INFORMATION TO USERS

The most advanced technology has been used to photograph and reproduce this manuscript from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.
Analysis of document encoding schemes: A general model and retagging toolset

Barnes, Julie Ann, Ph.D.

The Ohio State University, 1990

Copyright ©1990 by Barnes, Julie Ann. All rights reserved.
ANALYSIS OF DOCUMENT ENCODING SCHEMES: A GENERAL MODEL AND RETAGGING TOOLSET

DISSERTATION

Presented in Partial Fulfillment of the Requirements for the Degree Doctor of Philosophy in the Graduate School of The Ohio State University

By

Julie A. Barnes, B.S.Ed., B.S., M.A., M.S.

* * * * *

The Ohio State University

1990

Dissertation Committee
Sandra A. Mamrak
Chua-Huang Huang
Neelam Soundararajan

Approved by

Adviser
Department of Computer and Information Science
To my husband.
ACKNOWLEDGEMENTS

I wish to express my gratitude to my dissertation adviser, Dr. Sandra Mamrak, for her constant support and constructive criticism during the research and writing of this thesis. Thanks go to the other members of my reading committee, Dr. Neelam Soundararajan and Dr. Chua-Huang Huang. I am also indebted to the many members of the Chameleon Research Group for their friendship, advice, and technical expertise. Specifically, I want to thank Joan Bushek, Loyde Hales, Dr. Michael Kaelbling, Dr. Charles Nicholas, Dr. Conleth O'Connell, and Dr. Michael Share.

Thanks go to the wonderful staff in the department office, Chris, Ernie, Jane, Jeff, Judy, Marty, and Tom, whose help made the day-to-day tasks easier to endure. I also wish to thank Dana Frost, Michael Fortin, and Jim Clausing for their special friendship during my stay at The Ohio State University.

To my husband, Roger, I offer sincere thanks for your faith in me and your willingness to share the highs and lows of the past five years. I could not have done it without you.

Finally, I wish to thank the Applied Information Technologies Research Center, who provided the financial support for much of this work.
VITA

May 13, 1954 ........................................................Born - Bellevue, Ohio

1976 .................................................................B.S.Ed. Mathematics Education
B.S. Mathematics
Bowling Green State University
Bowling Green, Ohio

1976-1979 ..............................................................Graduate Teaching Assistant
Department of Mathematics and
Statistics
Bowling Green State University
Bowling Green, Ohio

1978 .................................................................M.A. Mathematics
Bowling Green State University
Bowling Green, Ohio

1979-1980 ..............................................................Math and Spanish Teacher
Mendon Union Local Schools
Mendon, Ohio

1980-1981 ..............................................................Math Teacher
McAuley High School
Toledo, Ohio

1981-1982 ..............................................................Instructor
Department of Mathematics and
Statistics
Bowling Green State University
Bowling Green, Ohio

1982-1985 ..............................................................Instructor
Department of Computer Science
Bowling Green State University
Bowling Green, Ohio
1984 .......................................................... M.S. Computer Science
Bowling Green State University
Bowling Green, Ohio

1985-1987 ..................................................... Graduate Teaching Associate
Department of Computer and Information Science
The Ohio State University
Columbus, Ohio

1986-present .................................................. Graduate Research Assistant
Chameleon Research Group
The Ohio State University
Columbus, Ohio

Publications


Fields of Study

Major Field: Computer and Information Science

Studies in:

Accommodating Heterogeneity: Prof. Sandra A. Mamrak
Programming Languages: Prof. Neelam Soundararajan
Theory: Prof. Timothy Long
# Table of Contents

ACKNOWLEDGEMENTS ...................................................... iii
VITA ................................................................................ iv
LIST OF TABLES .............................................................. ix
LIST OF FIGURES .......................................................... x

| CHAPTER |
|----------|----------------|
| I        | Introduction   | 1 |
| II       | A Uniform Representation for the Electronic-Document Domain | 6 |
| 2.1      | Token Classes in Encoding Schemes | 7 |
| 2.2      | The Proposed Lexical Intermediate Form | 9 |
| 2.2.1    | Abstract Syntax of the LIF | 10 |
| 2.2.2    | Concrete Syntax of the LIF | 12 |
| 2.2.3    | A Sample Document | 14 |
| III      | An Overview of Lexical Analysis | 16 |
| 3.1      | Theoretical Model for Lexical Analysis | 16 |
| 3.2      | Current Lexical-Analyzer Generators | 17 |
| IV       | Difficulties with the Automatic Generation of Action Routines | 19 |
| 4.1      | Specific Examples | 19 |
| 4.1.1    | Delimiters | 19 |
| 4.1.2    | Scope Rules | 21 |
| 4.2      | General Problems | 22 |
| 4.3      | Need for a New Toolset | 24 |
# List of Tables

<table>
<thead>
<tr>
<th>TABLE</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>21</td>
</tr>
<tr>
<td>2</td>
<td>67</td>
</tr>
<tr>
<td>3</td>
<td>69</td>
</tr>
<tr>
<td>4</td>
<td>74</td>
</tr>
<tr>
<td>5</td>
<td>76</td>
</tr>
<tr>
<td>6</td>
<td>79</td>
</tr>
<tr>
<td>7</td>
<td>82</td>
</tr>
<tr>
<td>8</td>
<td>87</td>
</tr>
<tr>
<td>9</td>
<td>90</td>
</tr>
<tr>
<td>10</td>
<td>92</td>
</tr>
<tr>
<td>11</td>
<td>97</td>
</tr>
<tr>
<td>12</td>
<td>101</td>
</tr>
<tr>
<td>13</td>
<td>105</td>
</tr>
<tr>
<td>14</td>
<td>113</td>
</tr>
</tbody>
</table>

ix
# List of Figures

<table>
<thead>
<tr>
<th>FIGURE</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Ideal Database-Processing Model</td>
</tr>
<tr>
<td>2</td>
<td>Current Database-Processing Model</td>
</tr>
<tr>
<td>3</td>
<td>Partitions of the Token Classes in Document Encoding Schemes</td>
</tr>
<tr>
<td>4</td>
<td>Partitions of the LIF Token Classes</td>
</tr>
<tr>
<td>5</td>
<td>Abstract Syntax of the LIF Tags</td>
</tr>
<tr>
<td>6</td>
<td>Concrete Syntax of the LIF Tags</td>
</tr>
<tr>
<td>7</td>
<td>A Scribe Document</td>
</tr>
<tr>
<td>8</td>
<td>LIF Version of a Scribe Document</td>
</tr>
<tr>
<td>9</td>
<td>The Replace Tag Tool</td>
</tr>
<tr>
<td>10</td>
<td>The Command Buttons of the RTT</td>
</tr>
<tr>
<td>11</td>
<td>The Alphabet Definition Windows</td>
</tr>
<tr>
<td>12</td>
<td>The Tag Mapping Editors</td>
</tr>
<tr>
<td>13</td>
<td>The RTT With the File Selection Dialog Box</td>
</tr>
<tr>
<td>14</td>
<td>The RTT Loaded With a TLG Specification</td>
</tr>
<tr>
<td>15</td>
<td>The RTT With the FSDB Prompting For a Specification Filename</td>
</tr>
<tr>
<td>16</td>
<td>The RTT With the Overwrite Warning Box</td>
</tr>
<tr>
<td>17</td>
<td>The RTT With the FSDB Prompting For an Object Filename</td>
</tr>
<tr>
<td>18</td>
<td>The RTT With the FSDB Prompting For a Replace_Tag Program</td>
</tr>
</tbody>
</table>
63  COCOA Text After Processing by Replace_Tag Program   98
64  COCOA Text After Processing to Insert Missing End Tags   100
65  Starting String Text After Preprocessing                101
66  Starting String Text After Processing by Replace_Tag Program  102
67  A Scribe Document                                       104
68  Scribe Document After Processing by the First Replace_Tag Program  106
69  Scribe Document After Processing by the Second Replace_Tag Program  108
70  Scribe Document After Processing by Special Insert-Tag Program  110
71  Scribe Document After Processing by the Insert_Tag Program  111
CHAPTER I

Introduction

Electronic databases provide access to data for a wide variety of application programs. For example, a manuscript in a document database might be accessed by text formatters, concordance packages, and linguistic-analysis packages. An ideal model of accessing a database would be similar to a client-server model. A common set of database-access functions would permit an application to retrieve or modify the data in the database. The application would be restricted to the application-dependent functions required to do the desired processing and the database would be accessed by calling a database-access function. This ideal model is depicted in Figure 1.

Figure 1: Ideal Database-Processing Model
A prime example of this model is the X Window System [19]. Although the resource to be accessed in this case is not a database, 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 [31] (SGML) for document databases. SGML provides a method for defining document types and declaring their markup. Documents in an SGML-encoded database can be accessed with a typical SGML parser. Such a parser provides the common database-access functions, but 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 database-access functions already exist. Also, it permits a single version of the database-access functions to be used by all applications that access the database.

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 doc­
ument databases [15, 20, 24, 37], corpora [8, 16], and dictionaries [5]. There are
also assorted text-formatting languages [22, 33, 35]. To further complicate mat­
ters, concordance-building programs [12, 32] permit the users to define their own
restricted encoding scheme to be used with that concordance program. Applica­
tions usually contain both the application-dependent functions and the necessary
database-access functions (see Figure 2). In this scenario, if an application has to
access more than one data representation, a different set of database-access func­
tions 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 repre­
sentation are dictated by the needs of application packages in the domain. A major
contribution of this dissertation is the identification of the necessary characteristics
of representations in the electronic-document domain and the definition of a uniform representation for this domain.¹ In Chapter II 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 tools like Lex [23], 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. A major contribution of this work is the recognition of the inadequacy of the current model of lexical analysis used in the programming language domain when that model is applied to other domains. In Chapter III we describe the current model of lexical analysis and why it is inadequate for data objects with embedded commands. In Chapter IV we give specific examples of how difficult the automatic generation of code is with current code-generation tools. In Chapter V we examine other code-generation tools relevant to retagging.

This detailed analysis resulted in the general classification and specification of problem areas in the lexical analysis of data objects with embedded commands. This general classification uniquely informs and empowers the retagging toolset. A major contribution of this work is the design and implementation of a retagging toolset based on this classification which supports the automatic generation of

¹General models for data have been developed for other domains such as databases. None have been developed for electronic documents. SGML is a meta-language for describing encoding schemes. It does not contain a model for electronic documents.
retagging software. In Chapter VI we describe the different tools in the toolset and their components.

In Chapter VII we describe 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 lines of code generated for each line of high-level specification. We also demonstrate that with the toolset, the user must provide only 2 per cent of the C source code required to do the retagging without the toolset. We summarize the contributions and future directions of the work presented in this dissertation in Chapter VIII.
CHAPTER II

A Uniform Representation for the Electronic-Document Domain

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. A concordance program relies on correct tokenization to generate a list of words in a document and display a sentence in which each word appears. 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 [27] 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 chapter 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) [37]. 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 is the distinction between the markup and the text that constitutes the document (see Figure 3). We refer to the markup as *tags*. 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. Some tags break the text into smaller segments. We will call these segmenting tags. Segmenting tags indicate some structural or physical property of the text segment. For example, segmenting tags can indicate page divisions in the document, or they can indicate that a string of text is to be italicized.

There are two methods in which segmenting tags can mark a portion of text. Sometimes both the start and the end of the text segment are explicitly marked by tags. We will call these explicit segmenting tags. For example, '{1' marks the start and '}1' marks the end of a title in the TLG.

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. We will call such tags 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.

For both explicit and implicit segmenting tags, the tags can be further classified as either start tags or end tags. It is understood that explicit tags always come in start/end pairs.

Tags which do not segment the text are nonsegmenting tags. These tags can be viewed as insertions in the text. These tags do not encapsulate a 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, †. Because parentheses are used to represent certain special symbols in the TLG, they are encoded as '[1' and '1]' where they occur in the original text.

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.

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. Section commands are used to mark the start of a new section, and several levels of sections are permitted. For the purpose of this example, we will use chapter, section, and subsection. A new chapter command implies not only the end of the previous chapter, but also the end of the last section in the previous chapter. If that section also had subsections, then the new chapter command would also imply the end of the last subsection of the last section in the previous chapter. Of
course, the first chapter command does not imply the end of any sections because there was no previous 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. This property states that two token classes are disjoint if their intersection is empty. This is very 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, right parentheses can be part of the text or a tag in Scribe [35]. 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 ‘@’. IATEX [22] restricts the use of many characters, most notably ‘\’. A reserved character ensures that the tag token classes and the text token classes are disjoint. Typically, a reserved character indicates the start of a tag, and the end
of a tag is implied by white space or a delimiter character. Sometimes, a reserved character is also used to indicate the end of a tag. 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, divide tags into segmenting and non-segmenting tags, segmenting tags into start and end tags, and non-segmenting tags into symbol and nonsymbol tags. Notice the absence of the implicit and explicit classes of segmenting tags. Because one of the goals in designing the LIF is that all markup be explicit, the class of implicit segmenting tags is 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.

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. Start tags, end tags, and symbols all have direct counterparts in SGML. Using the reference concrete syntax, we assign the following characters for the start and end characters of the LIF tags: S_TAG_START = '<', E_TAG_START = '</>', S_TAG_END = E_TAG_END = '>', SYM_START = '&', and SYM_END = ';'.

SGML does not specify identifiers for start and end tags because these are usually application or data-object dependent. SGML does define the identifier as a string starting with a name start character followed by zero or more name characters. We will define name start characters as letters and name characters as
SGML does include a list of identifiers for the current 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 public entity sets when possible.

