Translating ISO 12083 Mathematical Markup for Electronic Documents

Keith Shafer, Roger Thompson

Abstract

In this paper, we describe a general translation tool that can transform tagged text into arbitrary output formats. Specifically, we describe how OCLC makes scientific documents containing mathematical markup available on the World Wide Web. The translation capabilities we developed to do this help realize the potential of the Standard Generalized Markup Language (SGML) to provide users with a single, non-proprietary document representation that can be translated on demand to other output formats. This enables publishers who target the WWW as a delivery medium to use the latest advances in HTML without constant revision of their document archives. ***Keywords:*** *Mathematical markup, translation, ISO 12083, entities*

Introduction

The Hypertext Markup Language (HTML) is a specification language for describing the display characteristics of documents in a browser-independent manner [1]. Because of its small number of tags, simple structure, and declarative nature, HTML provides a relatively easy-to-use way of making documents available on the Internet. Another advantage of HTML is that it supports active documents. Authors can encode interface features into a document that allow readers to make selections, provide textual information, and, most significantly, jump to other related documents. Other document standards such as Postscript [2] and TeX [3] are print oriented and thus are passive. Readers cannot interact with a document encoded in these standards unless a special interface is used that supports interaction independent of the document.

While HTML provides a simple, convenient means to "publish" active World Wide Web (WWW) documents, it is not suitable for the construction of archival document databases that will be the core of online (scholarly or otherwise) publishing. There are several reasons for this. One is that HTML is undergoing constant revision: its first major revision (V2.0) was just completed, and the second (V3.0) is under consideration. Furthermore, because HTML is so strongly output oriented, advances in output capabilities like those of Sun's HotJava WWW browser will cause further revisions of the markup [4]. As a result, authors of HTML documents typically will choose some combination of the features specified in the various versions of HTML to encode their documents. The choice is usually dependent on how well the author's browser of choice responds to particular HTML features.

Another reason for HTML's unsuitabilty for tagging archival documents is that it is primarily an output specification. Most tags are devoted to either describing various formatting features, linking the document to other documents, or providing various kinds of user interaction features. HTML contains only a few tags that outline a document's structure, and the minimal structure defined is there for the convenience of WWW browsers. A document's true structure is only hinted at by the different heading levels (tags H1 through H6), and it is left to the document's author to use these heading tags consistently. Because the structure is not directly specified and cannot be enforced by an SGML document

parser, the temptation to use heading or other tags inconsistently to achieve desired visual effects is always present. A good example of this is the HTML markup required for documents accepted to the WWW '95 conference. The abstract and keywords were not specified by structurally oriented tags like <abstract> or <keywords>. Instead, tags designed to format definition lists had to be used (<dl>, <dt>, and <dd>). All of these factors can lead to collections of documents in which the markup is inconsistent, potentially obsolete, oriented towards a particular software vendor's browser, and in need of constant maintenance.

A better way to store the information is to use markup that reflects abstract document structure using the Standard Generalized Markup Language (SGML) [5]. SGML is a meta-language for writing Document Type Definitions (DTD). A DTD describes how a conforming document should be marked up (i.e., the tags that may occur in the document, the ordering of the tags, and a host of other features). HTML is itself an SGML application with each of its three versions corresponding to a different DTD.

A single, well-crafted SGML DTD can explicitly and precisely specify the structure of a wide variety of documents. For example, a DTD can define tags for a very deep structural hierarchy with many section/subsection levels, and at the same time allow a document to be very shallow. DTDs can be difficult and time-consuming to create by hand, depending on how many features of SGML are used, but straightforward DTDs can be generated automatically [6]. Thus, the cost of developing them can be greatly reduced. With a DTD available, SGML parsers can be used to ensure that tagged text conforms to the structure defined by the DTD and is therefore consistently and correctly marked up.

