LRE Project 62-050 MULTEXT
Workpackage 2

Milestone C Deliverable NSL

The Normalised SGML Library (NSL)

Henry Thompson, Steve Finch, David McKelvie

November 14, 1995

Contents

1 Introduction **4**

1.1 nSGML 4

2 The MULTEXT **Data and Process Architecture** **5**

2.1 The MULTEXT Data Architecture 5

2.1.1 Compound Annotation and Links 5

2.1.2 Versions 6

2.2 The MULTEXT Process architecture 7

2.2.1 Entities in nSGML 7

3 System level programs **9**

3.1 nsgml 9

3.2 unknit 10

4 SGML **utility programs built using the NSL API** **10**

4.1 sggrep 11

4.2 sgmltoken 11

4.3 sgmlseg 12

4.4 sgmlsb 12

4.5 sgmls 12

4.6 textonly 12

4.7 simple and simpleq 12

5 Overview of the NSL API **13**

5.1 Stream View 13

5.2 Tree View 13

6 simple.c – A model NSL application **14**

7 NSL queries **20**

7.1 Examples of NSL queries 21

8 simpleq.c - a model NSL application using queries **24**

9 Datatypes defined by the NSL API **26**

9.1 Char 26

9.2 NSL_Item 27

9.3 NSL_Data 27

9.4 NSL_Bit 28

9.5 NSL_Attr 29

9.6 NSL_Query 30

9.7 NSL_File 30

9.8 NSL_Doctype 30

9.9 NSL_ElementSummary 30

10 NSL Public function interface **31**

10.1 Initialisation of NSL 31

10.2 SGML Files 31

10.3 Reading SGML 32

10.4 SGML Queries 32

10.5 Printing 33

10.6 Freeing NSL data structures 34

10.7 Attributes 34

10.8 Creating NSL data structures 35

10.9 Copying NSL structures 35

10.10 Other functions 35

11 Definition of NSGML **36**

12 Restrictions, Bugs and Features **37**

Abstract

This document describes the Normalised SGML Library (NSL), which consists of a set of C programs for manipulating SGML files and a C application program interface (API) designed to ease the writing of C programs which manipulate SGML documents.

1 Introduction

In pursuit of a development environment for SGML-based corpus and document processing, with support for multiple versions and multiple levels of annotation, LTG have developed an integrated set of SGML tools and a developers tool-kit, including a C-based API. This software described here contains everything required to process a very wide range of conformant SGML documents. Its initial parsing module incorporates v0.4 of James Clark's SP software, arguably the broadest coverage SGML parser available anywhere, commercial or not.

The basic architecture is one in which an arbitrary SGML document is processed on the way in, as it were, yielding two results: 1) An optimised representation of the information contained in the document's DOCTYPE; 2) A normalised version of the document instance, which can be piped through any tools built using our API for augmentation, extraction, etc. The use of the cached DOCTYPE together with the normalisation of the SGML to nSGML means that applications processing nSGML streams can be very efficient.

This document assumes that the reader is familiar with SGML [Goldfarb 90] and the C programming language [Kernighan 88].

The structure of this document is as follows. The next section introduces the NSL system. The third and fourth sections describe the user-callable utility programs provided in the NSL system. We then give an overview of the data structures used to represent SGML structure in the API, followed by an annotated example of the use of the NSL API in a complete program. In section 7 we give a description of the NSL query language which provides a convenient way of referring to elements of an SGML document, followed by an annotated program showing the use of the query language. The final three sections give a detailed description of nSGML and the data structures and functions defined in the NSL API.

1.1 nSGML

Normalised SGML (hereafter called nSGML) is a restricted and slightly modified version of the SGML document format which is designed to (a) ease text processing by software tools and (b) improve the readability of SGML documents. These aims are achieved by a number of *syntactic* restrictions on allowed SGML and by demanding that the document's DTD is preprocessed into a form which allows fast processing by SGML-aware tools which use the NSL API. It also allows the NSL tools to know the structure of the documents they are reading, without doing a complete (and potentially costly) parse of the SGML.