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 and assign `<?` to `NSYM_START` and `>` to `NSYM_END`. SGML describes the string between these two delimiters as *system data*, and makes no restrictions on the characters in the string other than the exclusion of the end delimiter, `>`.

This form of SGML markup is to be used for system-specific data and is system dependent. The LIF nonsymbol tag serves as a marker for such data and an identifier similar to those for the other 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. The start and end of each tag is marked by a special character from the set \{<, </, >, &, ;, <?\}. The assignment of the start characters guarantees that segmenting tags are di-

\[
\begin{align*}
\text{start\_tag} & \ ::= \ `' <' \ identifier \ '>', \\
\text{end\_tag} & \ ::= \ `' />' \ identifier \ '>', \\
\text{symbol} & \ ::= \ `' \&' \ identifier \ ';', \\
\text{nonsymbol} & \ ::= \ `' <\?' \ identifier \ '>',
\end{align*}
\]

Figure 6: Concrete Syntax of the LIF Tags
This is an example of a formula that appears as a part of the text:
\begin{math}
\sum_{\text{from } n=1, \text{ to } \infty} \frac{\text{num } (-1)^n \begin{up} n \end{up}}{\text{denom } n}
\end{math}

Note the difference in this display version of the same formula:
\begin{mathdisplay}
\sum_{\text{from } n=1, \text{ to } \infty} \frac{\text{num } (-1)^n \begin{up} n \end{up}}{\text{denom } n}
\end{mathdisplay}

Figure 7: A Scribe Document

tinguishable from nonsegmenting tags, that start tags are different from end tags, and that symbol tags are different from nonsymbol tags. The rules for constructing identifiers will permit each nonsegmenting tag and each segmenting start/end tag pair to have a unique identifier.

2.2.3 A Sample Document

A sample Scribe document is presented in Figure 7. The corresponding LIF version of this document can be seen in Figure 8.
This is an example of a formula that appears as a part of the text:

$$\sum_{n=1}^{\infty} (-1)^n$$

Note the difference in this display version of the same formula:

$$\sum_{n=1}^{\infty} (-1)^n$$

Figure 8: LIF Version of a Scribe Document
CHAPTER III
An Overview of Lexical Analysis

Given a uniform representation of electronic documents, the next task is to convert existing documents to the representation. We define retagging as the process of replacing the nonuniform 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. A regular expression is often used to describe a pattern that specifies the set of strings that constitute a token class. The expression \([A-Za-z][A-Za-z0-9]*\) describes the token class \textit{identifier}\(^1\) in many programming languages.

\(^1\)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 \textit{stocknum}, \textit{X}, or \textit{Ddl2x}. 

16
Finite-state automata are used as a tool to design and model the actual code of the desired lexical analyzer. 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 [1].

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 [23] 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 \texttt{identifier} would be:

\begin{verbatim}
[A-Za-z][A-Za-z0-9]* { yylval=install_id(); return(ID); }
\end{verbatim}
where the action routine, \{yy1val=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.
CHAPTER IV

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.

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 '}'. If the delimiter characters need to be present in the text, there are special commands to represent them, namely '{' and '}'. In this way, there can be no confusion as to which regular expression represents the delimiter token classes and which represents a subset of the text token class.

In Scribe, this is not the case. A regular expression denoting a delimiter, such as ')', does not represent a unique token class. There are seven delimiter character pairs used for delimiting strings in Scribe. They are not reserved for this purpose and may also appear in text.
In terms of lexical analysis, it is necessary to determine if the delimiter character is part of the text or part of a tag. To do this, 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. To be a tag, a left delimiter must be immediately preceded by a Scribe command or the left delimiter must occur in an appropriate location in an argument list. 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.\footnote{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.}

The matching of delimiters requires a more powerful formal model than regular expressions and finite-state automata. Languages with matched delimiters fall into the class of context-free languages whose recognizers are based on push-down automata. These 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.\footnote{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 comparison, we estimate that for Pascal, the ratio would be approximately 1.5:1. The enormity 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.
Table 1: Examples of How to End a Scope

<table>
<thead>
<tr>
<th>Example:</th>
<th>End of Scope is Denoted by:</th>
</tr>
</thead>
<tbody>
<tr>
<td>{\texttt{em text} }</td>
<td>the end of scope character, }</td>
</tr>
<tr>
<td>{\texttt{em some \ bf text} }</td>
<td>a new declaration, \bf \begin{itemize} \texttt{em} \item some \item text \end{itemize}</td>
</tr>
<tr>
<td>\texttt{begin(itemize) \texttt{em}} \item some \item text \texttt{end(itemize)}</td>
<td>the end of an environment, \end{itemize}</td>
</tr>
</tbody>
</table>

Scribe has over 300 tokens that must be recognized and the TLG has over 1100.

4.1.2 Scope Rules

In \texttt{\LaTeX}, there is a subset of commands called \textit{declarations} that affect the way a string of text is printed. The text is said to be in the \textit{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. In fact, the end of a scope can be implied by the occurrence of another command. Table 1 shows several different methods used to indicate the end of the scope of the \texttt{\'} command.

In order to retag a \texttt{\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 \cite{6}. The action routine for just the token class \texttt{\'} contained over 120 lines of code.

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. 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.

An example of a citation record is "a"0059"b"034"c"Leg"x1.³ In the text, the character '"' is a symbol tag representing a double quotation mark. In a citation record, this character is used to indicate and delimit a non-numeric field value.

4.2 General Problems

These problems are not atypical with electronic encoding schemes. In our work with text-formatting languages, we discovered several general categories of lexical analysis problems [26], 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

³This citation record represents author code 0059 = Plato, work code 034 = Laws, preferred abbreviation = Leg, and x-level counter = 1.
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. In some instances, the end of the scope is implied by another tag. This is a common occurrence with implicit segmenting tags. 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. This is the case in Scribe and \LaTeX. 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) [24] all have explicit segmenting tags in which the end tags are not unique.
Both \LaTeX{} and DOSL use `\)' as the end tag for all explicit segmenting tags and permit the nesting of explicit segmenting tags. It has been proven that such languages cannot be recognized using finite-state automata \cite{13}.

### 4.3 Need for a New Toolset

As we have shown, when Lex-like tools are used 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.
CHAPTER V

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 [18], Lex [23], LAWS [9], Flex [29], Rex [11], and Wart [7] 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 [21] 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 γ-GLA [10], a translation rule for identifier would be:

$$[A-Za-z][A-Za-z0-9]* \quad [\text{mkidn}]$$

where mkidn is a library routine that performs the appropriate processing for an identifier. Lexical-analyzer generators that fall into this category are γ-GLA [10], Alex [28], Mkscan [14], and LexAGen [34].

These specialized lexical-analyzer generators for programming languages will not meet our requirements for document databases. 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.
CHAPTER VI

The Retagging Toolset

The Retagging Toolset consists of two major tools: the Replace Tag Tool and the Insert Tag Tool. In this chapter 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 Chapter II.

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. Thus, the user must be an expert in the encoding scheme to be retagged.

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. Two alphabets that always appear in encoding schemes are the text alphabet and the reserved-for-markup alphabet. A third alphabet that sometimes occurs is the ambiguous character alphabet. The text alphabet includes the alphanumeric characters plus some subset of the other characters. This subset usually includes common punctuation characters, but depends on the encoding scheme. 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.

An example of this problem would be 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 document. Based on this perception, a RTT user may classify a citation record as a nonsymbol tag and map a record such as "a"0059"b"034"c"Leg"x1 to the LIF tag <\?tilde>. 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"c"Leg"x1 might be mapped to <\tilde>a"0059"b"034"c"Leg"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 [31]. The Association of American Publishers has developed sets of SGML tags for tabular material [3], mathematical formulas [2], and several types of documents [4]. 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 windows. The general layout of the RTT can be seen in Figure 9.
Figure 9: The Replace Tag Tool
The Command Buttons

There are six buttons at the top of the RTT. (See Figure 10.) These buttons are used to permit file input and output with the user interface, to invoke the Replace_Tag program generator, and to execute a compiled Replace_Tag program.

The Load Button. The Load Button allows the user to retrieve a previously saved specification file and load it into the RTT. When the Load Button is clicked, the user is prompted for a filename. If the user selects a non-existent file, a warning is displayed and the user may cancel the selection of that file.

The Save Button. The Save Button is used to save the retag mapping specification currently residing in the RTT into a file. As with the Load Button, the user is prompted for a filename. If the user selects the name of an existing file, a warning is displayed and the user may cancel the save or choose to overwrite the file.

The Make Button. The Make Button invokes the Replace_Tag program generator. The user is prompted twice for filenames, first for the name of the file containing the specification and then for a name for the executable version of the program.
The Run Button. To execute a compiled Replace_Tag program, the user clicks the Run Button. The user is prompted for filenames three times. The first time the user selects the executable program. The second time the user selects an input document to be retagged. The final time the user selects a file in which to place the retagged document. Once the selections have been made, the selected Replace_Tag program is executed using the designated files.

The Reset Button. Selection of the Reset Button restores the RTT to its original configuration. A warning is displayed if there have been any changes to the data entry windows since the last save. This permits the user to cancel the reset action if desired.

The Quit Button. The Quit Button allows the user to exit the RTT. Like the Reset Button, a warning is displayed if there have been any changes to the data entry windows since the last save. This permits the user to cancel the quit action if desired.

The Alphabet Definition Windows

There are three dialog windows provided for the input of the character alphabets described in Subsection 6.1.1. To enter a new value or to edit the current value, the mouse cursor must be in an input box. These windows are shown in Figure 11.

The first alphabet definition window contains the text alphabet. It has a default value when first instantiated or when reset. This default value contains the alphanumeric characters as well as some common punctuation symbols. The user may add or delete characters to this set. As can be seen in the default string, the user may specify a range of alphanumeric characters using the hyphen character,
e.g., a-z represents all the lower case letters. Since the hyphen has a special meaning it must have a special representation to be included in the text alphabet. This representation is \-. The backslash character will often have special meaning in the RTT. When it is necessary to specify it as a character, the string \ is used.

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. Neither of these windows have initial values and the user must list the characters of each alphabet in the appropriate input box.

The Tag Mapping Editors

There are four tag mapping editors in the RTT, one for each of the four tag classes. These editors can be seen in Figure 12. Each of these windows are basic text editors and have a set of Emacs-like commands for traversing the contents of the window and for editing the text. To enter data in one of these windows the mouse cursor must appear in that window.

The contents of these windows are a list of statements. Each statement must minimally contain a tag description and a LIF identifier. These statements will be described in detail in Subsection 6.1.4.
6.1.3 A Sample Session

When the RTT is first invoked, it comes up with most of its editor windows empty as in Figure 9. The user may enter a new specification directly into the RTT by moving the mouse cursor to the appropriate window and typing. Or, the user may load a previously defined specification into the RTT by clicking the Load Button. In this simulation, the load option is chosen.

After the user clicks the Load Button, the File Selection Dialog Box (FSDB) is displayed, as seen in Figure 13. The user may either click a filename in the menu on the left side of the FSDB or type in a new filename in the entry box on the right side of the FSDB. Since the user wishes to load an existing file, the name
"tlg.rsf" is clicked. After the file is selected, the RTT appears as in Figure 14. The user may now modify this specification.

Once the specification is considered finished, the user must click the Save Button to save the current version of the specification. The user is prompted for a file name via the FSDB again (Figure 15). Here, the previously selected name appears

---

1 A Replace-Tag specification file must have the extension "rsf" attached to its name. The FSDB only displays filenames with this extension when a specification file is required.
Figure 14: The RTT Loaded With a TLG Specification
Figure 15: The RTT With the FSDB Prompting For a Specification Filename

in the entry box and the user may click the Open Button to select this filename. In this case, there already exists a file with the name “tlg.rsf” and the save action would destroy it. For this reason, the RTT presents a warning box to permit the user either to continue or to cancel the save action. This can be seen in Figure 16.

Once the specification has been saved, the user may generate a Replace_Tag program by clicking the Make Button. Once again the FSDB appears to permit the user to select the name of a specification file (Figure 15). Then the user is prompted
Figure 16: The RTT With the Overwrite Warning Box
for a filename for the object code of the Replace_Tag program (Figure 17).²

In order to run the Replace_Tag program that was just generated and compiled, the user clicks the Run Button. The user is then prompted to select a Replace_Tag program (Figure 18), an input document file (Figure 19), and an output file for the LIF version of the input document (Figure 20). In this instance, the document called “tlg.doc” will be converted and the LIF version will be written to “tlg.out”

²Object code files are created with the filename extension “.rx”.

Figure 17: The RTT With the FSDB Prompting For an Object Filename
Figure 18: The RTT With the FSDB Prompting For a Replace_Tag Program

for later viewing.

Figure 21 presents the situation when the user clicks the Reset or the Quit Button while there are changes to the specification that have not been saved. The Changes-Lost Warning Box permits the user to cancel the Reset or Quit action.
Figure 19: The RTT With the FSDB Prompting For an Input Document
Figure 20: The RTT With the FSDB Prompting For an Output Filename
Figure 21: The RTT With the Changes-Lost Warning Box
6.1.4 The Tag Mapping Statements

The tag mapping windows require a special type of statement to be entered. These statements define the mapping of the existing tags to their LIF counterparts. Minimally these statements require a regular expression for the existing tag and a LIF identifier for the LIF tag. In this first prototype, the rules for constructing regular expressions are those used for Lex [23].

