Robust Multilingual Parsing Using Island Grammars

Nikita Synytskyy, James R. Cordy, Thomas R. Dean
School of Computing, Queen's University
Kingston, Ontario, Canada K7L 3N6
nikita@mondenet.com, cordy@cs.queensu.ca, thomas.dean@ece.queensu.ca

Abstract

Any attempt at automated software analysis or modification must be preceded by a comprehension step, i.e. parsing. This task, while often considered straightforward, can in fact be made very challenging depending on the source code in question. Files that make up web applications serve as an example of such difficult-to-parse artifacts, for two reasons. Firstly, these files routinely contain several programming languages at once, sometimes with widely varying syntaxes, and intermingled at the statement level. Secondly, the code routinely contains syntax errors. Understanding such files calls for a robust parser that can handle multiple languages simultaneously.

An approach to creating such a parser, based on the concept of island grammars, is presented here. Island grammars have been used in the past for robust parsing and lightweight analysis of software. Some of the features of these grammars make them uniquely fit for parsing multiple languages simultaneously.

1 Introduction

Automated analysis of software is impossible without automated understanding of software. The process of understanding usually involves some form of parsing of software artifacts—most often source code. Parsing is well-understood and thus is often considered to be a "simple" process; actual analysis of the code is where the greatest difficulties seem to lie.

Nevertheless, the initial understanding step in software analysis can present some interesting challenges. This paper focuses on two common stumbling blocks for parsing—multilingual code, and code with syntax errors in it. Either of these characteristics makes parsing more difficult; they can also occur in documents together, thus creating a rather serious obstacle to parsing.

Multilingual documents. Multilingual code occurs frequently in software projects. Several languages (Java, Cobol) provide for embedding of SQL statements right in the language, and referring to results of SQL statement execution later in the program. Source code files containing such constructs are essentially written in several languages, although one language (e.g. Java in case of JDBC) usually dominates, in terms of both line count and the percentage of functionality implemented. Another common example of programs written in several languages are C/C++ source files. The preprocessor instructions, when taken together, comprise a very flexible and expressive language of their own, and are integral to achieving the final functionality of the program. In this case, again, the C/C++ code tends to dominate on a line-count basis.

Documents that make up web applications are perhaps the ultimate example of language mixing. Dynamic web pages contain mixtures of markup languages (HTML), and "true" programming languages, meant for execution rather than rendering. Unlike in most other language-mixing schemes, the languages in web documents are closely intertwined—as later examples will show, statements of one language can appear in statements in another language. Moreover, it is difficult to say with confidence which language is predominant, in line count or functionality.

For the purposes of reverse engineering, it is better to analyze all the languages in a multilingual document together, rather than separately. A great deal of information is contained not only in the languages themselves, but also in the ways in which they are nested, intertwined and used together. None of this information can be ignored in software analysis[1]. Having data about all the languages in the same parse tree also enables us to perform cross-language analysis and transformations on the code.

Robust Parsing. There are many situations in software engineering where incoming source code must be processed even though it contains errors or other unexpected content. Robustness allows parsing to get around a variety of obstacles. Two of these obstacles are syntax errors and language dialects. Browsers and compilers accept variations of the language that are not described in the manuals and web site designers will take advantage of anything the browser or

compiler will give them. In addition, web pages can and do contain errors. Browsers ignore unrecognized HTML tags and are very liberal about requirements for nesting of tags or use of close tags (i.e. omitting `</p>` or `</li>` tags is typically not an error). These observations are not new [2, 3, 4], but add an extra constraint when building a grammar that will accept multiple languages.

Approach. In this paper we present an approach to simultaneously parsing multiple languages into the same parse tree, and doing it in a robust way. The approach relies on island grammars, as introduced by van Deursen[5] and Moonen[2], and originally developed in the speech understanding community[6]. Island grammars have been used in the past for lightweight and/or robust analysis and data extraction[3]. This paper demonstrates that that many of the the properties that make island grammars suitable for robust parsing and lightweight analysis also make them useful for parsing multiple languages. Moreover, the switch to multiple languages does not diminish the ability to do robust parsing.

Island grammars readily adapt to multilingual parsing because they are very good at extracting multiple, and potentially very dissimilar, features from the input documents that are parser-unfriendly, i.e. contain errors or uninteresting input. By having an island (or several islands) devoted to each language that can be expected in the parse, separate languages can be extracted from the input without mix-ups. Any input that is not matched by any of the islands is either uninteresting to the grammar or malformed, and is treated as catch-all water.