Arbitrary valid SGML documents can be converted into nSGML by means of the `nsgml` program supplied as part of the NSL system. `nsgml` is therefore envisaged as the first step in a sequence of NSL tools. `nsgml` will create DDB files, containing an optimised representation of DOCTYPEs, where they are needed. The definition of nSGML is given in section 11.

In order to convert an nSGML document into a valid SGML document, all one needs to do is to add an appropriate <!DOCTYPE ... > statement to the start of the nSGML document.

2 The MULTEXT Data and Process Architecture

2.1 The MULTEXT Data Architecture

The data architecture answers the question "How is all the information included in a MULTEXT corpus organised and stored?"

We are committed to TEI-compliant SGML for all our data deliverables, but that still leaves a wide range of options as to just how the corpus components are organised as documents, and how those documents are stored as files.

We will 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.

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 *(HyTime) hyper-documents*, which are linked ensembles of documents.

2.1.1 Compound Annotation and Links

The implication of this is that corpus components *will* be hyper-documents, with low-density (i.e. above the token level) annotation being expressed indirectly in terms of links. In the first instance, this will be constrained to situations where element content at one level of one document is entirely composed of elements from another document. Suppose, for example, we already had tokenised the 3rd component of the English corpus resulting in a single document d marked up to level 1, and with a tokenisation marked with <w> tags:

```
. . .
<p id=en.c3.p4>
<w id=en.c3.p4.w1>
Time
</w>
<w id=en.c3.p4.w2>
flies
</w>
<w id=en.c3.p4.w3>
.
</w>
</p>
. . .
```

The output of a phrase-level segmentation might then be stored as follows:

```
. . .
<p id=en.c3.p4>
<phr id=en.c3.p4.ph1 linkend=[d:en.c3.p4.w1] type=n>
```

```
</phr>
<phr id=en.c3.p4.ph2 linkend=[d:en.c3.p4.w2] type=v>
</phr>
</p>
```

The details of exactly how the linking would be notated is not appropriate here, and will be addressed in the relevant section of the specification of the `unknit` program, given in (section 3.2). At a minimum, links will be able to target anything from single elements to sets of sequences of continguous elements. Note that although the example above shows links to only one document, this is artifactual, and it will be possible to link to several documents, e.g. to a token document and a lexicon document:

```
<word>
<source linkend=[d:en.c3.p4.w1]>
</source>
<lex linkend=[x:en.lex.40332]>
</lex>
</word>
```

It follows from this that the lowest level of processing, tokenisation, will establish the base level of elements on the basis of which all further annotation will be based in the data architecture. In other words, up through tokenisation, MULTEXT processing will result in complete self-contained documents, with added information literally incorporated in the file stream.

Note however that the proposed architecture is recursive, in that e.g. sentence-level segmentation could be expressed in terms of links into the phrase-level segmentation as presented above.

2.1.2 Versions

The data architecture needs to address not only multiple levels of annotation but also alternative versions at a given level. Since any linking mechanism will certainly use the SGML entity mechanism to implement the identification of target documents, we can use the entity manager's catalogue as a means of managing versions. For our example above, this means that the connection between the phrase encoding document and the tokenised document would be in two steps: the phrase document would use the PUBLIC identifier ". . ./MULTEXT English Comp 3, tokens . . .", which the catalogue would map to the particular file `d`. Since catalogue entries are interpreted by tools as local to the directory where the catalogue itself is found, this means that binding together groups of alternative versions can be easily achieved by storing them under the same directory.

Subdirectories with catalogue fragments can thus be used to represent both increasing detail of annotation and alternatives at a given level of annotation.

Note also that with a modest extension of functionality, it is possible to use the data architecture described here to implement patches, e.g. to the tokenisation process. If the above constraint on pointer content is relaxed to allow pointers to sequences of elements from more than one file, a patch file (call it 'e') for a tokenisation change might be incorporated like this, drawing words first from the original file, then from the patch file (which could contain only the words whose tokenisation had changed), then from the original file again:

```
<p id=en.c1.p1 linkend=[d:en.c1.p1.w1..en.c1.p1.w5;
                        e:en.c1.p1.w6a..en.c1.p1.w6b;
                        d:en.c1.p1.w7..en.c1.p1.w35]>
</p>
. . .
```

Whether such a patch would have knock-on effects on higher levels of annotation would depend, inter alia, on whether the change in tokenisation crossed any higher-level boundaries.