After documents are in a consistent structurally-oriented markup, they can be translated into other formats on demand. For example, they can be transformed into files for loading into a relational database system, or they can be selectively indexed for building a text retrieval system, as well as be formatted for viewing. Several general translation tools are available, but most force users to use a predefined DTD (which may be difficult or impossible to do) or do not offer sufficient options to meet users' translation needs. For instance, while there is now an international standard for SGML mathematical markup, ISO 12083 Mathematics DTD [7], there are no systems that produce formatted documents from the complete standard.

At OCLC, we receive tagged text, including ISO 12083 mathematical markup, that must be translated to other formats to support OCLC's Electronic Journals Online (EJO) service [8]. This service provides online access to full-text scientific journals, so it must be able to handle all sorts of mathematics and other kinds of equations, such as those found in Chemistry or Physics literature. Guidon, OCLC's proprietary document viewer and retrieval interface, receives records from the database engine that have been translated to TeX's "DVI" format [9]. Guidon renders these records to produce the screen image, and, if desired, typeset-quality paper output.

To provide access to the EJO service via non-proprietary WWW browsers, these same source documents are also translated into HTML. One of the major difficulties in translating tagged text to HTML is that neither HTML version 1.0 nor 2.0 support the markup of mathematics. While HTML 3.0 has mathematical markup in it, it is not yet stable as a standard, and only one vendor's WWW browser currently handles it. To overcome this obstacle, we translate the mathematical markup to TeX which can then be rendered into GIF images. These GIF images are then used in the HTML versions of the documents. So for both Guidon and the WWW browsers we are required to translate mathematical markup to TeX. To handle these translation requirements, as well as others, we added translation capabilities to our Grammar-Builder Engine (*GB-Engine*) software.

The GB-Engine is a library of C++ objects that has been developed to support the *SGML Docu-*

ment Grammar Builder project [10]. This project is an ongoing research effort at OCLC studying the manipulation of tagged text. The GB-Engine can be used to automatically create reduced structural representations of tagged text (DTDs), translate tagged text, combine DTDs, automate database creation, and automate interface design—all from sample tagged text.

While the GB-Engine is embedded in a number of systems, *Fred* is the most popular. *Fred* is the GB-Engine embedded into the Tcl/Tk [11] environment. Tcl is a complete string-based interpreted programming language with variables, strings, lists, functions, etc.; Tk is an X-based graphical user interface toolkit. As a result, Fred is a complete interpreter/shell that has access to the GB-Engine objects and can be used easily to build X interfaces. We have also embedded the GB-engine into Perl [12] and Scheme [13], and ported the GB-Engine to Microsoft's NT operating system, so that it can be embedded into environments such as Microsoft's OLE.

In the remainder of this paper, we present requirements for mathematical markup translation, a discussion of the basic GB-Engine translation tool capabilities, an explanation about how those capabilities are used to include mathematical markup in HTML documents, and some translation examples.

Mathematical Markup Translation Requirements

In this section, we present the requirements for translating mathematical markup. Specifically, we look at the requirements for translating ISO 12083 to TeX, since this motivated the addition of translation capabilities to GB-Engine. While this would appear to be ISO 12083 or TeX-specific, we have found that these same requirements exist for many other kinds of translations. Thus, the reader need not be familiar with ISO 12083 or TeX to appreciate these general translation requirements. We merely use these requirements to make our discussion concrete.

One of our major observations is that the proper translation of tagged text is often *context dependent.* A system may have to determine where a particular tagged structure occurs within the structure of all the tagged text to know what to do with it. For example, one might have some text delimited by *author* tags. In the context of a title page the text would be handled one way, but in the context of a bibliography entry it would be handled in another.

The same can be said about translating mathematical markup. Some of the ISO 12083 structures have direct mappings to TeX control sequences. For instance, the tag *bold* maps directly to the TeX sequence *bf.* However, other ISO 12083 structures require that structure of the mathematical markup be examined in order to choose the appropriate TeX control sequence or combination of control sequences to produce correct formatting. There are three common contextual possibilities needed in the translation: *ancestor*, *descendant*, and *sibling.*

Text justification is a good example of the use of *ancestor* information. The justification of a *fraction* in the ISO 12083 mathematical standard can be specified in the *fraction* start tag as an attribute. In TeX, horizontal fill is generally used to manually justify text by placing space before or after the element to be justified. To translate the ISO 12083 *fraction*, horizontal fill must be generated in the TeX numerator or denominator sub-structures. To do this, the translation program must look "up" at the enclosing *fraction* structure for the value of its alignment attribute to know where to properly insert the horizontal fill. In some instances, the program may need to look even farther "up" into the enclosing mathematical markup to get the proper alignment, as it may be specified in a variety of places. (See the text justification example below for an example.)

