Agile Parsing Techniques for Web Applications

Thomas Dean¹, Mykyta Synytskyy²

¹Electrical and Computer Engineering, Queen’s University
dean@cs.queensu.ca
²Deptartment of Computing Science, University of Waterloo
nikita@mondenet.com

Abstract. Syntactic analysis lies at the heart of many transformation tools. Grammars are used to provide structure to transformation system. This paper gives an overview of agile parsing techniques, and how they may be applied to Web Applications.

1. Introduction

Many program transformation tools [3,4,6,7,8,21,25] are syntactic in nature. That is, they parse the input into an abstract syntax tree (AST) or graph (ASG) as part of the transformation process. There is good reason for this: syntax is the framework on which the semantics of most modern languages is defined. One example is denotational semantics [24], which maps the syntax to a formal mathematical domain. We use syntax to define the scoping of names, and to define the precedence of operators.

The syntax of the language, and thus the structure of the tree or graph used by these tools is determined by a grammar. The grammar is typically some form of context free grammar, although it might be augmented with other information [1,2]. In most cases, the transformation rules that may be applied to the input are constrained by the resulting data structure. This is the main purpose of grammars. Grammars exists to impose structure on the input that can be used by the rules. Otherwise, we would have little more than complex lexical patterns and transformations. As a consequence, a lot of thought must go into the authoring of the grammars used for transformation.

Unlike traditional compiler grammars, which are focussed on speed, error detection and error recovery, the grammars used in the transformation tools are closer to the language definition as understood by the application programmers. This makes it easier to author new grammars, easier to write transformation rules[22], and easier to adapt the grammars for language dialects and embedded languages [6,15].

These grammars are general grammars and represent the compromise of authoring the best grammars that are suitable for a wide variety of tasks. However, for some tasks, minor changes to the grammar can make significant differences in the performance of the transformations. We call changing the grammar on a program by program basis agile parsing [12]. One particular area of analysis that agile parsing is useful for is web applications. Web applications are written in a variety of languages, and are the subject of a variety of analysis and transformation tasks.
2. Agile Parsing

Agile parsing refers to using a grammar customized to the task. It is a collection of techniques from a variety of sources that we have found useful for transformation tasks. In this section, we give an introduction to agile parsing, how it is accomplished in the TXL programming language and a quick overview of some of the techniques.

The parsing techniques generally serve several needs. One is to change the way the input is parsed. An example is in the C language. In the C reference grammar the keyword typedef is simply given in the grammar as a storage specifier, eliminating any syntactic difference between variable and type declarations. For many transformation tasks, this may be a reasonable approach. But rules that process type declarations must determine the difference using conditions or guards. Splitting the grammar into two sets of productions, one which requires a typedef keyword and one that does not permit the typedef keyword lets the parser make the distinction. The rules become much simpler since they no longer have to code the distinction between the two declarations.

The second need is to modify the grammar to allow information to be stored within the tree. An example is to modify the grammar to add XML style markup to various non-terminals in the grammar. This can be used to store temporary information part way through the transformation, or can be used as a vehicle for communicating the results of an analysis to the user. The last need is to unify input and output grammars so that the rules may translate from one to the other while maintaining a consistent and well typed parse tree.

2.1 Agile Parsing in TXL

TXL is a pure functional programming language particularly designed to support rule-based source-to-source transformation [7,8,10]. Each TXL program has two parts: a structure specification of the input to be transformed, which is expressed as an unrestricted context-free grammar; and a set of one or more transformation rules, which are specified as pattern/replacement pairs to define what actions will be performed on the input. Each pattern/replacement pair in a transformation rule is specified by example, and may be arbitrarily parameterized for more rule flexibility.

Figure 1 shows a simple TXL grammar for expressions. Square brackets denote the use of a non-terminal. Prefixing a non-terminal symbol with the keyword repeat denotes a sequence of the non-terminal and the vertical bar is used to indicate alternate productions. Thus an expression is a term followed by zero or more occurrences of addop_term. The addop_term non terminal parses an add operator (a plus or minus operator followed by a term).

The terminal symbols in the grammar are either literal values, such as the operator symbols in Fig. 1, or general token classes which are also referenced using square brackets ([id] for identifier in Fig 1).