2.2 The MULTEXT Process architecture

The process architecture determines the environment in which individual MULTEXT tools function. It is expressed in large part through two special tools: nsgml and unknit.

The overall conception of processing is as follows: One or more SGML documents are assembled (by nsgml, which interprets links as required to merge elements together), and piped via 'fat', (i.e. composed of nSGML elements), streams into a network of NSL programs. The eventual output nSGML document(s) are decomposed (by unknit, which installs links in new structure created by the tools back into the parts of the input documents which were not affected) into new and old components. Tools in the network are programmes which operate with (portions of) the document stream via the NSL API.

Figure 1 shows the overall dataflow picture which results from all this. Multiple invocations of nsgml and/or unknit will of course be required for richer configurations, and forking and joining will in due course also be supported.

2.2.1 Entities in nSGML

Should entities (especially character entities) be expanded by the stream interface before they are handed to the tool, and if so, should they be replaced in the stream before they are ouput?

Consider the following fragment of SGML marked up text:

```
<s>Fran&ccedil;ois Martin said yesterday that the following
companies announced quarterly results: IBM; AT&amp;T; &Xerox;</s>
```

Let us assume further that the character set is 7-bit ASCII, and therefore that there is no expansion of the entity `ç`. Which, if any, of these entities should be expanded? Now, there are some properties of these entities that a tool will need to know about if it is to do its job. For example, it may need to know that `ç` is a lower-case letter, while `&` is an ampersand sign, and `&Xerox;` is a string which expands to a word, or a string of words. If markup is to be preserved throughout the process of, for example, a word tokeniser, then clearly these are all potential problems.

Figure 1: MULTEXT data flow architecture

```
[Double lines are nSGML ‘fat’ streams, single lines are R/F ‘thin’ streams]

  file1.sgm . . . file1.htm . . .
  \     \      |       /       /
   \     \     |      /       /
    \     \___ | ____/       /
     \_______ ||| __________/
             |||||
             nsgml
              ||
              ||
              ||
        |---------------------
        |           |        |
        | nSGML   N A  C(++) |--(possible non-SGML resource files --)
        | stream  S P   tool |--(SGML resources come/go in fat streams)
        | parser  L I        |
        ----------------------
              ||
              ||
              ||
            unknit
              /\
             /  \
            /    \
         file.htm . . .
```

We therefore propose that tools must know about character entities such as `ç` and `&`. These will not be expanded, and can be reincorporated into the markup by the stream interface. All other entities, whether or not they contain markup, will be expanded and lost. This is necessary since markup may need to be added within their expansions (e.g. if `&Xerox;` expanded to "The Rank Xerox Corporation", then word token markup would have to be added within the entity. In general, if a string entity is expanded then markup may have to terminate in the entity which was started outside it. This would be obscure and not licensed by SGML in any case under its "obfuscatory markup" rules). We also insist that any insertion of a left angle bracket ('<') is represented by '`<`', and any insertion of an ampersand ('`&`') is represented by '`&`'. This is for SGML markup validity reasons.

The current implementation (version 1.4.2) achieves this by expanding *all* entity references *except* those of type **sdata** with explicit SGML notations.

3 System level programs

3.1 nsgml

The basic tool for converting SGML to NSGML and caching the DOCTYPE is called `nsgml`. This program converts arbitrary valid SGML documents into NSGML. NSGML is a variant of SGML which is designed to optimise the processing of SGML-marked up documents. All programs written using the NSL API assume that their SGML inputs are in NSGML format, thus the `nsgml` program should be called first in any chain of NSL applications.

Usage: nsgml [-wx] [-c catalog] [-o directory] filename ...