The islands can be arbitrarily complex, and therefore impose no restrictions on the languages that can be parsed. Separate islands are coupled rather loosely, so changes to one language do not affect changes in another language. Island grammars provide for nesting of islands within islands, which in turn enables processing of languages that are arbitrarily nested within each other.

We demonstrate this approach by presenting a grammar which is able to robustly parse documents that make up dynamic web sites conforming to Microsoft's Active Server Pages (ASP) standard. These pages contain three languages—HTML, often the main source of malformed or uninteresting code, and Visual Basic and JavaScript, used to implement server-side and client-side functionality, respectively.

2 Parsing Multilingual Code

The presence of more than one language in a document complicates parsing in several ways. The most basic challenge is to determine where the code in one language ends, and code in another language begins. Robust parsing makes this task even harder, since valid code in one language is erroneous code in another language. The parser has to be able to distinguish between code that is valid in at least one of the languages, and code that is invalid in any of them, and treat each code section accordingly. Most of the time, however, multilingual code contains delimiters or other clues that simplify language detection. The details of understanding multilingual code, however often prove difficult to get right.

Tokenization. Lexical rules can vary significantly between languages that are co-located in the same file. Since the input stream is split into tokens before the parsing proper begins, it is difficult to match tokenization rules used to the language that is currently being tokenized. If a parser is being developed for a fixed and known combination of languages, it is possible to hard-code the switch of tokenization rules, but solving the problem in general is considerably harder. This presents a considerable obstacle to parsing—if the input stream was split into tokens incorrectly, further stages of the parse will most likely fail, or give incorrect results.

Whitespace sensitivity. Another problem closely related to the issue of tokenization is the handling of whitespace. The amount of meaning assigned to whitespace differs greatly among different languages. Some popular languages, like C/C++, HTML and Java, completely ignore whitespace, and assign no meaning to it. In other languages whitespace has some syntactic meaning—Visual Basic for example, relies on new lines to detect ends of statements. Finally, some languages depend heavily on space to convey syntactic meaning—one example is Python, which uses indentation to determine nesting.

When two or more languages with different levels of whitespace sensitivity are parsed together, the differences in whitespace treatment must somehow be reconciled. Performing a simple whitespace-insensitive parse is not an option, because removing whitespace from whitespace-sensitive code would disrupt it to the point of destroying its meaning. On the other hand, performing a whitespace-sensitive parse is not an option either, because the meaning attached to whitespace propagates across all languages in the parse, and adds erroneous information to code written in whitespace-insensitive languages.

Comments. Different commenting conventions used by various languages present yet another obstacle to handling multilingual code. The differences between commenting conventions are quite drastic, and what is a comment marker in one language is often a valid identifier in others. Visual Basic for example uses the keyword `rem` or a single quote to denote the start of its comments. In most other languages, however, `rem` can be an identifier, whereas single quote is most frequently used to delimit character strings, not comments.

When unilingual source code files are processed for

```
<html>
<script>
function capitalize() {
  //capitalize the string
  temp= new String (theForm.textbox.value)
  theForm.textbox.value =
       temp.toUpperCase()
}
</script>
Welcome to our sample ASP page!
<%
  txtValue = Request.Form("textbox")
  if (txtValue <> "") then
    rem we have submissions, so
    rem display the value of txtValue
%>
  <table border=1>
    <tr>
      <td>
        You typed in: <% =txtValue %>
      </td>
    </tr>
  </table>
<%
  else
  'display the form for collecting data
%>
<form action="sample.asp" name=theForm>
       onSubmit="capitalize()">
  <input type="text" name="textbox"
         value="Type Something Here">
  <br>
  <input type="submit" value="Submit">
</form>
<%
  end if
%>
</html>
```

Figure 1. A sample of multilingual ASP code.

compilation or interpretation, the comments are usually removed by lexical matching before parsing proper begins. This approach is not as straightforward, or as fitting, when multilingual documents are parsed for reverse engineering purposes. First of all, lexical matching would have to somehow detect language boundaries, and match different comment patterns depending on which language is currently being processed. Second, for reverse engineering purposes it is beneficial to leave the comments intact, because they can hold important information about the code[7].

3 Web Analysis Grammar

As a proof of concept—and a showcase—for our parsing approach, we have developed a multilingual grammar for parsing web documents conforming to Microsoft's ASP specification. ASP allows for mixing of of client- and server-side languages together in the same file, and is thus similar to other technologies for developing dynamic web pages, such as Java Server Pages (JSP), ColdFusion, and others. ASP pages usually contain a mix of three languages—Visual Basic(VB), which is ASP's server side language of choice, and HTML and JavaScript, which are intended for client-side rendering or execution. A simple example of such a page is given in Figure 1.

