ELECTRONIC PUBLISHING, VOL. 4(2), 63–85 (JUNE 1991)

# A model and toolset for the uniform tagging of encoded documents

JULIE A. BARNES
*Department of Computer Science*
*Bowling Green State University*
*Bowling Green, OH 43403-0214, USA*

SANDRA A. MAMRAK
*Department of Computer and Information Science*
*The Ohio State University*
*Columbus, OH 43210-1277, USA*

**SUMMARY**

**In this paper we present a new, abstract model for textual data objects with embedded markup. Based on the model, we propose a uniform representation for these objects that borrows its concrete syntax from the ISO standard SGML. Such a uniform representation will greatly facilitate the development of software that analyzes, formats or otherwise processes these objects. We then describe a toolset that supports the retagging of existing encoded data objects to the new uniform representation. Our experience with the toolset demonstrates a savings of approximately 10:1 over a retagging effort without the toolset.**

KEY WORDS Data translation Lexical analysis Automatic code generation

## 1 INTRODUCTION

Digital library systems will be implemented in the next decade [1]. These systems of geographically distributed users and organizations, each with its own digital library containing information of both local and widespread interest, will make available large collections of electronic documents to a diverse group of users and applications. For example, a manuscript in a digital library might be accessed by text formatters, concordance packages, and linguistic-analysis packages. An ideal model of accessing a library would be similar to a client-server model. A common set of *library-access* functions would permit an application to retrieve or modify the data in the library. The application would be restricted to the *application-dependent* functions required to do the desired processing and the library would be accessed by calling a library-access function. This ideal model is depicted in Figure 1.

A prime example of this model is the X Window System [2]. Although the resource to be accessed is not a library, X provides a uniform view of the functionality required to build a window-type interface, independent of the underlying hardware. The common access functions are provided in a variety of forms such as a base function library (Xlib), toolkits of more sophisticated functions (Xt), and predefined window classes (widgets). In writing an application a programmer can concentrate on the application-dependent functions. The application is created by calling the common access functions without becoming involved in the low-level details of creating a window system.

A second example is the ISO Standard Generalized Markup Language [3] (SGML) for electronic documents. SGML provides a method for defining document types and declaring their markup. Documents in an SGML-encoded library can be accessed with a typical SGML parser. Such a parser provides the common library-access functions, but

0894–3982/91/020063–23$11.50
©1991 by John Wiley & Sons, Ltd.
© 1998 by University of Nottingham.
*Received 13 August 1990*
*Revised 28 March 1991*

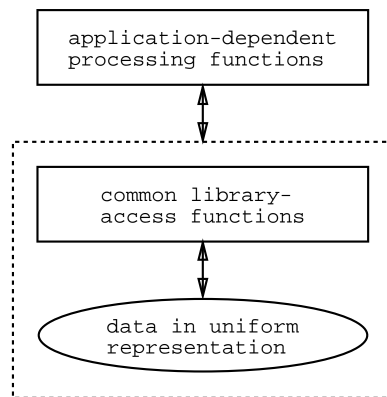


*Figure 1. Ideal library-access model*

does not process the document. Applications to perform services such as text formatting, validation, or linguistic analysis are written independently, using calls to the parser to access the components of the document.

There are many advantages to this ideal model. It eliminates the duplication of effort in the development of an application, because the library-access functions already exist. Also, it permits a single version of the library-access functions to be used by all applications that access the library.

In the electronic-document domain, the current situation is far from this ideal model, despite the existence of SGML and other document standards. A large number of nonuniform representations currently exist in the domain of electronic documents. There are different encoding schemes for specialized humanities document collections [4–7], corpora [8,9], and dictionaries [10]. There are assorted text-formatting languages [11–13]. To complicate matters, concordance-building programs [14,15] permit the users to define their own restricted encoding scheme. Applications usually contain both the application-dependent functions and the necessary library-access functions (see Figure 2). In this scenario, if an application has to access more than one data representation, a different set

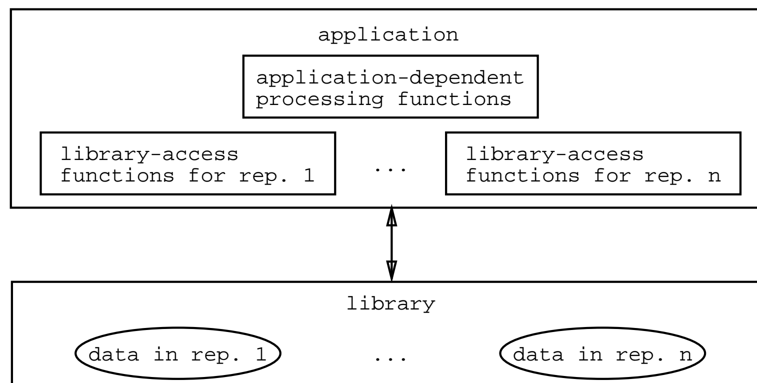


*Figure 2. Current access model*

of library-access functions has to be written for each representation. Thus, there is much duplication of effort within an application to access the different representations. There is duplication of effort among applications as well, since the same access code must be written for each.

A step toward approaching the ideal model in the electronic-document domain is to define a uniform representation of the data. The characteristics of this representation are dictated by the needs of application packages. In Section 2 we describe the characteristics of a uniform representation and provide an abstract and a sample concrete syntax for this representation.

Once such a uniform representation is in place, nonconforming document representations must be converted, or *retagged*, into the desired representation. An essential component of the retagging process is the lexical analysis of an encoded document. With the availability of code-generation tools like Lex [16], most computer scientists consider the problem of lexical analysis essentially solved. This may be true for programming languages, but it is not true for other encoded data objects. In Section 3 we describe the current model of lexical analysis and why it is inadequate for data objects with embedded commands. In Section 4 we give specific examples of how difficult the automatic generation of code is with current code-generation tools. In Section 5 we briefly review other code-generation tools relevant to retagging.

Based on our analysis of the inadequacy of existing retagging toolsets, we designed and implemented a new toolset described in Section 6. In Section 7 we summarize our experiences in using the toolset to generate retagging programs for six different document encoding schemes. These encoding schemes represent a broad spectrum of electronic-document encoding schemes in terms of applications and levels of complexity. We demonstrate savings in coding effort ranging from a factor of 4.3 to a factor of 23.2, measured in terms of lines of code generated for each line of high-level specification. We also demonstrate that with the toolset, the user must provide only two percent of the C source code required to do the retagging without the toolset. In Section 8 we describe some of the practical applications of the uniform representation. We summarize the contributions and future directions of the work presented in this paper in Section 9.

## 2 A UNIFORM REPRESENTATION FOR ELECTRONIC DOCUMENTS

All applications in the electronic-document domain minimally must recognize the significant *tokens* or character strings in an electronic document. The two primary classes of such strings in documents are the markup tokens and the content tokens. For example, in a document database for text-formatters, a markup token might be "<author>" and a content token "Julie Barnes".

An application needs to separate the markup from the content tokens. A word-frequency application, for example, needs to recognize and separate the markup tokens from the content tokens to build its table of words and counts. Similarly, a translation application needs to recognize the significant tokens before it can rename and possibly reorder them for a different target representation. The Chameleon translation project has reported [17] that the primary hindrance to automatically generating translation code was the inability to automatically generate tokenizers for representations to be translated.