In Figure 22, there are several examples of simple tag mapping statements. Statement (1) defines a symbol tag representing the character †. This statement would appear in the Symbol Tags Editor Window. The first string in the statement is a regular expression representing the tag string for the dagger symbol in the TLG. The second string is the LIF identifier for the symbol tag &dagger;. Statements (2) and (3) would appear in the Implicit Segmenting Tags Editor Window. Statement (2) equates the Greek font start tag in the TLG with its LIF equivalent <gr>. Statement (3) defines the end-of-page tag in the TLG. Here the slash character preceding the LIF identifier notifies the Replace.Tag program generator that this is an end tag, </page>. Statement (4) represents an explicit segmenting start/end pair and would appear in the Explicit Segmenting Tags Editor Window. The first two strings are the regular expressions for the start and end tags for a title in the TLG, respectively. The third string is the mutual LIF identifier.

The user may attach special clauses to these simple tag mapping statements in order to define context scopes. A context scope is a segment of a document where the tags are context-sensitive. Examples of these types of statements can be seen in Figure 23. Each of the example statements defines an explicit segmenting start/end tag pair.
Statements (5) and (6) are from the specification for the TLG. Statement (5) defines a citation record in the TLG. The tag signalling the end of a citation record is the newline character (\n). Since the newline character is normally part of the text alphabet, this use of the character as a tag is context-sensitive. For this reason, the newline character must be specified in a tag mapping statement with a context clause. Breaking statement (5) into its components, "\n" is the start tag, "\n" is the end tag and "tilde" is the LIF identifier. The key word ":DEFINES:" indicates a context scope starting with the character "\n" and ending with a newline character. This context scope is assigned the name "TILDE".

Statement (6) defines two tag strings, a" and ", that appear in the TILDE scope. The key word :IN: indicates that this interpretation of the tag strings is only valid inside a citation record. Outside of a citation record these strings will be interpreted as text or a different tag string. Inside the citation record, the
strings will be interpreted as the start and end tags of a subfield in the citation record. Statements (5) and (6) permit the multiple interpretations of the tag strings depending on the context in which they appear.

Statements (7) and (8) are from the specification for the DOSL. Here the context scope is defined by the start tag in statement (7) and the end tag in statement (8). The set of tags known as mnemonics in the DOSL have optional comment fields inside them. These comments are delimited by the characters colon and period which are usually part of the text alphabet. Statement (7) defines the start and end tags for an AD mnemonic with a comment field. The key word :START: indicates that it also defines the start of a context scope called MNEM. Statement (8) defines the colon and period characters as the start and end tags for the comment field when they appear in the MNEM context scope. The key word :END: indicates that the context scope, MNEM, ends with the first period after the colon character. The colon and period remain text characters outside of this scope.

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 will 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

In this case, the user must be aware of which implicit tags require the insertion of a mate tag. Thus, the user must be familiar with the LIF tag set used during the replace-tag phase of retagging.

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. An example of this is:

\[\text{<gr20>very large text<gr>text} \]
\[\text{in a more normal font size} \]

The appropriate Insert_Tag program will produce:

\[\text{<gr20>very large text</gr20><gr>text} \]
\[\text{in a more normal font size</gr> \]

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. An example is:

\[\text{<hd>heading text <folref>1r</hd>} \]

which becomes:
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. An example is:

```
<gr20>very large text<gr>text
in a more normal font size</page>
more text - short page</page>
```

which becomes:

```
<page><gr20>very large text</gr20><gr>text
in a more normal font size</page>
<page>more text - short page</page></gr>
```

Note how the `<gr>` tag segment starts in the middle of the first page and includes all of the second page. It is not entirely contained within either page. The text segments for the pages overlap with the text segment of the `<gr>` tag.

The ITT is based on the view of the document as a string containing properly nested delimiters. It has no problems with contiguous or nested tags, since they fit this model. 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

The ITT has two major areas of activity: the command buttons and the tag definition window. The general layout of the ITT can be seen in Figure 24.

The Command Buttons

The command buttons of the ITT are the same as those for the RTT. They provide the same functionality as they do in the RTT. After clicking a command button in the ITT, the user is presented with the same scenario as when the same button is clicked in the RTT.

The Tag Definition Window

The tag definition window of the ITT is similar to the tag mapping windows of the RTT. It is a basic text editor with a set of Emacs-like commands for traversing the contents of the window and editing the text.

The contents of this window are a list of statements defining the implicit seg-
menting tags that require mate tags to be inserted. Minimally, each statement must contain either a start or an end tag which requires a mate tag to be inserted. These statements will be described in more detail in Section 6.2.4.

### 6.2.3 A Sample Session

When the ITT is first invoked, it comes up with its editor window empty as in Figure 24. As with the RTT, the user may either enter a new specification directly into the ITT by typing in the editing window or the user may load a previously defined specification by clicking the Load Button. The ITT loaded with a specification can be seen in Figure 25.

Further simulation of a sample session with the ITT is similar to that for the RTT. The command buttons function the same way and the same dialog boxes are presented to the user. The user goes through the same process to create a specification, save it, make an Insert_Tag program, run an Insert_Tag program, and quit.
Figure 26: Tag Definition Statements for Font Tags in the TLG

6.2.4 The Tag Definition Statements

The Insert Tags Editor requires a special type of statement to be entered. These statements define which implicit segmenting tags are to be considered and the relationships between tagged text segments, if any.

Figure 26 contains the specification for two of the font tags in the TLG. Here it is sufficient to simply list the tags since the associated tagged text segments are contiguous.

The TLG also has page format tags that mark the end of the pages in the document and the end of the columns within a page. The specification of the page and column tags in the TLG is presented in Figure 27. Both the page and column tags are implicit segmenting tags that require matching start tags to be inserted. In this case, it is necessary to indicate that the segments tagged as columns are nested within the segments tagged as pages. The :IN: clause conveys this information. Since the page start tag is implied, only the page end tag is specified in the :IN: clause.

Figure 28 shows the specification for the folio reference tag in the DOSL. Recall that the folio reference tag only appears within the scope of a heading tag. The folio reference tag is an implicit segmenting tag, whereas the heading tag is an explicit segmenting tag. Since the heading tag is in another tag class, it is necessary to
specify both the start and end tag in the :IN: clause so that the ITT can generate the correct code.

As can be seen in Figures 27 and 28, it is necessary to list the exterior nesting tags prior to their use in an :IN: clause. It is possible to define explicit segmenting tags with an :IN: clause. It is also possible to multiply define tags with different :IN: clauses, so that they are permitted to appear at different levels of a hierarchy.

6.3 Functionality of the Toolset

Both the RTT and the ITT provide basic file-access functions through the use of command buttons. These functions include loading a specification into the tool, saving a specification in a file, creating a Replace_Tag or Insert_Tag program from a selected specification, and running said program on selected input files.

Each tool provides an environment in which the user may structure a solution to the different phases of retagging. The RTT prompts the user for the information necessary to do the initial tag mapping in the retagging process. The RTT provides support for the entry of the character alphabets and the tag mappings. The ITT
provides the user with the facility to enter an abstract specification of a hierarchy of implicit segmenting tags requiring the insertion of mate tags. Both tools permit the user to work at a higher conceptual level to solve the problem, eliminating the need to be concerned with the details of the implementation.

6.4 Code Generated by the Toolset

The RTT and ITT generate Lex and Yacc specifications for the generated Replace_Tag and Insert_Tag programs. These specifications are processed by Lex and Yacc to create C functions that are then compiled to yield the required programs. In this section we describe the Lex and Yacc specifications that are generated by the Toolset. For this purpose, we use a partial retagging specification for Scribe documents.

6.4.1 Code Generated by the RTT

Figure 29 contains the RTT loaded with a partial specification for Scribe. This specification utilizes all of the components of the RTT.

Code for the Alphabets

The text alphabet definition generates separate Lex rules for the alphabetic, numeric, special, and whitespace characters. The action routines make a copy of the string by calling the makestr() function and return a predefined token name to the corresponding Yacc routine. The generated Lex rules for the text alphabet specified in Figure 29 can be seen in Figure 30.

The predefined token names become alternatives to a production defining the text of the document. Every action routine consists of a call to the function
Figure 29: The RTT Loaded With a Scribe Specification

```plaintext
[A-Za-z]+ {makestr(); return(CDATA);}
[0-9]+ {makestr(); return(CDATA);}
[!#$%&++,-./:;=?\[\]\^_`{|}]+ {makestr(); return(CDATA);}
[ \t] {makestr(); return(WHITESPACE);}
\n {makestr(); linecount++; return(WHITESPACE);}
```

Figure 30: Lex Rules Generated for the Text Alphabet
concat(), which concatenates the designated tags or text, and a call to the function nfree(), which frees the space occupied by the copies of the text strings. The pseudo-variables ‘$$’, ‘$1’, and ‘$2’ are used to reference the text strings. The generated Yacc rules for the text alphabet can be seen in Figure 31.

For each character in the reserved alphabet, a special Lex rule is generated. The action routine first prints a warning message to the user and then treats the reserved character like a text character. The message indicates to the user that the RTT specification may not be complete. Because the reserved character is maintained in the text of the document, the user can quickly locate it and confirm if there is a problem. No additional Yacc rules are required, since the reserved characters are considered as text. The generated Lex rule for the <D character is presented in Figure 32.

For each character in the ambiguous character alphabet, a Lex rule with an action routine similar to that for the text characters is generated. Because ambiguous characters may appear as tags, a stylized token name is generated for each
\}
    fprintf(stderr, "Unresolved reserved character %s in line %d.\n",
        yytext, linecount);
    makestr(); return(CDATA);}

Figure 32: Lex Rules Generated for the Reserved Alphabet

"    {makestr(); return(TOKEN24);}
'    {makestr(); return(TOKEN25);}
(    {makestr(); return(TOKEN26);}
)    {makestr(); return(TOKEN27);}

Figure 33: Lex Rules Generated for the Ambiguous Alphabet

ambiguous character rather than using the predefined token names for text. A se­
lection of Lex rules for the ambiguous characters specified in Figure 29 is displayed
in Figure 33.

In order that the ambiguous characters may be recognized as text characters,
their token names are added as alternatives to the text production illustrated in
Figure 31. The action routines for these additional productions are the same as
the action routines for the productions currently in the figure.

Tag Mapping Code

The code generated for the statements in the Tag Mapping Windows is dependent
on the clauses that appear in the statements. We consider statements without
clauses first.

The statements in the Symbol Tags Editor, the Nonsymbol Tags Editor, and
the Implicit Segmenting Tags Editor generate essentially the same code. The only difference is the delimiter characters used to form the LIF tags. Each of the regular expressions are assigned a stylized token name in the Lex specification. The LIF tags are replaced for the Scribe tags in the Yacc specification. Examples of the generated Lex rules can be seen in Figure 34 and examples of the generated Yacc rules can be seen in Figure 35.

Statements without clauses in the Explicit Segmenting Tags Editor generate similar Lex and Yacc rules. The primary difference is that there are two regular expressions per statement. Thus two Lex rules may be generated and the Yacc production will contain two token names. An example would be the specification of the \subsection command in Figure 29. This statement generates the Lex rules in Figure 36 and the Yacc rules in Figure 37.

Statements with clauses in the Tag Mapping Editors are used to create and manage start conditions in the Lex specification. Start conditions indicate that a change in the scanning algorithm has occurred and they are used for context changes. A :START: clause defines a start condition and a regular expression that

\begin{verbatim}
\texttt{\textbackslash \texttt{Infnty}} \{ \texttt{makestr(); return(TOKEN0);} \}
\texttt{\textbackslash \texttt{device\([0-9a-zA-Z]+\)}} \{ \texttt{makestr(); return(TOKEN2);} \}
\texttt{\textbackslash \texttt{make\(\texttt{article}\)}} \{ \texttt{makestr(); return(TOKEN4);} \}
\texttt{\textbackslash (\textbackslash n)(\textbackslash n)+} \{ \texttt{makestr(); return(TOKEN5);} \}
\end{verbatim}

Figure 34: Lex Rules Generated for Simple Statements in the Symbol, Nonsymbol, and Implicit Segmenting Tags Editors

\footnote{The Yacc rules generated for these statements are the same as the Yacc rules for statements without clauses.}
nonsegmenting : TOKEN0
    {
        $$\text{trans} = \text{concat}(1,"\&infin;")
    }
    \| TOKEN2
    {
        $$\text{trans} = \text{concat}(1,"<\?device>")
    }
    \| TOKEN4
    {
        $$\text{trans} = \text{concat}(1,"<\text{article}>")
    }
    \| TOKEN5
    {
        $$\text{trans} = \text{concat}(1,"<\text{p}>")
    }

Figure 35: Yacc Rules Generated for Simple Statements in the Symbol, Nonsymbol, and Implicit Segmenting Tags Editors

\]
\] \{makestr(); return(TOKEN7);\}
\subsection{}
\{makestr(); return(TOKEN8);\}

Figure 36: Lex Rules Generated for Simple Statements in the Explicit Segmenting Tags Editor

segmenting : TOKEN8 textortagsstart TOKEN7
    {
        $$\text{trans} = \text{concat}(3,"<\text{st2}>","\text{\$2\text{.trans}}","</\text{st2}>")
        \text{nfrees}(1,\text{\$2\text{.trans}})
    }

Figure 37: Lex Rules Generated for Simple Statements in the Explicit Segmenting Tags Editor
\@begin\{math\} \@end\{math\} \textit{f} \text{ :DEFINES: } \textit{MATH} \\
\@over\{ \} \textit{fr} \text{ :IN: } \textit{MATH} \\
"num <" "\>" [ ]* \textit{nu} \text{ :IN: } \textit{MATH} \\
"denom <" \textit{de} \text{ :IN: } \textit{MATH}

Figure 38: Statements With Clauses

invokes that start condition. An :END: clause defines a regular expression that terminates the designated start condition. A statement in the Explicit Segmenting Tags Editor with a :DEFINE: clause acts as both a :START: and an :END clause. The first regular expression invokes the start condition and the second regular expression terminates the start condition. The :IN: clause is used with those statements defining regular expressions that are to be used only with the specified start condition.