Similarly, translation of the *radical* structure uses *descendant* information. TeX has two control sequences for radicals: one generates a simple square root and the other generates a general root with an explicit radix. To determine which

control sequence to use, one must count the number (there are only two possible) of immediate sub-structures of the ISO 12083 *radical* structure. If there is one sub-structure, indicating that there is no radix, the simple square root control sequence is selected. If there are two, the general root sequence is selected. (See the radical example below for an example.)

The generation of TeX array cell separators requires that *sibling* knowledge be used. In ISO 12083, every array cell is marked with a start tag and, usually, the cell is completely delimited by an end tag. TeX, on the other hand, marks only the separation of cells. This means that the translation program must be able to determine whether or not a cell is last in a list of cells (i.e., the cell has no right siblings). If it is the last, the translation program does not generate a separator. (See the array separator example below for an example.)

Translation in all of the previous situations involved simple substitutions or insertions of text. Some translations are more complex in that they require the placement of text in locations other than those where the tags occur. An example of this is the placement of superscripts and subscripts before an element. The ISO 12083 mathematical standard specifies that all of the superscripts and subscripts for an element follow the element. For example, an N with a leading superscript i and a trailing superscript j is encoded as: <subform>N</subform>ⁱ^j. The assignment of the value *pre* to the attribute *loc* specifies that the superscript i is to appear before the subform N. TeX encodes this whole structure as '$^iN^j$', so the ^i that corresponds to 'ⁱ' must be moved in front of the target subform, N, when the text is translated. (See the leading superscript example below for an example.)

One problematic requirement is with regard to the translation of arrays. The ISO 12083 DTD allows arrays to be marked up as a sequence of columns as well as a sequence of rows. TeX only allows them to be specified as rows. This means that the translation process must convert column order to row order, and at the same time preserve any justification information. Another problematic ISO 12083 structure is overlapping underlines and overlines. In ISO 12083 these are specified by reference *mark* tags that have an *id* attribute. These reference tags can be used by the *underline* and *overline* structures to determine where to start or finish. There is no corresponding TeX structure that directly encodes this.

The GB-Engine Translation Process

To meet these and related translation requirements at OCLC, we added translation capabilities to the GB-Engine. The GB-Engine translation capabilities provide a means for manipulation of tagged documents by translating, replacing, moving, or removing tags and their corresponding sub-structures. To accomplish this, GB-Engine translation requires three things:

- Tagged text to translate
- Translation script describing the desired transformation
- Optional entity translation table

We explain each of these parts in the following subsections. Examples will be presented in the Examples section below.

Tagged Text

The GB-Engine first processes the tagged text to construct a representation of its underlying structure. This is done by searching for start and end tags using traditional SGML syntax. These tags are matched to build a tree called a *tag structure* (or *document structure*). Once this structure is built, the translation capability can use it to determine the proper way to translate tags based on their context.

Translation Script

The GB-Engine translation is an interpreted process where the *translation script* is the user-supplied program of desired transformations. Every *translation script* is made up of *translation statements*. Each translation statement is composed of two parts, a *condition* and a block of *actions*:

```
if (condition)                { actions }
```

Translation *conditions* can be combined using the standard Boolean operators and can be parenthesized for grouping and readability. The conditions can test a tag in a variety of different ways, including whether it is a start or end tag, the presence or non-presence of attributes, the value of attributes, contextual location, as well as many of these same tests on ancestor, descendant, and sibling tags.