The process by which input data is scanned for significant clusters of characters is called *lexical analysis*. Because lexical analysis is the fundamental access function for

all applications in the electronic-document domain, the uniform representation should be *lexically* based, i.e., not based on some other properties of documents like their structure or layout features. For this reason, we call our proposed representation a *lexical intermediate form* (LIF).

In this section we present a detailed model of the token classes that exist in electronic documents and that, therefore, have to be recognized in their lexical analysis. A description of each class is given along with examples taken from the *Thesaurus Linguae Graecae* (*TLG*) [7]. Based on this model we derive the required abstract syntax of a LIF and give one example of a possible concrete syntax.

### 2.1 Token classes in encoding schemes

The coarsest partition of token classes in a document encoding scheme exploits the distinction between the markup tokens, or *tags*, and the text that constitute the document (see Figure 3). Tags may indicate the structure of the document, processing to be done by an application, or some other characteristic of a portion of text.

The major tag classes are based on the relationship of a tag with the surrounding text. Tags that break the text into smaller segments are called *segmenting tags*. Segmenting tags indicate some structural or physical property of the text segment, such as page divisions in a document or that a string of text is to be italicized.

There are two methods in which segmenting tags can mark a portion of text. When both the start and the end of the text segment are explicitly marked by tags, these tags are called *explicit segmenting tags*. For example, '{1' marks the start and '}1' marks the end of a title in the *TLG*. Explicit segmenting tags always occur as start-tag/end-tag pairs.

In other cases only the start or the end of the string, but not both, are explicitly marked by a tag. In these instances the segments are usually *contiguous*, meaning that the end of such a segment implies the start of the next similar segment. These tags are called *implicit segmenting tags* because either the start or the end of the segment is implied by another tag. Examples of this type of tag are '$' to indicate the start of a text segment in normal Greek font and '@1' to indicate the end of a page in a manuscript. In both cases no mate tag exists to indicate the corresponding end or start of the designated segment.

Tags which do not segment the text are *nonsegmenting tags*. These tags can be viewed

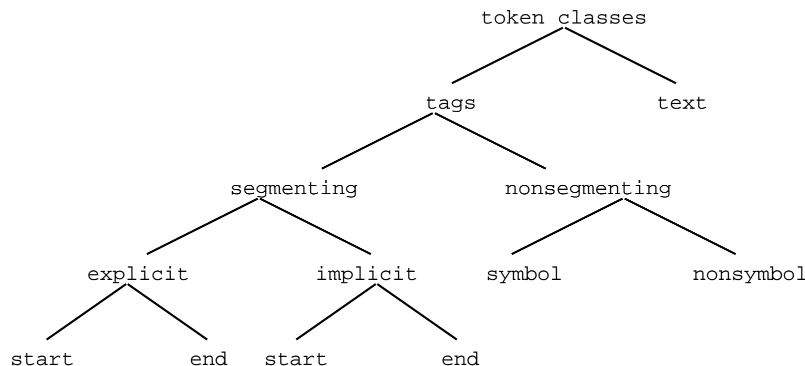


*Figure 3. Partitions of the token classes in document encoding schemes*

as insertions in the text. These tags do not encapsulate a text segment or attach any meaning to a segment like segmenting tags.

A major subclass of the nonsegmenting tags is the *symbol tags*. These tags represent characters or symbols in the document that do not occur on the keyboard of the input device and include the representations of any keyboard symbols whose use is restricted to markup. In the *TLG*, '%' represents the dagger character, †.

The other subclass of nonsegmenting tags is *nonsymbol tags*. These tags may serve a variety of purposes, such as general processing instructions for an application. An example would be '@' which is used as an indentation marker in the *TLG*.

## 2.2 The proposed lexical intermediate form

The token classes defined above indicate those classes that must appear in a lexical intermediate form (LIF) to serve the electronic-document domain. When deriving a syntax for the LIF, the ease with which a document in the LIF can be processed is the foremost concern.

We require that tags be *explicit* because the token classes associated with implicit tags cannot always be easily determined. For example, the section commands from most text formatters can imply different tags depending on the context. For the purpose of this example, we will use chapter, section, and subsection. The first chapter command in a document does not imply the end of any previous sections. The next chapter command implies the end of the previous chapter plus the end of the last section and the last subsection in that chapter. So, a chapter command can imply from zero to three different end tags. Determining which tags are implied requires information of what previously occurred, i.e., the context in which the chapter command appeared. Because implicit tags cannot always be determined without this auxiliary information, they should not be permitted in the LIF.

We require the token classes to satisfy the property of *disjointness*. Two token classes are *disjoint* if their intersection is empty. This is important because, if the token classes are disjoint, there will be no ambiguity during the lexical analysis. Each token will belong to a unique token class. Ambiguity complicates the lexical-analysis phase of accessing encoded documents. For example, a right parenthesis can be a text string or a tag in Scribe [13]. Hence, the LIF should satisfy the property of disjointness.

### *2.2.1 Abstract syntax of the LIF*

The construction of the abstract syntax of the LIF is driven by a partitioning of the token classes similar to that in Figure 3 for current encoding classes. The first partition of the token classes is that between tags and text. In a survey of current encoding schemes, one feature stands out as distinguishing tags and text. Each scheme reserves at least one printable keyboard character for the exclusive function of signaling the existence of a tag in the data stream. Scribe uses the symbol '@'. LaTeX [11] restricts the use of many characters, most notably '\'. A reserved character ensures that the tag token classes and the text token classes are disjoint. We require that the LIF provide reserved characters to indicate the start and the end of a tag.

Further partitions, as seen in Figure 4, eventually divide the tags into segmenting start tags, segmenting end tags, symbol tags, and nonsymbol tags. Because one of the goals in designing the LIF is that all markup be explicit, the class of implicit segmenting tags is

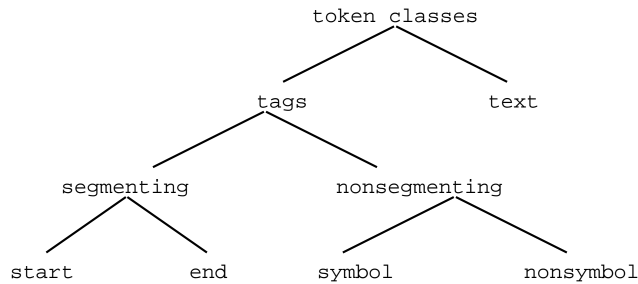


*Figure 4. Partitions of the LIF token classes*

eliminated. For each of these classes, the judicious selection of start and end characters will ensure disjointness of the different tag classes.

A final requirement is to make each of the tags unique, and hence each of the tag token classes will be pairwise disjoint. Each tag has a unique, special meaning to the application for which the tag was originally designed. For example, the commands '`@begin(math)`' and '`@begin(up)`' are both start tags, but it is the identifiers '`math`' and '`up`' that make the commands different and unique. A unique identifier is required in a LIF tag to preserve this meaning.