In Figure 38, four statements have been selected from the Explicit Segmenting Tags Editor to illustrate the Lex rules generated from statements with clauses. In Figure 39 the Lex rule for start tags in statements with a :DEFINE: clause is shown. The same rule is generated for tags in statements with a :START: clause. The BEGIN MATH statement indicates that the rules for the start condition MATH should be used until further notice. Because contexts in encoding schemes may nest and Lex does not have the functionality to maintain more than one start condition, a stack has been implemented for this purpose.

In Figure 40 the Lex rule for end tags in statements with :DEFINE: clauses is presented. The same rule is generated for tags in statements with :END: clauses. The stack is examined to determine whether a different start condition should be
invoked again or if the primary Lex rules should resume.

The Lex rules for the three statements in Figure 38 with :IN: clauses are shown in Figure 41. The action routines are similar to those for tags specified in statements without clauses. The difference is that the regular expression is prefaced by the start condition. For tag strings not containing a reserved character, the start condition is essential for the correct tokenization of the data stream.

```plaintext
<Math>over\{  {makestr(); return(TOKEN15);} 
<Math>\}    {makestr(); return(TOKEN16);} 
<Math>"num <" {makestr(); return(TOKEN17);} 
<Math>"[ ]*" {makestr(); return(TOKEN18);} 
<Math>"denom <" {makestr(); return(TOKEN19);} 
<Math>\}>    {makestr(); return(TOKEN20);} 
```

Figure 41: Lex Rules Generated for Tags in Start Conditions
Figure 42: Additional Yacc Rules Defining a Document

Other Code

The previous paragraphs describe all of the types of rules generated by the RTT for the Lex and Yacc specifications. The Yacc specification requires additional rules to describe the content of the document and the content of the explicit segmenting tags. These productions are displayed in Figure 42.

6.4.2 Code Generated by the ITT

Figure 43 presents the ITT loaded with a partial specification for Scribe. This specification is used to insert the missing article end tag, </article>, and the missing paragraph start tags, <p>. The specification describes a hierarchy of nesting tags. The article start tag, <article>, is at the highest level of the hierarchy and, with its inserted mate tag, encapsulates the other tags in the specification.
The section title tags, \(<\texttt{st1}\)> and \(<\texttt{st1}\>\), and the paragraph end tag, \(<\texttt{/p}\>\), are at the second level of the hierarchy and delimit text strings inside of the article tags. Any other tags that may appear in the document are ignored and are treated as text.

The Lex Rules

The Lex rules for the tags in this specification are very simple. The corresponding action routines only have to return a token name. Default text rules are also generated. Figure 44 displays the Lex rules for the specified tags and the default rules for text.

The Yacc Rules

The Yacc specification describes a document as a hierarchy of tagged strings. At different levels of the hierarchy, different tagged strings may appear. In the example from Figure 43, \(<\texttt{article}\>\) is defined at the uppermost level of the hierarchy. The tags \(<\texttt{st1}\>\), \(<\texttt{st1}\>\), and \(<\texttt{/p}\>\) are described at the second level of the hier-
archy. The lowermost level is always defined as text. An excerpt from the Yacc specification for this example is given in Figure 45.

The action routines for all productions consist of a call to the function concat() to gather the component strings of the document and a call to the function nfree() to free the space used by the component strings. The productions labeled (1) and (3) illustrate how the missing tags are inserted. The production labeled (2) shows how the specified explicit segmenting tags are preserved.

6.4.3 Summary

The RTT specification generates a Lex and a Yacc specification that specifies 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. Because contexts may nest in document encoding schemes and Lex does
document : textaglev1plus
;
textaglev1 : tagslev1
| textplus
;
(1) tagslev1 : TOKENO textaglev2plus
{
  $$\text{trans} = \text{concat}(3,"<article>",$
  $2.\text{trans},"</article>");$}
nfree(1,$2.\text{trans});
}
textaglev2 : tagslev2
| textplus
;
(2) tagslev2 : TOKEN1 textaglev3plus TOKEN2
{
  $$\text{trans} = \text{concat}(3,"<st1>";2.\text{trans},"</st1>");$}
nfree(1,$2.\text{trans});
}
(3) | textaglev3plus TOKEN3
{
  $$\text{trans} = \text{concat}(3,"<p>";1.\text{trans},"</p>");$}
nfree(1,$1.\text{trans});
}
textaglev3 : textplus
;

Figure 45: Some Yacc Rules for the ITT Specification
not support the nesting of start conditions, a stack mechanism is implemented to overcome this deficiency in Lex. 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.
CHAPTER VII

Experiences With the Toolset

The Retagging Toolset was used to retag a wide variety of document encoding schemes from the humanities, linguistics, and text-formatting domains. In this chapter, we describe each of the encodings, the success of the generated programs to retag the existing markup, and the degree to which the toolset decreases the amount of work that the user must perform. As a metric for the latter, we will compare the amount of work the user must perform using the Toolset (number of Toolset statements plus lines of C code) with the amount of work the user would have to perform using Lex and Yacc alone (number of rules plus lines of C code). For the latter, we will use the Lex and Yacc specifications for the automatically generated Replace_Tag and Insert_Tag programs. These specifications would have to be manually written if the Retagging Toolset was not used. A summary table containing the productivity statistics can be found in Section 7.7.

7.1 The Thesaurus Linguae Graecae

7.1.1 The Encoding Scheme

The encoding scheme for input manuscripts to the Thesaurus Linguae Graecae (TLG) [37] is composed of two types of markup: Beta codes and citation records. The Beta codes have a simple structure, consisting of a reserved character followed
Table 2: Reserved Characters in the TLG

<table>
<thead>
<tr>
<th>Characters</th>
<th>Class Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
<td>Greek Fonts</td>
</tr>
<tr>
<td>&amp;</td>
<td>Roman Fonts</td>
</tr>
<tr>
<td>%</td>
<td>Punctuation</td>
</tr>
<tr>
<td>&quot;</td>
<td>Quotation Marks</td>
</tr>
<tr>
<td>@</td>
<td>Page Formats</td>
</tr>
<tr>
<td>[ ]</td>
<td>Brackets</td>
</tr>
<tr>
<td>&lt; &gt;</td>
<td>Quasi-Brackets</td>
</tr>
<tr>
<td>{ }</td>
<td>Non-Text</td>
</tr>
<tr>
<td>#</td>
<td>Text Symbols</td>
</tr>
<tr>
<td>-</td>
<td>Citation Record</td>
</tr>
</tbody>
</table>

by an optional number, where the reserved characters may never appear as text characters. The reserved characters are listed in Table 2 with their class meanings. The citation records are indicated by a tilde as the first character of the record. A citation record is composed of one or more fields, where each field contains an identifier letter (a, b, c, v, w, x, y, or z) and a value. The value can either be numeric, a string enclosed in quotes, or null.

A short excerpt from a TLG document can be seen in Figure 46. Lines (1), (3), (5), and (12) are citation records. Line (2) contains examples of a page format code (0), non-text codes ({1 and }1), and font codes ($20 and $). Line (4) also contains page format codes and non-text codes. Lines (6) and (13) contain page format codes (0 and 01, respectively).

7.1.2 Retagging Documents Encoded for the TLG

The RTT was used to build a Replace_Tag program to process one complete TLG document. A specification was derived to replace all Beta codes in the document.
Figure 46: Excerpt From a TLG Document
Table 3: Classification of Tags in the TLG

<table>
<thead>
<tr>
<th>Description</th>
<th>Examples</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>punctuation</td>
<td>%</td>
<td>symbol</td>
</tr>
<tr>
<td>brackets</td>
<td>[ ]</td>
<td>symbol</td>
</tr>
<tr>
<td></td>
<td>[1 ] 1</td>
<td>symbol</td>
</tr>
<tr>
<td></td>
<td>[2 ] 2</td>
<td>symbol</td>
</tr>
<tr>
<td>quotation marks</td>
<td>&quot;3</td>
<td>symbol</td>
</tr>
<tr>
<td>page formats</td>
<td>@</td>
<td>nonsymbol</td>
</tr>
<tr>
<td></td>
<td>@1</td>
<td>implicit segmenting</td>
</tr>
<tr>
<td></td>
<td>@2</td>
<td>implicit segmenting</td>
</tr>
<tr>
<td>fonts</td>
<td>$</td>
<td>implicit segmenting</td>
</tr>
<tr>
<td></td>
<td>$20</td>
<td>implicit segmenting</td>
</tr>
<tr>
<td>non-text</td>
<td>{1 } 1</td>
<td>explicit segmenting</td>
</tr>
<tr>
<td>citation records and their fields</td>
<td>&quot; newline</td>
<td>explicit segmenting</td>
</tr>
<tr>
<td></td>
<td>a</td>
<td>implicit segmenting</td>
</tr>
<tr>
<td></td>
<td>or a&quot; &quot;</td>
<td>explicit segmenting</td>
</tr>
</tbody>
</table>

and to recognize the citation records and their subfields. The classification of the pertinent TLG tags is listed in Table 3. As can be seen in several of the table entries, tags starting with the same reserved character need not fall in the same tag class. Another phenomenon depicted in the table is the use of a whitespace character, newline, as part of the markup.

The current RTT specification for the TLG includes 15 Beta codes, the citation record, and six citation field identifiers. Although the number of Beta codes in the specification represents a small sample of the over 1100 possible tags, the simple structure of the Beta code tags guarantees their easy inclusion in future specifications. The current RTT specification contains 29 statements describing the two alphabets and the aforementioned tags. Figure 47 contains the TLG document of Figure 46 after processing by the Replace_Tag program.
(1) \textless \text{tilde} \textgreater 0003 \textless /\text{tilde}\textgreater \\
(2) \textless ?\text{ind}\textgreater ?\text{ind}\textgreater ?\text{ind}\textgreater ?\text{ind}\textgreater ?\text{ind}\textgreater ?\text{ind}\textgreater ?\text{ind}\textgreater ?\text{ind}\textgreater ?\text{ind}\textgreater ?\text{ind}\textgreater title\textgreater gr20 \*
*Q*K*U*D*I*D*U*U*U</w>
(3) \textless \text{tilde}\textgreater 1</xf> t</xf>/\text{tilde}\textgreater \\
(4) \textless ?\text{ind}\textgreater ?\text{ind}\textgreater ?\text{ind}\textgreater ?\text{ind}\textgreater ?\text{ind}\textgreater ?\text{ind}\textgreater ?\text{ind}\textgreater ?\text{ind}\textgreater ?\text{ind}\textgreater ?\text{ind}\textgreater ?\text{ind}\textgreater ?\text{ind}\textgreater ?\text{ind}\textgreater ?\text{ind}\textgreater title\textgreater *I*T*W*N *A</w>
(5) \textless \text{tilde}\textgreater xfn1</tilde>
(6) ?\text{ind} \*QOUKUDI/DHS \*)AQHNAI=OS CUNE/GRAYE TO\N PO/LEMON TW=N \*PELO-
(7) PONNHSI/WN KAI\ *=AQHNAI/ WN, W(S E)POLE/MHSAN PRO\S \A\LLH/LOUS,
(8) A)RCA/HEROS EU\S QAQISTAME/NOU KAI\ E)LPI/SAS ME/GAN TE \E)/SEQUI
(9) KAI\ A)CIOLOGW/TATON TW=N PROGEGENHME/NWN, TEKMAIRO/HEROS \O(/TI
(10) A)KMA/ZONTES TE H)=|SAN E S AU\TO\N A)MFO/TEROI PARASKEV=| 
TH|=|PA/SH|
(11) KAI\ TO\ A)/LLO *(ELLHNIKO\N O(RW=N CUNISTA/HEROS PRO\S \ E(KATE/ROUS,
(12) \textless \text{tilde}\textgreater xfn</tilde>
(13) TO\ ME\N EU\QU/S, TO\ DE\ KAI\ DIAOODU/HEROS. KA/NHSAS GA/R AU</TH>/page>

Figure 47: TLG Document After Processing by Replace_Tag Program
The current RTT specification for the TLG includes several implicit segmenting tags that require the insertion of mate tags. The ITT must be used more than once in order to build several Insert_Tag programs since these tags have overlapping text segments associated with them.

The first subset of implicit segmenting tags are the font tags. In the current RTT specification, there are two such tags and the ITT specification contains two lines listing the tags. Both of these tags are start tags and require matching end tags to be inserted.

The second subset of implicit segmenting tags includes two page-format tags, the page end tag and the column end tag. They both require start tags to be inserted and the column tags must nest inside of the page tags. The ITT specification for this set of implicit segmenting tags contains two lines listing the tags and their hierarchical relationship.

The last subset of implicit segmenting tags requiring mate tags are the citation record field identifiers that have numeric or null values. In these instances, the end of the value field is implied by the start of a new field or the end of the citation record. The ITT specification for this set of tags has thirteen statements describing the six nonstring-value identifiers, the six string-value identifiers, and their hierarchical relationship with the citation record tags.

The three generated Insert_Tag programs can be run in sequence to insert the missing mate tags. The result can be seen in Figure 48 where the TLG document of Figure 47 has been processed by the three Insert_Tag programs.