Translation *actions* can be nested and include sub-blocks of conditions and actions. Conditions are commonly enclosed in parentheses ()'s and action blocks are commonly enclosed in braces {}'s. Actions enable the translation to perform a wide variety of transformations ranging from simple textual substitution to reconfiguring the structure of a document. A more detailed description of the translation script syntax can be found in [14].

Given a well-tagged document structure and a translation script, the GB-Engine applies the *complete* translation script to each tag in the document structure in succession by performing a depth-first traversal of the document structure. (This tag traversal corresponds to the natural reading order of the document.) That is, *each tag* is checked against *each* statement condition in the translation script. If a statement condition evaluates to TRUE for a tag, the corresponding actions are applied to that tag. Thus, multiple translation statements may be applied to a single tag and a single translation statement may be applied to multiple tags.

The translation process has no effect on tags that have no conditions that evaluate to TRUE for them in the translation script. They are simply passed through into the output of the translation. Accordingly, a null translation script will reproduce the original document—the only difference being that some non-tagged white space will be removed. (Many people add white space like carriage returns, tabs, and spaces to tagged documents to make them easier to read. In most cases, this white space is *not* part of the document structure because it is *not tagged*. Since the translation process allows for text movement, we do not attempt to retain non-tagged white space in the translated text. For that matter, we have no way of knowing where the non-tagged white space should go and arbitrary insertion of such non-tagged white space may produce invalid translation results.)

Entity Translation Table

The entity translation table is used after the translation script has been applied to all of the tagged text. The table contains simple mappings of SGML entity references to arbitrary text strings. The standard syntax for an SGML entity reference is an ampersand "&" followed by a sequence of alphanumeric characters, followed by a semicolon ";". For example, the entity representing the capital Greek delta, "&Dgr;", is replaced by the TeX delta, "\Delta". In the radical example below the use of the entity translation table is shown. Entities can be handled in this way because they are designed to be a representation of special characters that are not contained in a standard character set.

Putting the Mathematics in HTML

Given the understanding of how the GB-Engine translation works, we can now describe how the ISO 12083 mathematical markup is included in HTML documents (also see [15]). First, the document is processed to build the tree-structured representation. The structured representation is then used to extract and save the mathematical markup, which is delimited by *formula* tags for inline mathematics, or by *dformula* or *dformgrp*

tags for display mathematics. These separate pieces are each passed through a Fred translation script for mathematical markup, resulting in a TeX translation for each piece. The TeX is then used to generate a DVI file, and the DVI file is rendered into a GIF image. Finally, a pointer to the GIF image is placed in the HTML document. When the document is loaded by a WWW browser, the image is brought along with it and displayed in the appropriate place.

Examples

Having presented the general GB-Engine translation process, we can now show how the GB-Engine handles the translation problems presented in the requirements section above. The sample tagged text, translation script, and resultant translation all appear immediately before the discussion of each example. Note that the line numbers in the examples are included for reference only and are not part of the actual syntax.

Example 1 shows how the GB-Engine can use ancestor information to generate proper text justification.

Example 1 shows how multiple conditions are met and applied to a tag during translation. The condition Start_Tag on line 1 of the script matches the *fraction* tag on line 1 of the sample text because the tag has the traditional SGML syntax for a start tag. In this case, the *fraction* start tag also has an attribute value assignment of "left" to "align". This assignment is not used in the translation of this tag, but is important later. The action "Literal" simply puts whatever is in its parentheses into the developing translation. If whitespace is desired, then the output must be enclosed in quotes. In this case, nothing is put into the translation, so the *fraction* start tag is "consumed." This will also be true for the *fraction* end tag as well. This is done by line 2 of the script.