The final abstract syntax for LIF tags can be seen in Figure 5. The upper-case strings in Figure 5 represent the start and end characters for each of the tags. These characters will be assigned their final values in the concrete syntax.

```
tag             ::= segment_tag
                  | nonsegment_tag
segment_tag     ::= start_tag
                  | end_tag
nonsegment_tag ::= symbol
                  | nonsymbol
start_tag ::= S_TAG_START identifier S_TAG_END
end_tag   ::= E_TAG_START identifier E_TAG_END
symbol    ::= SYM_START identifier SYM_END
nonsymbol ::= NSYM_START identifier NSYM_END
```

*Figure 5. Abstract syntax of the LIF tags*

### 2.2.2 *Concrete syntax of the LIF*

To specify a concrete syntax for LIF tags, we will adopt part of the reference concrete syntax of SGML, because it satisfies most of the requirements specified in the abstract syntax of the LIF tags. For example, start tags, end tags, and symbols all have direct counterparts in SGML.

SGML includes a list of identifiers for a set of widely used graphic characters. Symbols are called *character entity references* in SGML and their declarations are grouped into ISO *public entity sets*. We will use these identifiers in the LIF when possible.

SGML does not specify identifiers for start and end tags because these are usually application or data-object dependent. We define an identifier as a string starting with a letter followed by zero or more letters and digits.

The SGML construct that most closely resembles the class of nonsymbol tags is *processing instructions*. We adapt the notation for processing instructions in SGML for nonsymbol tags. This SGML markup is used for system-specific data. The LIF nonsymbol tag serves as a marker for such data and an identifier similar to those for the start and end tags is used to indicate its function.

The concrete syntax for the LIF is given in Figure 6. This concrete syntax satisfies all of the syntactic properties described for the abstract syntax.

```
start_tag ::= ‘<’  identifier ‘>’
end_tag   ::= ‘</’ identifier ‘>’
symbol    ::= ‘&’  identifier ‘;’
nonsymbol ::= ‘<?’ identifier ‘>’
```

*Figure 6. Concrete syntax of the LIF tags*

#### *2.2.3 A sample document*

A sample Scribe document is presented in Figure 7. A corresponding LIF version of this document can be seen in Figure 8. The amount of information retained in a LIF version is dependent in how the tags are classified and the choice of an identifier.[1] Different classifications of the tags will yield different LIF versions of the document.

## 3 AN OVERVIEW OF LEXICAL ANALYSIS

Given a uniform representation of electronic documents, the next task is to convert existing documents to that representation. We define *retagging* as the process of replacing the markup in an encoded document with its LIF equivalent. In general, retagging is a complex process because many existing encoding schemes contain ambiguous or context-sensitive markup. Thus, techniques like global string replacements are not applicable in a large number of cases, and special, sophisticated software tools are needed to support the process. We examine current theory and practice in lexical analysis as a prelude to proposing the development of a customized toolset for our purposes.

### 3.1 Theoretical model for lexical analysis

The current techniques for implementing lexical analyzers are based on the theory of regular expressions and finite-state automata [18]. A regular expression is often used to describe a pattern that specifies the set of strings that constitute a token class. The

[1] In this example, the device and libraryfile commands have been mapped to nonsymbol tags which do not retain the information in their parameters. If it is necessary to retain the parameter data, then a different identifier could be chosen to reflect the value, i.e., dev_ps for @device(postscript), or the commands could be classified differently.

```
@device(postscript)
@make(article)
@libraryfile(mathematics10)
@section(In-text Math Formulas)
This is an example of a formula that
appears as a part of the text:
@begin(math)
@sum{from <n=1>, to <@infty >}
@over{num <(-1)@begin(up)n@end(up)>,
 denom <n>}
@end(math)

@section(Display Math Formulas)
Note the difference in this display
version of the same formula:
@begin(mathdisplay)
@sum{from <n=1>, to <@infty >}
@over{num <(-1)@begin(up)n@end(up)>,
 denom <n>}
@end(mathdisplay)
```

*Figure 7. Example Scribe document*

```
<?device>
<article>
<?libfile>
<sec><st>In-text Math Formulas</st>
<p>This is an example of a formula that
appears as a part of the text:
<f>
<sum><ll>n=1</ll><ul>&infin;</ul></sum>
<fr><nu>(-1)<sup>n</sup></nu>
 <de>n</de></fr>
</f></p></sec>
<sec><st>Display Math Formulas</st>
<p>Note the difference in this display
version of the same formula:
<fd>
<sum><ll>n=1</ll><ul>&infin;</ul></sum>
<fr><nu>(-1)<sup>n</sup></nu>
 <de>n</de></fr>
</fd></p>
</sec></article>
```

*Figure 8. LIF version of example document*

expression `[A-Za-z][A-Za-z0-9]*` describes the token class *identifier*[2] in many programming languages.

Finite-state automata are used as a tool to design and model the actual code of the desired lexical analyzer [18]. They represent the actions to be taken upon reading the next input character. A finite-state automaton can be represented as either a directed graph or a transition table [19].

The theory of regular expressions and finite-state automata includes theorems which show the equivalence of regular expressions and finite-state automata. The proofs of these theorems include algorithms for constructing finite-state automata from regular expressions. This forms the basis for tools to generate automatically the code for a lexical analyzer.

### 3.2 Current lexical-analyzer generators

With this well-understood model of lexical analysis in place, several tools have been developed to assist in the writing of code for lexical analyzers. These tools, known as lexical-analyzer generators, usually take a high-level specification and generate code for a lexical analyzer. The high-level specification often has the format of a list of *translation rules* each containing a *regular expression* followed by an *action routine*. The action routine is a program fragment that is to be executed when the lexical analyzer matches a segment of the input stream with the regular expression. The program fragment is written in a target implementation language. The Unix™ tool Lex [16] accepts action routines written in the C programming language and generates a lexical-analyzer routine written in C. An example of a translation rule in Lex for the token class *identifier* would be:

```
[A-Za-z][A-Za-z0-9]*      {yylval=install_id(); return(ID);}
```

where the action routine, `{yylval=install_id(); return(ID);}`, updates the symbol table and notifies the parser that an identifier has been found.

Lexical-analyzer generators, such as Lex, automatically generate the transition table representing the required finite-state automaton from the specified regular expression and the necessary code to simulate the finite-state automaton. They do not automatically generate the action routines. In the above example, the function `install_id` must be written by the implementor. In some domains, like programming languages, the token classes and their requisite action routines are well-defined. In these cases, the action routines can be automatically generated.

## 4 DIFFICULTIES WITH THE AUTOMATIC GENERATION OF ACTION ROUTINES

Because we desire to support the automatic generation of retagging code, reasonable candidates for this support are the currently existing lexical-analyzer generators. However, these tools are inadequate for the task, as we illustrate below.

[2] This regular expression describes an identifier as a string that starts with an alphabetic character and may be followed by zero or more alphanumeric characters. Strings that satisfy this regular expression are *stocknum, X,* or *Dd12x*.

