A Textual Object Management System*

Scott C. Deerwester†
Keith Waclena
Michelle LaMar

University of Chicago
Center for Information and Language Studies

Abstract

Computer programs that access significant amounts of text usually include code that manipulates the textual objects that comprise it. Such programs include electronic mail readers, typesetters and, in particular, full-text information retrieval systems. Such code is often unsatisfying in that access to textual objects is either efficient, or flexible, but not both. A programming language like Awk or Perl provides very general facilities for describing textual objects, but at the cost of rescanning the text for every textual object. At the other extreme, full-text information retrieval systems usually offer access to a very limited number of kinds of textual objects, but this access is very efficient. The system described in this paper is a programming tool for managing textual objects. It provides a great deal of flexibility, giving access to very complex document structure, with a large number of constituent kinds of textual objects. Further, it provides access to these objects very efficiently, both in terms of time and auxiliary space, by being very careful to access secondary storage only when absolutely necessary.

1. Introduction and Motivation

In this paper we present a system for managing textual objects called TOMS (Textual Object Management System). Textual Objects are the structural components of texts, such as words, paragraphs, chapters, or mail headers. The management of these objects includes the description of structural relationships between them, their recognition within the text files that contain them, and the efficient access of the bytes of text that comprise them. The system that provides these features is available in several forms: as a library of C functions, as primitives in several programming languages, and as a server that can be accessed from other programs across a network.

Text has logical structure, but for the vast majority of texts stored on most computer systems, this structure is represented in files as a flat sequence of bytes. Writing computer programs that operate on the structure of text is difficult because of the requirement to continually write programs to parse this flat encoding. For example, the Unix operating system provides many text manipulation tools that operate solely in terms of lines, and perhaps words (typically naively defined as a sequence of non-whitespace characters).

In addition, there are many encodings of text structure in use. Encodings range from simple con-

* This research was supported by a State of Illinois Technology Challenge Grant.

† Author's current affiliation: The Hong Kong University of Science and Technology, Department of Computer Science.

ventions used for data files, to complex *markup languages* used for text. Markup languages for document production can be procedural (e.g., the Troff and TEX typesetting languages) or structural (e.g., SGML), and in both cases are typically very complex. Structured text that is usually thought of as relatively simple often reveals surprising complexity. For example, the structure of electronic mail messages on the Internet and other networks has been rigorously defined for years by the grammar in the Internet standard RFC-822 [5]. Yet some mailers don't correctly parse or generate legal mail messages, and many mail utilities are written to make simplifying assumptions about the structure of mail, assumptions that are frequently violated.

There are a number of text-scanning programming languages, such as Awk [2], Perl[14], and Icon[8], which provide language features, such as regular expressions and other kinds of pattern matching, for easy manipulation of flat files. We would like to be able to use these kinds of languages in preference to a low-level language like C, but they are inappropriate for the following reasons:

- When used most naturally, these tools are line-oriented. This line-orientation often bears no relationship to text structure.
- These languages are designed primarily to scan text: that is, text structure is recognized by performing pattern matching in a linear scan of the text. For large collections of text this approach is too inefficient.
- Any structure recognized by a program in a text-scanning language is represented implicitly *by the algorithms*, and these algorithms are written by a programmer, rather than a database administrator — a programmer who may not be fully aware of the actual complexity of the text structure. Any new program written to manipulate the same structures can, at best, share the algorithm or, at worst, reimplement the structure recognition. If there are any bugs in any of these algorithms, textual objects will be incorrectly identified.

Rather than rely on structure recognition algorithms in several programs all being correct, it would be better if a database administrator could describe the structure of a given collection of texts beforehand, in an auxiliary data file much like a grammar for a language. Then all the programs could be written using a tool that used the grammar to recognize textual objects. If there is a bug in the grammar, it can be corrected without all the programs being rewritten (or even so much as recompiled). Ideally, this tool could be made accessible from both high- and low-level languages. The TOMS was designed to be such a tool.

The TOMS was developed and used as part of our research on full-text information retrieval systems[6]. The focus of the group's work was to develop a coherent information system architecture for making accessible the information resources of a loosely coupled, widely distributed network of computers. This system will be used to replace the current retrieval system used by the ARTFL (American and French Research on the Treasury of the French Language) Project, which provides access to the Trésor de la Langue Française[10] (TLF), a 700 megabyte database of French literary texts. ARTFL subscribers currently access the TLF approximately 150 times per month by login connections to the host system.

1.1. A Primary Indexing Tool

In order to understand the design of the TOMS, it is important to distinguish two forms or representatives of abstract textual objects, and two varieties of indexes. We use the term *token* to refer to representatives of occurrences of particular objects: the total number of *word tokens* in a document is what's usually meant by the phrase "total number of words in a document." The term *type* refers to those sets of tokens that all have the same content: the number of *word types* in a document is what's usually meant by the phrase "total number of unique words in a document."