Agile parsing is supported in TXL through the use of the redefine and the not keyword. The redefine keyword is used to change a grammar. Figure 2 shows an example. In this example, the factor grammar production is changed to include XML markup on the identifier. The not keyword is used to prevent a production from matching and is described later.
The TXL processor first tokenizes the input using a standard set of tokens which may be extended by the grammar definition, parses the input using the grammar and then applies the rules. The rules are constrained by the grammar to keep the tree well formed. Thus the grammar includes not only the productions for parsing the input, but also the productions for the output and intermediate results. This may lead to ambiguities which is resolved through the use of an ordered parse. That is, the order of the rules in the grammar is used to determine the form of the input.

3. Agile Parsing in Web Applications

Web applications deliver services over the HTTP protocol, usually to a browser of some sort. Early web applications provided dynamic content to standard web browsers using server side scripting such as CGI scripts, servlets, JSP and ASP. Recent innovations include Real Simple Syndication (RSS), SOAP and AJAX. Web applications have multiple tiers. At one end you have the client software, usually a browser, but it may also be a desktop news ticker or a part of another application. In the middle you have a web or application server, and at the back end you have some information source, a database, for example.

Web applications present a particular challenge for parsing and for transformation. While some of the files in the web application contain conventional languages such as Java, the core files of the web application (the web pages) are typically comprised of a mixture of several languages. The web pages typically contain HTML, augmented with JavaScript, references to applets, and server side languages such as Java, Visual

```
define expression
[term] [repeat addop_term]
end define

define term
[factor] [repeat mulop_factor]
end define

define addop_term
[add_op] [term]
end define

define mulop_factor
[mul_op] [factor]
end define

define add_op
+ | -
end define

define factor
[id]
end define
```

**Figure 1. Simple Expression Grammar in TXL**

```
redefine factor
[XmlStart][id][XMLEnd]
end redefine

define XMLEnd
</[id]>
end define

define XMLStart
<![id] [repeat XMLParm]>
end define

define XMLParm
[id] = [stringlit]
end define
```

**Figure 2. Adding XML Markup to Identifiers in Expressions**
Basic, or PHP. Other components of the application may be in XMLs such as various configuration files, or the XML transfer schema such as that used in SOAP.

For some web transformation tasks, it may be possible to convert to a single language [11], but if it is desired to translate between client and server side technologies, we must be able to represent all of the languages in a single tree. A grammar for web applications must also be robust. That is, most browsers accept malformed HTML and do their best to render them and the tool must also accept and deal with the malformed HTML.

3.1 Island Grammars and Robust Parsing

Thus, the first agile parsing technique is island grammars [5,16,18,19,23]. In the context of web applications, island grammars allow us to do two things: find and parse application elements within the natural language on the web page and to handle erroneous input in a reasonable way. The first of these is that most of the page text is unimportant. Island grammars provide us with a mechanism to identify the interesting elements. However, what constitutes an interesting element depends on the task. In some contexts we are only interested in the Java or Visual Basic code that is embedded within the web page and the links from that code to other components such as other code modules or databases. In other contexts we may be interested in some of the structural components of the web pages such as tables, forms, anchors and links. In still other contexts, elements of the natural language may be interesting such as semantic web analysis.

The second use of the island grammars is to provide robustness. The general access by almost anyone to web publishing means that browsers have to deal with errors in the HTML markup in the pages. Two common errors are:

Style markup tags in web documents may not be nested. Browsers do not require that style markup such as bold, italic and other be nested. So the following, although technically illegal, is accepted: “<B><I>bold italic text</B></I>”.

Closing tags for some constructs may be missing. An example is the closing tags for tables, table elements and forms. In some cases, the tags are implicitly closed when a surrounding markup is closed (table rows closed by the table, nested tables closed by surrounding tables, tables closed by the end of the document.

Since browsers allow these errors, any analysis or transformation must also allow these errors and more importantly, interpret them in the same way as browsers.