## 4.1 Specific examples

### 4.1.1 Delimiters

Delimiter characters play an important role in text-formatting languages. They mark the start and end of strings to be modified and encapsulate arguments of commands. LaTeX reserves one pair of delimiter characters for this purpose and they are always part of a markup string. In Scribe, this is not the case. A regular expression denoting a delimiter, such as ‘`)`’, does not represent a unique token class. To determine if the delimiter character is part of the text or part of a tag, it is necessary to check the left context in which the delimiter character occurs, i.e., the tokens which have occurred prior to the delimiter character. For example, to be a tag, a right delimiter character must have been preceded by a command and the matching left delimiter. Because commands may be nested in Scribe, it is necessary to remember more than one of these contexts.[3]

The matching of delimiters requires a more powerful formal model than regular expressions and finite-state automata. Push-down automata have a memory stack that can be used to record previously seen tokens. In a Lex-like specification, a programmer must implement a stack in the action routines to achieve this functionality. In our specification of a lexical analyzer for a subset of Scribe commands, we were required to write 370 lines of code in the action routines and auxiliary functions, primarily to create and manage a stack required for matching delimiters. We saw a ratio of approximately 6:1 for lines of code to Scribe tokens to generate the desired lexical analyzer.[4] In comparison, we estimate that for Pascal, the ratio would be approximately 1.5:1. The size of the task is further emphasized when one observes that the average number of tokens in a typical programming language (e.g., Pascal has 101) is considerably smaller than that for the typical document encoding schemes (e.g., Scribe has over 300 tokens and the TLG has over 1100).

### 4.1.2 Scope rules

In LaTeX, there is a subset of commands called *declarations* that affect the way a string of text is printed. The text is said to be in the *scope* of the declaration. The declaration itself marks the beginning of the scope, but there is not a single, definitive markup to denote the end of the scope. Table 1 shows several different methods used to denote or imply the end of the scope of the ‘`\em`’ command.

Table 1. Examples of how to end a scope

| Example: | End of scope is indicated by: |
|---|---|
| `{\em text}` | the end of scope character, } |
| `{\em some \bf text}` | a new declaration, `\bf` |
| `\begin{itemize} \em`<br>`\item some`<br>`\item text`<br>`\end{itemize}` | the end of an environment,<br>`\end{itemize}` |

[3] The start-condition mechanism in Lex to handle left context-sensitivity is not sufficient because the left contexts may nest and start conditions do not.

[4] These tokens were a subset of the simpler forms of markup. The ratio of 6:1 would probably be much higher if more complicated forms were included.

In order to retag a LaTeX document, we need to know where the scope ends, since the LIF requires an explicit tag to mark the end. To do this, it is necessary to keep track of the scopes of all of the commands. In a Lex specification, the accessing and updating of the necessary data structure involved over 250 lines of code in the various action routines [20]. The action routine for just the token class '`}`' contained over 120 lines of code.

#### *4.1.3 Additional information*

Occasionally, additional information that is not part of the original text is added to an electronically encoded document. In the TLG, citation records are added for identification and navigation purposes. These records occur on separate lines and are indicated by a '~' (tilde) as the first character. The record is divided into fields, and each field contains a level identifier followed by an optional value. An example of a citation record is `~a"0059"b"034"c"Leg"x1`.

The problem is that the scanning algorithm for these records is different than that for the rest of the encoded document. A separate, special scanning procedure is required. This procedure can be placed in the action routine of a regular expression for citation records. Such a scanning procedure to analyze and process the record requires at least 35 lines of code.

### 4.2 General problems

These problems are not atypical with document encoding schemes. In our work with text-formatting languages, we discovered several general categories of lexical analysis problems [21], all requiring the writing of sometimes lengthy and complex action routines. Broad descriptions of these would be (1) context-sensitive markup, (2) implicit markup, (3) white space, and (4) matching start/end tag pairs.

Context-sensitive markup refers to the fact that some tags change meaning in different environments within text-formatters. When an environment changes, a particular tag may (1) retain its current meaning, (2) take on a new meaning, or (3) no longer be regarded as a tag but must be recognized as part of the text. In Scribe, for example, in certain mathematical forms the word '`from`' has special meaning, but outside these forms it is considered to be part of the text. Because environments can be nested, it is necessary to record nesting information as well as the current context.

Implicit markup is markup implied by other tags. An example of implicit markup is the declaration scope problem in LaTeX (see Section 4.1.2). The lexical analyzer must determine the end of the string that is not marked by the segmenting tag, but is implied by some other markup. This often requires keeping track of the scopes of certain tags.

White space is used to indicate certain types of processing in many text-formatting languages. The most common example of this is the use of a blank line to separate paragraphs in the body of a document. White-space tags are particularly difficult to analyze for a variety of reasons. First, they frequently occur in text strings, so it must be determined if the white space is a tag or part of the text. Second, in the newline case, single newlines have to be distinguished from various classes of consecutive newlines: we have found the classes *one*, *two*, and *two or more* may be significant from a lexical analysis point of view. Third, the analysis of white-space characters is sometimes context-sensitive. In

LaTeX, consecutive newlines are ignored in the preamble of the document and in the list environments, but they have special meaning elsewhere in the document.

Another basic problem is the need to pair explicit segmenting tags. Scribe, LaTeX, and the encoding scheme for the *Dictionary of the Old Spanish Language* (*DOSL*) [6] all have explicit segmenting tags in which the end tags are not unique. All three permit the nesting of explicit segmenting tags. It has been proven that such languages cannot be recognized using finite-state automata [18].

### 4.3 Need for a new toolset

As we have shown, when Lex-like tools are used exclusively to build lexical analyzers in the electronic-document domain, the automatic generation of code for the lexical analyzers has to be complemented in large degree by code that is hand-written by the specifier. What is required to reduce this effort is a toolset for generating lexical analyzers that is domain-specific. Such a toolset would be based on the token classes relevant to the particular domain and would provide libraries of already-coded action routines commonly required in that domain. The specifier is thus relieved of writing or rewriting this commonly occurring code.

## 5 RELATED WORK

Many lexical-analyzer generators have been developed. These existing tools fall into one of two categories: general-purpose or domain-specific.

### 5.1 General-purpose lexical-analyzer generators

Lexical-analyzer generators similar to Lex are general-purpose tools in the sense that the user can specify *any* action in the translation rule. Such general-purpose tools often provide too much general functionality, and too little aid in generating actions for more specific function sets. Action routines can become very complicated in order to determine the token class of a given token.

RWORD [22], Lex [16], LAWS [23], Flex [24], Rex [25], and Wart [26] are all general-purpose lexical-analyzer generators. They each use regular expressions as their specification mechanism. They all accept arbitrary action routines and generate code for a lexical analyzer that may be used in conjunction with a parser routine. INR [27] also has the capability of recognizing arbitrary regular languages, but it generates a regular language parser.

### 5.2 Domain-specific lexical-analyzer generators

In certain domains, such as programming languages, the token classes and their corresponding action routines are well-defined. The use of a general-purpose lexical-analyzer generator in these domains requires the programmer to rewrite existing code for the action routines. For this reason, domain-specific lexical-analyzer generators have been developed that facilitate the definition of token classes and provide libraries of action routines.

Programming languages is one domain where the token classes are clearly defined and each token class has a specific action to be performed in the lexical-analysis phase. For example, the token class *identifier* requires specific actions to be performed. These include:

(1) installing the string that represents an identifier into the symbol table and (2) notifying the parser that an identifier has been found. These actions can easily be automatically generated. For example, in $\gamma$–GLA [28], a translation rule for identifier would be:

```
$[A-Za-z][A-Za-z0-9]*      [mkidn]
```

where `mkidn` is a library routine that performs the appropriate processing for an identifier. Lexical-analyzer generators that have been designed specifically for programming languages are $\gamma$–GLA [28], Alex [29], Mkscan [30], and LexAGen [31].