A *token representative*, or token for short, is some form of identifier appropriate to an indexing technique. It may be as simple as a sequence number, or a pair (*offset*, *length*), or it may be something more complex. A *primary index* is a function that maps tokens to text. A list of tokens is called a *concordance*, and a function that maps types to concordances (a list of their occurrences) is a *secondary index*. These distinctions become important in viewing the differences between the approach the TOMS takes to text and that of most other text processing tools.

Most information retrieval systems assume only rudimentary structure for documents. This is because early information retrieval systems dealt almost entirely with bibliographic databases. Full-text information retrieval systems are relatively recent. Most commonly, retrieval systems deal only with two kinds of textual objects: the word, and the document containing it—any intermediate structure is left unrepresented, and therefore inaccessible.

When there are only two types of textual objects, the temptation is to "hard-wire" them into the application. After all, one may say, words and documents are quite different, sharing few characteristics. Documents contain words, but words don't contain anything of interest, and word tokens are searchable by their types while documents aren't usually divided into document-types and document-tokens.

The relatively few systems that represent any structure beyond the document and the word do so either by hard-coding a particular structure (see, for example, Smith, Weiss, and Ferguson[12]) or by adapting a traditional structured database approach, treating textual objects as though they were fields[7].

Sophisticated applications demand the ability to manipulate many more sorts of textual object than these approaches allow. This means that each sort of textual object deserves its own primary index (at least conceptually), thus allowing the application to handle queries such as:

- What is the text of the third sentence in this paragraph?
- How many sentences does this paragraph contain?
- What words are contained in this chapter?

Common user queries, such as proximity searches, require the retrieval engine to pose these questions.

The TOMS is designed to be a *primary indexing toolkit*. It abstracts the notion of primary index so that it can be readily applied to any sort of textual object, and provides a language for describing:

- the textual objects in a given set of texts
- the relationships between these objects
- how to recognize these objects within a text

Finally, the TOMS provides tools for efficiently indexing these objects.

It's important to note that the TOMS isn't intended to handle *secondary* indexing at all. There is no shortage of secondary indexing tools available, such as the standard Unix DBM family of disk-based hash tables[1], or any of a number of commercial B-tree packages. Also, few textual objects other than words seem to generalize to require secondary indexes of object-types. At any rate, TOMS objects of any kind can be readily used with any secondary indexing scheme.

The TOMS also provides a facility to associate with a textual object information not present in the text. Each object can have associated with it a list of attribute/value pairs. Both the attribute and the value are arbitrary strings of text, though the attribute is expected to be a short identifier. For example, a document object might have associated with it an `AUTHOR` attribute, with the value identifying the author. Attribute values can be computed from information in the text, such as summary statistical information, or come from another source entirely.

The TOMS provides functions to associate values with attributes, and to retrieve attributes of an object. Retrieval is fast because hash tables are used.

1.2. Users of the TOMS

The TOMS is intended for two groups of users: database *administrators* and programmers who write TOMS *client* applications; see Figure 1.

Note that when we say "database administrator" we are not implying that the TOMS must be used with a conventional database management system. The database referred to is simply a collection of online texts. The administrator is the person who describes the structure of the documents and textual objects and provides resources for the database: that is, disk space for the texts and indexes.

The client programmer writes application programs that manipulate text. These applications use TOMS functions or services which work with the grammars and recognizers specified by the database administrator.

2. Defining Documents

Before an end user can access documents in a database, a database administrator must use the TOMS to register definitions of the documents. In this section, we discuss how this is done. Briefly, the administrator uses the TOMS to perform the following three steps:

Figure 1: An Overview of the TOMS

- Define what kinds of textual objects occur in each class of documents
- Define how these objects fit together to form the structure of documents of this class
- Map this structure onto an individual document or set of documents

These steps correspond to these TOMS features:

Recognizer functions — The TOMS uses *recognizer functions* to find instances of textual objects of a particular class in text.

Document grammars — A *document grammar* defines the structure of a class of documents (e.g., letters, technical reports, etc.).

Markings — The TOMS combines a set of recognizer functions and a document grammar to create a *marking*, through which end users may access individual documents and their components in terms of their structure.

We present these three features in more depth in the remainder of this section.

2.1. Recognizing Instances of Textual Objects

A key innovation in the TOMS is the role of *recognizer functions*. The job of these functions is to find instances of a particular textual object class in text, assuming a particular markup language. The TOMS calls a recognizer function, in response to a client request, to find a textual object in the stream of text that makes up a document.

