Using SGML as a Basis for Data-Intensive Natural Language Processing

D. McKelvie, C. Brew and H.S. Thompson
Language Technology Group, Human Communication Research Centre, University of Edinburgh

February 2, 1999

Abstract. This paper describes the LT NSL system (McKelvie et al, 1996), an architecture for writing corpus processing tools. This system is then compared with two other systems which address similar issues, the GATE system (Cunningham et al, 1995) and the IMS Corpus Workbench (Christ, 1994). In particular we address the advantages and disadvantages of an SGML approach compared with a non-SGML database approach.

Key words: SGML, Natural language processing, Corpus-based linguistics

Abbreviations: SGML - Standard Generalised Markup Language; nSGML - Normalised SGML; TEI - Text Encoding Initiative; DTD - Document Type Description; NLP - Natural Language Processing.

1. Introduction

The theme of this paper is the design of software and data architectures for natural language processing using corpora. Two major issues in corpus-based NLP are: how best to deal with medium to large scale corpora, often with complex linguistic annotations; and what system architecture best supports the reuse of software components in a modular and interchangeable fashion.

In this paper we describe the LT NSL system (McKelvie et al, 1996), an architecture for writing corpus processing tools, that we have developed in an attempt to address these issues. This system is then compared with two other systems which address some of the same issues, the GATE system (Cunningham et al, 1995) and the IMS Corpus Workbench (Christ, 1994). In particular we address the advantages and disadvantages of an SGML approach compared with a non-SGML database approach. Finally, in order to back up our claims about the merits of SGML-based corpus processing, we present a number of case studies of the use of the LT NSL system for corpus preparation and linguistic analysis.

The main focus of this paper is on software architectural issues which will be of primary interest to developers of NLP software. Nevertheless, the issues addressed have relevance both for the choice of corpus annotation schemes and for (non-computationally minded) users of NLP software such as KWIC viewers, automatic taggers, statistical profiling tools and (semi-)automatic annotation software. Our basic position is that one should annotate your corpus using SGML because it is descriptively adequate and it allows the use of generic software across different corpora, i.e. reuse of software and user interfaces.

2. SGML

Standard Generalised Markup Language (SGML) (Goldfarb, 1990) is a meta-language for defining markup languages. A markup language is a way of annotating the structure (of some sort) of a class of text documents. SGML was originally designed as a kind of text formatting language that drew a clear distinction between logical structure and formatting issues. However, its flexibility and the ease with which new markup schemes could be defined (by means of DTDs (Document Type Descriptions)[1]) meant that SGML was quickly taken up by linguists who were working with large computer readable corpora as a good solution to their needs for a standardised scheme for linguistic annotation.

2.1. A brief introduction to SGML

SGML is a standard for annotating texts. It allows one to decompose a document into a hierarchy (tree structure) of *elements*. Each element can contain other elements, plain text or a mixture of both. Associated with each element is a set of *attributes* with associated values. These attributes allow one to add further information to the markup, e.g. to define subtypes of elements. Attributes can also be used to link together elements across the document hierarchy. *Entities* provide a structured means of defining and using abbreviations. Elements, attributes and entities are user definable, thus providing the flexibility to support any markup scheme. They are defined in a DTD that also includes a grammar defining which elements can occur inside other elements and in which order. The following example of SGML shows the basic structure, with SGML markup shown in bold.

<S ID=S1>
<PHR ID=P1>The cat**</PHR>** sat on the mat. **</S>**
<S ID=S2>
<PRO ID=PRO1 REF=P1 GENDER=N>It**</PRO>**
was sleeping. **</S>**

[1] SGML's way of describing the structure (or grammar) of the allowed markup in a document.

The example consists of two S elements, representing (in this case) sentences with associated ID attributes. The first S contains a PHR element and some text. The second contains a PRO element. The REF attribute on the PRO refers back to the element with an ID attribute of 'P1', i.e. the PHR element.

2.2. SGML AND COMPUTATION LINGUISTICS

We use SGML in the context of collecting, standardising, distributing and processing very large text collections (tens and in some case hundreds of millions of words in size) for computational linguistics research and development. The LT NSL API and associated tools were developed to meet the needs that arise in such work, in the first instance for the creation, elaboration and transformation of markup for such collections (usually called corpora). Not only are the corpora we work with large, they have a very high density of markup (often each word has associated markup). We needed to support pipelined transformation programs running in batch mode (to allow modular, distributed software development) and specialised interactive editors to allow hand correction of automatically added markup.

Given the increasingly common use of SGML as a markup language for text corpora, the question arises as to what is the best way of processing these corpora. For example, the task (common in linguistic applications) of tokenising a raw corpus, segmenting out the words therein and then looking the results up in a lexicon, becomes more complex for SGML marked-up corpora (as indeed for any marked up corpus).

The approach that we have taken in the LT NSL library is that SGML markup should not only be retained and used as the input and output format for tool pipelines, but should also be used for inter-tool communication. This has the advantage that SGML is a well defined language which can be used for any markup purpose. Its value is precisely that it closes off the option of a proliferation of ad-hoc markup notations and their associated I/O software. A second advantage is that it provides a notation that allows an application to access the document at the right level of abstraction, attending to text and markup that are relevant to its needs, and ignoring that which is not. LT NSL defines a query language and retrieval functions that make the selection of relevant text content a straight forward task.

3. The LT NSL system

LT NSL is a tool architecture for SGML-based processing of (primarily) text corpora. It generalises the UNIX pipe architecture, making it possible to use pipelines of general-purpose tools to process annotated corpora. The original UNIX architecture allows the rapid construction of efficient pipelines of conceptually simple processes to carry out relatively complex tasks, but is restricted to a simple model of streams as sequences of bytes, lines or fields. LT NSL lifts this restriction, allowing tools access to streams which are sequences of tree-structured text (a representation of SGML marked-up text).

The use of SGML as an I/O stream format between programs has the advantage that SGML is a well defined standard for representing structured text. The most important reason why we use SGML for all corpus linguistic annotation is that it forces us to formally describe the markup we will be using and provides software for checking that these markup invariants hold in an annotated corpus. In practise this is extremely useful. SGML is human readable, so that intermediate results can be inspected and understood. Although it should be noted that densely annotated text is difficult to read, and for ease of readability really requires transformation into a suitable display format. Fortunately, customisable display software for SGML is readily available. It also means that it is easy for programs to access the information that is relevant to them, while ignoring additional markup. A further advantage is that many text corpora are available in SGML, for example, the British National Corpus (Burnage&Dunlop, 1992).

However, using SGML as the medium for inter-program communication has the disadvantage that it requires the rewriting of existing software, for example, UNIX tools which use a record/field format will no longer work. It is for this reason that we have developed an API library to ease the writing of new programs.