The generated Replace_Tag program successfully replaced all of the original markup strings with the specified LIF equivalent. The generated Insert_Tag pro-
Figure 48: TLG Document After Processing by Insert_Tag Programs
grams correctly inserted the missing font end tags, the missing column and page start tags, and the missing citation field end tags. The end result is a fully marked LIF document.

The RTT and ITT both automatically generate Lex and Yacc specifications for the respective Replace_Tag and Insert_Tag programs. Whereas the user had to specify a total of 46 statements in the toolset, the generated Lex and Yacc specifications contained 192 rules and 500 lines of C code. This represents a savings in coding effort of 15.0:1.

7.2 The Dictionary of the Old Spanish Language

7.2.1 The Encoding Scheme

The encoding scheme for input manuscripts to the Dictionary of the Old Spanish Language (DOSL) [24] is moderately sophisticated. It utilizes the concept of properly nested delimiters. The majority of the tags are called mnemonics and have the structure:

\[
\{ \text{name} \text{ optional\_remark . text\_or\_markup} \}
\]

A mnemonic may appear inside another mnemonic as part of the text\_or\_markup as long as the inner mnemonic ends before the outer one. A list of the reserved characters and their meanings can be seen in Table 4.

A short excerpt from a DOSL document can be seen in Figure 49. Line (1) contains a folio tag enclosing a folio number. Lines (2), (3), (5), (6) and (7) include mnemonic tags. The heading mnemonic in line (2) contains an expanded abbreviation, \texttt{<IBRO>} and an old folio reference. The column boundary mnemonic
Table 4: Reserved Characters in the DOSL

<table>
<thead>
<tr>
<th>Characters</th>
<th>Meaning or Function:</th>
</tr>
</thead>
<tbody>
<tr>
<td>{ }</td>
<td>enclose mnemonics</td>
</tr>
<tr>
<td>[ ]</td>
<td>enclose folio numbers and insertions</td>
</tr>
<tr>
<td>&lt; &gt;</td>
<td>enclose expanded abbreviations</td>
</tr>
<tr>
<td>( )</td>
<td>enclose deletions</td>
</tr>
<tr>
<td>\</td>
<td>precedes old folio number in heading</td>
</tr>
<tr>
<td>&quot;</td>
<td>indicates scribal insertion or deletion, used with [ ] and ( )</td>
</tr>
<tr>
<td>*</td>
<td>indicates reconstructed text, used with [ ]</td>
</tr>
<tr>
<td>=</td>
<td>indicates placement of pictures, part of mnemonic name</td>
</tr>
<tr>
<td>!</td>
<td>signals divisions in diagrams</td>
</tr>
<tr>
<td>'</td>
<td>previous character is superscript</td>
</tr>
<tr>
<td>+</td>
<td>continuation character</td>
</tr>
</tbody>
</table>

that starts in line (3) contains lines (4) through (15) of the document. The diagram mnemonic in line (5) has a remark field. The rubric mnemonic in line (6) contains the superscript tag indicating that the 's' of the word 'Las' was written as a superscript in the original manuscript. The initial mnemonic in line (7) has no content. In line (14) is a scribal deletion, (^es).

7.2.2 Retagging Documents Encoded for the DOSL

The RTT was used to build a Replace_Tag program to process one complete document and excerpts from several other documents encoded for the DOSL. The classification of the DOSL tags can be seen in Table 5. The current RTT specification for the DOSL includes 23 different mnemonic tags, two types of deletion tags, and four types of insertion tags as well as the continuation, superscript, fo-
Amatalo en uinagre & con esto se desfara todo.

Dixo aristotil. tauro es signo de tierra & su natura es fria & seca. & es signo fixo & conuiene le delas cosas miner-

de tauro assi cuemo el fierro & su semeiante. Et cuando quisieres fazer ymagen(~es) pora af-
fimar estado o de alguna uilla o de algun lo-

Figure 49: Excerpt From a DOSL Document

The user was required to enter 61 lines of specification for the 36 tags and 2 alphabets. The discrepancy between the number of specification lines and the number of tags is the result of the need to specify the mnemonic tags twice: once for when there is no remark field, as in line (6) of Figure 49, and again for when there is a remark field, as in line (5) of Figure 49. Figure 50 contains the DOSL document of Figure 49 after processing by the Replace_Tag program.

The RTT specification included two implicit segmenting tags requiring the insertion of mate tags. The folio reference tag is a start tag. It indicates the beginning of an old folio number in a heading and requires the insertion of a
Table 5: Classification of Tags in the DOSL

<table>
<thead>
<tr>
<th>Description</th>
<th>Examples</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>continuation</td>
<td>+</td>
<td>nonsymbol</td>
</tr>
<tr>
<td>non sequitur</td>
<td>[....]</td>
<td>nonsymbol</td>
</tr>
<tr>
<td>folio reference</td>
<td>\</td>
<td>implicit segmenting</td>
</tr>
<tr>
<td>superscript</td>
<td>\</td>
<td>implicit segmenting</td>
</tr>
<tr>
<td>folio</td>
<td>[fol. ]</td>
<td>explicit segmenting</td>
</tr>
</tbody>
</table>
| mnemonic               | For example:  
{HD. }  
or  
{HD }  | explicit segmenting      |
| optional remark field  | : .      | explicit segmenting      |
| expanded abbreviation  | < >      | explicit segmenting      |
| deletions              | ( )      | explicit segmenting      |
|                       | ( )      |                          |
| insertions             | [ ]      | explicit segmenting      |
|                       | [~ ]     |                          |
|                       | [^2 ]    |                          |
|                       | [* ]     |                          |

(1) <fol> 26v</fol>
(2) <hd> L<exp>IBRD</exp> <folref> xxiii</folref> <hd>
(3) <cb2>
(4) amatelo en unagre & con esto se desfara todo.
(5) <diagl><com> zodiac chart</com></diagl>
(6) <rub> Las</sup> ymagenes de tauro.</rub>
(7) <in4></in4> Diremos loque a onel capitulo seg-
(8) undo que es la figura de tauro.
(9) Dixo aristotil. tauro es signo de
(10) tierra & su natura es fria & seca. &
(11) es signo fixo & conuiene lo delas cosas miner-
(12) ales las dela natura; que semeian ala natura
(13) de tauro assi cuemo el fierro & su semeiante. Et
(14) quando quiesieres fazer ymagen<sdel>es</sdel> pora af-
(15) firmar estado o de alguna uilla o de algun lo-</cb2>

Figure 50: DOSL Document After Processing by Replace_Tag Program
amatalo en uinagre & con esto se desfara todo.
Dixo Aristotil. taura es signo de

Las ymagenes de taura.

Diremos loque a en el capitulo seg-
vido que es la figura de taura.

Dixo aristotil. taura es signo de

tierra & su natura es fria & seca. &
es signo fixo & conuiene le delas cosas miner-

ales las dela natura; que semeian ala natura
de taura assi cuemo el fierro & su semeiante. Et

quando quisiere fazer ymagen pora af-

firmar estado o de alguna uilla o de algun loc-

Figure 51: DOSL Document After Processing by Insert_Tag Program

matching end tag. The superscript tag is an end tag. It indicates that the character preceding the superscript character is raised above the normal text and requires a start tag to be inserted.

The ITT was used to build an Insert_Tag program to insert the missing folio reference end tags. The user had to write two statements to describe the folio reference tag and its context. Figure 51 contains the DOSL document of Figure 50 after processing by the Insert_Tag program.

The ITT could not be used to insert the missing superscript start tags. The ITT uses a model for the text of a document that collects the longest possible alphabetic string at any one time. The superscript tag requires a different model of text: one that permits the isolation of individual characters. This view of
the text is not common among the different encoding schemes. For this reason, the superscript tags require a special post-processor program to resolve them and convert the document into a fully tagged LIF document. This special program must be provided by the user. A Lex specification for such a special program contained 16 lines of C code.

The generated Replace_Tag program successfully recognized all of the markup strings in the test documents and replaced the markup with its specified LIF equivalent. This represents a 100 per cent success rate in recognizing and replacing the specified markup. Also, the generated Insert_Tag program for the folio reference tags inserted the missing end tags correctly. This left only the superscript start tag to be inserted by other means.

The RTT and ITT both automatically generate Lex and Yacc specifications for the respective Replace_Tag and Insert_Tag programs. The user had to specify a total of 63 statements in the Toolset and the 16 lines of C code for the superscript postprocessor. The generated Lex and Yacc specifications contained 165 rules and 445 lines of C code. This represents a savings in coding effort of 7.9:1.

7.3 The Lancaster-Oslo/Bergen Corpus

7.3.1 The Encoding Scheme

The Lancaster-Oslo/Bergen (LOB) Corpus [16, 17] is a million word collection of British English text. It consists of 500 text samples of approximately 2000 words. The encoding scheme for the horizontal version of the Corpus has two phases of markup: codes from the original transcription of the texts to electronic form and word-class tags added later.
Table 6: Reserved Characters in the LOB

<table>
<thead>
<tr>
<th>Characters</th>
<th>Meaning or Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>special symbol, diacritical mark, or formatting code</td>
</tr>
<tr>
<td>\</td>
<td>'non-English' word</td>
</tr>
<tr>
<td>-</td>
<td>sentence start</td>
</tr>
<tr>
<td>~</td>
<td>included sentence start</td>
</tr>
<tr>
<td>_</td>
<td>word-class tag start</td>
</tr>
</tbody>
</table>

The majority of the transcription codes pertain to special symbols, non-English words or alphabets, and some formatting information. There are also codes to indicate the start of sentences. Every word and punctuation mark in an LOB text has a word-class tag associated with it. A word-class tag immediately follows the word or symbol and is prefaced by an underscore character. The identifier of a word-class tag is composed of capital letters, digits, and some of the non-alphanumeric characters. The reserved characters and their meanings are listed in Table 6.

An excerpt from an LOB document can be found in Figure 52. Line (1) contains the sentence start character (\`). Line (3) contains the included sentence start character (\`), the begin quote (\*) and the end quote (\**) characters, and the abbreviation tag (\0). Each record begins with the reference field which contains an identification code and line number. Every word and punctuation mark is followed by a word-class tag. Also, the start of a paragraph is indicated by three spaces at the beginning of the text portion of the record, as can be seen in lines (1) and (8).
(1) AOS 188 "a_AT reference_NN to_IN the_ATI reunification_NN of_IN Germany_NP
(2) AOS 188 brought_VBD a_AT bark_NN of_IN
(3) AOS 189 "*'_**' start_VB another_DT war_NN !_! **'_**' from_IN
   \OMr_NPT
(4) AOS 189 Ellis_NP Smith_NP (_(\\0Lab_NN ,_ , Stoke_NP on_IN
(5) AOS 190 Trent_NP \OS_NP )_)_._. . then_RN \OMr_NPT Healey_NP
   launched_VBD
(6) AOS 190 out_RP on_IN his_PP$ pet_NN theme_NN of_IN
(7) AOS 191 limitation_NN of_IN armaments_NNS in_IN Europe_NP ._.
(8) AOS 192 "this_DT could_MD ,_, he_PP3A suggested_VBD ,_,
   be_BE linked_VBN
(9) AOS 192 with_IN prohibition_NN of_IN the_ATI
(10) AOS 193 production_NN of_IN atomic_JJ weapons_NNS in_IN
    any_DTI part_NN of_IN
(11) AOS 193 Europe_NP ._. "inspection_NN and_CC
(12) AOS 194 control_NN would_MD be_BE much_RB easier_JJR to_TO
    establish_VB in_IN
(13) AOS 194 these_DTS territories_NNS ._.

Figure 52: Excerpt From an LOB Document
7.3.2 Retagging the LOB

An LOB Corpus text must be preprocessed before it can be accessed by the Retag Toolset. This is due to the storage format of the Corpus. The Corpus texts are stored in 80 column records with the first seven columns of the record reserved for a reference code and the last 72 columns for the tagged text. The reference code is not part of the text. A preprocessing program must recombine the physical records into the logical records and remove the reference code. Such a program using a Lex specification required 23 lines of C code. The preprocessed version of the document in Figure 52 can be seen in Figure 53.

Although the user also would have had to write the preprocessor code, we will not include those statements in our calculations of the user's contribution. This is due to the fact that the preprocessor probably would have had to be written even if the Toolset was not used.

The RTT was used to build a Replace_Tag program to convert several LOB documents. The classification of some of the tags in the specification can be seen in Table 7. The table only contains a small sample of the possible tags in the RTT specification.

The current RTT specification contains 170 statements describing the two alphabets, 17 transcription tags and all of the word-class tags. Figure 54 contains the LOB document of Figure 53 after processing by the Replace_Tag program.

