XML in a Web-based Grammar Development Environment

Eugene Koontz
151 Calderon Ave Apt #125
Mountain View CA 94041
ekoontz@hiro-tan.org

1 Introduction

Example-based Development of Grammars (EDG) is a natural language parsing and grammatical knowledge representation system, implemented in Common Lisp, that uses XML to display its output. The system parses sentences and allows querying and editing of its linguistic knowledge base. The system's response to user input is in XML, which is then transformed by an XSLT processor to form HTML, which is then displayed by the user's browser. The XML dialect used by the system to represent linguistic structure is described, together with details of how the XSLT stylesheet transforms it into HTML. Further information about the system, including an interactive demonstration of the system, may be found on the Web at `http://edg.sf.net`

1.1 Background

EDG is an implementation of Head Driven Phrase Structure Grammar (HPSG) as described in (Pollard and Sag, 1994). HPSG posits a uniform representation of syntactic structure and semantics using feature structures, which are sets of feature-value pairs. Feature structures are used to represent all linguistic knowledge, both lexical and grammatical, and these structures are organized hierarchically in a type hierarchy. Typed feature structures and the type hierarchy obey certain mathematical constraints (Carpenter, 1992). Below we discuss how these data structures are represented in XML form, and how the XML representations are transformed into HTML via XSLT.

1.2 User Interface

A web-based client-server grammar development model has a number of attractive features:

- It is cross-platform; ideally any client able to run a web browser should be able to use to the system.
- It allows multiple users to connect to the system simultaneously. Existing http mechanisms such as cookies can be used to save user state and allow multiple users to edit a single grammar and lexicon concurrently.
- HTML (in particular, the `<table>` tag) is well suited for representing complex data structures such as parse trees and typed feature structures. XSLT allows a straightforward mapping between the linguistic data (grammar, lexicon, etc) in XML, and their graphical representation in HTML. In section 4 we discuss this mapping in detail.

2 System Architecture

Figure 1 shows the flow of information through the system, sequentially ordered as follows. Using a web browser, a user sends information via http to a web server. The web server then invokes a CGI (Common Gateway Interface) script that translates the http request to a Lisp s-expression. This expression is passed over a socket to the core of the EDG system, the Linguistic Engine (LE), which is a Lisp program that does all linguistic processing. After receiving the s-expression from the web server's CGI script, the LE evaluates the expression according to its lexicon and grammar. The s-expression could be a request to parse a sentence, a query on the lexicon/grammar, or a command to update the lexicon/grammar, as described in detail in Section 5. The LE returns its evaluation in the form of an XML document to the CGI script, which then transforms this XML document according to an

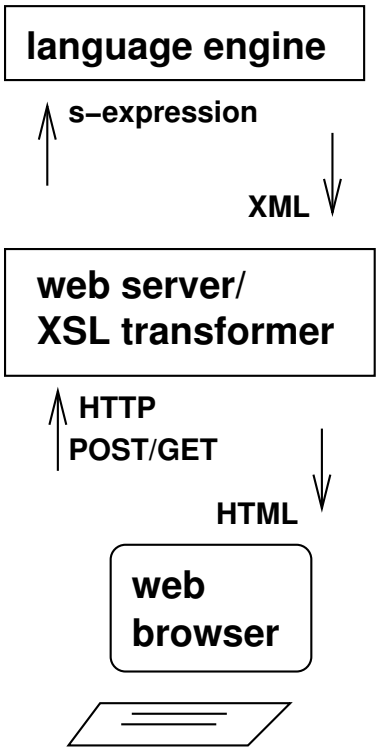

Figure 1: System Architecture

XSLT stylesheet, which transforms the XML into HTML. Finally this HTML is displayed on the client web browser.

3 XML and HTML encoding of Data Structures

We here describe how to represent data such as feature structures and parse trees. These data structures are used to represent the grammars, lexicons, and phrase structures of parsed phrases.