The LT NSL system is released as C source code. The software consists of a C-language Application Program Interface (API) of function calls, and a number of stand-alone programs that use this API. The current release is known to work on UNIX (SunOS 4.1.3, Solaris 2.4 and Linux), and a Windows-NT version will be released during 1997. There is also an API for the Python programming language.

One question that arises in respect to using SGML as an I/O format is: what about the cost of parsing SGML? Surely that makes pipelines too inefficient? Parsing SGML in its full generality, and providing validation and adequate error detection is indeed rather hard. For efficiency reasons, you wouldn't want to use long pipelines of tools, if each tool had to reparse the SGML and deal with the full language. Fortunately, LT NSL doesn't require this, since parsing SGML is easy and fast if you handle only a subset of the full notation, and don't validate, i.e. assume valid input.

Accordingly, the basic architecture underlying our approach is one in which arbitrary SGML documents are parsed only once using a full validating SGML parser, regardless of the length of the tool pipeline used to process them. This initial parsing produces two results:

1. An optimised representation of the information contained in the document's DTD, cached for subsequent use;
2. A normalised version of the document instance, that can be piped through any tools built using our API for augmentation, extraction, etc.

Thus the first stage of processing *normalises* the input, producing a simplified, but informationally equivalent form of the document. Subsequent tools can and often will use the LT NSL API which parses normalised SGML (henceforth nSGML) approximately ten times more efficiently than the best parsers for full SGML. The API then returns this parsed SGML to the calling program as data-structures (see figure 1).

nSGML is a fully expanded text form of SGML informationally equivalent to the ESIS output of SGML parsers. This means that all markup minimisation is expanded to its full form, SGML entities are expanded into their value (except for SDATA entities), and all SGML names (of elements, attributes, etc) are normalised. The result is a format easily readable by humans and programs. The translation into nSGML is a space-time trade-off. Although we can expect that processing will be faster, the normalised SGML input files will tend to be larger than unnormalised SGML if markup minimisation has been extensively used.

The LT NSL programs consist of `mknsg`, a program for converting arbitrary valid SGML into normalised SGML[2], the first stage in a pipeline of LT NSL tools; and a number of programs for manipulating normalised SGML files, such as `sggrep` which finds SGML elements that match some query. Other of our software packages such as LT POS (a part of speech tagger) and LT WB (Mikheev&Finch, 1997) also use the LT NSL library.

In addition to the normalised SGML, the `mknsg` program writes a file containing a compiled form of the DTD, which LT NSL programs read in order to know what the structure of their nSGML input or output is.

How fast is it? Processes requiring sequential access to large text corpora are well supported. It is unlikely that LT NSL will prove the rate limiting step in sequential corpus processing. The kinds of repeated search required by lexicographers are more of a problem, since the

[2] `mknsg` is based on James Clark's SP parser (Clark, 1996).

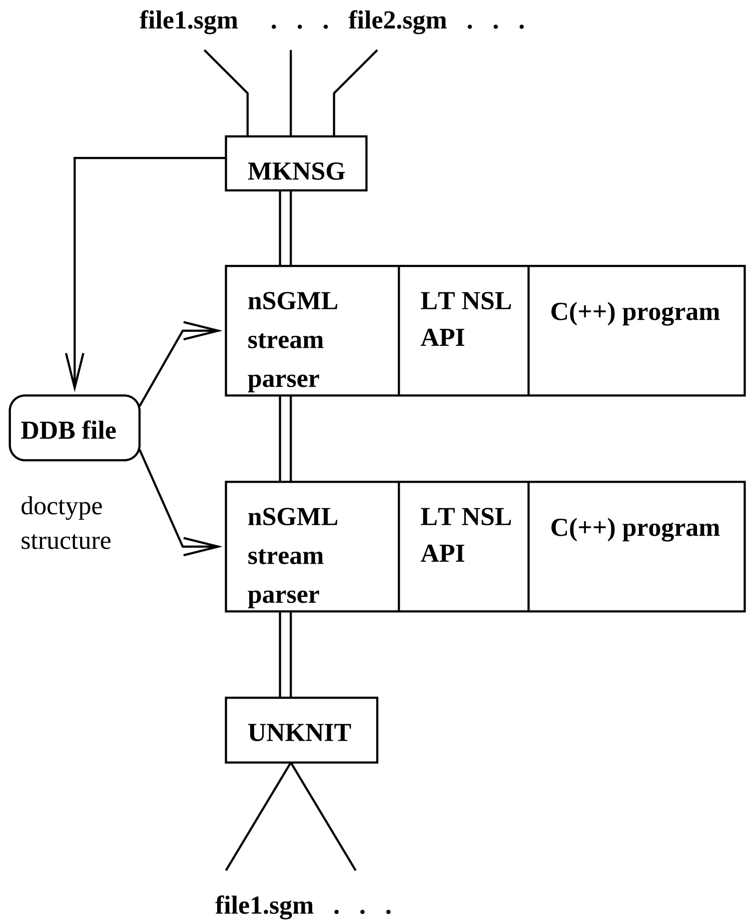

Figure 1. Dataflows in a pipeline of LT NSL programs.

system was not designed for that purpose. The standard distribution is fast enough for use as a search engine with files of up to several million words. Searching 1% of the British National Corpus (a total of 700,000 words (18 Mb)) is currently only 6 times slower using LT NSL `sggrep` than using `fgrep`, and `sggrep` allows more complex structure-sensitive queries. A prototype indexing mechanism (Mikheev&McKelvie, 1997), not yet in the distribution, improves the performance of LT NSL to acceptable levels for much larger datasets.

Why did we say "primarily for text corpora"? Because much of the technology is directly applicable to multimedia corpora such as the Edinburgh Map Task corpus (Anderson et al, 1991). There are tools that interpret SGML elements in the corpus text as offsets into files of audio-data, allowing very flexible retrieval and output of audio information using queries defined over the corpus text and its annotations. The same could be done for video clips, etc.

In passing, we should say that there is a great similarity between the idea of normalised SGML as defined here and the Extensible Markup Language (XML) being defined by the WWW Consortium(Bray & Sperberg-McQueen, 1996), which is intended to be an easily processed subset of SGML suitable for WWW page annotation. The current release of the LT NSL library is capable of processing XML documents.

3.1. Hyperlinking

We are inclined to steer a middle course between a monolithic comprehensive view of corpus data, in which all possible views, annotations, structurings etc. of a corpus component are combined in a single heavily structured document, and a massively decentralised view in which a corpus component is organised as a hyper-document, with all its information stored in separate documents, utilising inter-document pointers. Aspects of the LT NSL library are aimed at supporting this approach, which was first formulated in the context of the MULTEXT project (Ballim & Thompson, 1995), (Ide, 1995). It is necessary to distinguish between *files*, which are storage units, *(*SGML*) documents*, which may be composed of a number of files by means of external entity references, and *hyper-documents*, which are linked ensembles of documents, using e.g. HyTime or TEI (Sperberg-McQueen&Burnard, 1994) link notation.