Sample Target. The example in Figure 1 is a simplified version of many ASP pages that accept input from a user, perform initial client-side processing, and then perform an action based on the input. The code in Figure 1 displays an input form, expecting a single string of input from the user. After the "Submit" button is pressed, but before the data is sent over to the server, a JavaScript function converts the string to uppercase. Finally, the server-side code performs an action on the string—in this case, simply echoing it to the user. The page follows a popular ASP design pattern, submitting its input to itself. The way the page looks is determined by server-side code—if no submission is detected, the input form is displayed; otherwise, the code performs data processing and displays the result.

This small example presents almost all of the parsing challenges discussed above. Some of the languages are whitespace-sensitive (Visual Basic) while others are not (HTML and JavaScript). Commenting conventions vary considerably between these languages. HTML contains syntactically unpredictable constructs that require robustness—text encountered inside tables and forms, and elsewhere in the document can be arbitrary. Furthermore, we will not be processing all HTML tags; our interest is limited to HTML tags that define structure and behavior of the document, namely tables, forms, links and client-side scripts. Other tags, such as the `<br>` tags encountered inside the form, are uninteresting from our point of view, and should be ignored, but preserved in the code for possible further analysis.

The additional level of complexity presented by ASP pages is language intertwining. For example, the `if` statement in the document is separated into three separate parts by "snippets" of HTML—the table and the form. The challenge in this case is to recognize that pieces of the `if` statement, even though stranded in their own VB snippets, are not in fact malformed, but make up a single, congruous statement.

The Goals. To parse multilingual, and potentially malformed, code as presented in Figure 1, we must construct a

```
define program
  [repeat html_document_element]
end define

% Input is split into islands and water.
define html_document_element
  [interesting_element]
  | [uninteresting_element]
end define

define uninteresting_element
% Only basic structure is
% recovered in water elements.
  [html_tag_unknown]
  | [ampersand_symbol]
  | [html_tag_parameter]
  | [token_or_key]    % catch-all option
end define

% Islands are split into languages
define interesting_element
  [asp_code_block]    % Visual Basic
  | [html_script_tag] % JavaScript
  | [html_block]      % HTML
end define
```

Figure 2. TXL definition of a robust multilingual island grammar core.

grammar that is able to perform the following tasks. First of all, it must be able to differentiate between the three languages involved in the parse. This is comparatively easy—VB code is always contained between `<% %>` delimiters, and JavaScript is found predominantly in bodies of `<script>` tags. Anything outside these delimiters can be assumed to be HTML. Second, it must be able to handle intertwined code; the split-up `if` statement described in the paragraph above, for example, must be recognized as a single VB statement. Finally, it must be robust enough to selectively analyze HTML; this means being able to ignore uninteresting and malformed tags as well as text without breaking the parse, and were possible detect "interesting" tags despite minor syntactic errors they might have.

3.1 Grammar Core

Through creative use of island grammars, we are able to achieve all our goals. The main strength of island grammars is the ability to separate the text to be parsed in two general categories—interesting *islands*, and uninteresting *water*. Definitions making up both categories can then be enhanced and refined to achieve desired parsing results.

Figure 2 shows definitions that make up the core of the grammar, presented in TXL's[8] grammar definition syntax, which is very close to BNF-style definitions. The core is made up of just a few high-level non-terminal definitions that give a bird's eye view of how the grammar is structured. These definitions set up the foundations for separating islands from water, and identify three kinds of islands the grammar is interested in, namely code in Visual Basic, JavaScript, and HTML.

The target non-terminal `program` shows that all documents parsed are expected to be a sequences of zero or more productions of type `html_document_element`. The definitions of `html_document_element` and one of its children, the `uninteresting_element` that makes this grammar an island grammar. `Html_document_element` can be either one of `interesting_element` (i.e. an island) or `uninteresting_element` (i.e. water). TXL prioritizes productions in the order they are listed in the definition, so the priority of interesting islands is higher than that of water. Because of this, the TXL parser will always attempt to parse the input as an island. The water productions will only be resorted to if the input matches none of the island definitions included in the grammar.