The current RTT specification for the LOB has a large number of implicit segmenting tags requiring the insertion of mate tags. These include all of the word-class tags as well as the abbreviation tag, the sentence initial tags, and the paragraph tag. The ITT was used to build one Insert_Tag program to insert all
Figure 53: LOB Document After Preprocessing

Table 7: Classification of Tags in the LOB

<table>
<thead>
<tr>
<th>Description</th>
<th>Examples</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>transcription</td>
<td>*-</td>
<td>symbol</td>
</tr>
<tr>
<td>- dash</td>
<td>*</td>
<td>symbol</td>
</tr>
<tr>
<td>- open quote</td>
<td>*</td>
<td>symbol</td>
</tr>
<tr>
<td>- close quote</td>
<td>**</td>
<td>symbol</td>
</tr>
<tr>
<td>- subscript</td>
<td>*; **;</td>
<td>explicit delimiting</td>
</tr>
<tr>
<td>- superscript</td>
<td>*; **;</td>
<td>explicit delimiting</td>
</tr>
<tr>
<td>- abbreviation</td>
<td>\O</td>
<td>implicit segmenting</td>
</tr>
<tr>
<td>- sentence start</td>
<td>\O</td>
<td>implicit segmenting</td>
</tr>
<tr>
<td>- included sentence start</td>
<td>~</td>
<td>implicit segmenting</td>
</tr>
<tr>
<td>- paragraph start</td>
<td>3 spaces at start</td>
<td>implicit segmenting</td>
</tr>
<tr>
<td></td>
<td>of text field</td>
<td></td>
</tr>
<tr>
<td>word-class</td>
<td>_ABL</td>
<td>implicit segmenting</td>
</tr>
<tr>
<td></td>
<td>_AP&quot;</td>
<td>implicit segmenting</td>
</tr>
<tr>
<td></td>
<td>_CD1</td>
<td>implicit segmenting</td>
</tr>
<tr>
<td></td>
<td>_NN$</td>
<td>implicit segmenting</td>
</tr>
<tr>
<td></td>
<td>_!</td>
<td>implicit segmenting</td>
</tr>
<tr>
<td></td>
<td>_&amp;FW</td>
<td>implicit segmenting</td>
</tr>
</tbody>
</table>
(1) <p>a reference to the reunification of Germany brought a bark of</p>

(2) &quot;start another war!&quot; from Mr Ellis Smith (Lab), Stoke on Trent.

(3) Trent (S) Trent Mr Healey launched out on his pet theme of.

(4) limitation of armaments in Europe.

(5) this could, he suggested, be linked with prohibition of the

(6) production of atomic weapons in any part of Europe.

(7) control would be much easier to establish in these territories.

Figure 54: LOB Document After Processing by Replace_Tag Program
but the missing included sentence end tag. The ITT specification contains 308
statements describing the tags and their hierarchical relationship. Most of the
tags had to be specified twice, because they could occur both inside and outside
the paragraph and sentence tags.

The ITT could not be used to insert the included sentence end tag, because
there is no specific subset of tags which signal the end of an included sentence.
Sometimes the included sentence is a quote. Sometimes it is delimited by paren­
theses. Because of this, the included sentence tag must be manually inserted.

The RTT and ITT both automatically generate Lex and Yacc specifications
for the respective Replace_Tag and Insert_Tag programs. The user had to specify
478 statements through the Toolset. The generated specifications contained 846
rules and 1309 lines of C code. For both methods the 23 lines of C code for the
preprocessor had to be written. This represents a savings in coding effort of 4.3:1.

7.4 WATCON-2

7.4.1 The Encoding Scheme

WATCON-2 is a concordance-generator program package. It does not have one
specific encoding scheme, rather it permits the users of the package to specify
their own encoding scheme within certain restrictions. The control parameters
that can be defined by the user of interest to retagging are UPPER, LOWER,
ACCENTS, ALTERNATE, ELIMINATE, SUSPEND, START, STOP, CEASE,
MORE, GENERATE, DELIMITER, and MARKER. These control parameters
define the the text alphabet, the reserved alphabet and the tag strings. The names
of the control parameters reflect the concordance-building process associated with
(1) A reference to the reunification of Germany brought about a bark of off the sq.

(2) "Another war started another war from Ellis Smith."

(3) Trent Stoke on Trent (Lab) .. .

(4) Limitation of armaments in Europe, this could be linked with prohibition of atomic weapons in any part of Europe.

(5) Control would be easier to establish in these territories.

Figure 55: LOB Document After Processing by Insert_Tag Program
the strings defined by them, but not necessarily the meaning associated with the tags.

The parameters UPPER, LOWER, and ACCENTS define the text alphabet. ALTERNATE permits the definition of tags with the same start and end character, e.g., *a6*. ELIMINATE defines a list of characters to be ignored by WATCON-2 in building a concordance, but these characters may be markup strings in the original transcription, e.g., in the string Lo{rd ha}th, the braces contain a reconstruction of unintelligible text. SUSPEND defines left and right delimiter characters to mark text not to be included in the concordance, e.g., in [SARGENTO], the brackets delimit the name of a character in a play. START and STOP characters serve a similar function as the SUSPEND characters, except they indicate which part of a data record is to be processed. A CEASE character is similar to a STOP character and is used in conjunction with a START character. The MORE character appears in the last column of the data field of a record and indicates that the next physical record is a continuation of the current logical record. The GENERATE parameter indicates that the data file contains a special label-definition record. The DELIMITER and MARKER values are used in combination to define tag strings with identifying names, e.g., <B< and >B>, where the DELIMITER characters are < and > and the MARKER value is the letter B.

Some of the documents are encoded as fixed-length physical records with label fields and MORE characters. These documents have to be preprocessed to recombine the physical records into the logical records.

We had excerpts from ten documents encoded for WATCON-2 available to us for retagging purposes. Only two of these contained markup strings. We describe
Table 8: Reserved Characters for the García-Lorca Text

<table>
<thead>
<tr>
<th>Characters</th>
<th>Control Parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>ALTERNATE</td>
</tr>
<tr>
<td>&lt;&gt;</td>
<td>SUSPEND</td>
</tr>
<tr>
<td>[]</td>
<td>SUSPEND</td>
</tr>
<tr>
<td>+</td>
<td>MORE</td>
</tr>
<tr>
<td>~</td>
<td>STOP</td>
</tr>
</tbody>
</table>

our analyses and experiences with these two documents.

Excerpt From a Play by García-Lorca

This sample text has values defined for the control parameters ALTERNATE, SUSPEND, MORE, and STOP. The list of reserved characters defined by these parameters can be seen in Table 8. An excerpt from the document can be seen in Figure 56. The characters [ and ] delimit the names of speakers in the play. The tags <t, t>, <s, s>, <b, and b> delimit descriptive parts of the play. The strings *a6* and *a7* delimit text that was italicized in the original document. The ^ character represents special processing information. The MORE character, +, will be removed during preprocessing.

Excerpt From “The Waves” by Virginia Woolf

An excerpt from this document can be seen in Figure 57. This sample text has values defined for the control parameters GENERATE, DELIMITER and MARKER. There are two reserved characters defined by the DELIMITER parameter, < and >. The delimited markers (<B<, >B>, <S<, and >S>) can be seen in Lines (2), (8), and (9). The document also includes a label-definition record in Line (1) to be
(1) 328119 0<ESCENA DEL TENIENTE CORONEL DE LA GUARDIA%CIVIL t>
(2) 328119 0<s Cuarto de banderas s>
(3) 328119 1[TENIENTE CORONEL] Yo soy el teniente coronel de la +
(4) 328119 $Guardia%Civil.~
(5) 328119 2[SARGENTO] S’i.
(6) 328119 3[TENIENTE CORONEL] Y no hay quien me desmienta.~
(7) 328119 4[SARGENTO] No.~
(8) 328119 5[TENIENTE CORONEL] Tengo tres estrellas y veinte cruces.~
(9) 328119 4[SARGENTO] S’i.~
(10) 329119 7[TENIENTE CORONEL] Me ha saludado el cardenal arzobispo /~
(11) 329119 8/ con sus veinticuatro borlas moradas.~
(12) 329119 9[SARGENTO] S’i.~
(13) 329119 10[TENIENTE CORONEL] Yo soy el teniente. Yo soy el teniente. +
(14) 329119 11Yo soy el teniente coronel de la $Guardia%Civilor.~
(15) 329119 12<b*a6*(Romeo y Julieta, celeste, blanco y oro*a7*/,b>
(16) 329119 13<b*/a6*se abrazan sobre el jardín de tabaco de la caja de +
(17) 329119 puros#a7*.b>

Figure 56: Excerpt From the García-Lorca Text
Figure 57: Excerpt From the Woolf Text

**Retagging Documents Encoded for WATCON-2**

Retagging the García-Lorca Text

This document is one that has to be preprocessed. It has a label field in columns one through nine and a continuation character defined by the MORE parameter. A preprocessing program was developed using a Lex specification containing 10 lines of C code. The preprocessed version of the document in Figure 56 can be seen in Figure 58.

There are eleven tag strings defined by the control parameters. These are listed in Table 9 with their tag classification. The RTT specification contains 8 statements for the two alphabets and the tags. The document after processing by the generated Replace_Tag program can be seen in Figure 59. Since the encoding did not include any implicit segmenting tags, the ITT was not used.

The generated Replace_Tag program successfully replaced all of the specified...
(1) <t ESCENA DEL TENIENTE CORONEL DE LA GUARDIA%CIVIL t>
(2) <s Cuarto de banderas s>
(3) [TENIENTE CORONEL] Yo soy el teniente coronel de la
Guardia%Civil.^ 
(4) [SARGENTO] S'i.^
(5) [TENIENTE CORONEL] Y no hay quien me desmienta.^ 
(6) [SARGENTO] No.^ 
(7) [TENIENTE CORONEL] Tengo tres estrellas y veinte cruces.^ 
(8) [SARGENTO] S'i.^
(9) [TENIENTE CORONEL] Me ha saludado el cardenal arzobispo / 
(10) / con sus veinticuatro borlas moradas.^ 
(11) [SARGENTO] S'i.^
(12) [TENIENTE CORONEL] Yo soy el teniente. Yo soy el teniente.
Yo soy el teniente coronel de la $Guardia%Civil.^ 
(13) <b*a6*(Romeo y Julieta, celeste, blanco y oro*a7*/b> 
(14) <b/*a6*se abrazan sobre el jard'in de tabaco de la caja de 
puros**a7*/b> 

Figure 58: García-Lorca Text After Preprocessing 

Table 9: Classification of Tags in the García-Lorca Text

<table>
<thead>
<tr>
<th>Tag String</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>nonsymbol</td>
</tr>
<tr>
<td>&lt;t t&gt;</td>
<td>explicit segmenting</td>
</tr>
<tr>
<td>&lt;s s&gt;</td>
<td>explicit segmenting</td>
</tr>
<tr>
<td>&lt;b b&gt;</td>
<td>explicit segmenting</td>
</tr>
<tr>
<td>[ ]</td>
<td>explicit segmenting</td>
</tr>
<tr>
<td><em>a6</em> <em>a7</em></td>
<td>explicit segmenting</td>
</tr>
</tbody>
</table>
ESCENA DEL TENIENTE CORONEL DE LA GUARDIA\%CIVIL

Cuarto de banderas

TENIENTE CORONEL
Yo soy el teniente coronel de la Guardia\%Civil.

SARGENTO
S'i.

TENIENTE CORONEL
Y no hay quien me desmienta.

SARGENTO
No.

TENIENTE CORONEL
Tengo tres estrellas y veinte cruces.

SARGENTO
S'i.

TENIENTE CORONEL
Me ha saludado el cardenal arzobispo

/ con sus veinticuatro borlas moradas.

SARGENTO
S'i.

TENIENTE CORONEL
Yo soy el teniente. Yo soy el teniente. Yo soy el teniente coronel de la $Guardia\%Civil.$

(Romeo y Julieta, celeste, blanco y oro, se abrazan sobre el jardín de tabaco de la caja de puros)
Table 10: Classification of Tags in the Woolf Text

<table>
<thead>
<tr>
<th>Tag String</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;B&lt; B&gt;</td>
<td>explicit segmenting</td>
</tr>
<tr>
<td>&lt;S&lt; S&gt;</td>
<td>explicit segmenting</td>
</tr>
<tr>
<td>**** , ,</td>
<td>explicit segmenting</td>
</tr>
</tbody>
</table>

tag strings with the specified LIF equivalent. The user had to specify eight statements in the RTT. The generated Lex and Yacc specifications contained 33 rules and 109 lines of C code. The preprocessor program contained 10 lines of C code. This represents a savings in coding effort of 8.4:1.

Retagging the Woolf Text

There are six tag strings in this text excerpt. They are listed in Table 10. The complete document has many more delimited markers which are included in the RTT specification.

The RTT specification consists of 22 statements defining the two alphabets and 20 pairs of tags. The retagged document can be seen in Figure 60. Since the encoding did not include any implicit segmenting tags, the ITT was not used.

The generated Replace.Tag program correctly replaced all of the specified tag strings. The user had to specify 22 statements in the RTT. The generated Lex and Yacc specifications contained 80 rules and 183 lines of C code. This represents a savings in coding effort of 12.0:1.
(1) <id>TW.1.14</id>
(2) <Bern>"Susan has passed us,"</Bern> said Bernard. <Bern>"She has passed
(3) the tool-house door with her handkerchief screwed
(4) into a ball. She was not crying, but her eyes, which are
(5) so beautiful, were narrow as cats' eyes before they
(6) spring. I shall follow her, Neville. I shall go gently behind
(7) her, to be at hand, with my curiosity, to comfort
(8) her when she bursts out in a rage and thinks, <Su>'I am
(9) alone.'</Su></Bern>

Figure 60: Woolf Text After Processing by Replace_Tag Program

7.5 The Oxford Concordance Program

7.5.1 The Encoding Scheme

The Oxford Concordance Program (OCP) [12] is another concordance-generator program package. It does not have one specific encoding scheme, rather it permits the users of the package to specify their own encoding scheme within certain restrictions. The options that can be defined by the user of interest to retagging are ALPHABET, DIACRITICS, PUNCTUATION, PADDING, NEWLINE, CONTINUE, COMMENTS, and REFERENCES. These options define the text alphabet, the reserved alphabet and the tag strings. Once again, the names of the options reflect the processing to be done by the OCP. Many tag strings may have other meanings in the original encoding.