The LE returns representations of these structures as XML documents conforming to the DTD in Figure 2. Note that every XML document begins with the top-level element `response`. Note also the use of the `ID` DTD type for the `ref` attribute. `ID`s are used to represent indexed parts of an XML document. We use them to represent structure sharing in feature structures and to represent multiple inheritance in type hierarchies. A fragment of an XML document conforming to this DTD is given in Figure 3.

```
<!ELEMENT response (parse?,sub-hierarchy?)>
<!ELEMENT parse (agenda,chart)>
<!ELEMENT agenda (type*)>
<!ELEMENT chart (type*)>
<!ELEMENT sub-hierarchy (type,sub-hierarchy)>
<!ELEMENT type (object*)>
<!ELEMENT object (feature*)>
<!ATTLIST type name CDATA #REQUIRED>
<!ATTLIST type ref ID #IMPLIED>
<!ELEMENT feature (CDATA,object*)>
<!ATTLIST feature name CDATA #REQUIRED>
<!ATTLIST feature ref ID #IMPLIED>
```

Figure 2: DTD for Linguistic Engine output

```
<response>
  <sub-hierarchy>
    <type name="HIRO">
      <feature name="SYNSEM">
        <type name="T244">
        .
        .
        </type>
      </feature>
      <feature name="PHON">"hiro"</feature>
    </type>
  </subhierarchy>
</response>
```

Figure 3: XML output from Linguistic Engine

3.1 AVMs

Attribute Value Matrices (AVMs) are commonly used as a compact representation of feature structures. A LaTeXsty file called `avm.sty` (Manning, 1992) can be used to transform a textual encoding of an AVM into a typeset version suitable for printing. Similarly, we use an XSLT stylesheet to transform an XML encoding of an AVM, such as Figure 3, into HTML for viewing by a web browser. Figure 4 shows such an HTML-rendered AVM.

A feature structure may contain re-entrances, that is, values which are shared by more than one feature. Structure-sharing within a feature structure are indicated in the XML document using the `id` and `ref` attributes. The LE only elaborates the contents of a shared subgraph only once, marking it with a `id` attribute; the other occurrences of the subgraph are simply marked by an empty element with the corresponding `ref` attribute. The LE obeys the official XML specification that a given value of the

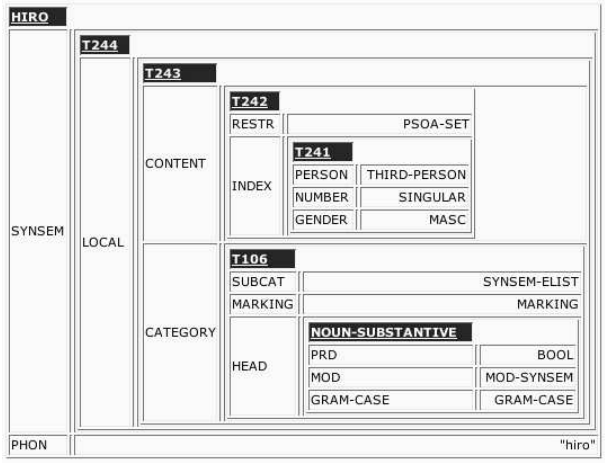

Figure 4: Attribute Value Matrix represented by nested HTML tables.

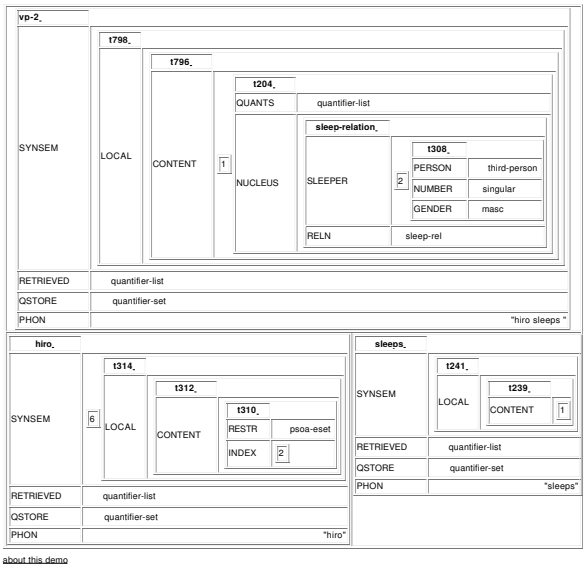

Figure 5: Parse tree for *hiro sleeps* represented by nested HTML tables

`id` attribute appears only once in a single XML document.