A recognizer function is given nothing more than a string of characters within a document. It returns a list of offset and length pairs of all of the objects of its class that begin and end within that text.

Recognizer functions are something like parsers, but do not actually correspond to any particular parsing technology. This is because of the generality of recognizers; many are trivially simple, being not much more than scanners or lexers, while others may actually invoke complex parsers as external programs. The entire complex of recognizers that makes up a TOMS database description, taken as a whole, constitutes something like a parser for that database.

Although all recognizer functions are functionally equivalent — all take a string of characters and return a list of offset and length pairs — the TOMS allows several different ways of defining them. This allows for a wide range of textual object classes with a minimum of effort on the part of the database administrator, while retaining efficiency. In fact, it's not necessary that the database administrator be a C programmer.

The types of recognizer functions are:

Regular Expressions — The TOMS automatically constructs a recognizer function by using a pattern-matcher driven by a user's regular expression[13]. Quite complex textual objects can be specified by regular expressions.

Internal — Recognizer functions can be written as C functions and linked into TOMS clients along with the TOMS library. These recognizers can be very efficient and powerful, but are somewhat complicated to write and maintain. Several basic internal recognizers for simple words, sentences, and paragraphs are provided with the TOMS. We have also developed internal recognizers for some of the macro packages used by the Unix typesetter Troff.

Enumerations — Object classes that are difficult or impractical to recognize algorithmically can be enumerated by the specification of the exact offsets and lengths of its objects. This allows database administrators to specify objects in a completely *ad hoc* fashion. The enumeration might be the result of manual corrections of a heuristic parser, or it might simply be a file of offset/length pairs resulting from the markup of a document by a human editor. Any tool can be used to generate the enumeration; a programmable editor could be used to allow objects to be specified via mouse sweeps, for example. Certain distinguished object classes (documents and files) are automatically enumerated by the TOMS.

2.2. Document Grammars — Describing Classes of Documents

A *document grammar* specifies the structure of documents of a particular class. Examples of document classes could include letters, scholarly papers, USENET articles, etc. Each grammar specifies what textual object classes occur and how they are related to each other within documents.

Document grammars are composed of object class references and structuring constructs. An object class reference is a label used in a grammar to stand for an object class. When a document grammar is compiled, object class references are bound to particular recognizer functions that find instances of objects of their respective class in text.

Structuring constructs describe how objects are knit together to form higher level objects, and ultimately entire documents. The structuring constructs of the TOMS are a superset of those defined in the ISO Office Document Architecture (ODA)[3].

The structuring constructs are:

REP A repetition of similar objects (series). A paragraph, for example, might be defined as a repetition of sentences using **REP**.

CHO A choice between objects (disjoint union). To define the components that make up a chapter as either figures, paragraphs or tables, one would use **CHO**.

SEQ A fixed sequence of different object classes (cartesian product). A letter, for example, could be as defined as a sequence containing a greeting, a body, and a closing using **SEQ**.

PAR A set of conjoint, or parallel, objects (union). The markings indicated by **PAR** *all* apply. As an example, a Bible chapter might be defined to be a list of paragraphs *and* a list of verses using **PAR**.

CHO, **SEQ** and **PAR** are indicated in a grammar by the object class reference, the symbol for the structuring construct, and a parenthesized list of subtrees, separated by commas, e.g.,

```
figure CHO (graphic, table)
```

The **REP** construct, because it occurs so frequently, has a special shorthand: the object name is enclosed in square brackets, e.g., `[word]` to indicate a list of words.

The **PAR** structure is an addition to the constructs provided by the ODA, and is one of the most interesting features of the TOMS. It allows a text to be seen from a number of different views, which need not share a clear hierarchical relationship. A good example is a parallel marking with one branch describing the document in terms of pages and lines, while a second branch describes the hierarchical structure of words, sentences, paragraphs, etc. Text lines don't contain sentences; nor do sentences contain lines: they overlap each other. The TOMS can map between the two branches, however, and find which lines a sentence spans. Parallel markings can be much more complex than this; for example, they could be used to record and relate different linguistic parses of a text.

2.3. Describing Individual Documents

Given a grammar and a set of recognizer functions, the TOMS compiles them and produces a marking. Once a marking exists, the database administrator simply tells the TOMS which files make up a document, and which marking should be used to access it. The TOMS stores a reference to both, and gives the administrator an ID that must be used as a unique key to identify the document in all future interactions.

2.4. An Example: Electronic Mail Messages

As mentioned earlier, standard e-mail has a complex structure which must be parsed by any programs which interact with it. Mail handlers in particular must be able to effectively parse the various fields and structure of an e-mail message. In addition to such applications, it is easy to imagine how a database of e-mail could be interesting for purposes of information retrieval. Thus e-mail provides an interesting case of a document structure which, once defined by the TOMS, can be used by many different kinds of applications.