Filename is an SGML file, if no filenames are given, then standard input is used.
The output is an NSGML version of filename written to standard output.
-w turns on warnings. If this option is given, then warning messages from the SGML parser about duplicate declarations, undefined elements and default entities will be written to stderr.
-x inhibits writing of a .ddb file.
-c specifies the name of an SGML catalog file.
-o provides an alternative directory (rather than .) to write the ddb file in.

As well as producing NSGML output, `nsgml` also creates a .ddb file called `filename.ddb` which contains the condensed form of the DTD. This .ddb file will then be used by subsequent NSL programs. If you modify the DTD describing the input SGML file, then you should delete the .ddb file before calling `nsgml` to ensure that a new .ddb file is created. There should be more consistency checking built in.

`Nsgml` will also handle *hyperlinks*, i.e. links between different SGML files. SGML elements in the input file which are of the form

```
<W FROM='id (C78.T1)' TO='id (C78.T3)' DOC=dd ID=W1440 ></W>
```

will be expanded into their complete form, by including all material between the elements with ids `C78.T1` and `C78.T3` in the file given by the value of the SGML entity `dd`. If TO is omitted then only the FROM element is included. The DOC attribute only needs to be given on the first <W> element, subsequent ones will use the same file. A processing instruction of the form

```
<?Multext LINKS W>
```

is needed to tell `nsgml` that W elements may be hyperlinked, and an external entity declaration for `dd` to identify the source, e.g.

```
<!ENTITY dd PUBLIC ... >
```

3.2 unknit

`Unknit` is a program which creates hyperlinked SGML files from NSGML files. The present version is still somewhat experimental. However, suppose `test.w.nsg` is an NSGML file which contains <w> markup around words; `test.s.nsg` is an NSGML file which contains <s> markup around sentences (consisting of a sequence of <w> elements. Running the command

```
unknit test.w.nsg W < test.s.nsg > test.s.sgm
```

will create the SGML file `test.s.sgm` which contains the <s> markup from `test.s.nsg`, but with all <w> elements replaced with hyperlinks back to `test.w.nsg`. This can be useful to save space and separate annotation markup from the 'original' content of the document.

Usage: unknit [-d ddbfile] basefile targetGI [sourceGI] < IN > OUT

-d allows the specification of an optional ddb file.

basefile is an NSGML file.

IN is an NSGML file.

OUT is an SGML file.

4 SGML utility programs built using the NSL API

The other tools in the bin directory are:

- a suite of tools developed for the MULTEXT project, including a tokeniser (sgmltoken), a toy segmenter (sgmlseg, actually a perl program) and a sentence boundary finder (sgmlsb);
- a simple version (element structure and text only) of sgmls;

- a program to output all the text from an NSGML file, textonly;
- two illustrative applications, simple and simpleq, whose source code (in src/nsg/simple.c and src/nsg/simpleq.c) is intended as a starting point for application developers.

4.1 sggrep

This useful program works like the `grep` program in searching a file for regular string expressions. However, unlike `grep`, it is aware of the tree structure of SGML files.

Usage: sggrep [-d ddb-file] [-v] [-n] query [sub-query] regexp < input > output

-d ddb-file: Take DOCTYPE from specified ddb file

-v: invert sense of sub-query+regexp

-n: don't force a newline between output matches. The default is to print a newline between each match.

query: pattern on items to select, basically path based with terms separated by /, see section 7 for the details of what a query looks like.

sub-query: if present, selects sub-elements of query-selected item for regexp to match

regexp: Regular expression to match against text directly contained in query-selected item (if no sub-query) or in any sub-query selected sub-element of query-selected item. If empty (i.e. "") matches anything, including empty elements, indeed this is the ONLY way to get empty elements if required.

The `query` selects a subset of SGML elements which are potentially to be printed. `sub-query` is a query relative to elements found by `query` which selects elements containing text. SGML elements matching `query`, which contain sub elements matching `sub-query` which in turn contain text matching `regexp` are printed. For example:

```
sggrep ".*/S" "S/.*/W" "the"
```

will print out all sentences which contain a word element containing the string "the". Since the output of `sggrep` is NSGML, it can be used as input to another call of `sggrep`, thus allowing more complex queries to be build up in stages.