Water Definitions. Water, the production of last resort, has some structure of its own. Of particular interest is the last, catch-all production; it is the presence of this production in water definition that gives the grammar its robustness. The catch-all `token_or_key` nonterminal, as its name suggests, matches any one token or key. Thus, it can match absolutely any single input lexeme. In case a particular part of the input cannot be matched by any of the island definitions, or any of the higher-priority water productions, it is matched to the `token_or_key` non-terminal, thus preventing the parse from breaking, and consuming one token/key of the input. The parse then resumes starting from the following token. Because `token_or_key` matches absolutely any input, it also has the lowest priority of any nonterminal in the grammar—it is the production of very last resort.

Other non-terminals that make up the definition of `unintersting_element` are there because they represent elements that, while uninteresting at present, are frequently present in HTML code and have easily identifiable structure that is worthwhile to recover. Since our grammar is currently interested in tables, forms, links and scripts, all tags not related to these HTML constructs are considered uninteresting and are treated as water. The `html_tag_unknown` and `html_tag_parameter` productions match HTML tags and parameters that are not at present interesting to the grammar and not included in island definitions. The `ampersand_symbol` is designed to match HTML-encoded characters, which provide a way to include in HTML characters that have special meaning. The $<$ and $>$ signs, for example, if included in an HTML page, would be internally specified as `<` and

```
define html_table_tag
  <table [repeat html_any_tag_parameter]>]
  [repeat html_table_content]
  [opt html_table_tag_closing]
end define

define html_table_content
  [html_legitimate_table_content]
  | [html_bad_table_content]
end define

define html_bad_table_content
  [not html_table_tag_stop]
    [html_document_element]
end define

define html_table_tag_stop
  </table
  | <table
  | [html_table_tag]
end define

define html_table_tag_closing
   </table>
end define

define html_legitimate_table_content
  [html_tr_tag]
  | [html_tfoot_tag]
  | [html_thead_tag]
  | [html_tbody_tag]
end define
```

Figure 3. Island grammar definition of the HTML table tag.

>, respectively, and would both be matched by the ampersand_symbol water nonterminal.

Island Definitions. The grammar core does not contain many specifics about islands—island definitions make up the bulk of the grammar and are discussed later in the paper. In the core, islands are established as a class of high-priority productions different from water. Furthermore, the core shows that there are three types of islands—asp_code_block islands, corresponding to passages of Visual Basic code, html_script_tag islands, which contain JavaScript code, and finally html_block islands, corresponding to HTML tags that the grammar finds interesting.

3.2 HTML Islands

Of all the languages involved in the parse, HTML presents the most challenges. First of all, HTML is not nearly as structured as other programming languages—arbitrary text can appear in many places in HTML, and the grammar has to take it into account. Even were it not so, the grammar is only interested in a small subset of HTML tags (chiefly tables and forms); all others have to be effectively ignored. The parse of HTML by necessity has to be robust. Furthermore, HTML is prone to containing syntactical errors in the mark-up; reasons for this are discussed by P. Brereton *et. al.*[9].

It is important to note that there are two degrees to the "badness" of HTML, i.e. all syntax errors encountered can be classified into two major categories. The "mild" errors are those that violate HTML syntax, but have no impact on the rendering of HTML by the browsers. An example of such an error would be markup like <i>bold italic</i>, which violates HTML syntax by closing the <b> tag before the <i> one. The vast majority of the browsers, however, would successfully render the text inside the tags as ***bold italic***, conveying the original intent despite the error. Severe errors are errors that make correct rendering of HTML by the browsers impossible. Trying to nest a table directly in another table is one example:

```
<table>
  <table>
   .......
  </table>
</table>
```

In this case, most browsers will be unable to discern the original intent—that the tables were meant to be nested. Most browsers will in fact consider the first table closed when they encounter the second one, so the tables will be consequent rather than nested.

In our efforts to parse HTML, we try to follow the behavior of browsers as much as possible. We consider structures with mild errors valid even though they are technically not. On the other hand, we don't attempt to detect structures with severe errors in them even if it is possible to do so—if we stray from the browsers' interpretation of HTML meaning, we would necessarily get erroneous results.

To achieve all these goals, the grammar again relies on its island structure. Figure 3 shows an island grammar definition of the HTML <table> tag. The definition has been slightly shortened for presentation purposes, but retains all the crucial features that enable robust parsing of <table> tags. Other tag definitions in the grammar largely conform to a similar pattern.

Unsurprisingly, the definition of a table tag to a certain extent mirrors the definitions presented above in the discussion of core grammar definitions since both are island-based. The table tag is defined as an entity that starts with the character sequence <table>, has parameters, and contains zero or more entities of type html_table_content.

The closing tag for the table is treated is optional, because an omitted closing tag is a very common problem in HTML markup. As shown below, the grammar does not rely on closing tags exclusively to determine where the table tag really ends.