The implication of this is that corpus components can be hyper-documents, with low-density (i.e. above the token level) annotation being expressed indirectly in terms of links.

There are at least three reasons why separating markup from the material marked up ("standoff annotation") may be an attractive proposition:

1. The base material may be read-only and/or very large, so copying it to introduce markup may be unacceptable. This however requires either on-line normalisation during the process of accessing hyperlinked data or that the base material is already in a normalised form.

2. The markup may involve multiple overlapping hierarchies[3];

3. Distribution of the base document may be controlled, but the markup is intended to be freely available.

[3] For example, lines and sentences in poetry, inline footnotes, transcriptions of multi-party dialogues, multi-media corpora.

Here, we introduce two kinds of semantics for hyperlinks to facilitate this type of annotation, and describe how the LT NSL toolset supports these semantics.

The two kinds of hyperlink semantics which we describe are (a) *inclusion*, where one includes a sequence of SGML elements from the base file; and (b) *replacement*, where one provides a replacement for material in the base file, incorporating everything else.

The crucial idea is that standoff annotation allows us to distribute aspects of virtual document structure, i.e. markup, across more than one actual document, while the pipelined architecture allows tools that require access to the virtual document stream to do so.

This can be seen as taking the SGML paradigm one step further: Whereas SGML allows single documents to be stored transparently in any desired number of entities; our proposal allows virtual documents to be composed transparently from any desired network of component documents.

Note that the examples which follow use a simplified version of the draft proposed syntax for links from the XML-LINK draft proposal (Bray & DeRose, 1997), which notates a span of elements with two TEI extended pointer expressions separated by two dots ('..').

3.1.1. *Adding markup from a distance*

Consider marking sentence structure in a read-only corpus of text which is marked-up already with tags for words and punctuation, but nothing more:

```
. . .
<w id='w12'>Now</w><w id='w13'>is</w><w id='w14'>the</w>
. . .
<w id='w27'>the</w><w id='w28'>party</w><c id='c4'>.</c>
```

With an inclusion semantics, we can mark sentences in a separate document as follows:

```
. . .
<s xml-link='simple' href="#ID(w12)..ID(c4)"></s>
<s xml-link='simple' href="#ID(w29)..ID(c7)"></s>
. . .
```

Now crucially (and our LT NSL and LT XML products already implement this semantics), we want our application to see this document collection as a single stream with the words nested inside the sentences:

```
. . .
<s>
 <w id='w12'>Now</w><w id='w13'>is</w><w id='w14'>the</w>
 . . .
 <w id='w27'>the</w><w id='w28'>party</w><c id='c4'>.</c>
```

```
</s>
<s>
 . . .
</s>
```

Note that the linking attribute is gone from the <S> start-tag, because its job has been done.

We believe this simple approach will have a wide range of powerful applications. We are currently using it in the development of a shared research database, allowing the independent development of orthogonal markup by different sub-groups in the lab.

3.1.2. *Invisible Mending*

We use *inverse replacement* semantics for e.g. correcting errors in read-only material. Suppose our previous example actually had:

```
. . .
<w id='w15'>tiem</w>
. . .
```

If we interpret the following with inverse replacement semantics:

```
<mend xml-link='simple' href="#ID(w15)">
 <w id='w15'>time</w>
</mend>
```

We mean "take everything from the base document except word 15, for which use my content". In other words, we can take this document, and use it as the target for the references in the sentence example, and we'll get a composition of linking producing a stream with sentences containing the corrected word(s).

3.1.3. *Multiple point linking*

There are more uses of hyperlinks than the inclusion and replacement uses discussed above. For example, in multilingual alignment, one may want to keep the alignment information separate from each of the single language documents e.g:

```
<ALIGNMENT REF1=Eng037 REF2=Fre066>
```

where REF1 and REF2 refer to paragraphs in the English and French documents respectively. The LT NSL API supports the accessing of the separate aligned paragraphs via their identifiers, although one does not necessarily want to expand all alignments into a virtual document structure.

3.1.4. *Hyperlinking and overlapping hierarchies*

Using the hyperlinking inclusion we can create invalid SGML if the included material is not well-formed, i.e. if it contains a start tag without a corresponding end tag. This is a problem which we have not yet addressed.

In terms of asking questions about the overlap of two different markup hierarchies (or views), the best we can do so far is to express both overlapping hierarchies in terms of a base SGML file which contains the material that is common to both views. It is then possible to read both view files in parallel and calculate the intersection (in terms of the base) of two sets of markup elements in each of the views, see for example the `intersect` program in the LT NSL release which addresses this problem.

3.2. `sggrep` AND THE LT NSL QUERY LANGUAGE

The API provides the program(mer) with two alternative views of an nSGML stream: an object stream view and a tree fragment view. The first, lower level but more efficient, views an SGML document as a sequence of 'events'. Events occur for each significant piece of the document, such as start (or empty) tags with their attributes, text content, end tags, and a few other bits and pieces. LT NSL provides data structures and access functions such as `GetNextBit` and `PrintBit` for reading and processing these events. Using this view, a programmer will write code to take place at these events. This provides a simple but often sufficient way of processing documents.

The alternative, higher level, view, lets one treat the nSGML input as a sequence of tree-fragments in the order defined by the order of the SGML start tags[4]. The API provides functions `GetNextItem` and `PrintItem` to read and write the next complete SGML element. It also provides a function:

```
GetNextQueryElement(infile,query,subquery,
regexp,outfile)
```

In this call, `query` is an LT NSL query that allows one to specify particular elements on the basis of their position in the document structure and their attribute values. The `subquery` and `regexp` allow one to specify that the matching element has a subelement matching the subquery with text content matching the regular expression. This function call returns the next SGML element which satisfies these conditions. Elements which do not match the query are passed through unchanged to `outfile`.

Under both models, processing is essentially a loop over calls to the API, in each case choosing to discard, modify or output unchanged each `Bit` or `Element`. For example, the function call:

```
GetNextQueryElement(infile,".*/TEXT/.*/P",
```

[4] As for the PRECEDING and FOLLOWING keywords in the XML-LINK proposal(Bray & DeRose, 1997).