4.2 sgmltoken

Text tokenization. All text inside <TEXT> elements is tokenized, i.e. split into tokens and marked up with <C> elements.

Usage: sgmltoken [-d ddb file] < input > output

-d gives an optional ddb file.

4.3 sgmlseg

This is a PERL program which segments an NSGML file which has already been tokenised, into words using <w> markup.

Usage: sgmlseg input > output

The use of PERL demonstrates that normalising SGML (`nsgml`) can be useful even if one does not use the NSL API to process the result.

4.4 sgmlsb

`sgmlsb` is a quick and dirty sentence boundary marking application, i.e. it adds <S> elements to an NSGML file which has already been tokenised and segmented.

Usage: sgmlsb [-d ddb file] < input > output

-d gives an optional ddb file.

4.5 sgmls

`sgmls` is a trivial version of the `nsgmls` program, i.e. it takes NSGML input and produces output in the form that `nsgmls` does (ESIS format). `sgmls` does not yet print out attribute values for elements, it should and could.

Usage: sgmls [-d ddb file] < input > output

The optional -d option allows one to specify a .ddb file externally. If not given, then we assume that the input file contains a <?NSL DDB ... > line.

4.6 textonly

`textonly` expects NSGML as input, outputs text only, i.e. removes all SGML element markup.

Usage: textonly [-t tag] [-s c] < input > output

If the -t option is present, only text somewhere inside <tag ...> elements is printed. If -s is present, the SINGLE CHARACTER c (e.g. ' ' or newline) is printed between each bit of text.

4.7 simple and simpleq

These two programs have been written as demonstrations of how to write C programs using the NSL API. They are explained in detail in following sections of this document.

5 Overview of the NSL API

There are two common ways of looking at SGML documents. First, as a linear stream of text with embedded markup tags, and secondly as a hierarchic tree structure. Both of these views can be useful in some circumstances, and accordingly, the NSL API provides data structures and access functions to support both these views.

5.1 Stream View

To view SGML documents as a linear stream of text plus markup, we use the `NSL_Bit` data structure (see section 9.4). A `NSL_Bit` is either an SGML start tag, an end tag, an empty tag, an SGML processing instruction or text with no SGML elements in it. For example, the SGML text:

> <P> This is some text with a name, <NAME>Fred Bloggs</NAME> and a page break <PB> in it. </P>

would be split into the following `NSL_Bits` (one per line).

<P>	type=NSL_start_bit
This is some text with a name,	type=NSL_text_bit
<NAME>	type=NSL_start_bit
Fred Blogs	type=NSL_text_bit
</NAME>	type=NSL_end_bit
and a page break	type=NSL_text_bit
<PB>	type=NSL_empty_bit
in it.	type=NSL_text_bit
</P>	type=NSL_end_bit

The NSL function `GetNextBit` (see section 10.3) reads the next `NSL_Bit` from an NSGML file. Function `PrintBit` (see section 10.5) writes `NSL_Bits` to an output NSGML file.

5.2 Tree View

In order to view an SGML document as a hierarchic structure, the NSL API constructs a C data structure made up of `NSL_Item` and `NSL_Data` data structures (see section 9.2) which mirrors the tree structure of the document.

The `NSL_Item` type describes an SGML element and all its contents in a document, i.e. it represents a complete subtree of the document structure. So that, in the above example, there would be one `NSL_Item` for the <P> element, with an nested `NSL_Item` for the <NAME> element.

The `NSL_Data` data structure represents a chunk of SGML element content, i.e. either an SGML element or a piece of text without element structure. `NSL_Data` structures for the contents of an `NSL_Item` are organised into a linked list of mixed `NSL_Items` and text in mixed content, or all `NSL_Items` in element-only content. In the above example, the `NSL_Item` corresponding to <P> will point to five `NSL_Data` structures corresponding to "This ... name,", <NAME>, "and ... break", <PB>, and "in it."