The options ALPHABET, DIACRITICS, and PUNCTUATION define the text alphabet. PADDING allows the definition of markers. These tag strings mark features of the original text such as italics, foreign words, and proper names. The
NEWLINE option permits the user to include more than one logical record per physical record. This character will have to be preprocessed and replaced with a newline character. The CONTINUE option defines a continuation character that indicates that a logical record has been broken into more than one physical record. The COMMENTS option defines delimiter characters for comments.

The REFERENCES option describes the type of "reference" strings in the document. Reference strings indicate different components of a document. These include title, author, character, verse numbers, line numbers, and others. As such, they represent a large class of markup in OCP documents. There are three types of references: COCOA, starting string, and fixed format.

COCOA references were first formulated for one of the first general purpose concordance programs. The general format of a COCOA tag is

    open_character category space reference close_character

The user can specify the open and close character. Original COCOA references used < and > for the open and close characters, respectively. COCOA references have two special categories L and R, representing line number and record number, respectively.

Starting string references appear in the first few columns of a record. Their general structure is

    reserved_character identifying_string

The user has to specify each of the strings in the REFERENCES option.

Fixed format references occur in the same column positions of every input record. This reference field may have several subfields. The Retagging Toolset
does not have an option for processing specific fields of a record. Documents containing this type of reference will need to be preprocessed. Since we have only one such document available with fixed format references and it does not contain any other markup, we will exclude it from our considerations.

Excerpt From A Text With COCOA References

We have three documents with COCOA references available to us. Each of these have different sets of categories to be used within the COCOA references. A sample document with COCOA references can be seen in Figure 61. The COCOA references can be seen in lines (1), (2), (3), and (5). This document also contains markers for proper names of people ("), place names (%"), and Latin words ($). In addition, Roman numerals are delimited by [ and ] (line 4) and reconstructed text is delimited by << and >> (line 16). The reserved alphabet for this document is \{\^, %, $, <, >, [, ]\}.

Excerpt From A Text With Starting String References

There was one file containing two documents using starting string references. One of these documents included a NEWLINE character. An excerpt from this document can be seen in Figure 62. The starting string references are *Title and *Verse and can be seen in lines (1), (2), and (5). The NEWLINE character is // and is removed during preprocessing. The only reserved character is the slash, /.

The asterisk is ambiguous, because it is also used as an acute accent.
(1) <A WILLIAM PASTON I>
(2) <N 2>
(3) <L 0>
(4) #DO WRITEN [IJ] COPIES OF QIS NOTE IN PAPIER WYDE WRITEN, AND
GETE A COPIE OF QE WRITTE IN QE #ESCHEKYR AGEYN YOW.
(5) <L 1>
(6) #RIGHT WORTHY AND WORSEPEFULL SER, #I RECOMAUNDE ME TO YOW
AND THANK
(7) YOW FOR QE GOOD, TREW, AND DILIGENT LABOUR YE HAUE HADDE FOR
QE MATIER
(8) BETWEN QE #PRIOUR OF %BROMHOLM AND HIS COMMOigne APOSTATA
~JOHN ~WORTES,
(9) QAT NAMYTH HYM-SELF '~PASTON AND AFFERMITH HYM VNTREWLY TO BE MY
(10) COUSYN. ~GOD DEFENDE QAT ANY OF MY POURE KYN SHULD BE OF SWYCH
(11) GOUERNANUCSE AS HE IS OF, #MAISTER ~JOHN ~IXWORTH TOLD ME QAT
HE HADDE
(12) LETTRES FRO A FRENDE OF YOWRES IN QE COURT OF %ROME QAT IS OF
QE SEYD #PRIOURSES
(13) COUNSEILL IN QIS MATER AS YE BE, WHOS NAME #I KNOWE NOUGHT,
SPECIFYENG
(14) QAT QE SEYD ~JOHN ~WORTES, $ADUERSARIUS $#PRIORIS, $DESPERAT
IN $CAUSA $ET $CONCORDIAM
(15) $QUERIT. #IT IS TOLD ME SITHEN QAT QE SEYD ~JOHN ~WORTES IS IN
QE
(16) C<<DURT>> OF %ROME SACRED A BYSSHOP OF %IRLAND, $VIDELICET
$EPISCOPUS %CORCAGENSI;

Figure 61: Excerpt From Text With COCOA References
(1) *Title LA LUNA ASONA
(2) *Verse 1
(3) Cuando sale el luna // se perden las campanas
(4) y aparecen las sendas // impenetrables.
(5) *Verse 2
(6) Cuando sale la luna // el mar cubre la tierra
(7) y el corazón se siente // isla en el infinito.

Figure 62: Excerpt From Text With Starting String References

<table>
<thead>
<tr>
<th>Tag String</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>implicit segmenting</td>
</tr>
<tr>
<td>%</td>
<td>implicit segmenting</td>
</tr>
<tr>
<td>$</td>
<td>implicit segmenting</td>
</tr>
<tr>
<td>&lt;A &gt;</td>
<td>explicit segmenting</td>
</tr>
<tr>
<td>&lt;T &gt;</td>
<td>explicit segmenting</td>
</tr>
<tr>
<td>&lt;N &gt;</td>
<td>explicit segmenting</td>
</tr>
<tr>
<td>&lt;L &gt;</td>
<td>explicit segmenting</td>
</tr>
<tr>
<td>&lt;&lt; &gt;&gt;</td>
<td>explicit segmenting</td>
</tr>
<tr>
<td>[ ]</td>
<td>explicit segmenting</td>
</tr>
</tbody>
</table>

7.5.2 Retagging Documents Encoded for the OCP

Retagging Texts With COCOA References

For the document in Figure 61, there are twelve tag strings. These tags are listed with their tag classification in Table 11. The RTT specification contains eleven statements for the two alphabets and the tags. The document after processing by the generated Replace.Tag program can be seen in Figure 63.

There are three markers that are implicit segmenting tags requiring end tags to be inserted. The end tags are implied by the space terminating the word that
(1) <author>WILLIAM PASTON I</author>
(2) <num>2</num>
(3) <lnum>0</lnum>
(4) #DO WRITEN <rnum>IJ</rnum> COPIES OF QIS NOTE IN PAPIER WYDE WRITEN, AND GETE A COPIE OF QE WRITTE IN QE #ESCHEKYR AGETYN YOW.
(5) <lnum>1</lnum>
(6) #RIGHT WORTHY AND WORSEPEFULL SER, #I RECOMAUNDE ME TO YOW AND THANK
(7) YOW FOR QE GOOD, TREW, AND DILIGENT LABOUR YE HAUE HADDE FOR QE MATIER
(8) BETWEN QE #PRIOUR OF <pn>BROMHOLM AND HIS COMMOIGNE APOSTATA <pnp>JOHN <pnp>WORTES,
(9) QAT NAMYTH HYM-SELF <pnp>PASTON AND AFFERMITH HYM VNTREWLY TO BE MY
(10) COUSYN. <pnp>GOD DEFENDE QAT ANY OF MY POURE KYN SHULD BE OF SWYCH
(11) GUERNAUNCE AS HE IS OF, #MAISTER <pnp>JOHN <pnp>IXWORTH TOLD ME QAT HE HADDE
(12) LETURES FRO A FREND OF YOWRES IN QE COURT OF <pn>ROME QAT IS OF QE SEYD #PRIOURES
(13) COUNSEILL IN QIS MATIER AS YE BE, WHOS NAME #I KNOWE NOUGHT, SPECIFYENG
(14) QAT QE SEYD <pnp>JOHN <pnp>WORTES, <lat>ADUERSARIUS <lat>#PRIORIS, <lat>DESPERAT <lat>IN <lat>CAUSA <lat>ET <lat>CONCORDIAM
(15) <lat>QUERIT. #IT IS TOLD ME SITHEN QAT QE SEYD <pnp>JOHN <pnp>WORTES IS IN QE
(16) C<rec>OURT</rec> OF <pn>ROME SACRED A BYSSHOP OF <pn>IRLAND, <lat>VIDELICET <lat>EPISCOPUS <pn>CORCAGENSIS;

Figure 63: COCOA Text After Processing by Replace_Tag Program
is marked. The ITT cannot be used to insert the tags, because it assumes that missing tags are implied by other tags. A postprocessor to insert the missing tags contained three lines of C code in a Lex specification. The fully marked document can be seen in Figure 64.

The generated Replace_Tag program successfully replaced all of the specified tag strings. The generated Lex and Yacc specifications of the Replace_Tag program contained 37 rules and 114 lines of C code. The user had to specify 11 statements in the RTT. The postprocessor required three lines of C code. This represents a savings in coding effort of 11.0:1.

The second document with COCOA references is an excerpt from the *The Merchant of Venice*. It contained four different COCOA reference strings and delimiter markers for stage directions. There were no implicit segmenting tags. The RTT specification had 7 statements in it and the generated Lex and Yacc specifications contained 29 rules and 105 lines of C code, for a savings in coding effort of 19.1:1.

The third document contains only COCOA references as markup. Its RTT specification had 11 statements and the generated Lex and Yacc specifications contained 36 rules and 116 lines of C code. This represents a savings in coding effort of 13.8:1.

**Retagging Texts With Starting String References**

The document in Figure 62 must be preprocessed to remove the NEWLINE character. A Lex specification for such a program contains one line of C code. The resulting document can be seen in Figure 65.
(1) <author>WILLIAM PASTON I</author>
(2) <num>2</num>
(3) <lnum>0</lnum>
(4) #DO WRITEN <rnum>IJ</rnum> COPIES OF HIS NOTE IN PAPIER WYDE WRITEN, AND GETE A COPIE OF THESE WRITTE IN ESCHKYR AGEYN YOW.
(5) <lnum>1</lnum>
(6) #RIGHT WORTHY AND WORSEPEFULL SER, #I RECOMAUNDE ME TO YOW AND THANK
(7) YOW FOR QE GOOD, TREW, AND DILIGENT LABOUR YE HAUE HADDE FOR QE MATIER
(8) BETWEN QE #PRIOUR OF <pn>BROMHOLM</pn> AND HIS COMMOIGNE APOSTATA <pnp>JOHN</pnp> <pnp>WORTES</pnp>,
(9) #AT NAMYTH HYM-SELF <pnp>PASTON</pnp> AND AFFERMITH HYM VNTREWLY TO BE MY COUSYN. <pnp>GOD</pnp> DEFENDE #AT ANY OF MY POURRE KYN SHULD BE OF SWYCH
(11) GOUERNAUNCE AS HE IS OF, #MAISTER <pnp>JOHN</pnp> <pnp>IIXWORTH</pnp> TOLD ME #AT HE HADDE
(12) LETTRES FRO A FREND OF YOWRES IN QE COURT OF <pn>ROME</pn> #AT IS OF QE SEYD #PRIOURES
(13) COUNSEILL IN HIS MATIER AS YE BE, WHOS NAME #I KNOWE NOUGHT, SPECIFYENG
(14) #AT QE SEYD <pnp>JOHN</pnp> <pnp>WORTES</pnp>,<lat>AUDERSARIUS</lat> <lat>#PRIORIS</lat>,
(15) <lat>DESPERAT</lat> <lat>IN</lat> <lat>CAUSA</lat> <lat>ET</lat> <lat>CONCORDIAM</lat>
(16) #IT IS TOLD ME SITHEN #AT QE SEYD <pnp>JOHN</pnp> <pnp>WORTES</pnp> IS IN QE
(16) C<rec>OURT</rec> OF <pn>ROME</pn> SACRED A BYSSHOP OF <pn>IRLAND</pn>, <lat>VIDELICET</lat> <lat>EPISCOPUS</lat> <pn>CORCAGENSIS</pn>;

Figure 64: COCOA Text After Processing to Insert Missing End Tags
Figure 65: Starting String Text After Preprocessing

Table 12: Classification of Tags in the Starting String Text

<table>
<thead>
<tr>
<th>Tag String</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>*Title newline</td>
<td>explicit segmenting</td>
</tr>
<tr>
<td>*Verse newline</td>
<td>explicit segmenting</td>
</tr>
</tbody>
</table>

For the document in Figure 65, there are 2 starting string references. These tags are listed with their tag classification in Table 12. The RTT specification contains 5 statements for the three alphabets and the tags. The document after processing by the generated Replace_Tag program can be seen in Figure 66. The ITT was not used, because there are no missing implicit segmenting tags.

The generated Replace_Tag program successfully replaced all of the specified tag strings. The user had to specify 5 statements in the RTT. The preprocessor required one line of C code. The generated Lex and Yacc specifications of the Replace_Tag program contained 27 rules and 111 lines of C code. This represents a savings in coding effort of 23.2:1.
7.6 Scribe

7.6.1 The Encoding Scheme

Scribe [30, 35, 36] is a text-formatting program. Documents encoded for processing by Scribe are a combination of text strings and formatting commands. The encoding scheme is robust in that it permits a multiple number of ways to express a tag string. In the previously discussed encoding schemes, there was usually only one way to type the tag strings. The encoding scheme is also very sophisticated in that it permits commands to nest inside of other commands and some commands have parameter lists.

An example of the robustness of Scribe is the role of delimiter characters in Scribe. Scribe does not reserve any characters for the function of delimiting strings. The role of a delimiter character is determined by the context in which it appears. Scribe permits the use of seven different delimiter character pairs. If we want to
indicate that the word *text* should be italicized, we may encode it as \( \textbf{i}(\text{text}) \), \( \textbf{i}\{\text{text}\} \), \( \textbf{i}\langle\text{text}\rangle \), \( \textbf{i}''\text{text}'\), \( \textbf{i}'\text{text}' \), or \( \textbf{i}'\text{text}' \).