Table I. Syntax of LT NSL query language

<table>
<tr><td><query></td><td>:=</td><td><term> ('/' <term>)*</td></tr>
<tr><td><term></td><td>:=</td><td><bTerm> '*'?</td></tr>
<tr><td><bTerm></td><td>:=</td><td><aTerm> ('|' <aTerm>)*</td></tr>
<tr><td><aTerm></td><td>:=</td><td><GI> <cond>?</td></tr>
<tr><td><GI></td><td>:=</td><td><elementName> | '.'</td></tr>
<tr><td><cond></td><td>:=</td><td>'[' (<index> | <atests> | <index> <atests>) ']'</td></tr>
<tr><td><index></td><td>:=</td><td><number></td></tr>
<tr><td><atests></td><td>:=</td><td><atest> (' ' <atest>)*</td></tr>
<tr><td><atest></td><td>:=</td><td><aname> ('=' <aval>)?</td></tr>
</table>

```
"P/.*/S","th(ei|ie)r", outfile)
```

would return the next <P> element dominated anywhere by <TEXT> at any depth, with the <P> element satisfying the additional requirement that it contain at least one <S> element at any depth with text containing at least one instance of 'their' (possibly mis-spelt).

3.2.1. *The* LT NSL *query language*

LT NSL queries are a way of specifying particular nodes in the SGML document structure. Queries are coded as strings which give a (partial) description of a path from the root of the SGML document (top-level element) to the desired SGML element(s). For example, the query

".*/TEXT/.*/P"

describes any <P> element that occurs anywhere (at any level of nesting) inside a <TEXT> element which, in turn, can occur anywhere inside the top-level document element.

A query is basically a path based on terms separated by "/", where each term describes an SGML element. The syntax of queries is defined in table 1.

That is, a query is a sequence of terms, separated by "/". Each term describes either an SGML element or a nested sequence of SGML elements. An item is given by an SGML element name, optionally followed by a list of attribute specs (in square brackets), and optionally followed by a "*". An item that ends in a "*" matches a nested sequence of any number of SGML elements, including zero, each of which match the item without the "*". For example "P*" will match a <P> element, arbitrarily deeply nested inside other <P> elements. The special GI "." will match any SGML element name. Thus, a common way of finding a <P> element anywhere inside a document is to use the query ".*/P". *Aname* (attribute name) and *aval* (attribute value) are as per SGML .

Table II. Example DTD

```
<!ELEMENT CORPUS - - (DOC+)>
<!ELEMENT DOC - - (DOCNO,TITLE,BODY,IT,NI) >
<!ELEMENT DOCNO - - (#PCDATA) >
<!ELEMENT TITLE - - (s+) >
<!ELEMENT BODY - - (s+) >
<!ELEMENT IT - - (#PCDATA) >
<!ELEMENT NI - - (#PCDATA) >
<!ELEMENT s - - (#PCDATA—w)* >
<!ELEMENT w - - (#PCDATA) >
<!ATTLIST BODY id ID #IMPLIED >
<!ATTLIST IT id ID #IMPLIED>
<!ATTLIST w rend CDATA #IMPLIED>
```

A term that consists of a number of *aTerms* separated by '|' will match anything that any one of the individual *aTerms* match.

A condition with an *index* matches only the *index*'th sub-element of the enclosing element. Index counting starts from 0, so the first sub-element is numbered 0. Conditions with *indices* and *atests* only match if the *index*'th sub-element also satisfies the *atests*. Attribute tests are not exhaustive, i.e. P[rend=it] will match <P n=45 rend=it> as well as <P rend=it>. They will match against both explicitly present and defaulted attribute values, using string equality. Bare *anames* are satisfied by *any* value, explicit or defaulted. Matching of queries is bottom-up, deterministic and shortest-first.

3.2.2. *Examples of* LT NSL *queries*

In this section we show some examples of LT NSL queries, assuming the DTD in table 2.

The SGML structure of a sample document that uses this DTD is shown in figure 2.

The query `CORPUS/DOC/TITLE/s` means all S elements directly under TITLE's directly under DOC. This is shown graphically in figure 3. The LT NSL query functions return the indicated items one by one until the set denoted by the query is exhausted.

The query `CORPUS/DOC/./s` means all s's directly under anything directly under DOC, as shown in figure 4.

The query `CORPUS/DOC/.*/s` means all s's anywhere underneath DOC. ".*" can be thought of as standing for all finite sequences of "." For the example document structure this means the same as CORPUS/DOC/./s, but in more nested structures this would not be the

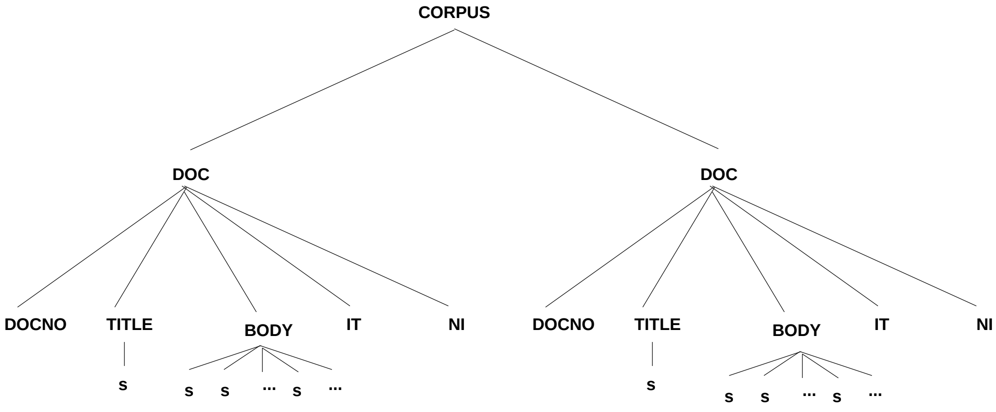

Figure 2. The hierarchical structure of an example document.

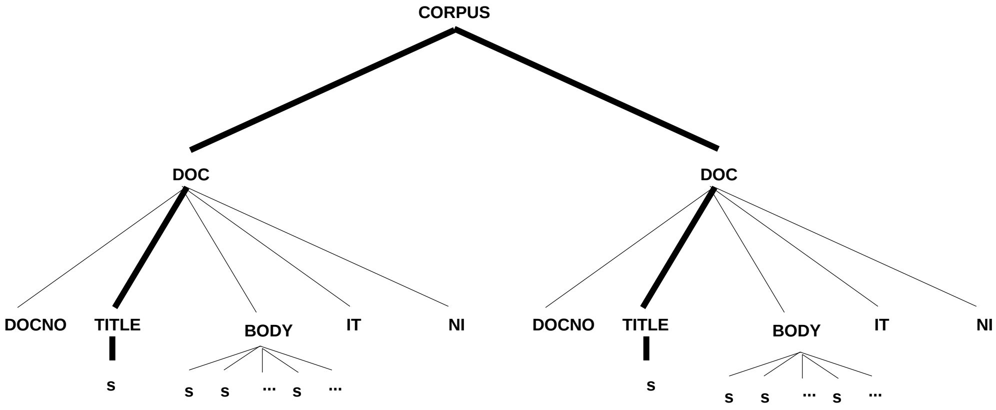

Figure 3. Result of query "CORPUS/DOC/TITLE/s"