3.2 Phrase Structure Trees

HPSG considers syntactic structure to be just another type of information stored in a typed feature structure. Specifically, the daughters of a phrasal sign are pointed to by a feature typically called DAUGHTERS. The XML representation in the system is similarly organized. However, for the user's graphical display, this features is handled specially so that the phrasal structure is reflected in the tabular structure of the HTML. Figure 5 is an example of such an HTML-represented parse tree for the sentence *hiro sleeps*.

3.3 Type Hierarchies

Linguistic knowledge (both grammatical and lexical) in HPSG is represented by typed feature structures. The types are organized hierarchically, and multiple inheritance is allowed; that is, a single type having one or more possible supertypes. As with parse trees, we use nested HTML tables to represent these hierarchies, using co-indices to represent multiple inheritance. As with AVMs, the LE performs a depth-first traversal of the graph to find shared sub-graphs and indicates these with `id` and `ref` attributes. Figure 6 shows a small type hierarchy transformed from the LE's XML output to nested HTML tables.

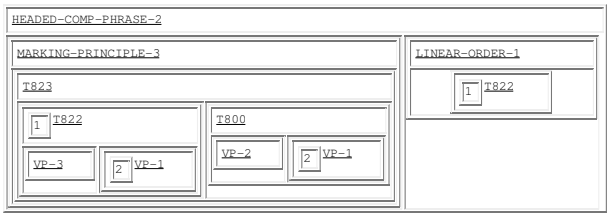

Figure 6: HTML tabular representation of a type hierarchy

4 Stylesheet Details

As mentioned above, an XSLT stylesheet is used to transform the system's XML representation of linguistic information into user-browsable HTML. Below we discuss how the stylesheet represents typed features structures and parse trees. This stylesheet is available, along with the rest of the source code for the EDG system, at the URL given in the Introduction.

4.1 Feature Structures

Representing a feature structure is done with two stylesheet templates, one for the feature structure object itself (the `object` template), and one for each feature-value pair in that object (the `feature` template). Both templates are shown below in Figures 7 and 8. These two templates are mutually recursive, that is, they

```
<xsl:template match="object">
  <xsl:when
       test="feature[@name='DAUGHTERS']">
  ...
  </xsl:when>
  <xsl:otherwise>
    <table border="1">
      <tr>
        <td>
          <xsl:value-of select="@type"/>
        </td>
      </tr>
      <xsl:apply-templates
           select="feature"/>
    </table>
  </xsl:otherwise>
</xsl:template>
```

Figure 7: XSLT template for the `object` element

call each other to recursively output the HTML for the feature structure.

First, consider the `object` template. (For now, ignore the test at the beginning; this will be discussed below in section 4.2.) This template (within the `xsl:otherwise` element) first creates a HTML table, and then writes out, as the table's first row, the type of the object. Next, it applies the `feature` template to every `feature` element in the object.

Now we turn to the `feature` template. Note that every feature element is wrapped in a `<tr>` (table row) HTML element. This corresponds to a separate table row within the `<table>` of the object in which this feature is contained. Each table row contains two cells : the first is the feature name; the second is the feature value. The second cell is populated by calling the `object` template on the `object` element within this feature. For feature values that are atomic (not containing features themselves), there are separate templates not shown here. Also omitted for brevity are the rendering of index tags.