These specialized lexical-analyzer generators for programming languages will not meet our requirements for digital libraries. They are limited to understanding the set of token classes in the programming-language domain. The domain of electronic documents has a different set of token classes that require different action routines.

## 6 THE RETAGGING TOOLSET

We have identified two major tasks in the process of retagging a document: (1) identifying and replacing the existing markup strings and (2) inserting missing start and end tags in order to satisfy the LIF requirement that all segmenting tags have explicit start and end tags. Each task requires a different type of information be provided in order to generate a program that performs that task. For this reason the Retagging Toolset consists of two major tools: the Replace Tag Tool and the Insert Tag Tool.

Each of the current prototypes of the tools in the Retagging Toolset consist of an editor interface and a specification compiler. The interface permits the user to enter and modify statements in the high-level specification. Once the user is satisfied with the specification, the compiler can be invoked in order to check the specification and generate the corresponding program modules.

The Retagging Toolset is used for *developing* the necessary program modules for the retagging process. Once the development process has been completed, the modules can be used as independent programs. In this section we describe the different parts of each tool, the typical user, and a sample session using the tool.

### 6.1 The Replace Tag Tool

The primary objective of the Replace Tag Tool (RTT) is to generate automatically a program that will *replace* all existing markup in a document with the LIF equivalent. For this discussion we will call the generated program Replace_Tag. In order to achieve this objective, the RTT facilitates the entry of a high-level specification of the replace-tag process. This high-level specification is based on the tag classes described in Section 2.

#### *6.1.1 The intended user*

Since the generated program is based upon a specification given by the user, the quality of the Replace_Tag program depends upon the expertise of the RTT user. The user must have expert knowledge of the encoding scheme in order to describe the tag strings. Without some specification of the tag strings, retagging cannot take place because it would not be possible to find the tags to replace. The user must know the composition of the existing

tag strings and be able to translate that knowledge into simple strings or more elaborate regular expressions.

Another aspect of the encoding scheme of which the user must be aware is the partition of the keyboard characters with regard to their use in markup. There are three alphabets that may occur in encoding schemes. The text alphabet includes the alphanumeric characters plus some subset of the nonalphanumeric characters. The reserved-for-markup alphabet contains those characters that only appear in tag strings of the encoded document. Examples of this alphabet are {@} for Scribe and {$,&,%,",@,[,],<,>,{,},#} in the *TLG*. The ambiguous alphabet includes characters that are used as markup, but not reserved for that purpose. In Scribe, this alphabet is {(,),[,],{,},<,>,',",`}. Since each of these alphabets requires different actions to be performed in the Replace_Tag program, the user must specify each.

In order to transfer the user's knowledge of the encoding scheme to the purpose of retagging, the user must assign each tag string to one of the categories: symbol tag, nonsymbol tag, implicit segmenting tag, or explicit segmenting tag. This assignment can be difficult and may not be unique. The assignment can be further complicated by questions about what constitutes the text of the document.

To illustrate this problem, we will consider the citation records of the *TLG*. These records are used to identify the document and its different parts. As such, they are not actually part of the text of the original document. Based on this perception, an RTT user may classify a citation record as a nonsymbol tag and map a record such as `~a"0059"b"034"x1` to the LIF tag <?tilde>, discarding the information in the record. Another *TLG* expert may decide to classify the citation record as data surrounded by segmenting tags. In this case the record `~a"0059"b"034"x1` might be mapped to <tilde>`a"0059"b"034"x1`</tilde>. Both of these retag mappings are valid, but the viability of future applications that access the LIF version of the document will depend on which one is chosen.

A final consideration on the part of the RTT user is the set of identifiers to be used in the LIF tags. Many SGML tag sets already have been established. For example, the ISO has established a set of identifiers for the common graphic characters used in publishing [3]. The Association of American Publishers has developed sets of SGML tags for tabular material [32], mathematical formulas [33], and several types of documents [34]. Rather than invent a new set of identifiers, the user may wish to incorporate an existing tag set.

### *6.1.2 Parts of the RTT*

The RTT has three major areas of activity: the command buttons, the alphabet definition windows, and the tag mapping editors. The general layout of the RTT can be seen in Figure 9.

**The command buttons** There are six buttons at the top of the RTT. (See top of Figure 9.) These buttons are used to permit the input (Load) and output (Save) of specification files with the user interface, to invoke the Replace_Tag program generator (Make), to execute a compiled Replace_Tag program (Run), to reset the components of the RTT to their default values (Reset), and to exit the tool (Quit).

**The alphabet definition windows** There are three dialog windows provided for the input of the character alphabets described in Section 6.1.1. These windows can be seen in the middle of Figure 9. The first alphabet definition window contains the text alphabet. It has a

default value when first instantiated or when reset. The second alphabet definition window is used to input the reserved character alphabet and the third window is used to input the ambiguous character alphabet. There are special rules that permit the specification of a range of characters and govern the specification of certain special characters.

**The tag mapping editors** There are four tag mapping editors in the RTT, one for each of the four tag classes. These editors are shown at the bottom of Figure 9. Each of these windows is instantiated as a text widget from the Athena widget set of the X Window System [35]. A set of basic commands is provided with the widget for traversing the contents of the window and for editing the text.

Statements in the Symbol, Nonsymbol, and Implicit Segmenting Tags Editors have the same structure. Simple statements consist of a regular expression followed by a LIF tag identifier. In our current prototype, the rules for constructing regular expressions are those used for Lex [16]. In Figure 9, the first statement in the Symbol Tags Editor contains a regular expression for the percent character and the LIF identifier “dagger”. This statement

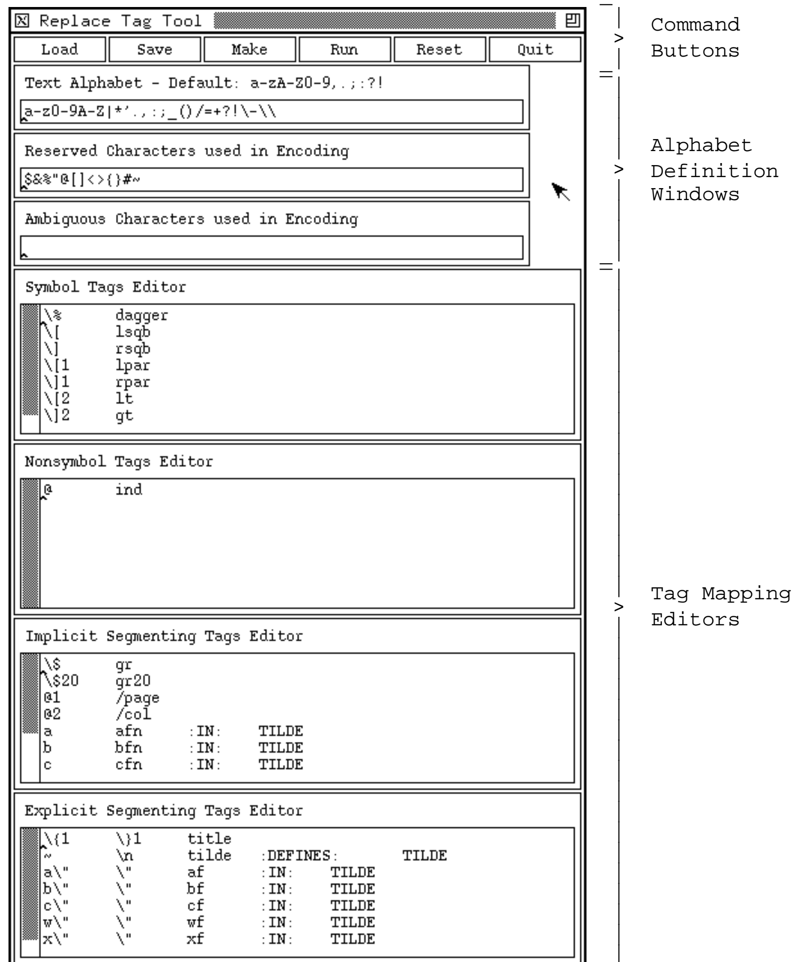


*Figure 9. The RTT loaded with a TLG specification*

generates code that will replace all occurrences of "`%`" with "`†`". In the Implicit Segmenting Tags Editor (ISTE), the user must preface the LIF identifier with "`/`" if the tag is an end tag.