In the remainder of this section, we use this example to show how a database administrator defines documents: in this case, Internet electronic mail messages. A sample message is shown below:

```
From daemon Sun Jan  5 17:30:25 1992
Return-Path: <michelle@tira.uchicago.edu>
To: michelle
Cc: keith@curry.uchicago.edu,
     scott@uxmail.ust.hk
Subject: Your TOMS paper section
Date: Sun, 05 Jan 92 17:32:33 -0600
From: scott@tira.uchicago.edu

Please hurry up with your section of the
paper.  All other sections are done and
we'd like to get the thing put together
and formatted ASAP!

Scott
```

In the next section we continue the example to show how the TOMS could facilitate several IR tasks using an e-mail database. Our example programs are written in C.

2.4.1. Electronic Mail Message Grammar

As we have stated, the first task of a database administrator is to define the structure of documents by providing a grammar and a set of recognizer functions. The grammar file, `email.mk`, shown in Figure 2, describes the structure of e-mail messages.

```
doc PAR (
  file,
  [
    message SEQ (
      status,
      headers [
        CHO (
          adhead SEQ (
            label,
            [address]
          ),
          nahead SEQ (
            label,
            value [valword]
          )
        )
      ],
      body [sentence [word]]
    )
  ]
)
```

Figure 2: Grammar File

Documents may span several files; the presence of the parallel file branch allows documents to be related to the files that contain them. Next, the grammar indicates that documents are a series (**REP**) of `messages` (note the square brackets), each of which is defined as a sequence of `status`, `headers`, and `body` objects. The `headers` are a series of either `adhead` (address headers, which are composed of `address`'s) or `nahead` (non-address headers, which are composed of words, here called `valwords`), while the body contains `sentences` which, in turn, contain `words`.

2.4.2. Corresponding Recognizer Functions

The recognizer function file corresponding to the e-mail grammar is `email.rf`, shown in Figure 3.

```
doc        1   ENUM
file       1   ENUM
message    1   message_rec
status     1   status_rec
headers    1   headers_rec
adhead     1   addrhead_rec
label      1   label_rec
address    1   address_rec
nahead     1   nonaddrhead_rec
value      1   value_rec
body       1   body_rec
valword    0   vword_rec
sentence   1   /[A-Z][^.?!]*[.?!]/
word       0   /[a-zA-Z0-9-']+/
```

Figure 3: E-mail Message Recognizer Functions

The object class references in the first column (being the same labels that were used in the grammar) are matched with a description of their recognizer functions in the third column. "ENUM" indicates an enumerated recognizer. `doc` and `file` are special object classes, internally enumerated by the TOMS. Regular expression recognizers are written between slashes. The regular expression for `word` specifies a non-empty sequence of alphanumerics, hyphens or apostrophes, while the `sentence` recognizer simply looks for a terminating period, question mark, or exclamation point. All the other classes use internal recognizer functions. The tags such as `message_rec` are keys into a table of internal recognizer functions: corresponding to each is a C function written by the database administrator. The second column is the memoization flag. A "1" tells the TOMS to memoize that object class while a "0" tells it not to. This flag allows the database administrator flexibility in balancing the space vs. time trade off. In a large database high frequency objects (such as words) can be left unmemoized saving a great deal of index space while only slightly decreasing the TOMS efficiency since the object's parent (e.g., sentence) remains memoized.

2.4.3. Creating a Marking

The installation of a document into our database is carried out by the simple TOMS client program shown in Figure 4. We assumed the document to be contained in a file named `mbox`. (In all sample code, error checking is omitted for clarity.)

```
#include <stdio.h>
#include <toms.h>

int main ()
{
  long mid, rid;
  char *docfiles[] = { "mbox" };

  TomsInit();

  mid = MarkInit("email");
  rid = DocInit(mid, 1, docfiles, NULL);

  printf(stderr, "rid\t%lu\n", rid);

  TomsExit();

  exit(0);
}
```

Figure 4: Initialization Function

The program begins by calling the TOMS function `TomsInit` which reads in some auxiliary data files. Then `MarkInit` compiles the grammar and recognizer files with the base name "email" (i.e. the files `email.mk` and `email.rf` in Figures 2 and 3), and returns the unique marking identifier, *mid*. This identifier is then passed to `DocInit` along with the number of files (1) and the array of filenames that make up this document, *docfiles*. `DocInit` returns the root ID — an identifier which is required for further access to this document; in this example program it is simply printed.

When this initialization is complete, the TOMS database is ready to be used by TOMS client programs.