4.2 Parse Trees

As mentioned above, HPSG uses a feature called DAUGHTERS to represent syntactic structure within feature structures, which is faithfully reproduced in the XML that the LE generates. The stylesheet is responsible for treating the DAUGHTERS feature specially so that a parse

```
<xsl:template match="feature">
  <tr>
    <td>
      <xsl:value-of select="@name"/>
    </td>
    <td>
      <xsl:apply-templates
           select="object"/>
    </td>
  </tr>
</xsl:template>
```

Figure 8: XSLT template for the `feature` element

tree is rendered in HTML tables. Figure 9 shows the processing of the DAUGHTERS feature within the object template, done by the `xsl:when` test that we ignored in the above section.

A parse tree is rendered as a table with two rows. The top row consists of a single cell containing the parse tree's mother. This row contains a single cell that spans (using the `colspan` HTML element) all of the cells in the bottom row. The bottom row contains the parse tree's daughters, one daughter per cell. The XSLT `count()` function is called to determine the number of daughters, and thus the needed `colspan` value for the mother's cell.

4.3 Hiding Excessive Data

One useful aspect of XSLT is that parameters can be passed to the XSLT transformation engine at run-time. This allows control over what aspects of the underlying XML should be displayed. This is important when the XML documents become so large as to be difficult to view as a single HTML page. A single AVM can contain so much information that its XML representation, when transformed into HTML becomes too large to conveniently view on a single web page : the user must scroll excessively to view the entire AVM. We compensate for this by providing a parameters to the stylesheet that specify names of AVM features to hide from the HTML output. The user can select which features to show or hide with a web form. Groups of related features, such as all features encoding semantic information can be grouped for the user's convenience and referred to as "semantics" on the user's web form.

```
<xsl:template match="object">
  <xsl:when
       test="feature
             [@name='DAUGHTERS']">
    <table border="1">
      <tr>
        <td align="center">
          <xsl:attribute name="colspan">
            <xsl:value-of
                 select="
 count(feature[@name='DAUGHTERS']/
       object//
       feature[@name='FIRST'])"/>
          </xsl:attribute>
          <xsl:value-of select="@type"/>
        </td>
      </tr>
      <tr>
        <xsl:apply-templates
             select=
             "feature[@name='DAUGHTERS']"/>
      </tr>
    </table>
  </xsl:when>
  .
  .
</xsl:template>
```

Figure 9: XSLT template for the `object` element, showing section for handling objects that are phrases.

The LE need not know about the details of the user's display preferences, however, and this enforces modularity by separating the underlying XML generation from its presentation to the user.

5 Input/Output Specification

As mentioned before, the LE is a program written in Lisp that listens on a TCP port and receives s-expressions. It then processes the expression and returns an XML document to the client. More details of the format of these input and output expressions are given in this section.

Each input s-expression is a function that the LE evaluates. The possible functions are given in the following table.

There are three input expressions that can be passed to the LE:

- `(parse` *quoted string*`)`. This causes the LE to parse the given quoted string and returns the resulting agenda (completely parsed constituents) and the chart (partially parsed constituents).
- `(sub-hier` *upper-bound lowerbound*`)`. The LE returns the set of types that is both more specific than the given upper-bound and more general than the given lower-bound.
- `(declare-type` *new-type supertypes subtypes*`)`. The LE updates the type hierarchy by creating a new type with the given name and situated in the type hierarchy according to the supplied supertypes and subtypes.

The following table summarizes the input-output relationships described in this section.

input s-exp.	*output elem.*
`(parse` *expr*`)`	`<parse>`
`(show-hier` *type-name*`)`	`<sub-hier>`
`(declare-type` *type-name*`)`	`<sub-hier>`

References

Bob Carpenter. 1992. *The Logic of Typed Feature Structures with Applications to Unification-based Grammars, Logic Programming and Constraint Resolution*, volume 32 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, New York.

Chris Manning. 1992. Documentation for avm.sty. http://www.essex.ac.uk/linguistics/clmt/latex4ling/avms/.

Carl J. Pollard and Ivan A. Sag. 1994. *Head-Driven Phrase Structure Grammar*. University of Chicago Press, Chicago.