In the Explicit Segmenting Tags Editor (ESTE), it is necessary to specify two regular expressions, one for the start tag and another for the end tag. In Figure 9, the first statement in the ESTE generates code that will replace all occurrences of "`{1`" with "<title>" and all occurrences of "`}1`" with "</title>".

It is also possible to define and reference contexts by using special clauses with the statements. The second statement in the ESTE defines a context called TILDE for citation records in the *TLG*. The scope of the context begins when a "~" is found in the document and terminates with the newline character (`\n`) that follows. The last three statements in the ISTE and the last five statements in the ESTE define the tag strings that are to be recognized only in the TILDE context. Note that the characters a, b, and c are interpreted as field start tags in the TILDE context. Outside of this context, they are viewed as text characters. Additional information about these types of statements can be found in Reference [36].

### *6.1.3 A sample session*

When the RTT is first invoked, it comes up with most of its editor windows empty. The user may enter a new specification directly into the RTT by typing in the appropriate windows. Or, the user may load a previously defined specification by using the Load Button. After editing the specification, the user must click the Save Button to save the current version of the specification.

Once the specification has been saved, the user may generate a Replace_Tag program by using the Make Button. This invokes the Replace_Tag program generator which checks the specification file for possible problems and generates a Replace_Tag program. If there are no errors, the Replace_Tag program is compiled. In order to execute the compiled Replace_Tag program from the RTT, the user clicks the Run Button. At this point, the user is prompted for input and output files. If results of executing the Replace_Tag program are not satisfactory, the specification can be modified and the process repeated.

## 6.2 The Insert Tag Tool

The objective of the Insert Tag Tool (ITT) is to generate automatically a program that will *insert* the missing start or end tag of the previously specified implicit segmenting tags. We call this generated program Insert_Tag. This definition of the ITT implies that the input of an Insert_Tag program is the output of a Replace_Tag program or some other partially marked-up LIF document.

### *6.2.1 The intended user*

The ITT views a document as a set of properly nested delimited strings. For this reason, the user must also be aware of the relationships among the text segments associated with the segmenting tags. There are three categories of interaction between tagged text segments: contiguous, nested, and overlapping.

Tagged text segments are *contiguous* if the end of one segment implies the beginning of the next segment. This is the case in the *TLG* with font tags. Only the beginning of a text

segment is tagged. The next font tag indicates the end of the text segment for the previous font tag.

A tagged text segment is *nested* if it is enclosed within a larger tagged segment. An example from the *DOSL* is the folio-reference tag. This tag only appears within the scope of the heading tag. The heading end tag implies the folio-reference end tag.

Tagged text segments *overlap* when the start and end of one segment is not encapsulated inside the start and end of the other segment. This is the case with the font and page tags in the *TLG*. A string of text in one font may start in the middle of a page and run many pages before ending on another page.

The ITT has no problems with contiguous or nested tags, because they fit the model of properly nested delimited strings. It cannot handle overlapping tag segments directly. For a set of tags with overlapping text segments, the ITT must be used repeatedly to create Insert_Tag programs for different subsets of implicit segmenting tags. These subsets would contain only tags with nested or contiguous text segments. In the case of the TLG, the user could build one Insert_Tag program for the font tags and another Insert_Tag program for the page tags, where each individual program is concerned with only a subset of contiguous or nested tags.

### *6.2.2 Parts of the ITT and a sample session*

The ITT has the same general layout as the RTT (see Figure 10). The command buttons are the same as those for the RTT and provide the same functionality.

The ITT has only one editor window, the Insert Tags Editor. The contents of this window are a list of statements describing the implicit segmenting tags that require mate tags to be inserted. This set of statements must list the segmenting tags to be considered and the nesting hierarchy between the specified tags. In Figure 10, the tags <article> and </p> require mate tags to be inserted where the start tag, <p>, may be implied by the <st*> and <fig> tags. A detailed explanation of these statements can be found in Reference [36].

A sample session with the ITT is similar to that for the RTT. The user creates a specification and then makes an Insert_Tag program that can be executed later.

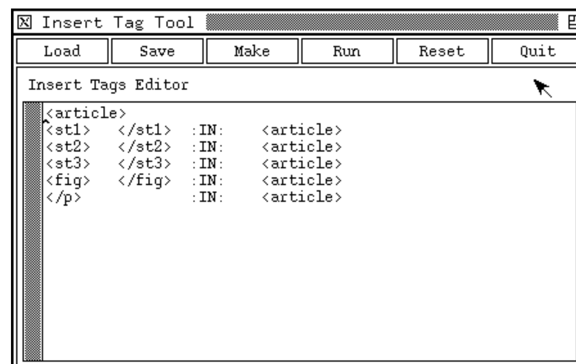


*Figure 10. The ITT loaded with a TLG specification*

### 6.3 Functionality of the toolset

The RTT specification generates a Lex and a Yacc specification for a Replace_Tag program that will recognize the markup strings in a document and replace these strings with LIF tags. The generated Lex specification may contain start conditions in order to recognize markup strings defined in certain contexts. Although Lex does not support nested start conditions, the RTT generates code to support the nesting of contexts. The generated Yacc specification yields a general description of a document as a mixture of tags and text strings without any formal structure. It does provide for the nesting of tagged text strings. This permits the replacement of ambiguous tags like delimiter characters with the correct LIF equivalent.

The ITT specification generates Lex and Yacc specifications describing a document as a hierarchy of tagged strings. These generated specifications are used to create an Insert_Tag program that will insert missing mate tags of any implicit segmenting tags listed in the ITT specification.

Although the RTT and ITT tools generate programs that are implemented with Lex and Yacc, the existence of these tools represents a significant improvement compared to the prevailing situation. With the RTT and ITT, a user is able to specify retagging and insertion at a higher cognitive level, with no knowledge of Lex and Yacc and with virtually no need to write any C code. Without the toolset, the user would be faced with either using Lex and Yacc directly or, if Lex and Yacc are not used, writing hundreds of lines of C code to scan and parse a document. The user would also have to create a model for encoded documents, write a fair amount of C code for the action routines to perform the retagging and insertion tasks, as well as manage any interface issues.