3. Using Documents

Once a document is registered with a marking in the TOMS database, any TOMS client can access the structure, the textual objects, and the actual text of the document using functions in the TOMS library. Continuing our example from the previous section, we now look at TOMS client programs written for the e-mail database whose grammar and recognizer functions were defined above.

3.1. Creating Secondary Indexes with the TOMS

The first application that we describe here is a very common one for any information retrieval system — using the TOMS in the creation of a secondary index.

We maintain the abstraction barrier between primary and secondary indexes by using a separate program to actually create the latter. (Depending on the secondary indexing package used, this program can be as short as two or three lines of code.) We will assume that this indexing program reads a file of lines, each containing the text of a word (i.e., a word type), a tab character, and a token representative. Of course we use TOMS object IDs as token representatives.

So, the TOMS client we write must create the file that maps types to tokens. Since the more complicated details of structure and object recognition have already been taken care of by the grammar and recognizer functions, this program is easy to write and maintain.

For our e-mail database we are interested in the words actually written by the sender of the message, thus we've chosen to index the words contained in the "Subject" header and in the body of the message. The processing is carried out by the function `ListWords`, which takes the root ID as an argument and prints, for each word of interest, the text of the word and the TOMS object that it corresponds to. This function is sketched below:

```
ListWords(id_t rootid)
{
  /* Identify the cursors needed. */

  /* Output the words of the body. */

  /* Output the words of the Subject line. */
}
```

The rest of this section will describe in greater detail how `ListWords` uses the TOMS to accomplish this task. The full code of all example functions are included in the appendix.

3.1.1. Finding Cursors of Interest

The first job of any TOMS client is to identify the object classes of the marking which are of interest to the program. A client program can keep track of any number of positions in a marking using data structures called *cursors*. Each cursor represents a particular class of textual objects in a particular location in the document structure. Any client will need access to the root of the marking (the top node of the cursor tree), so we refer to this important position as the *root cursor*. The root cursor generally corresponds to the document object class. Even though common types of textual objects such as "words" usually occur in more than one place in the grammar (and thus the marking), each occurrence can have a separate and distinct cursor. This means that words in a sentence in the body of a mail message and words in the Subject line of the message can be distinguished by the TOMS. The root cursor is returned by the TOMS function `marking`, which takes the root ID as it's only argument.

```
root_cur = marking(rid);
```

Given the root cursor, all other cursors can be accessed either by position using tree traversal functions (`Next`, `Prev`, `Parent`, and `Child`) or by label using the `Project` function. `Project` returns the first cursor which is a descendant of the given cursor and which matches the given label. `ListWords` retrieves only those cursors needed to find the object classes we wish to index:

```
body_cur = Project(root_cur, "body");
word_cur = Project(body_cur, "word");
nahead_cur = Project(root_cur, "nahead");
label_cur = Project(nahead_cur, "label");
val_cur = Project(nahead_cur, "value");
vw_cur = Project(val_cur, "valword");
```

3.1.2. Using Context to Find Textual Objects

The heart of the TOMS' functionality lies in the `Context` function. It is `Context` that drives the recognizer functions, identifying instances of object classes and determining the relationships between various textual objects. `Context` can be thought of as "converting" a given textual object to some set of other objects of another class which is an ancestor or descendent of the first. That is, `Context` describes a containment relation between objects in the grammar. Given a sentence object for example, `Context` can tell you all the word objects which are contained in it, or which paragraph object it is itself contained in.

The `Context` function takes an object and a cursor as arguments and returns a list of objects which contain or are contained by the given object; the cursor determines the type of object returned. If `Context` is given the document itself as the object, it returns *all* instances of the given cursor in the text. In order to start using `Context` in any way, however, the client needs at least one object as a reference. Thus TOMS provides the bootstrapping function `RootObject`.

```
root_obj = RootObject(root_cur);
```

`ListWords` is now ready to find the words in the bodies of the e-mail messages of our database. Since we want *all* the instances of the cursor *word_cur* — all the words in the message bodies — we can call `Context` with *root_obj* as the reference object.

```
word_ol = Context(root_obj, word_cur);
```

`Context` returns an object list, in this case *word_ol*, which is manipulated using special TOMS object-list functions. Here we use `GetObject` in a `while` loop to retrieve the objects in the list sequentially, and for each one, print the text of the word, returned by the TOMS `Get` function, and an external (string) form of the TOMS object, given by `ObjectId`.

```
while( (word_obj =
  GetObject(word_ol)).o_cursor
  != NULL )
{
  printf("%s\t%s\n", Get(word_obj),
    IdtoStr(ObjectId(word_obj)));
}
```