As usual in island grammars, the content of the table tags is split into two categories—legitimate content, such as rows and table header/footer definitions, and "bad" content. Bad content can be arbitrary, because it can only be found in a table that is syntactically malformed. Therefore, `html_bad_table_content` is defined as `html_document_element`, which, as core definitions in Figure 2 show, can potentially match any input at all. Because legitimate content is given higher priority than the bad one, the grammar tries to evaluate everything found inside a table tag as legitimate content before resorting to treating is as "bad" content. However, because the content is found inside a table rather than at the top level of the document, the definition imposes certain restrictions—namely, that the document element cannot be of type `html_table_tag_stop`.

There are some HTML elements that can never be found inside a table. For example, another complete table can not be found directly inside another table—tables can be nested by placing one of them in the *cell* of the other, but not directly. Similarly, strings `<table>` and `</table>` can not occur in a table tag, because they would instantly either end the table, or start a new one. It is these items that are defined as `html_table_tag_stop`, and therefore effectively prohibited inside tables. If one of these is found inside a table during a parse, the parser will come to the conclusion that the table tag has already ended (and this was not detected previously perhaps the closing tag is missing or misspelled). Since the grammar relies on this information as well as on the closing `</table>` tags to understand where the table truly ends, that malformed tables with their closing tags missing and with malformed content inside them.

The grammar handles most other tags in a similar way. The approach described above enables us to achieve two important goals in HTML parsing. First, we do not have to rely on the oft-forgotten closing tags to determine where a tag truly ends; this enables us to detect tags even when their structure is incomplete. Second, and related achievement, is to be able to handle unexpected tag content—content that according to HTML syntax does not belong inside a tag. The second goal is achieved by effectively giving each interesting HTML tag a miniature island grammar of its own.

3.3 VB and JavaScript Islands

Visual Basic. While only a few select tags were processed in HTML, Visual Basic is interpreted in its entirety. This is a lot easier to achieve in VB than in HTML, since the grammar can expect Visual Basic to be well-formed. While browsers are quite lax in what HTML they accept and render, servers that execute VB code are as strict as any compiler, making robustness in processing VB code itself largely unnecessary.

One of the most interesting and challenging aspects in handling VB code in the context of ASP pages is the issue of code intermingling. Just as VB code can be inserted into HTML, so HTML can be inserted into VB code; what's more, the insertions can be arbitrarily nested. The nesting lets the code be very agile and flexible, but it also makes it difficult to parse. Figure 1 gives an example of such nesting, where passages of HTML are embedded into the middle of a VB `if` statement; the statement is used to decide which passage to show.

Even though the `if` statement is split into parts by HTML pieces, it must be treated as a single entity—a VB statement. It is, at least semantically, a parent node of the passages splitting it up. The question arises, then, what kind of entity these passages are.

The meaning of the HTML nested in VB code is equivalent to that of a print statement—it instructs the server to print the HTML in question to the output stream going to the browser. This is true for all static text in an ASP page—it evaluates to itself, and is equivalent to a print statement with itself as the argument. It is therefore possible to convert all HTML found in examples above to print statements, and thus simplify the parsing in the process considerably.

Such simplification, however, will severely reduce the amount of design information that can be extracted from the code. All structure of HTML that is found inside the Visual Basic statements will be lost, because they will be reduced to ordinary text strings—parameters of the print methods. The distinction between items that are generated by the Visual Basic and printed, and static items that were included as HTML, will be lost as well.

To allow for nested HTML and Visual Basic code without having to destroy the structure of either, the grammar allows HTML passages—so-called "snippets"—to appear inside Visual Basic code as a special Visual Basic statement. These snippets start with a "`%>`" marker and end with a "`<%`" marker, and can contain arbitrary HTML inside. This approach preserves all information that has been contained in the code—the language the code was written in, the nesting structure, and whether it was static or dynamic.

The snippets are restricted to appear only in sub-scopes, meaning they are only recognized inside constructs like loops, decision statements, and subroutines. If HTML snippets were allowed as elements at the top-most Visual Basic scope, the information on multiple Visual Basic passages will be lost, because HTML code is the only thing that can separate two VB passages. Without prohibition of snippets at the top level, the parse would detect at most one

```
define if_statement
  if [expn] then
    [sub_scope]
  else
    [sub_scope]
  end if
end define

define sub_scope
  [repeat sub_scope_content]
end define

define asp_sub_scope_content
  [asp_legitimate_element]
  | [asp_html_snippet]
end define

define asp_html_snippet
  %>
    [repeat snippet_content]
  <%
end define

define snippet_content
  [html_only_interesting_element]
  | [not asp_delimiter]
     [uninteresting_element]
end define
```

Figure 4. Grammar definitions for inclusion of HTML snippets in sub-scopes.