Example 1: Text Justification

SAMPLE TAGGED TEXT:

```
1  <fraction align=left>
2     <num>1</num>
3     <den>ax + b</den>
4  </fraction>
```

TRANSLATION SCRIPT:

```
1    if Start_Tag (fraction)                                  { Literal( ) }
2    if   End_Tag (fraction)                                  { Literal( ) }

3    if Start_Tag (num)                                       { Literal( { ) }
4    if Start_Tag (num) && Match_Parent (align,right)         { Literal ("\hfill ") }

5    if   End_Tag (num) && Match_Parent (align,left)          { Literal (" \hfill")  }
6    if   End_Tag (num)                                       { Literal ("}\over")  }

7    if Start_Tag (den)                                       { Literal ( { )  }
8    if Start_Tag (den) && Match_Parent (align,right)         { Literal ("\hfill ")  }

 9   if   End_Tag (den) && Match_Parent (align,left)          { Literal (" \hfill") }
10   if   End_Tag (den)                                        { Literal ( } )  }
```

TRANSLATION OUTPUT:

```
{1 \hfill}\over {ax + b \hfill}
```

When the script is applied to the *num* start tag, the Start_Tag condition on line 3 of the script evaluates to true, and so the action "Literal" generates an opening brace to enclose the numerator. Line 4 succeeds on the Start_Tag condition, but fails on the "Match_Parent" condition. This condition checks attribute/value pair assignments for a node's immediate ancestor. In this case, *num*'s immediate ancestor is *fraction*, and has the value *left* and not *right* for its *align* attribute.

The next tag processed is the *num* end tag. This tag matches the conditions on both lines 5 and 6. End_Tag is true if a tag has the standard syntax of an SGML end tag. The result is that on line 5 the horizontal fill is generated and then, on line 6, the enclosing brace along with the TeX "over" control sequence is generated. The *den* start and end tags on line 3 of the sample tagged text are processed in the same way by line 7 through 10 of the script.

While the condition Match_Parent restricts the context search to a tag's immediate ancestor, there are a variety of other conditions for looking both up and down beyond the immediate context to find occurrences of specific tags, attributes, and attribute values.

Example 2 shows how a translation script can use descendant information. .

In the ISO 12083 Mathematics DTD the *radical* can have only one or two sub-structures, since the *radix* structure is optional and the *radicand* is required. If it has none or more than two, the markup is not valid. This constraint is encoded in the use of the "Child_Count" condition.

The first tag processed by this script is the *radical* start tag. Line 1 of the script checks to see if it is a start tag (true), and if it has only one immediate substructure (false). This line generates nothing since the whole condition part failed. Line 2 also checks to see if it is a start tag, and if the tag has two immediate substructures, which it does. The result of this line is that a TeX "\root" command is generated. The rest of the script is straight forward, processing the *radix* and *radi-*

Example 2: Radical Example

```
SAMPLE TAGGED TEXT:
1  <radical>
2     <radix>3</radix>
3     <radicand>&Dgr;</radicand>
4  </radical>

TRANSLATION SCRIPT:
1    if Start_Tag (radical) && Child_Count (1)          { Literal (\sqrt) }
2    if Start_Tag (radical) && Child_Count (2)          { Literal (\root) }
3    if   End_Tag (radical)                             { Literal ( ) }

4    if Start_Tag (radix)                               { Literal ( { ) }
5    if   End_Tag (radix)                               { Literal ("}\of") }

6    if Start_Tag (radicand)                            { Literal ( { )  }
7    if   End_Tag (radicand)                            { Literal ( } )  }

ENTITY TRANSLATION TABLE:
"Dgr"                                "\Delta "

TRANSLATION OUTPUT:
  \root{3}\of {\Delta }
```

Example 3: Array Separator

SAMPLE TAGGED TEXT:

```
1  <array>
2     <arrayrow>
3        <arraycel> A </arraycel> <arraycel> B </arraycel>
4     </arrayrow>
5     <arrayrow>
6        <arraycel> C </arraycel> <arraycel> D </arraycel>
7     </arrayrow>
8  </array>
```

TRANSLATION SCRIPT:

```
1    if Start_Tag (array)                          { Literal ("\matrix{ ") }
2    if   End_Tag (array)                          { Literal ( } )  }

3    if Start_Tag (arrayrow)                       { Literal ( )  }
4    if   End_Tag (arrayrow)                       { Literal (" \cr ") }

5    if Start_Tag (arraycel)                       { Literal ( )  }
6    if   End_Tag (arraycel) &&  Right_Peer        { Literal (" & ")  }
7    if   End_Tag (arraycel) && !Right_Peer        { Literal ( )  }
```

TRANSLATION OUTPUT:

```
\matrix{ A  &  B  \cr  C  &  D  \cr }
```

cand tags. This example also demonstrates entity sub-stitution.