Finding the words of the "Subject" header is slightly more complicated as we must first find the correct header line through comparison of the label object, and then parse the value field of the matching header. Since the header we are interested in is not an address, we start by finding the list of non-address headers in the document using `Context`:

```
nahead_ol = Context(root_obj, nahead_cur);
```

Now we iterate through the list much as we did for *word_ol*, but for each header we must find it's corresponding label and compare it to our desired label. In the case of a match we retrieve the value field of that header and break it down into value-words (`valwords`).

```
while( (nahead_obj =
  GetObject(nahead_ol)).o_cursor
  != NULL )
{
  label_ol = Context(nahead_obj, label_cur);
  label_obj = GetObject(label_ol);
  if( strcmp(Get(label_obj),"Subject") == 0 )
  {
    val_ol = Context(nahead_obj, val_cur);
    val_obj = GetObject(val_ol);

    vw_ol = Context(val_obj, vw_cur);
    while( (vw_obj =
       GetObject(vw_ol)).o_cursor
       != NULL )
    {
      printf("%s\t%s\n", Get(vw_obj),
          IdtoStr(ObjectId(vw_obj)));
    }
  }
}
```

Figure 5: Indexing Subject Line Words

With the words of the Subject line, the program is complete. The resulting index can now be used for retrieval purposes, but by using the TOMS objects the index also provides the means to further access the full structure of the texts.

3.2. Further Retrieval Tasks with the TOMS

Not only is the TOMS an aid to creating a secondary index, but now that our e-mail database is indexed with TOMS objects, we can write further clients with the TOMS to gain more information about words of interest. If, for example, we wish to take a list of word tokens (i.e., TOMS objects from our index) and retrieve for each the list of all words which occur in the same sentence, we could write a TOMS function like our next example, `SentenceOf`. `SentenceOf` takes an external TOMS object as it's argument, finds the sentence that contains it, and prints out the list of words in that sentence. If the word is in a Subject line, `SentenceOf` prints out the words contained in that Subject.

In this case the object which our client program starts with is not the document, but instead the word object *extw_obj* (the hit) passed into it from the outside. Thus `SentenceOf`'s first step is to convert this external TOMS object back to it's internal representation with `OInternalize`.

```
w_obj = OInternalize(extw_obj, NULL);
```

Now we want to find the parent of the cursor of the object *w_obj*. *parent_cur* will either be a sentence cursor (from the body) or a value cursor (from a non-address header).

```
w_cur = CursorOf(w_obj);
parent_cur = parent(w_cur);
while( GetLabel(parent_cur) == NULL)
     parent_cursor = parent(parent_cur);
```

The while loop assures that we skip over any NULL (purely structural) nodes.

With object and cursor in hand we are ready to apply `Context` again, this time going up the marking to find the parent object (either a sentence or a Subject header value).

```
parent_ol = Context(w_obj, parent_cur);
parent_obj = GetObject(parent_ol);
```

Since there is only one parent object, we only have to call `GetObject` once. Now we reverse the `Context` to retrieve the list of words in the parent and print them out.

```
w_ol = Context(parent_obj, w_cur);
while((w_obj = GetObject(w_ol)).o_cursor
  != NULL )
{
  printf("%s\n", Get(w_obj) );
}
```

The simplicity of this function (note that it is only 12 lines long) points out the facility with which the TOMS allows application programmers to access textual structure.

4. Implementation Considerations

The concept of a TOMS is innovative in itself. The implementation of the TOMS, however, contains several innovations that are worthy of mention. In this section, we describe some of the more important aspects of this implementation, with particular attention to their impact on time and space performance.

4.1. Memos

One of our most important design goals was that access to indexed objects should be very fast. Pragmatically, this means that the system is careful to keep the number of disk accesses required to access a textual object to a minimum — less than one on the average. But another major goal was that while textual objects will in general be indexed this indexing needn't be done ahead of time. These two goals together imply a dynamic, incremental indexing scheme, akin to the programming language optimization known as *memoization*.[9]

The rationale for memoization is simple: indexes take up space on disk, and they also take time to precompute. It may well be the case that, for a collection with complex structure, certain classes of textual objects may not be manipulated as often as others. For these objects, it can save space and initialization time to put off the indexing of these objects until they are actually used.

If an unindexed object is referenced by a client, the TOMS will have to find that object by scanning text. Of course, we don't want the system to slow down by indexing all objects in the process of finding the one that was referenced: hence the incremental nature of the memoization. As more higher-level textual objects are memoized (either incrementally or ahead of time), the TOMS needs to do less scanning to find lower-level objects.

For example, if chapters have been memoized but words haven't been, and the last sentence in the document is requested, text scanning will start at the beginning of the last chapter rather than at the beginning of the text. In addition, all objects of the type being searched for which are recognized in the process will be memoized: that is, the next time any one of them is requested by the client, it will be retrieved as quickly as if it had been preindexed. Because of this behavior, a database administrator may want to precompute high-level memos even if most memoization is left to be done incrementally; the highest level memos also tend to be very small.