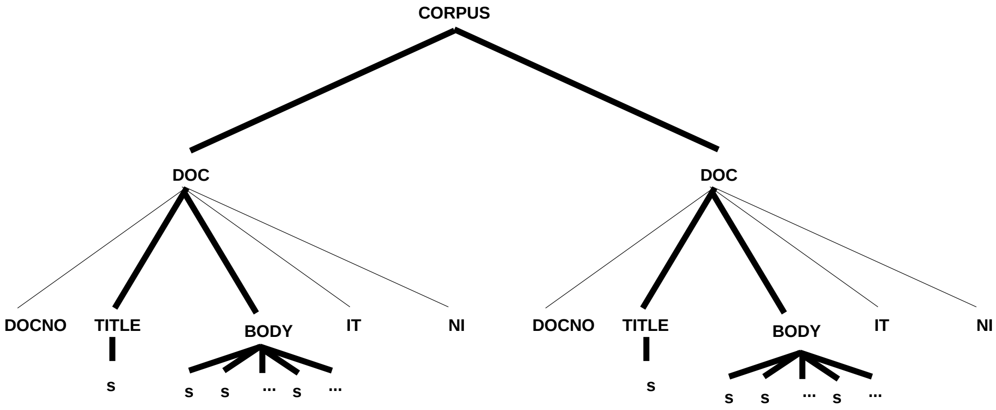

Figure 4. Result of query "CORPUS/DOC/.*/s"

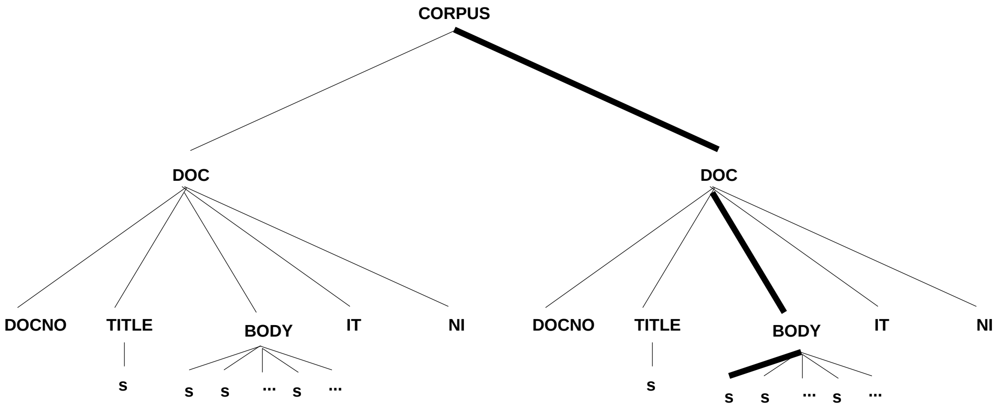

Figure 5. Result of query "./.[1]/.[2]/.[0]"

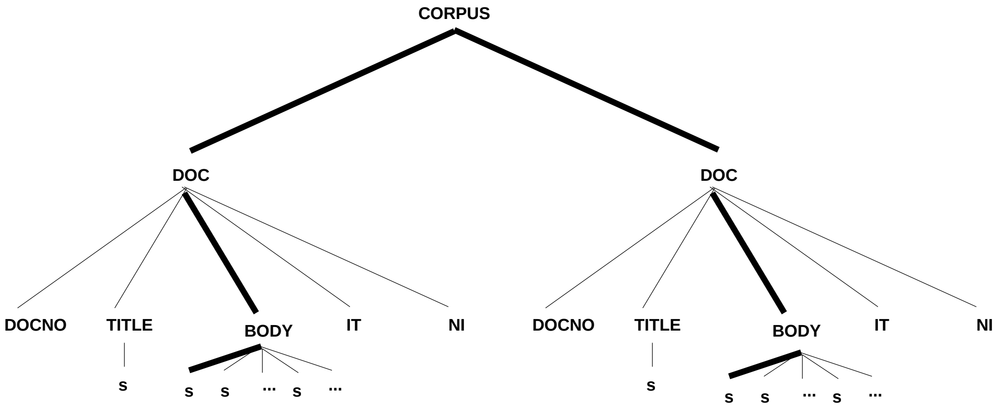

Figure 6. Result of query ".*/BODY/s[0]"

case. An alternative way of addressing the same sentences would be to specify .*/s as query.

We also provide a means of specifying the Nth node in a particular local tree. So the query `./.[1]/.[2]/.[0]` means the 1st element below the 3rd element below the 2nd element in a stream of elements, as shown in figure 5.

This is also the referent of the query `CORPUS/DOC[1]/BODY[2]/s[0]` assuming that all our elements are s's under BODY under DOC, which illustrates the combination of positions and types. The query `.*/BODY/s[0]` refers to the set of the first elements under any BODY which are also s's. The referent of this is shown in figure 6.

Additionally, we can also refer to attribute values in the square brackets: `.*/s/w[0 rend=lc]` gets the initial elements under any <s>

element so long as they are words with rend=lc (perhaps lower case words starting a sentence).

As will be obvious from the preceding description, the query language is designed to provide a small set of orthogonal features. Queries which depend on knowledge of prior context, such as "the third element after the first occurrence of a sentence having the attribute quotation" are not supported. It is however possible for tools to use the lower-level API to find such items if desired. The reason for the limitation is that without it the search engine might be obliged to keep potentially unbounded amounts of context.

Other systems have taken a different view of the tradeoffs between simplicity and expressive power and have defined more complex query languages over SGML documents. For example, the SgmlQL language (Le Maitre et al, 1996) based on SQL, the `sgrpg` program (part of the LT NSL software), or the SDQL language defined by DSSSL. The work of S. Abiteboul and colleagues e.g. (Abiteboul et. al., 1997), (Christophides et. al., 1994) on SQL based query languages for semi-structured data is also of interest in this respect as they define a formal framework for path-based query languages.

4. Comparisons with other systems

The major alternative corpus architecture that has been advocated is a database approach, where annotations are kept separately from the base texts. The annotations are linked to the base texts either by means of character offsets or by a more sophisticated indexing scheme. We will discuss two such systems and compare them with the LT NSL approach.

4.1. GATE

The GATE system (Cunningham et al, 1995), currently under development at the University of Sheffield, is a system to support modular language engineering.

4.1.1. *System components*

It consists of three main components:

- GDM - an object oriented database for storing information about the corpus texts. This database is based on the TIPSTER document architecture (Grishman, 1995), and stores text annotations separate from the texts. Annotations are linked to texts by means of character offsets[5].

[5] More precisely, by inter byte locations.

