of LAPIS parser composition.
6. Conclusion
The research proposed in this paper builds a methodology for composing independent structure detectors to find and utilize the structural information found in embedded structures. The research identifies the problem of embedded structures in that current parsers are tied to the input they are given. Using LAPIS, we can work around this problem by dynamically selecting the regions of the embedded structures and then parsing those regions. This enables LAPIS users to make use of the structural information available to all applicable parsers for the region set selected.

References