Visual Basic passage, going from the very first `<%` sign to the very last `%>` sign, interpreting any and all HTML found in-between as being embedded in the amalgamated Visual Basic passage.

Grammar definitions relevant to handling language intermingling appear in Figure 4. HTML snippets are described in the nonterminal `asp_html_snippet`. These snippets start with the closing ASP delimiter (`%>`), end with an opening asp delimiter (`<%`) and contain inside, as the definition for the nonterminal `snippet_content` shows, a sequence of HTML elements—either islands or water. The only restriction placed on the content of HTML snippets is that they can't contain ASP delimiters—to avoid the risk of missing one and parsing Visual Basic code as HTML. For reasons discussed above, HTML snippets are only allowed to appear in sub-scopes, which in turn only appear inside loops, decision statements, or subroutines. Figure 4 shows the `if` statement definition as an example.

JavaScript. When compared to HTML or Visual Basic, JavaScript presents relatively few challenges for parsing. The need for robustness is rather relaxed, because while several competing, and partially overlapping, implementations of JavaScript exist, the differences lie mostly in areas of functionality, not syntax.

JavaScript code is expected only inside HTML `<script>` tags. In keeping with the island parsing approach, inside these tags, the grammar expects to find either JavaScript code, or simple text. As usual, an attempt to parse input is code is made before falling back to the plain text option. The plain text option is nevertheless necessary, because HTML comments are often inserted inside JavaScript tags, either to provide comments or to hide JavaScript form browsers that are not JavaScript compliant.

4 Preprocessing

Despite the extensibility and flexibility that island grammars afford, the differences between languages involved are too great to easily resolve during parsing. The two obstacles to easy multilingual parsing that require lexical preprocessing to remove are comment formats and whitespace sensitivity.

Comments. The standard way of dealing with comments is to identify and remove them from the code before the parsing actually begins. The parser relies on predefined comment markers to identify and remove the comments. This scheme works well when only one language is in use; when two or more are parsed together, however, this approach fails, because what is a comment in one language is not a comment in the other. Therefore, if the comment markers of all languages are simultaneously provided to the parser as means for deciding what parts of input to ignore, there is a real possibility that some code pieces will be erroneously classified as comments and ignored. The standard way of dealing with comments is therefore not an option.

Our solution is to modify the standard approach—we rely on comment markers to find and remove comments before the parse, and assume that the comment markers stay the same across different languages. The assumption clearly does not hold in real life, so a lexical preprocessor performs a code transformation to force all languages to a single commenting convention.

There are three commenting conventions to address—HTML, Visual Basic, and JavaScript. The target commenting convention is as follows: anything that is contained inside the `<comment> ... </comment>` tags is considered a comment. The `<comment>` tag is actually a valid way to denote comments in HTML, but is not widely used. To coerce the code to this convention, all the code is scanned for comments, and any comments found are enclosed in the `<comment>` markers. The search is context-sensitive, so VB-style comments are only converted when found between the ASP code delimiters `<% ... %>`, JavaScript comments—only inside `<script>` tags, and HTML comments—everywhere else.