The retagging alphabets for Scribe are easy to define. There is only one reserved character, \( \textbf{©} \), which indicates the start of a command. The delimiter characters described above are in the ambiguous alphabet. Any character, except \( \textbf{©} \), may appear in text.

Although Scribe categorizes its tag strings as commands, environments, or forms, the structure of the tag strings in the different categories is primarily the same. This structure is

\[
\textbf{©} \text{ name optional_delimited_string}
\]

where the *optional_delimited_string* is *not* necessarily delimited text.

Some commands are not followed by a delimited string, e.g., \( \textbf{©©} \) indicates that an \( \textbf{©} \) character is to be printed in the document. For those commands that are followed by a delimited string, the delimited string may be

- text, e.g., \( \textbf{©section(In-text Math Formulas)} \),
- a parameter list, e.g., \( \textbf{©sum\{from <n=i>, to \infty\}} \),
- a predefined value, e.g., \( \textbf{©libraryfile(mathematics10)} \),
- a user-defined value, e.g., \( \textbf{©cite(AH086)} \),
- an environment name\(^1\), e.g., \( \textbf{©begin(math)} \).

\(^1\)This permits a long form for the environment-type tags, e.g., \( \textbf{©begin(i)text©end(i)} \), as well as the common form, \( \textbf{i}(\text{text}) \).
and there are other possible categories for the delimited string. It is interesting to note that a parameter list may contain delimited text or a command as a parameter value because this will yield tag strings, the parameter names, to be specified in the RTT that do not contain the reserved character. How the delimited string for a particular command is to be interpreted, i.e., as markup or text, is determined by the individual defining the retagging.

An example of a Scribe document can be seen in Figure 67. Lines (1)-(4), (7)-(10), (12), and (15)-(18) contain commands. Lines (5), (6), (8), (9), (13), (14), (16), and (17) contain the text of the document. The blank line in (11) indicates the end of a paragraph.

Figure 67: A Scribe Document
Table 13: Classification of Tags in Scribe

<table>
<thead>
<tr>
<th>Description</th>
<th>Examples</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>commands or forms representing symbols</td>
<td><code>@</code></td>
<td>symbol</td>
</tr>
<tr>
<td></td>
<td><code>@ldots</code></td>
<td>symbol</td>
</tr>
<tr>
<td>commands without delimited text strings</td>
<td><code>@bibliography</code></td>
<td>nonsymbol</td>
</tr>
<tr>
<td></td>
<td><code>@libraryfile(stable)</code></td>
<td>nonsymbol</td>
</tr>
<tr>
<td>environments and commands with delimited text strings</td>
<td><code>@begin(up) text @end(up)</code></td>
<td>explicit segmenting</td>
</tr>
<tr>
<td></td>
<td><code>@b(text)</code></td>
<td>explicit segmenting</td>
</tr>
<tr>
<td></td>
<td><code>@section[ text ]</code></td>
<td>explicit segmenting</td>
</tr>
<tr>
<td>commands that mark the start or end of document components</td>
<td><code>@make(article)</code></td>
<td>implicit segmenting</td>
</tr>
<tr>
<td></td>
<td><code>@\</code></td>
<td>implicit segmenting</td>
</tr>
<tr>
<td></td>
<td>a blankline</td>
<td>implicit segmenting</td>
</tr>
<tr>
<td>commands or forms with parameter lists containing delimited text strings</td>
<td><code>@over(num &quot; text &quot;, denom &quot; text &quot;)</code></td>
<td>explicit segmenting (several)</td>
</tr>
</tbody>
</table>

7.6.2 Retagging Documents Encoded for Scribe

We tested the Retagging Toolset on ten Scribe documents. These documents contained a wide variety of Scribe commands and many representations for some of these commands. A small sample of the commands and their classification can be seen in Table 13.

The RTT was used to build two Replace_Tag programs for the sample of ten Scribe documents. The first Replace_Tag program replaces the Scribe markup strings, excluding whitespace tags, with their LIF equivalents. The second Replace_Tag program takes the partially retagged documents from the first Replace_Tag program and replaces the whitespace tags with their LIF equivalents.

The first RTT specification contains 22 environments, 36 commands, and 5 forms. Many of the tags have multiple representations in the specification. The
In-text Math Formulas

This is an example of a formula that appears as a part of the text:

\[
\sum_{n=1}^{\infty} \frac{(-1)^n}{n}
\]

Display Math Formulas

Note the difference in this display version of the same formula:

\[
\sum_{n=1}^{\infty} \frac{(-1)^n}{n}
\]

Figure 68: Scribe Document After Processing by the First Replace_Tag Program

reserved alphabet is \{@\}. The ambiguous alphabet is \{ (, ), {, }, [], <, >, ", ', ' \}. The text alphabet contains the remaining characters. The RTT specification contains 109 statements describing the three alphabets and the different representations of the included Scribe tags. Figure 68 contains the Scribe document of Figure 67 after processing by the first Replace_Tag program.

Scribe has one whitespace tag: a blankline indicates the end of a paragraph, if the blankline is interpreted as part of the text. The Scribe Manual [35] describes the interpretation of carriage returns (newline characters) as:

Whether a carriage return is part of a command or part of the manuscript text depends on where the @ sign for the command is.
When the @ sign is the first character on a line, then the end-of-line following it is part of the command, not part of the text. When the @ sign is anywhere else on the line, then an end-of-line following it is part of the manuscript text.

To further complicate the situation, newline characters within the verbatim environment are always interpreted as part of the text.

If the newline characters were included in the first RTT specification, the size of the specification would double and it would be significantly more difficult to write. To simplify the specification of the whitespace tag, the RTT was used to build a second Replace_Tag program. This second Replace_Tag program takes the partially retagged output of the first Replace_Tag program as input and properly replaces the whitespace tags. Because the LIF tags in a partially retagged document have only one representation, this specification is much simpler. The RTT specification for this Replace_Tag program contains 45 statements. Figure 69 contains the sample document of Figure 68 after processing by the second Replace_Tag program.

A curious situation arises during the processing by the Replace_Tag program that affects the function of the ITT. There are instances where some implicit segmenting tags are not placed in a document because there is no tag string in the document for the Replace_Tag program to recognize. These implicit segmenting tags are implied by other tags or by the end of the document.

An example of this situation is a row of cells in a table. In Scribe, a row of cells is encoded as

\texttt{cell 1 } \& \texttt{\ cell 2 } \& \texttt{\ cell 3}
This is an example of a formula that appears as a part of the text:

\[ \sum_{n=1}^{\infty} \frac{(-1)^n}{n} \]

Note the difference in this display version of the same formula:

\[ \sum_{n=1}^{\infty} \frac{(-1)^n}{n} \]

Figure 69: Scribe Document After Processing by the Second Replace_Tag Program
with a blankline after each row. In the RTT, the tag @\ is interpreted as the subcomponent end tag </cpmt> and the Replace_Tag program converts the row to

```plaintext
    cell 1 </cpmt> cell 2 </cpmt> cell 3 </r>
```

After processing by the Insert_Tag program the row becomes

```plaintext
    <r><cpmt> cell 1 </cpmt><cpmt> cell 2 </cpmt> cell 3 </r>
```

The last cell in the row is not tagged. This problem arises because the end of the row implies the end of the last cell.

Another example is the last paragraph in a Scribe document. Usually a paragraph is followed by a blankline to indicate the end of the paragraph. The last paragraph does not have to be followed by a blankline, since the end of the document implies the end of the last paragraph. The Replace_Tag program recognizes the blanklines and replaces them with paragraph end tags, but it does not insert a paragraph end tag at the end of the document. Without the end tag, the Insert_Tag program does not realize that there is a missing start tag to insert and the last paragraph remains untagged.

The RTT tool is not capable of inferring more than one tag from a markup string, nor is it capable of inferring tags from the end-of-file marker. The ITT is not capable of inserting both missing start and end tags. A new tool is needed in the toolset to insert implicit segmenting tags implied by other tags or by the end-of-file marker.

Because the tool does not exist at his time, a special program was developed for the purpose of inserting these types of tags. The Lex specification for this program

---

2If the @\ tag had been interpreted as a start tag, then the first cell would not have been tagged.
Figure 70: Scribe Document After Processing by Special Insert-Tag Program

contained 15 lines of C code. The effect of this program on the sample document can be seen in Figure 70.

The current RTT specification for Scribe includes three implicit segmenting tags that require the insertion of mate tags. Scribe defines several different document types. Because these document types have different hierarchical tags, the ITT must be used to build Insert_Tag programs for each type of document. There is one type of document in the current specification, article.

The ITT was used to build an Insert_Tag program to insert the missing tags for article end, paragraph start, and subcomponent\(^3\) start. The user had to write

\(^3\)The subcomponent end tag corresponds to the Scribe tag \@\text\. It is used to separate table cells
23 statements to describe the missing tags and their hierarchical relationship with other tags. Figure 71 contains the sample document of Figure 70 after processing by the Insert_Tag program.

The generated Replace_Tag program successfully recognized all of the specified Scribe tags in the test documents and replaced the tags with the specified LIF equivalents. The generated Insert_Tag program inserted the missing mate tags correctly except for one case. It did not properly insert the paragraph start tag when a list was in the body of the paragraph.

and definition terms, i.e., subcomponents of larger entities.
The RTT and the ITT both automatically generate Lex and Yacc specifications for the respective Replace_Tag and Insert_Tag programs. The user had to specify a total of 177 statements in the Toolset. The generated Lex and Yacc specifications contained 403 rules and 922 lines of C code. The special Insert-Tag program required 15 lines of C code. This represents a savings in coding effort of 7.0:1.

7.7 Summary of Results

In Table 14 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 statements includes the number of statements in the RTT and ITT specifications plus the number of lines of C code in any postprocessor program that inserted missing tags. The number of lines of generated code is the number of rules plus the total number of lines of C code in the Lex and Yacc specifications of the generated Replace_Tag and Insert_Tag programs. 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.

The savings for the LOB Corpus is smaller than it need be. Recall that approximately 155 tags had to be specified twice in the ITT specification, because they appeared at different levels in the hierarchy of nesting tags. Enhancements to the
Table 14: Results of the Experiences With the Toolset

<table>
<thead>
<tr>
<th>Encoding Scheme</th>
<th>Number of User Specified Statements</th>
<th>Lines of Generated Code</th>
<th>Savings</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Specification</td>
<td>C code</td>
<td>Specification</td>
</tr>
<tr>
<td>TLG</td>
<td>46</td>
<td>0</td>
<td>192</td>
</tr>
<tr>
<td>DOSL</td>
<td>63</td>
<td>16</td>
<td>165</td>
</tr>
<tr>
<td>LOB</td>
<td>478</td>
<td>23</td>
<td>846</td>
</tr>
<tr>
<td>WATCON-2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>- García-Lorca</td>
<td>8</td>
<td>10</td>
<td>33</td>
</tr>
<tr>
<td>- Woolf</td>
<td>22</td>
<td>0</td>
<td>80</td>
</tr>
<tr>
<td>OCP</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>- COCOA Ex. 1</td>
<td>11</td>
<td>3</td>
<td>37</td>
</tr>
<tr>
<td>- COCOA Ex. 2</td>
<td>7</td>
<td>0</td>
<td>29</td>
</tr>
<tr>
<td>- COCOA Ex. 3</td>
<td>11</td>
<td>0</td>
<td>36</td>
</tr>
<tr>
<td>- Start String Ex.</td>
<td>5</td>
<td>1</td>
<td>27</td>
</tr>
<tr>
<td>Scribe</td>
<td>177</td>
<td>15</td>
<td>403</td>
</tr>
</tbody>
</table>

specification language of the ITT to reduce the number of statements necessary for this situation would improve the 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 representative because of the influence of the values for the LOB Corpus. As stated above, the number of lines of specification for the LOB Corpus can be reduced. 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 the user only has to write less than 2 per cent of the required retagging code.
CHAPTER VIII

Conclusion and Future Work

This chapter summarizes the main contributions of this dissertation and reviews the areas for possible future work.

8.1 Contributions of this Dissertation

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. The new model for textual data
objects is based on the relationships between markup and text strings. The model includes four classes of markup strings: symbol, nonsymbol, implicit segmenting, and explicit segmenting tags.

We proposed a uniform representation called a Lexical Intermediate Form with the following lexical properties: 1) the tags are easy to distinguish from the text, 2) the tags are unambiguous, and 3) the tags are explicit. 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 this 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 Thesaurus Linguae Graecae, the Dictionary of the Old Spanish Language, the Oxford Concordance Program, WATCON-2, the LOB Corpus, and Scribe. 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.

8.2 Future Work

The future work identified in this section involves studying and enhancing the Retagging Toolset.
8.2.1 Evaluation of the Toolset

The usefulness of the Retagging Toolset has been established by the work in this thesis. 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. It 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 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) [25]. 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.

8.2.2 Enhancements to the Toolset

When the studies are done, their results may include enhancements to the prototype. Other enhancements presently identified are listed below.
• Provide a method of importing a predefined set of LIF tag identifiers into the RTT.

• Provide a method of importing a set of implicit segmenting tags into the ITT from the tags specified in the RTT.

• Provide additional support in the RTT to identify improperly formed regular expressions.

• 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.

• Design and implement a tool that will generate a program that will insert tags implied by other tags or the end-of-file.
BIBLIOGRAPHY