Figure 2 shows the `NSL_Item` and `NSL_Data` data structures created for the text

<name>David<surname>McKelvie</surname></name>

Full details of the NSL API functions and data structures are given in sections 9 and 10.

6 simple.c – A model NSL application

This model application program (`simple`) has been written to demonstrate the use of the NSL API. The `simple` program reads an NSGML file containing paragraph and word markup. It assumes that each word element has an attribute which contains part of speech (POS) information. The program then outputs a modified version of the input file where the text of each word element has been replaced by some text which shows the word and the POS tag associated to the word. For example, if the input file looks like:

```
<HEADER>blah blah</HEADER>
<TEXT>
<P>
<W TYPE=det>The</W>
<W TYPE=nn>cat</W>
</P></TEXT>
```

then the output file will look like:

```
<HEADER>blah blah</HEADER>
<TEXT>
<P>
<W TYPE=det>The/det</W>
<W TYPE=nn>cat/nn</W>
</P></TEXT>
```

`Simple.c` is not intended to be a particularly useful program, rather to be an example of the use of the NSL API.

The program can be called as follows: simple [*options*] *nsgmlfile*

Allowed options (all of which are optional) are:

-d The name of the cached DOCTYPE .ddb file
-t name of attribute containing the POS information (default TYPE)
-w name of word element (default W)
-t print format for output words and their POS tags (default "%s/%s")