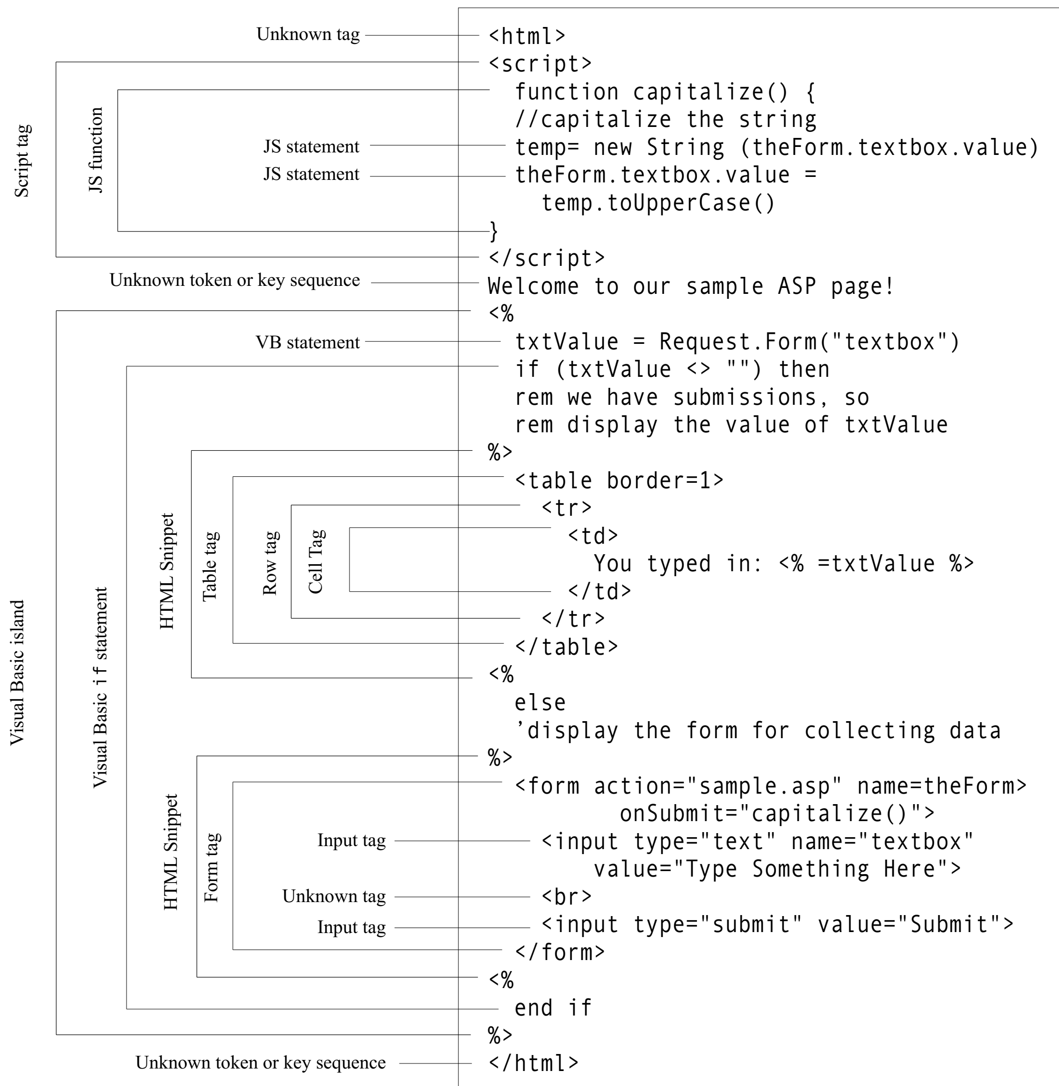

Figure 5. Multilingual parse of an ASP web page.

Whitespace. To reconcile differences between Visual Basic, which is whitespace-sensitive, and other languages in the parse, which are not, we perform a whitespace-insensitive parse, and inject additional information into the code to replace the information lost by ignoring the whitespace. VB relies on carriage return characters to separate its statements. Since information about locations of carriage returns is destroyed during a whitespace-insensitive parse, we insert a special new-line marker (`~~NL~MARKER~~`) at the end of every line. The grammar than can then rely on these markers, rather than on carriage returns to determine where statements begin and end.

5 Parsing Results

Figure 5 shows the results of parsing the original example text presented in Figure 1. The grammar was able to successfully process all the languages in the code, and coped with unstructured or uninteresting input well. It is not feasible to show a full parse tree, because parse trees for even relatively small examples, like the one shown, are too large to be visualized. Rather, the figure presents a selective, high-level view of the parse tree, and highlights the structure detected in the document by our island grammar.

Unstructured and uninteresting content. A number of items in the input were either of no interest, or did not have any structure, and had to be processed as water productions. These are the opening and closing `<html>` tags, the `<br>` tags inside the form, and all text content, such as the phrases "`Welcome to our ASP page!`" and "`You typed in:`". The structure that was available in these items was recovered—the `<html>` and `<br>` tags were recognized as generic HTML tags. With the help of the productions of last resort, text that has no discernible structure was classified as sequences of tokens.

Visual Basic. The grammar correctly identified the sole VB island in the code. The island contains two VB statements, of which the second one—the split `if` statement—is of particular interest. The `if` statement is identified as a single entity, containing in its `then` and `else` parts two HTML snippets, which are treated by the grammar as a special kind of VB statement. Inside the snippets, all HTML structure is preserved.