We have developed an HTML island grammar that is the basis of our approach to analyzing and transforming web applications. Figure 3 shows the core of our HTML grammar. The predefined non-terminal program is the goal symbol of the grammar. In this case, it is a sequence (i.e. repeat) of document elements (html_document_element). Each document element is either an interesting element or an uninteresting element. Since TXL uses an ordered parse, only input that cannot be parsed as interesting elements can be parsed as uninteresting elements.
```xml
define program
 [repeat html_document_element] [html_interesting_element]
end define

define html_document_element
    [interesting_element]
    | [uninteresting_element]
end define

define uninteresting_element
    [html_uninteresting_element]
end define
```

Figure 3. Core HTML Grammar in TXL

The base grammar defines interesting elements as interesting html elements (the non-terminal `html_interesting_element`). In our case interesting elements include anchors (including links), tables and forms. The main purpose of the `interesting_element` non-terminal is to act as a extension point when adding other languages to the grammar.

Uninteresting elements use the grammar production `token_or_key` which is defined as any token or keyword. The `not` keyword is a guard. Before the `token_or_key` production is attempted, the `doc_guard` production is tried. If it succeeds in parsing the next elements in the input stream, the `uninteresting_element` production as a whole fails. Thus, the `doc_guard` production is used to prevent the `token_or_key` production from consuming any interesting tokens. It is not strictly necessary because of TXL’s ordered parse, but is useful during grammar development. In the `html_interesting_element` production also forms an extension point for other languages.

The `html_uninteresting_element` production called from uninteresting element is used to handle formatting issues. TXL uses a generalized unpars to write out the results of the transform. The `token_or_key` is occasionally inappropriate for HTML text. The `html_uninteresting_element` production allows the parser to recognize those cases and provide formatting cues in the grammar to handle them.

Figure 4 shows the grammar productions used to recognize the form elements (one of the interesting elements). The production `html_form_tag` (which is one of the alternatives of `html_interesting_element`) is rather straightforward. It is a form tag with parameters (`html_any_tag_param`) followed by form content and an optional end element. The end element must be optional since non-terminated forms are accepted by most browsers.

The content is either legitimate form content which includes form elements and other formatting elements such as tables. It also permits “bad” content which are `html_document_elements` (which is the top level of the gramar). The bad content is guarded by `html_form_tag_stop` which prevents any of the form tags, table close tags or table element tags from matching.

Similar approaches are used to parse tables and anchor elements as interesting elements of the grammar.

Extending the grammar to handle scripting languages (server or client side) is done by extending the `interesting_element` and `doc_guard` non-ter-
minals given in Figure 3. Figure 5 shows how the grammar can be extended to handle Java server pages. As mentioned before, the interesting_element and doc_guard productions are extended to include the JSP elements. The tokens jsp_start and jsp_end represent the "<%" and "%>" tokens respectively. So interesting elements are java expressions and scriptlets that are enclosed between jsp_start and jsp_end elements, jsp bean declaratives, declarations include directives, include actions and forward directives. The doc_guard grammar production is extended to include jsp_start and jsp_end elements to ensure that all java code is parsed as interesting elements.

The other side of the extension is the ability of java statements to include HTML. For example:

```html
<table><%
for(i = 0; i < 10; i++){
> <TR><TD> <%=i %> </TD></TR><%
} %>
</table>
```

In this case we have a Java for loop that generates the first 10 integers in a table. With the grammar additions shown in Figure 6, the end of the first scriptlet (containing the start of the for statement, will be treated as if it was a Java statement. The statement consists of HTML elements which in turn contain a nested java expression. The jsp_html_segment, is the reverse of a jsp_interesting_element. it starts with a jsp_end token and ends with a jsp_start token. The content is similar to the regular html content, but does not include scriptlets. opening another scriplet thus results in a statement at the same scope level as the html text.

One final wrinkle is that Java code may also be invoked within the attribute values of tags. They are usually enclosed in double quotes. This causes some difficulty in TXL since string literals are primitive tokens in TXL. We handle this with a simple lexical preprocessor that translates double quotes containing java scriptlets into double square brackets. The grammar for at-

```plaintext
define html_form_tag
[SPOFF] <form>[SP]
[repeat html_any_tag_parm][SPON]
[repeat html_form_content]
[opt html_form_tag_closing]
end define

define html_form_tag_closing
[SPOFF]</form>[SPON]
end define

define html_form_content
[html_licitimate_form_content]
end define

define html_form_bad_content
[not html_form_tag_stop]
[html_document_element]
end define

define html_form_tag_stop
</form]<form|</table|<tr|<td|<th
end define
```

**Figure 4. Grammar for Form Elements**
tributes in HTML tags is then extended to permit double square brackets as well as identifiers and string literals.