– Creole - A library of program and data resource wrappers, that allow one to interface externally developed programs/resources into the GATE architecture.

– GGI - a graphical tool shell for describing processing algorithms and viewing and evaluating the results.

A MUC-6 compatible information extraction system, VIE, has been built using the GATE architecture.

4.1.2. *Evaluation*

Separating corpus text from annotations is a general and flexible method of describing arbitrary structure on a text. It may be less useful as a means of publishing corpora and may prove inefficient if the underlying corpus is liable to change.

Although TIPSTER lets one define annotations and their associated attributes, in the present version (and presumably also in GATE) these definitions are treated only as documentation and are not validated by the system. In contrast, the SGML parser validates its DTD, and hence provides some check that annotations are being used in their intended way. SGML has the concept of *content models* which restrict the allowed positions and nesting of annotations. GATE allows any annotation anywhere. Although this is more powerful, i.e. one is not restricted to tree structures, it does make validation of annotations more difficult.

The idea of having formalised interfaces for external programs and data is a good one.

The GGI graphical tool shell lets one build, store, and recover complex processing specifications. There is merit in having a high level language to specify tasks which can be translated automatically into executable programs (e.g. shell scripts). This is an area that LT NSL does not address.

4.1.3. *Comparison with* LT NSL

In (Cunningham et al , 1996), the GATE architecture is compared with the earlier version of the LT NSL architecture which was developed in the MULTEXT project. We would like to answer these points with reference to the latest version of our software. See also (Cunningham et. al. , 1997) which addressed some of the points discussed here.

It is claimed that using normalised SGML implies a large storage overhead. Normally however, normalised SGML will be created on the fly and passed through pipes and only the final results will need to be stored. This may however be a problem for very large corpora such as the BNC.

It is stated that representing ambiguous or overlapping markup is complex in SGML. We do not agree. One can represent overlapping markup in SGML in a number of ways. As described above, it is quite possible for SGML to represent 'stand-off' annotation in a similar way to TIPSTER. LT NSL provides the hyperlinking semantics to interpret this SGML.

The use of normalised SGML and a compiled DTD file means that the overheads of parsing SGML in each program are small, even for large DTDs, such as the TEI.

LT NSL is not specific to particular applications or DTDs. The MULTEXT architecture was tool-specific, in that its API defined a predefined set of abstract units of linguistic interest, words, sentences, etc. and defined functions such as `ReadSentence`. That was because MULTEXT was undecided about the format of its I/O. LT NSL in contrast, since we have decided on SGML as a common format, provides functions such as `GetNextItem` which read the next SGML element. Does this mean the LT NSL architecture is application neutral? Yes and no.

Yes, because there is in principle no limit on what can be encoded in an SGML document. This makes it easier to be clear about what happens when a different view is needed on fixed-format read-only information, or when it turns out that the read-only information should be systematically corrected. The details of this are a matter of ongoing research, but an important motivation for the architecture of LT NSL is to allow such edits without requiring that the read-only information be copied.

No, because in practice any corpus is encoded in a way that reflects the assumptions of the corpus developers. Most corpora include a level of representation for words, and many include higher level groupings such as breath groups, sentences, paragraphs and/or documents. The sample back-end tools distributed with LT NSL reflect this fact.

It is claimed that there is no easy way in SGML to differentiate sets of results by who or what produced them. But, to do this requires only a convention for the encoding of *meta-information* about text corpora. For example, SGML DTDs such as the TEI include a 'resp' attribute which identifies who was responsible for changes. LT NSL does not require tools to obey any particular conventions for meta-information, but once a convention is fixed upon it is straightforward to encode the necessary information as SGML attributes.

Unlike TIPSTER, LT NSL is not built around a database, so we cannot take advantage of built-in mechanisms for version control. As far as corpus annotation goes, the UNIX rcs program has proved an adequate solution to our version control needs. Alternatively, version control can be provided by means of hyperlinking.

Some processing architectures use non-sequential control structures e.g. agent-based or blackboard systems. The pipe-lined architecture of LT NSL, unlike GATE, does not provide builtin support for these kinds of architecture. However, in our experience, many common tasks in NLP are sequential in nature, i.e. they only require access to a limited size window of the text in order to work. Examples of this kind of algorithm are tokenisation, morphological analysis, sentence boundary markup, parsing, and multilingual alignment. LT NSL is designed for this kind of application.

The GATE idea of providing formal wrappers for interfacing programs is a good one. In LT NSL the corresponding interfaces are less formalised, but can be defined by specifying the DTDs of a program's input and output files. For example a part-of-speech tagger would expect <W> elements inside <S> elements, and a 'TAG' attribute on the output <W> elements. Any input file whose DTD satisfied this constraint could be tagged. SGML architectural forms (a method for DTD subsetting) could provide a method of formalising these program interfaces.

As Cunningham et. al. say, there is no reason why there could not be an implementation of LT NSL which read SGML elements from a database rather than from files. Similarly, a TIPSTER architecture like GATE could read SGML and convert it into its internal database. In that case, our point would be that SGML is a suitable abstraction for programs rather than a more abstract (and perhaps more limited) level of interface. We are currently in discussion with the GATE team about how best to allow the interoperability of the two systems.

4.2. The IMS Corpus Workbench

The IMS Corpus Workbench (Christ, 1994) includes both a query engine (CQP) and a Motif-based user visualisation tool (*xkwic*). CQP provides a query language which is a conservative extension of familiar UNIX regular expression facilities[6]. XKWIC is a user interface tuned for corpus search. As well as providing the standard keyword-in-context facilities and giving access to the query language it gives the user sophisticated tools for managing the query history, manipulating the display, and storing search results. The most interesting points of comparison with LT NSL are in the areas of query language and underlying corpus representation.

[6] Like LT NSL IMS-CWB is built on top of Henry Spencer's public domain regular expression package

4.2.1. *The CQP model*

CQP treats corpora as sequences of attribute-value bundles. Each attribute[7] can be thought of as a total function from corpus positions to attribute values. Syntactic sugar apart, no special status is given to the attribute `word`.

4.2.2. *The query language*

The query language of IMS-CWB, which has the usual regular expression operators, works uniformly over both attribute values and corpus positions. This regularity is a clear benefit to users, since only one syntax must be learnt.

Expressions of considerable sophistication can be generated and used successfully by beginners. Consider:

```
[pos="DT" & word !="the"] [pos="JJ.*"]?
[pos="N.+"]
```

This means, in the context of the Penn treebank tagset, "Find me sequences beginning with determiners other than *the*, followed by optional adjectives, then things with nominal qualities". The intention is presumably to find a particular sub-class of noun-phrases.

The workbench has plainly achieved an extremely successful generalisation of regular expressions, and one that has been validated by extensive use in lexicography and corpus-building.