Example 3 shows how a script can determine if a tag is the last in a sequence.

The sample tagged text encodes a simple 2x2 array. Lines 1 and 2 of the script handle the *array* start and end tags and generate respectively the TeX matrix control sequence, and an enclosing brace for it. Lines 3 and 4 handle the *arrayrow* tags. In this case the start tag is consumed and the end tag is translated to a row terminator. Line 5 consumes the *arraycel* start tag. Lines 6 and 7 check the *arraycel* end tag to see if it does or does not have a right peer in the document structure. If it does, a TeX array cell separator is put into the translation; if not, the tag is consumed.

The three previous examples have all shown translation occuring right where the tag occurs in the document text. Example 4 shows that, in some cases, proper translation requires text to be inserted in a place other than where the tag actually occurs.

Example 4: Leading Superscript

SAMPLE TAGGED TEXT:

```
1  <subform> N </subform>
2  <sup loc=pre> i </sup>
3  <sup> j </sup>
```

TRANSLATION SCRIPT:

```
1    if Start_Tag (subform)                        { Literal ( { )  }
2    if   End_Tag (subform)                        { Literal ( } )  }

3    if Start_Tag (sup)                            { Literal (^{ )  }
```

Example 4: Leading Superscript (Continued)

```
4   if Start_Tag (sup) && Match (loc,post)          { Literal ( )  }
5   if Start_Tag (sup) && Match (loc,pre)           { Move_Relative_Left }
6   if   End_Tag (sup)                              { Literal ( } )  }
```

TRANSLATION OUTPUT:

```
^{ i }{ N }^{ j }
```

Example 4 is the solution to the leading superscript problem presented in the requirements section above. Text "movement" actions do not alter the input text and its underlying structure. As translation is performed, an output structure is constructed that may be freely restructured by the translation script.

The *subform* start and end tags are handled by lines 1 and 2, and generate the enclosing braces. Next, the first *sup* start tag is translated by line 3, which generates a TeX superscript command and a brace to enclose any items that will be superscripted. Lines 4 and 5 check the value of the *loc* attribute. Since *loc* has the value *pre*, the translation of the superscript structure is moved to the left of the immediately preceding sibling tagged structure; the *subform* structure. The *sup* end tag is translated by line 6 of the script, and a closing brace is generated. Line 3 of the text is processed in the same manner except that it is not moved, since it has no *loc* attribute.

In summary, we have shown some specific examples of how the GB-Engine translation tool capability meets the requirements imposed by the task of translating ISO 12083 mathematical markup to TeX. These examples by no means show all the capabilities of the translation tool. There are nearly 40 conditions [18] to examine various properties of the tags and tree structure and nearly 70 processes to format and alter the structure of the output. In addition, function callbacks provide access to the outer programming environment enabling arbitrarily complex transformations. Translation to other formats is possible by simply using different scripts.

Conclusion

In this paper, we have described how the GB-Engine translation capability provides a means whereby richly tagged documents can be transformed into other arbitrary formats. As a result, SGML is made more attractive as the underlying representation for archival document storage. This allows publishers who target the WWW as a delivery medium to take advantage of developments in HTML without having to constantly revise their document archives. GB-Engine translation also shows how some of the capabilities of advanced style sheet languages such as those suggested by Sperberg-McQueen [16] can be implemented.