For this reason we give the database administrator a lot of flexibility in determining exactly how to index her collection; for each class of textual object, she must choose:

- to preindex all of these objects; or
- to leave them unindexed, but allow indexes to be built dynamically, in response to client actions, via memoization; or
- to specify that this class of objects never be indexed — that they should always be scanned for.

Finally, individual TOMS indexes are designed to occupy at most a fraction of the size of the text. This is achieved partially by only storing compressed versions of memos on disk. Memos are read and written in blocks, and compression (or decompression) is applied when an individual block is written (or read). The TOMS comes configured to use LZW[15] compression, but since the compression and decompression is performed by a separate process, the database administrator can use any compression method available.

4.2. The TOMS as Tool

We see the TOMS as a toolkit that can be used in a number of different ways. The lowest level interface to the TOMS is a library of C functions which can be called directly from C programs and other compiled languages. Using the TOMS in this fashion is straightforward, but writing a simple application does require the usual edit–compile–-debug cycle. This is especially inconvenient for a TOMS database administrator who needs to write (or use) many simple TOMS applications to create, maintain, and test TOMS collections.

Because of this inconvenience, we have compiled the TOMS library into a customized interpreter for the Perl programming language. Each TOMS function appears as a Perl primitive, some slightly modified.

Perl is an interactive, high-level language that provides many facilities for both systems programming and text manipulation. Useful features include persistent associative arrays (lookup tables) which are implemented as disk-based hash tables, regular expression pattern matching, and interprocess communication (IPC) and networking primitives. Perl's dynamic memory management means that the programmer need not bother with storage allocation and deallocation and its consequent bugs.

However, Perl still encourages the use of text-scanning for the recognition of text structure. Since our version of Perl allows us to use the TOMS for this purpose, we can exploit Perl's other features and still be assured of efficient access to large volumes of text. This special Perl interpreter has proved useful for many tasks; we have used it already to write a number of applications to support our retrieval system. For example, the IPC facilities allowed us to write prototype and production versions of a server that provides Internet-wide access to the text of the TLF, stored on a CD-ROM and mounted on a local machine. Also, the persistent associative arrays of Perl have been used to write the short secondary indexing programs mentioned in §3.1.

5. Future Work

One future direction for the TOMS, inspired by our success with Perl, is to integrate it into other high level languages such as Tcl[11], Icon, and Alfonzo[16] — each of which would benefit from the TOMS primitives themselves while providing more programming flexibility to TOMS client programmers.

5.1. A TOMS Server

The text server described in §4.2 provides TOMS access to any client anywhere on the Internet. In our current application it allows us to isolate access to our database, which is important to us for reasons of efficiency and security. The server language is a simple line-oriented language that's very easy to parse; client programs that interact with the server can be as short as 1 or 2 lines and can be written in any language, even if it doesn't support linking with the TOMS library.

Currently the server only provides a limited subset of TOMS functionality: it basically allows contextualization to documents, and `SubGet`'s of the documents' text. We will be extending the server language so that it provides complete access to all TOMS facilities.

5.2. Graph Structure / Hypertext

There remain several interesting extensions that can be made to the TOMS itself. At present, the TOMS' abstraction of document structure is strictly hierarchical. Although hierarchical relationships appear to account for a large portion of the structure of most collections of documents, there are certainly non-hierarchical relationships that we would like to be able to describe using the TOMS. Footnotes and recursive containment (e.g., an electronic mail message that contains a forwarded electronic mail message) are obvious examples. A more modern example is hypertext, where text is organized as nodes in a graph rather than as a tree. In future work, we intend to explore the consequences of such structures in the context of applications that require them.

5.3. Dynamicity

Although the TOMS is in many senses very dynamic — the TOMS' memoization makes use of this dynamicity to achieve time and space efficient access of textual objects — there are some areas in which the TOMS would benefit from more dynamicity.

The most obvious such area is markings. Once a marking is compiled, the TOMS assumes that it never changes. The reason for this assumption is that, once a document is known to the TOMS, the TOMS guarantees to its clients that the objects within it will persist. It is clear, however, that markings should be allowed to change. One can easily imagine a user wanting to add a new object class to an existing set of documents, for example. For the TOMS to allow such evolution, one of two things must be true.

- It must be able to either detect that objects that were once part of a document no longer exist, and be able to notify clients when they ask for obsolete objects.
- It must be able to guarantee that no object ever becomes obsolete, perhaps by effectively limiting changes to markings to be only additive.

Either possibility entails further development.