## 7 EXPERIENCES WITH THE TOOLSET

The Retagging Toolset was used to retag a variety of document encoding schemes from the humanities, linguistics, and text-formatting domains. In particular, the Toolset was used to build retagging software for documents encoded for the *Thesaurus Linguae Graecae* (*TLG*), the *Dictionary of the Old Spanish Language* (*DOSL*), the *Lancaster-Oslo/Bergen Corpus* (*LOB*), the *Waterloo Concordance Package* (*WATCON-2*), the *Oxford Concordance Program* (*OCP*), and *Scribe*. These encoding schemes represent a diverse level of data-entry technologies and syntactic sophistication. A complete description of how the Toolset was used to build the retagging software for each of these can be found in Reference [36].

In Table 2 we summarize the statistics gathered pertaining to the amount of work that was required in developing retagging software for the different encoding schemes. The number of user-specified specification statements is the total number of statements entered into the RTT and ITT. The number of user-specified C code statements represents the additional C code that had to be written. This additional code consisted of preprocessors that converted physical records to logical records, for those schemes limited to fixed length records for data entry, and processors that inserted missing tags which were not implied by other tags.

The lines of generated code reflect the effort that would be required to use Lex and Yacc directly to build the same retagging software. The number of specification lines corresponds to the number of rules in the Lex and Yacc specifications. The number of lines of C code includes the statements in the action routines of the Lex and Yacc specifications, auxiliary functions, and any additional processors.

Table 2. Results of the experiences with the toolset

| Encoding scheme | Number of user specified statements | | Lines of generated code | | Savings |
|---|---|---|---|---|---|
| | Specification | C code | Specification | C code | |
| *TLG* | 46 | 0 | 192 | 500 | 15.0 : 1 |
| *DOSL* | 63 | 16 | 165 | 461 | 7.9 : 1 |
| *LOB* | 478 | 23 | 846 | 1332 | 4.3 : 1 |
| *WATCON-2* | | | | | |
| —García-Lorca | 8 | 10 | 33 | 119 | 8.4 : 1 |
| —Woolf | 22 | 0 | 80 | 183 | 12.0 : 1 |
| *OCP* | | | | | |
| —COCOA Ex. 1 | 11 | 3 | 37 | 113 | 11.0 : 1 |
| —COCOA Ex. 2 | 7 | 0 | 29 | 105 | 19.1 : 1 |
| —COCOA Ex. 3 | 11 | 0 | 36 | 116 | 13.8 : 1 |
| —Start String Ex. | 5 | 1 | 27 | 112 | 23.2 : 1 |
| *Scribe* | 177 | 15 | 403 | 937 | 7.0 : 1 |

The entries for the concordance building packages (*OCP* and *WATCON-2*) differ from the other entries, because these packages do not have one specific encoding scheme. Each permits the user to set different parameters, thus creating a different encoding scheme for each document that we retagged.

The savings is calculated by dividing the lines of generated code by the number of user specified statements. The savings range from a factor of 4.3 to a factor of 23.2. The savings factor is in part a reflection of either the complexity or the number of tags in an encoding scheme. Schemes that require a large number of statements in their specifications to describe the tag sets or the relationships between tags will have smaller savings factors. Encoding schemes that require a large number of context clauses will have more substantial savings.

An average savings can be calculated by totaling the lines of specification and the lines of C code. This process yields an average savings of 6.5:1. This value is not totally representative because of the undue influence of the values for the *LOB Corpus*.[5] If the *LOB Corpus* values are removed from the calculations, then the average savings becomes 9.2:1.

Overall, the amount of C code that must be generated by the user is substantially reduced. By comparing the 68 lines of C code that the user had to generate while using the toolset to the 3978 lines of C code without the toolset, we see that over 98 per cent of the required retagging code was automatically generated.

## 8 PRACTICAL SIGNIFICANCE OF THE LIF

The Retagging Toolset significantly reduces the time to develop software to convert encoded documents to LIF. Once a document is encoded in LIF, it becomes accessible

[5] The savings for the *LOB Corpus* is deflated. Because they appeared at different levels in the hierarchy of nesting tags, approximately 155 tags had to be specified twice, rather than only once, in the ITT specification. Planned enhancements to the specification language of the ITT to reduce the number of statements necessary for this situation will substantially improve the savings.

to more applications. The structure of the LIF tags facilitates the development of new applications by eliminating many of the lexical idiosyncrasies found in existing encoding schemes.

For many applications, the LIF version of a document is sufficient. New applications can be generated, or existing ones applied directly, without any further manipulation of the tag set. For example, if a document was originally encoded for a corpus, then the LIF version still contains tags denoting parts of speech. Linguistic analysis packages such as those yielding certain word counts can be written based on the LIF tags, without any further manipulation of the tagset.

Also, because the LIF is encoded using SGML-like tags, some existing applications can access LIF-encoded documents. An example is the searching tool, PAT, developed at the Centre for the New Oxford Dictionary at the University of Waterloo [10]. PAT is used to search for instances of a specified string. It can match SGML-like tags similar to those in the LIF. Thus PAT can be used to extract information from LIF-encoded documents.

For other applications, converting documents to their LIF equivalents may be only a first step in a more complex analysis process. An example of one such application is the Integrated Chameleon Architecture (ICA) developed at the Ohio State University. One activity of the ICA is the generation of translators among different data representations. The ICA can be used to build translators for documents in syntactically different, but philosophically similar, encoding schemes such as LaTeX and Scribe. The Retagging Toolset and LIF address and solve only the *lexical* analysis step of this translation process. Another tool in the ICA is then applied to address issues relating to *parsing* of the tagset, e.g., the ordering and occurrence rules for tags [37].

## 9 SUMMARY AND FUTURE WORK

There exist many document encoding schemes and software applications to process electronically encoded documents. The plethora of schemes complicates the development of applications that must access documents in more than one representation. A uniform representation of electronic documents would greatly facilitate software development.

Unfortunately, the retagging of existing electronic documents is difficult, given the current development tools. The fundamental problem of distinguishing the markup from the text strings is complicated by problems such as context-sensitive markup, implicit markup, white space, and the matching of start and end tags. Lexical-analyzer generators such as Lex are based on formal models that are inadequate to handle these problems. Because of this, much of the retagging code must be written by hand.

Based on a generalization of these problems, we developed a new model for textual data objects with embedded markup. We then proposed a uniform representation called a Lexical Intermediate Form, LIF. The LIF borrows its concrete syntax from the ISO standard SGML, but it is not encumbered with the SGML concept of document-type definitions. Based on our model and the proposed LIF, we identified two steps in the retagging process: the replacement of existing tags with their LIF equivalents and the insertion of missing implicit segmenting tags. We developed software tools that automatically generate the code for each of these steps.

The toolset was exercised by using it to specify the retagging of six encoding schemes of varying complexity. The resulting savings indicated an increase in productivity ranging

from 4.3:1 to 23.2:1. Overall, any additional C code that had to be written by hand represented less than two percent of the C code generated by the toolset.

The *usefulness* of the Retagging Toolset has been established by the work in this paper. A study of the *usability* of the Retagging Toolset would provide an indication of the effect the Retagging Toolset's interface has on the task of developing retagging software. The results of this study may identify additions, enhancements, or deletions to the Retagging Toolset's interface.