The same technique can be used to handle other web application technologies such as ASP or PHP. It can also be used to include the javascript grammar as well.

3.2 Markup Grammars

Another agile parsing technique we use for analyzing and transforming web applications is markup grammars. Markup grammars allow us to annotate elements for a variety of purposes. Sometimes the markup is used as part of a transformation. For example, the use of a variable may be marked up with unique identifier linking the declaration and all uses of the variable [13]. Markup can be used to store details about a transformation. Some transformations can be complex to identify, but rather simple to carry out once they have been completed. The complex identification problem can be broken down into multiple simple transforms, each of which analyses some part of the application and adds markup to encode the information that has been deduced. Subsequent transforms can combine markup to produce more sophisticated markup.

---

**Figure 5. Grammar for Form Elements**

```plaintext
redefine interesting_element  define jsp_interesting_element
  ...  [jsp_start]
  |[jsp_interesting_element]  [jsp_expression]
end redefine  [jsp_end]
redefine doc_guard  [repeat jsp_scriptlet]
  ...  [jsp_end]
  |[jsp_delimiter]  [jsp_useBean]
end redefine  [jsp_start]
define jsp_delimiter  [jsp_formal_declaration]
  [jsp_start]  [jsp_end]
  |[jsp_formal_declaration]
end define  [jsp_include_directive]
define jsp_element_or_html_seg  define jsp_html_segment_content
  [jsp_declaration_or_statement]  [jsp_useBean]
  |[jsp_expression]  [jsp_formal_declaration]
  |[jsp_html_segment]  [jsp_include_directive]
end define  [jsp_start]
define jsp_html_segment  [jsp_forward]
  [jsp_end]  [jsp_forward]
  |[jsp_include_action]
  [not jsp_delimiter]
define jsp_html_segment_content  [html_only_interesting_element]
  [jsp_start]  [html_element]
end define
end define
```

**Figure 6. Grammar for Form Elements**
Once the information has been identified and the appropriate elements annotated, then the final transformation is relatively straightforward.

A recent example is the transformation of JSP applications with scriplets into custom tags [26,27]. Each of the scriplets and each java variable is identified with a unique identifier. For example:

```xml
<tag id="Block2_Block1_Else">
  String[] <uid id="items ex.jsp Block2_Block1_Else">items</uid> = <uid id="cart ex.jsp Block0">cart</uid>.getItems();
</tag>
```

This information is supplemented with information the tags to which each statement will belong. The markup is used by the transform that removes the scriplets from the JSP pages, replaces them with custom markup and generates the java classes that implements the custom tags.

Markup can also be used to convey information back to the user. For example, a slice is the minimal subset of a program that can affect the slicing criteria[16]. The slicing criteria is the value of one or more variables at a particular point in the program. Markup can be used to show both the slice and how the slice was computed in the context of the larger program. Figure 7 shows an small example of computing slices in JSP applications using markup. The slicing criteria is identified using the slice tag. The set of variables active at each point is calculated by a transform and stored in the varset tag, and finally the elements of the slice are identified using backslice tag.

### 4. Conclusions

This paper has presented of the ways agile parsing can be used as a basis for analyzing and transforming web applications. Web applications provide a particular challenge to transformation since they often contain a multiple of languages, some of which are executed on the server side, while others are executed on the client side. We have examined two techniques in particular: the use of robust island grammars to uniformly represent all of the languages in a single parse and markup grammars which allow us to add annotations to web applications as part of various analysis and transformation tasks.

```xml
<varset set="rs, con, userName">
  <backslice distance=2>if(! rs.next())</backslice></varset>{
    <varset set="con, userName"><backslice distance=1>
      PreparedStatement stm = con.prepareStatement("INSERT INTO Employee" + "(Username,Password,FirstName,LastName) VALUES(?,?,?,?)");
    </backslice></varset>
    stm.setString(1, request.getParameter("userName"));
  </slice></varset>
```

**Figure 7:** Implementing Slicing in Web Applications Using Markup
References