JavaScript. JavaScript code located in the `<script>` tag was detected successfully and parsed correctly. The grammar purposely does not look for JavaScript located in tag attributes, such as the function call in the `onSubmit` parameter of the `<form>` tag. These calls are instead classified as text strings. JavaScript code can occur only in a fixed set of HTML tag parameters (of which `onSubmit` is a member), if the need to analyze such code arises, the strings can be located and re-parsed as needed.

HTML. HTML is without a doubt the primary challenge to parsing in this example, and the grammar was able to handle HTML code well. It detected items of interest (form and table tags) in an input stream contaminated with unstructured and uninteresting items. Moreover, the grammar is robust enough to permit these "water" items inside the islands, an important trait when processing HTML tags like tables and forms, which are designed to hold other (and not necessarily interesting) items.

6 Related Work

Partial parsing for lightweight fact extraction has been proposed by M. Collard *et. al.*[13], who used XML tools to extract static information from C++ programs. A. Cox and C. Clarke[14] have proposed using iterative lexical analysis to process code that is syntactically malformed and thus difficult to parse.

A considerable amount of work has also been done in analyzing multilingual web documents. In particular, C. Boldyreff and R Kewish[10] have used an error-tolerant HTML parser for analysis and detection of clones in web pages. F. Ricca, P. Tonella and I. Baxter[11] have proposed a way to modify web pages and sites by rewrite rules, which search for specific patterns in web site code and substitute them for more desirable ones. A. Hassan and R. Holt[12] have proposed a framework for migrating web applications between different server-side languages using a series of transformation.

7 Conclusion

Parsing is the basic step necessary to enable automated comprehension, analysis and transformation of source code. Because they contain both multiple intermingled programming languages and frequent minor syntax errors, dynamic web pages pose a particular problem for parsers. In this paper we have described a method for extending the idea of island grammars to achieve robust multilingual parsing of mixed language programs, and have demonstrated how this idea can be used to simultaneously parse the multiple languages used in ASP dynamic web pages. The technique generalizes well and can be as easily applied to JSP and other dynamic web content paradigms.

References

[1] R. C. Holt, M. W. Godfrey, A. J. Malton, "The Build/Comprehend Pipelines", Proceedings of the Second ASERC Workshop on Software Architecture, February 2003.

[2] Leon Moonen, "Generating Robust Parsers Using Island Grammars", Proceedings of Eighth Working Conference On Reverse Engineering. October 2001.

[3] Leon Moonen, "Lightweight Impact Analysis Using Island Grammars", Proceedings of Tenth International Workshop On Program Comprehension. June 2002.

[4] T. R. Dean, J. R. Cordy, K. A. Schneider, A. J. Malton "Using Design Recovery Techniques to Transform Legacy Systems", Proceedings of IEEE International Conference on Software Maintenance (ICSM'01), November 2001.

[5] A. van Deursen, T. Kuipers, "Building Documentation Generators", Proceedings of International Conference on Software Maintenance, August-September 1999.

[6] John A. Carrol, An Island Parsing Interpreter For The Full Augmented Transition Network Formalism, Proceedings of First Conference of the European Chapter of the Association for Computer Linguistics, September 1983.

[7] M. L. Van De Vanter, "The Documentary Structure of Source Code", Information and Software Technology, October 2002, Volume 44, Issue 13, pp. 767–782.

[8] Cordy, J.R., Carmichael, I.H. and Halliay, R., "The TXL Programming Language - Version 10", Queen's University at Kingston and Legasys Corporation, Kingston, January 2000 (65 pp).

[9] Pearl Brereton, David Budgen and Geoff Hamilton, "Hypertext: The Next Maintenance Mountain", IEEE Computer, Vol. 31, No. 12. pp 49-55, 1998.

[10] Cornelia Boldyreff, Richard Kewish, "Reverse Engineering To Achieve Maintainable WWW Sites", Proceedings of Eighth Working Conference on Reverse Engineering, October 2001.

[11] Filippo Ricca, Paolo Tonella, Ira D. Baxter, "Restructuring Web Applications Via Transformation Rules", Proceedings of IEEE Workshop on Source Code Analysis and Manipulation, November 2001.

[12] Ahmed E. Hassand and Richard C. Holt, "Migrating Web Applications", Proceedings of the Second ASERC Workshop on Software Architecture, February 2003.

[13] M. L. Collard, H. H. Kagdi, J. I. Maletic, "An XML-Based Lightweight C++ Fact Extractor", Proceedings of the Eleventh International Workshop on Program Comprehension, May 2003.

[14] A. Cox, C. Clarke, "Syntactic Approximation Using Iterative Lexical Analysis", Proceedings of the Eleventh International Workshop on Program Comprehension, May 2003.