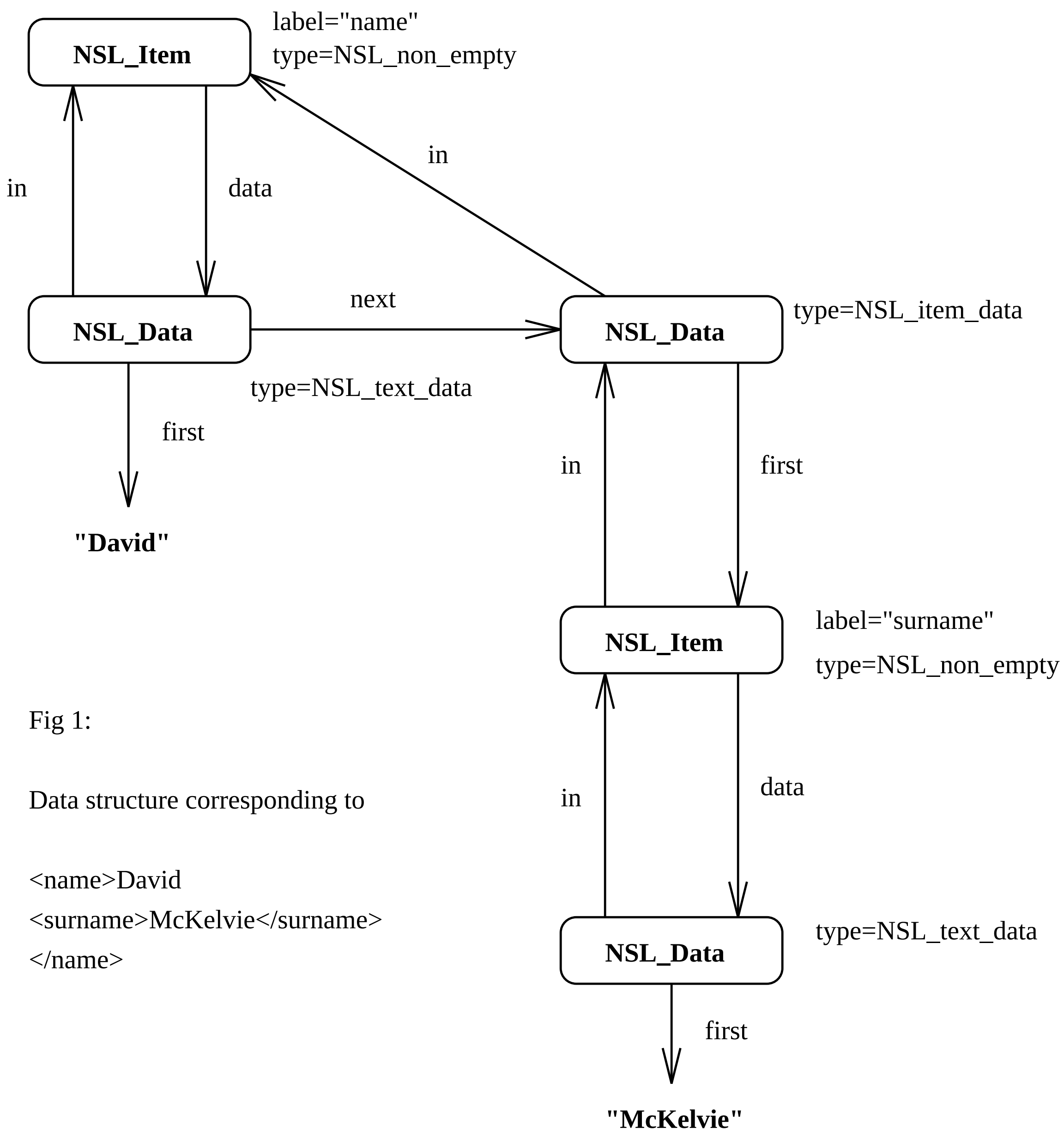

Figure 2: NSL API data structure

The source of the `simple` program can be found in `src/nsg/simple.c`. The example data input file can be found in `data/simple.sgm` .

The annotated code of the `simple` program is as follows:

Include header file for character functions.

```
#include <ctype.h>
```

Include header file for string functions.

```
#include <string.h>
```

Include the header file for the NSL API.

```
#include "nsl.h"
```

Main program.

```
void main(int argc, char **argv) {
  NSL_Bit *bit;
  NSL_File inf, outf;
  NSL_Doctype dct=NULL;
  char *paraLabel,*wordLabel,*textLabel,*ptr,*label,*tagVal=NULL,buf[100];
  int in_para=0,in_text=0,ac=1,in_word=0,len;

  /* defaults for command line arguments */
  /* Name of attribute carrying tag -- set with -t */
  char* tagAttr=(char*)"TYPE";
  /* Name of word element -- set with -w */
  char* wordTag=(char*)"W";
  /* Format string for word, tag -- set with -f */
  char* textFormat=(char*)"%s/%s";
```

Initialise the NSL SGML API.

```
  NSLInit();
```

Read the command line options. The optional -d option allows one to specify a .ddb file externally. If not given, then we assume that the input file contains a <?NSL DDB ... > processing instruction.

```
  while (ac<argc) {
    if (STREQ(argv[ac], "-d")) {
      dct=DoctypeFromDdb(argv[++ac]);
      ac++;
    }
```

```
    else if (STREQ(argv[ac], "-t")) {
      ptr=tagAttr=argv[++ac];
      ac++;
      /* need upper case for attribute comparison */
      while (*ptr) {
        *ptr=toupper(*ptr);
        ptr++;
      };
    }
    else if (STREQ(argv[ac], "-w")) {
      ptr=wordTag=argv[++ac];
      ac++;
      /* need upper case for tag comparison */
      while (*ptr) {
        *ptr=toupper(*ptr);
        ptr++;
      };
    }
    else if (STREQ(argv[ac], "-f")) {
      textFormat=argv[++ac];
      ac++;
    }
    else {
      break;
    };
  };
```

Open the input NSGML file, passing in the doctype declaration `dct` if any has been specified by a -d option. If the doctype declaration was not specified by a -d option, then it will be set by reading from the input file on opening.

```
  inf=SFFopen(stdin, dct, NSL_read);
  dct=DoctypeFromFile(inf);
```

Open the output NSGML stream using the same doctype declaration.

```
  outf=SFFopen(stdout, dct, NSL_write_normal);
```

Get the unique name of the elements we care about.

```
  textLabel=ElementUniqueName(dct,(char*)"TEXT",4);
  paraLabel=ElementUniqueName(dct,(char*)"P",1);
  wordLabel=ElementUniqueName(dct,wordTag,0); /* length will be computed */
```