Further experimentation with the Retagging Toolset in building retagging software for other electronic-document encoding schemes is also warranted. The text-formatting language LaTeX is an excellent candidate for this work and this retagging effort is under way. LaTeX has a level of sophistication similar to that of Scribe plus it has its own idiosyncrasies, whereas the encoding schemes in the humanities and linguistic domains usually have a simpler structure. The results of this study may identify enhancements or discrepancies in the specification languages of the Retagging Toolset.

Another area of evaluation is the applicability of the toolset to other domains. There is currently an effort under way to create an exchange format for patient records in the medical community, called Patient Record Exchange Format (PREF) [38]. The existing database records must be converted to the PREF format and the Retagging Toolset may be useful in the development of the conversion software.

Other enhancements to the toolset currently under consideration are to: (1) provide a method of importing a predefined set of LIF tag identifiers into the RTT; (2) provide a method of importing a set of implicit segmenting tags into the ITT from the tags specified in the RTT; (3) provide additional support in the RTT to identify improperly formed regular expressions; (4) expand the ITT specification language to eliminate the need to specify an implicit segmenting tag more than once when it appears in more than one level of the hierarchy; and (5) design and implement a tool that will generate a program that will insert tags implied by other tags or the end-of-file.

## ACKNOWLEDGEMENTS

This work has been supported in part by a grant from the Applied Information Technologies Research Center, Columbus, Ohio.

## REFERENCES

1. Robert E. Kahn and Vinton G. Cerf, 'An open architecture for a digital library system and a plan for its development', (March 1988). Draft.
2. Oliver Jones, *Introduction to the X Window System*, Prentice Hall, Englewood Cliffs, NJ, 1988.
3. *Information Processing—Text and Office Systems—Standard Generalized Markup Language (SGML)*, October 1986. ISO 8879-1986 (E).
4. Claus Huitfeldt, *The Norwegian Wittgenstein Project: Transcription Methods, Software, Editorial Principles*, Norwegian Computing Centre for the Humanities, Bergen.
5. Stephen A. Kaufman, *The Comprehensive Aramaic Lexicon Text Entry and Format Manual*, The Department of Near Eastern Studies of The Johns Hopkins University, Baltimore, 1987.
6. David Mackenzie and Victoria A. Burrus, *A Manual of Manuscript Transcription for the Dictionary of the Old Spanish Language*, The Hispanic Seminary of Medieval Studies, Ltd., Madison, fourth edition, 1986.
7. University of California, Irvine, California, *Thesaurus Linguae Graecae: Beta Manual*, 1988.

8. W. N. Francis and Henry Kučera, *Frequency Analysis of English Usage: Lexicon and Grammar*, Houghton Mifflin Co., 1982.
9. Stig Johansson, Eric Atwell, Roger Garside, and Geoffrey Leech, *The Tagged LOB Corpus: User's Manual*, Norwegian Computing Centre for the Humanities, Bergen, 1986.
10. Donna Lee Berg, Gaston H. Gonnet, and Frank Wm. Tompa, 'The New Oxford English Dictionary Project at the University of Waterloo', Technical Report OED-88-01, Centre for the New Oxford English Dictionary, University of Waterloo, (February 1988).
11. Leslie Lamport, LaTeX *User's Guide and Reference Manual*, 1985.
12. Sun Microsystems, Inc., *Using nroff and troff on the Sun Workstation*, 1986.
13. Unilogic, Ltd., *Scribe Document Production System User Manual*, fourth edition, 1984.
14. Susan Hockey and Ian Marriot, *Oxford Concordance Program, Version 1.0, User's Manual*, Oxford University Computing Service, 1986.
15. Philip H. Smith, Jr., *WATCON-2: A Concordance-Generator Program Package*, Waterloo, Ontario, 1985.
16. M. E. Lesk, 'Lex – a lexical analyzer generator', Technical Report Computing Science Technical Report 39, AT&T Bell Laboratories, Murray Hill, NJ, (1975).
17. S. A. Mamrak, M. J. Kaelbling, C. K. Nicholas, and M. Share, 'Chameleon: A system for solving the data translation problem', *IEEE Transactions on Software Engineering*, **15**(9), (September 1989).
18. John E. Hopcroft and Jeffrey D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley, Reading, MA, 1979.
19. A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, Reading, MA, 1986.
20. Joan Bushek, LaTeX *and SGML: A Functional Evaluation of One Approach to Automatic Generation of Translators*, Master's thesis, The Ohio State University, 1988.
21. S. A. Mamrak, J. Barnes, J. Bushek, and C. K. Nicholas, 'Translation between content-oriented text formatters: Scribe, LaTeX and Troff', Technical Report OSU-CISRC-8/88-TR23, Department of Computer and Information Science, The Ohio State University, (August 1988).
22. Walter L. Johnson, James H. Porter, Stephanie I. Ackley, and Douglas T. Ross, 'Automatic generation of efficient lexical processors using finite state techniques', *Communications of the ACM*, **11**(12), 805–813, (December 1968).
23. Robert C. Gammill, *A Portable Lexical Analyzer Writing System*, The Rand Corp., Santa Monica, CA, 1983.
24. Vern Paxson. Flex manual page, Sun release 3.5, May 1987.
25. Josef Grosch, *Rex—A Scanner Generator*, Gesellschaft für Mathematik und Datenverarbeitung mbH, Forschungsstelle an der Universität Karlsruhe, December 1987.
26. Jeff Damens. Wart source code, version 1A(006), 1989.
27. Rick Kazman, *Structuring the Text of the Oxford English Dictionary through Finite State Transduction*, Master's thesis, University of Waterloo, 1986.
28. Robert W. Gray, '$\gamma$–GLA: A generator for lexical analyzers that programmers can use', in *Proceedings of the Summer 1988 USENIX Conference*, pp. 21–24, San Francisco, (June 1988).
29. H. Mössenböck, 'Alex – a simple and efficient scanner generator', *SIGPLAN Notices*, **21**(12), 139–148, (December 1986).
30. R. Nigel Horspool and Michael R. Levy, '*Mkscan* – an interactive scanner generator', *Software Practice and Experience*, **17**(6), 369–379, (June 1987).
31. Duane Szfaron and Randy Ng, *LexAGen User's Manual*, Department of Computing Science, University of Alberta, Edmonton, Alberta, Canada, February 1989.
32. Association of American Publishers, *Markup of Tabular Material*, April 1986. Electronic Manuscript Series.
33. Association of American Publishers, *Markup of Mathematical Formulas*, April 1986. Electronic Manuscript Series.
34. Association of American Publishers, *Standard for Electronic Manuscript Preparation and Markup*, 1986. Electronic Manuscript Series.
35. Chris D. Peterson, *Athena Widget Set–C Language Interface, X Window System, X Version 11, Release 4*, MIT X Consortium, 1989.
36. Julie A. Barnes, 'Analysis of document encoding schemes: A general model and retagging

toolset', Technical Report OSU-CISRC-7/90-TR19, Department of Computer and Information Science, The Ohio State University, (July 1990).

37. S. A. Mamrak, C. S. O'Connell, and J. A. Barnes, 'The Integrated Chameleon Architecture: A software toolset to support data translation', Technical Report OSU-CISRC-11/90-TR37, The Ohio State University, (November 1990).
38. S. A. Mamrak. Patient record exchange project: Specification and software tools. Proposal submitted to the Agency for Health Care Policy and Research, May 1990.