Another area where TOMS could use greater dynamicity is in the memoization itself. While the memos are created dynamically, there are as yet no facilities within the TOMS for deleting or altering already created memos. Thus in order to use the TOMS on a fully dynamic database like a newswire feed, one where texts can be deleted and edited as well as being added, these facilities need to be added to the TOMS current memoization system.

6. Conclusions

In previous work[4, 6] our group focused on the separation of the user interface from the retrieval engine by using an interactive programmable language as the interface between the two; current research extends this work to a more powerful and expressive programming language paradigm.[16] The TOMS represents an improvement in information retrieval systems by identifying a new level of abstraction and clearly separating the representation of text structure from the algorithms used to manipulate it. The importance of this division is that it allows us to write retrieval engines that are highly independent of the data representation. Secondary indexing programs can be written with no knowledge of the format of the text, relying solely on the TOMS to provide primary object access. The retrieval software proper can ask the TOMS about the structure of a given text, pose user queries in terms of the revealed relations of textual objects, and display results identified not in terms of lines and file names, but in terms of the actual discourse elements used in the text. In addition, the efficiency of the TOMS primitives we have added to Perl has allowed us to use a higher level, interactive programming language where previously we were forced to use C to gain acceptable performance.

We feel that the resulting system meets the goals of efficiency and flexibility that we set for it. It is quicker, easier, and much more efficient to describe the structure of a textual database using the TOMS than using any of the other tools that we had previously used. It has especially had the effect of making the *clients* that we have written subsequent to developing the TOMS simpler, more coherent, and better structured.

References

1. Free Software Foundation, *GNU dbm, Release 1.4,* FSF, Cambridge, MA, 1989. Computer software.
2. Aho, Alfred V., Brian W. Kernighan, and Peter J. Weinberger, *The AWK Programming Language,* Addison Wesley, Reading, MA., 1988.
3. Ansen, Debra, ‘‘Document Architecture Standards Evolution,’’ *AT&T Technical Jouurnal*, vol. 68, no. 4, pp. 33-44, July/August 1989.
4. Bookstein, Abraham, Scott Deerwester, Robert Morrissey, Keith Waclena, and Donald Ziff, ‘‘A System for Integrated Bibliographic and Full-text Retrieval in a Distributed Computing Environment,’’ in *Computers and the Humanities: Today's Research, Tomorrow's Teaching*, pp. 285-291, University of Toronto, Toronto, March, 1986.
5. Crocker, D., ‘‘Standard for the Format of ARPA Internet Text Messages,’’ RFC-822, Network Information Center, 1982.
6. Deerwester, Scott C., Donald A. Ziff, and Keith Waclena, ‘‘An Architecture for Full Text Retrieval Systems,’’ *DEXA 90: Database and Expert Systems Applications*, pp. 22-29, Vienna, September 1990.
7. Goldstein, C. M., ‘‘Online Reference Works and Full Text Retrieval,’’ *National Online Meeting Proceedings*, pp. 171-177, Medford, N. J., 1989.
8. Griswold, Ralph E. and Madge T. Griswold, *The Icon Programming Language,* Prentice-Hall, Englewood Cliffs, New Jersey, 1983.
9. Michie, Donald, ‘‘‘‘Memo’’ Functions and Machine Learning,’’ *Nature*, vol. 218, pp. 19-22, 6 April 1968.
10. Morrissey, Robert and C. del Vigna, ‘‘A Large Natural Language Data Base,’’ *Educom Bulletin*, vol. 18, no. 1, pp. 10-13, Spring 1983.
11. Ousterhout, John K., ‘‘Tcl: An Embedded Command Language,’’ *USENIX Proceedings*, pp. 133-146, Winter 1990.
12. Smith, John B., Stephen F. Weiss, and Gordon J. Ferguson, *MICROARRAS: An Advanced Full-Text Retrieval and Analysis System,* University of North Carolina Department of Comuter Science, Chapel Hill, N. C., [n.d.].
13. Thompson, Ken, ‘‘Regular Expression Search Algorithm,’’ *CACM*, vol. 11, no. 6, pp. 419-422, June 1968.
14. Wall, Larry and Randal L. Schwartz, *Programming Perl,* A Nutshell Handbook, O'Reilly and Associates, Sebastopol, CA, 1991.
15. Welch, Terry A., ‘‘A Technique for High-Performance Data Compression,’’ *Computer*, vol. 17, no. 6, pp. 8-19, June 1984.
16. Ziff, Donald A., Keith Waclena, and Stephen P. Spackman, ‘‘Using a Lazy Functional Language for Textual Information Retrieval,’’ TR 92-02, University of Chicago Center for Information and Languages Studies, Chicago, 1992.