There is only limited access to structural information. While it is possible, if sentence boundaries are marked in the corpus, to restrict the search to within-sentence matches, there are few facilities for making more refined use of hierarchical structure. The typical working style, if you are concerned with syntax, is to search for sequences of attributes that you believe to be highly correlated with particular syntactic structures.

4.2.3. *Data representation*

CQP requires users to transform the corpora which will be searched into a fast internal format. This format has the following properties:

- Because of the central role of corpus position it is necessary to tokenise the input corpus, mapping each word in the raw input to a set of attribute value pairs and a corpus position.

- There is a logically separate index for each attribute name in the corpus.

[7] In CQP terminology these are the "positional attributes".

- CQP uses an integerised representation, in which corpus items having the same value for an attribute are mapped into the same integer descriptor in the index which represents that attribute. This means that the character data corresponding to each distinct corpus token need only be stored once.

- For each attribute there is an item list containing the sequence of integer descriptors corresponding to the sequence of words in the corpus. Because of the presence of this list the storage cost of adding a new attribute is linear in the size of the corpus. If the new attribute were sparse, it would be possible to reduce the space cost by switching (for that attribute) to a more space efficient encoding[8]

4.2.4. *Evaluation*

The IMS-CWB is a design dominated by the need for frequent fast searches of a corpus with a fixed annotation scheme. Although disk space is now cheap, the cost of preparing and storing the indices for IMS-CWB is such that the architecture is mainly appropriate for linguistic and lexicographic exploration, but less immediately useful in situations, such as obtain in corpus development, where there is a recurring need to experiment with different or evolving attributes and representational possibilities.

Some support is provided for user-written tools, but as yet there is no published API to the potentially very useful query language facilities. The indexing tools that come with IMS-CWB are less flexible than those of LT NSL since the former must index on words, while the latter can index on any level of the corpus annotation.

The query language of IMS-CWB is an elegant and orthogonal design, which we believe it would be appropriate to adopt or adapt as a standard for corpus search. It stands in need of extension to provide more flexible access to hierarchical structure[9]. The query language of LT NSL is one possible template for such extensions, as is the opaque but powerful `tgrep` program (Pito, 1994) which is provided with the Penn Treebank.

[8] IMS-CWB already supports compressed index files, and special purpose encoding formats would presumably save even more space.

[9] This may be a specialised need of academic linguists, and for many applications it is undoubtedly more important to provide clean facilities for non-hierarchical queries but it seems premature to close off the option of such access.

5. Case studies

5.1. Creation of marked-up corpora

One application area where the paradigm of sequential adding of markup to an SGML stream fits very closely, is that of the production of annotated corpora. Marking of major sections, paragraphs and headings, word tokenising, sentence boundary marking, part of speech tagging and parsing are all tasks which can be performed sequentially using only a small moving window of the texts. In addition, all of them make use of the markup created by earlier steps. If one is creating an annotated corpus for public distribution, then SGML is (probably) the format of choice and thus an SGML based NLP system such as LT NSL will be appropriate.

Precursors to the LT NSL software were used to annotate the MLCC corpora used by the MULTEXT project. Similarly LT NSL has been used to recode the Edinburgh MapTask corpus into SGML markup, a process that showed up a number of inconsistencies in the original (non-SGML) markup. Because LT NSL allows the use of multiple I/O files (with different DTDs), in (Brew&McKelvie, 1996) it was possible to apply these tools to the task of finding translation equivalencies between English and French. Using part of the MLCC corpus, part-of-speech tagged and sentence aligned using LT NSL tools, they explored various techniques for finding word alignments. The LT NSL programs were useful in evaluating these techniques. See also (Mikheev&Finch, 1995), (Mikheev&Finch, 1997) for other uses of the LT NSL tools in annotating linguistic structures of interest and extracting statistics from that markup.

5.2. Transformation of corpus markup

Although SGML is human readable, in practice once the amount of markup is of the same order of magnitude as the textual content, reading SGML becomes difficult. Similarly, editing such texts using a normal text editor becomes tedious and error prone. Thus if one is committed to the use of SGML for corpus-based NLP, then one needs to have specialised software to facilitate the viewing and editing of SGML. A similar problem appears in the database approach to corpora, where the difficulty is not in seeing the original text, but in seeing the markup in relationship to the text.

5.2.1. *Batch transformations*
To address this issue LT NSL includes a number of text based tools for the conversion of SGML: `textonly`, `sgmltrans` and `sgrpg`. With these tools it is easy to select portions of text which are of interest (using the query language) and to convert them into either plain text or another text format, such as LaTeX or HTML. In addition, there are a large number of commercial and public domain software packages for transforming SGML. In the future, however, the advent of the DSSSL transformation language will undoubtably revolutionise this area.

5.2.2. *Hand correction*
Specialised editors for SGML are available, but they are not always exactly what one wants, because they are *too* powerful, in that they let all markup and text be edited. What is required for markup correction are specialised editors which only allow a specific subset of the markup to be edited, and which provide an optimised user interface for this limited set of edit operations.

In order to support the writing of specialised editors, we have developed a Python (vanRossum, 1995) API for LT NSL, (Tobin&McKelvie, 1996). This allows us to rapidly prototype editors using the Python/Tk graphics package. These editors can fit into a pipeline of LT NSL tools allowing hand correction or disambiguation of markup automatically added by previous tools. Using this API we are developing a generic SGML editor. It is an object-oriented system where one can flexibly associate display and interaction classes to particular SGML elements. Already, this generic editor has been used for a number of tasks; the hand correction of part-of-speech tags in the MapTask, the correction of turn boundaries in the Innovation corpus (Carletta et al, 1996), and the evaluation of translation equivalences between aligned multilingual corpora.

We found that using this generic editor framework made it possible to quickly write new editors for new tasks on new corpora.

6. Conclusions

SGML is a good markup language for base level annotations of published corpora. Our experience with LT NSL has shown that:

– It is a good system for sequential corpus processing where there is locality of reference.

- It provides a modular architecture that does not require a central database, thus allowing distributed software development and reuse of components.
- It works with existing corpora without extensive pre-processing.
- It does support the Tipster approach of separating base texts from additional markup by means of hyperlinks. In fact SGML (HyTime) allows much more flexible addressing, not just character offsets. This is of benefit when working with corpora which may change.

LT NSL is not so good for:

- Applications that require a database approach, i.e. those that need to access markup at random from a text, for example lexicographic browsing or the creation of book indexes.
- Processing very large plain text or unnormalised SGML corpora, where indexing is required, and generation of normalised files is a large overhead. We are working on extending LT NSL in this direction, e.g. to allow processing of the BNC corpus in its entirety.

In conclusion, the SGML and database approaches are optimised for different NLP applications and should be seen as complimentary rather than as conflicting. There is no reason why one should not attempt to use the strengths of both the database and the SGML stream approaches. It is recommended that future work should include attention to allowing interfacing between both approaches.