It is interesting to note that this paper was itself written as tagged text using GB-Engine via Fred to simultaneously translate the single tagged source to ASCII, HTML, and TeX (PostScript). GB-Engine translation services are freely available via a WWW Fred server [6]. ■

References

1. T. Berners-Lee and D. Connolly, *Hypertext Markup Language—2.0*, 1995. Accessible at *http://www.w3.org/hypertext/WWW/MarkUp/html-spec/html-spec_toc.html*
2. Adobe Systems Incorporated, *PostScript Language Reference Manual*, Addison-Wesley Publishing Company, Reading, MA, 1985.
3. Donald E. Knuth, *The TeXbook*, Addison-Wesley Publishing Company, Reading, MA, 1984.
4. Sun Microsystems, *HotJava Home Page*, 1995. Accessible at *http://java.sun.com/*
5. *Information Processing—Text and Office Systems—Standard Generalized Markup Language (SGML)*, International Organization for Standardization, Ref. No. ISO 8879:1986, 1986.

6. Keith Shafer, *Fred: The SGML Grammar Builder*, Fred's WWW home page, 1994. Accessible at *http://www.oclc.org/fred/*

7. *Electronic Manuscript Preparation and Markup*, ANSI/NISO/ISO 12083, 1994.

8. Andrea Keyhani, *The Online Journal of Current Clinical Trials: An Innovation in Electronic Journal Publishing*, *Database*, February 1993, pages 14-23.

9. Donald E. Knuth, *TeXWare*, Dept. of Computer Science, Stanford UniversityTechnical Report STAN-CS-89-1097, 1986.

10. Keith Shafer, *SGML Grammar Structure*, *Annual Review of OCLC Research July 1992-June 1993*, pages 39-40, 1994.

11. John K. Ousterhout, *Tcl and the Tk Toolkit*, Addison-Wesley Publishing Company, Reading, MA, 1994.

12. Larry Wall and Randal L. Schwartz, *Programming Perl*, O'Reilly & Associates, Inc., Sebastopol, CA, 1992.

13. Harold Abelson, Gerald Jay Sussman, and Julie Abelson, *Structure and Interpretation of Computer Programs*, The MIT Press, Cambridge, MA, 1985.

14. Keith Shafer and Roger Thompson, *Introduction to Translating Tagged Text via the SGML Document Grammar Builder Engine*, 1994. Accessible at *http://www.oclc.org:80/fred/docs/translations/intro.html*

15. Stuart Weibel, Eric Miller, Ralph LeVan, and Jean Godby, *An Architecture for Scholarly Publishing on the World Wide Web, Proceedings from the Second International WWW Conference: Mosaic and the Web*, 1994. Accessible at *http://www.oclc.org:5046/publications/weibel/web_pub_arch/*, pages 739-748.

16. C.M. Sperberg-McQueen and Robert F. Goldstein, "HTML to the Max: A Manifesto for adding SGML Intelligence to the World Wide Web," *World Wide Web Fall 1994 Papers*. Accessible at *http://www.ncsa.uiuc.edu/SDG/IT94/Proceedings/Autools/sperberg-mcqueen/sperberg.html*

17. Diane Vizine-Goetz, Jean Godby, and Mark Bendig, "Spectrum: A Web-Based Tool for Describing Electronic Resources," presented at the *Third International World Wide Web Conference*. Darmstadt, Germany, 1995.

18. Keith Shafer, *Quick Translation Reference for Fred*, 1994. Accessible at *http://www.oclc.org/fred/docs/help/quick.html*

19. Keith Shafer, *Fred Translation Information*, 1994. Fred's WWW translation home page. Accessible at *http://www.oclc.org/fred/docs/translations/*

20. Thomas B. Hickey and Terry Noreault, "The Development of a Graphical User Interface for The Online Journal of Current Clinical Trials," *The Public-Access Computer Systems Review*, 3(2):4-12, 1992.

21. Thomas B. Hickey, "Reference Client Software Design," *Annual Review of OCLC Research July 1992-June 1993*, pages 37-39, 1994.

About the Authors

Keith Shafer
[*http://www.oclc.org:5046/~shafer/*]
shafer@oclc.org

Roger Thompson
thompson@oclc.org

OCLC Online Computer Library Center, Inc.
6565 Frantz Road, Dublin, Ohio 43017-3395
FAX: (614) 764-6096