7. Acknowledgements

This work was carried out at the Human Communication Research Centre, whose baseline funding comes from the UK Economic and Social Research Council. The LT NSL work began in the context of the LRE project MULTEXT with support from the European Union. It has benefited from discussions with other MULTEXT partners, particularly ISSCO Geneva, and drew on work at our own institution by Steve Finch and Andrei Mikheev. We also wish to thank Hamish Cunningham and Oliver Christ for useful discussions, and the referees for their detailed comments.

References

S. Abiteboul, D. Quass, J. McHugh, J. Widom, J.L. Wiener, "The Lorel Query Language for Semistructured Data", *Journal on Digital Libraries*, 1(1). 1997.

A. H. Anderson, M. Bader, E. G. Bard, E. H. Boyle, G. M. Doherty, S. C. Garrod, S. D. Isard, J. C. Kowtko, J. M. McAllister, J. Miller, C. F. Sotillo, H. S. Thompson, and R. Weinert. "The HCRC Map Task Corpus". *Language and Speech*, 34(4):351–366, 1991.

A. Ballim and H. Thompson. "MULTEXT Task 1.2 Milestone B Report". Technical Report, available from Laboratoire Parole et Langage, Universite de Provence, Aix-en-Provence, France.

T. Bray and S. DeRose, eds. 1997 "Extensible Markup Language (XML) Version 1.0". WD-xml-link-970406, World Wide Web Consortium. (See also `http://www.w3.org/pub/WWW/TR/`

T. Bray and C. M. Sperberg-McQueen (eds). 1996. "Extensible Markup Language (XML) version 1.0". World Wide Web Consortium Working Draft WD-xml-961114. Available at `http://www.w3.org/pub/WWW/TR/WD-xml-961114.html`.

C. Brew and D. McKelvie. 1996. "Word-pair extraction for lexicography". In *Proceedings of NeMLaP'96*, pp 45-55, Ankara, Turkey.

G. Burnage and D. Dunlop. 1992. "Encoding the British National Corpus". In *13th International Conference on English Language research on computerised corpora*, Nijmegen. Available at `http://www.sil.org/sgml/bnc-encoding2.html` See also `http://info.ox.ac.uk/bnc/`

J. Carletta, H. Fraser-Krauss and S. Garrod. 1996. "An Empirical Study of Innovation in Manufacturing Teams: a preliminary report". In *Proceedings of the International Workshop on Communication Modelling (LAP-96)*, ed. J. L. G. Dietz, Springer-Verlag, Electronic Workshops in Computing Series.

O. Christ. 1994. "A modular and flexible architecture for an integrated corpus query system". In *Proceedings of COMPLEX '94: 3rd Conference on Computational Lexicography and Text Research* (Budapest, July 7-10, 1994), Budapest, Hungary. CMP-LG archive id 9408005

V. Christophides, S. Abiteboul, S. Cluet, M. Scholl, "From Structured Documents to Novel Query Facilities", *SIGMOD 94*, 1994.

J. Clark. 1996 "SP: An SGML System Conforming to International Standard ISO 8879 – Standard Generalized Markup Language". Available from `http://www.jclark.com/sp/index.htm`.

H. Cunningham, K. Humphreys, R. J. Gaizauskas, and Y. Wilks. 1997. "Software Infrastructure for Natural Language Processing". In *5th Conference on Applied Natural Language Processing*, Washington, April 1997.

H. Cunningham, Y. Wilks and R. J. Gaizauskas. 1996. "New Methods, Current Trends and Software Infrastructure for NLP". In *Proceedings of the Second Conference on New Methods in Language Processing*, pages 283–298, Ankara, Turkey, March.

H. Cunningham, R. Gaizauskas and Y. Wilks. 1995. "A General Architecture for Text Engineering (GATE) - a new approach to Language Engineering R&D". *Technical Report, Dept of Computer Science, University of Sheffield.* Available from `http://www.dcs.shef.ac.uk/research/groups /nlp/gate/`

C. F. Goldfarb. 1990. "The SGML Handbook". Clarendon Press.

R. Grishman. 1995. "TIPSTER Phase II Architecture Design Document Version 1.52". *Technical Report, Dept. of Computer Science, New York University.* Available at `http://www.cs.nyu.edu/tipster`

N. Ide et. al. "MULTEXT Task 1.5 Milestone B Report". Technical Report, available from Laboratoire Parole et Langage, Universite de Provence, Aix-en-Provence, France.

J. Le Maitre, E. Murisasco, and M. Rolbert. 1996 "SgmlQL, a language for querying SGML documents". In *Proceedings of the 4th European Conference on Information Systems (ECIS'96).* Lisbon, 1996, pp. 75-89. Information available from `http://www.lpl.univ-aix.fr/projects/multext/MtSgmlQL/`

D. McKelvie, H. Thompson and S. Finch. 1996. "The Normalised SGML Library LT NSL version 1.4.6". *Technical Report, Language Technology Group, University of Edinburgh.* Available at `http://www.ltg.ed.ac.uk/software/nsl`

D. McKelvie, C. Brew & H.S. Thompson. 1997 "Using SGML as a Basis for Data-Intensive NLP". In *Proc. ANLP'97, Washington April 1997.*

A. Mikheev and S. Finch. 1995. "Towards a Workbench for Acquisition of Domain Knowledge from Natural Language". In *Proceedings of the Seventh Conference of the European Chapter of the Association for Computational Linguistics (EACL'95).* Dublin, Ireland.

A. Mikheev and S. Finch. 1997. "A Workbench for Finding Structure in Texts". In *Proc. ANLP'97, Washington, April 1997*

A. Mikheev and D. McKelvie. 1997. "Indexing SGML files using LT NSL". *Technical Report, Language Technology Group, University of Edinburgh.*

R. Pito. 1994. "Tgrep Manual Page". Available from `http://www.ldc.upenn.edu/ldc/online/treebank/man/cat1/tgrep.1`

G. van Rossum. 1995. "Python Tutorial". Available from `http://www.python.org/`

C. M. Sperberg-McQueen & L. Burnard, eds. 1994. "Guidelines for Electronic Text Encoding and Interchange". Text Encoding Initiative, Oxford.

R. Tobin and D. McKelvie. 1996. "The Python Interface to the Normalised SGML Library (PythonNSL)". *Technical Report, Language Technology Group, University of Edinburgh.*

Address for correspondence:
Language Technology Group, Human Communication Research Centre,
University of Edinburgh, 2 Buccleuch Place,
Edinburgh EH8 9LW, Scotland
Email: `David.McKelvie@ed.ac.uk`,
`Chris.Brew@ed.ac.uk`,
`H.Thompson@ed.ac.uk`