Loop round, reading *bits* of the SGML input text. A *bit* is either a single piece of SGML markup (a start tag, an end tag, or a processing instruction) or it is text without SGML element markup.

```
  while ((bit=GetNextBit(inf))) {
```

Now we perform a case statement on the type of the NSL_bit we have just got.

```
    switch (bit->type) {
    case NSL_start_bit:
```

Case 1: We have found the start tag for an SGML element. Note that the item value of this NSL_Bit is of type `NSL_inchoate`, meaning that unless you call `ItemParse`(see section 10.3) on it, it's got just the start tag info and no contents.

```
      if((label=bit->label)==textLabel) {
        in_text=1;
      }
```

If we are inside a <text> element, then note this fact.

```
      else if (in_text && label==paraLabel) {
        in_para=1;
      }
```

If we are inside a paragraph (<P>) inside text, then note this fact.

```
      else if (in_para && label==wordLabel) {
```

We have found a word in a text paragraph. Note this fact and save the POS tags which are the value of the SGML attribute of the <W> that we just found, given by the `tagAttr` variable.

```
        in_word=1;
        tagVal=GetAttrVal(bit->value.item,tagAttr);
      };
```

In any case, we fall through to the next `case` to print out the NSL_bit we just read.

```
    case NSL_empty_bit:
```

Case 2: We have found an empty SGML element. There is no point in comparing labels of empty items. Note that `PrintItem` is smart and will only print a start tag for empty or inchoate items (not an end tag), but will get into a different state depending on which.

```
      PrintItem(outf, bit->value.item);
      break;

    case NSL_text_bit:
      if (in_word) {
```

Case 3: If we are inside the text of a word element, then strip off trailing whitespace, and write the word and its POS in a user defined format. Note that we use `PrintText`, rather than `printf` to actually write the string to the output file. This is in order to keep the NSGML output state up-to-date.

```
        len=strlen(bit->value.body);
        while (strchr((char*)" \t\n",bit->value.body[len-1])) {
          bit->value.body[--len]='\000';
        };
        sprintf(buf,textFormat,bit->value.body,tagVal);
        PrintText(outf,buf);
      }
      else {
        /* text in some other context -- just print it */
        PrintText(outf, bit->value.body);
      }
      break;

    case NSL_end_bit:
```

Case 4: We have found an SGML end tag, so we keep track of where we are in the SGML markup tree.

```
      if (in_para) {
        if (bit->label==paraLabel) {
          /* no longer in para */
          in_para=0;            /* NOTA BENE assume no nested para's! */
        }
        else if (bit->label==wordLabel) {
          /* no longer in word */
          in_word=0;
        };
      };
      /* print it no matter what */
      PrintEndTag(outf,bit->label);
      break;

    default:
      SHOULDNT;
    }; /* end switch */
```

Note: processing instructions are not dealt by the above code. If this is required, then a new case should be added with label `NSL_pi_bit` (see section 9.4).

NSL_Bits are fixed but their contents are not freed unless we do it ourselves.

```
    FreeBit(bit);
  }; /* end while */
```

At the very end we need to close the output NSGML stream.

```
  SFclose(outf);
}
```

7 NSL queries

It will be noted that in the above program (`simple.c`), a largish part of the code is devoted to implementing a finite-state machine which keeps track of where we are in the SGML structure of the document (cases 1 and 4 in the annotated listing of `simple.c`). One feels that this tracking of where we are could be supported by the NSL API, and in fact this can be done.

The reason that this API support for tracking SGML positions was not used in `simple.c`, is that, for simple cases, it is much more efficient to spell out this processing explicitly rather than to call the API query processing mechanism, which, by necessity, must be able to deal with the general case. If however, ease of programming is more important than processing speed or you need to handle a wider range of document structures, then the NSL query processing functions may provide what you need.

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 which 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 as follows:

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

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 which 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 .

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.

7.1 Examples of NSL queries

In this section we show some examples of NSL queries, assuming the following 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>
```

The SGML structure of a sample document which uses this DTD is shown in figure 3.

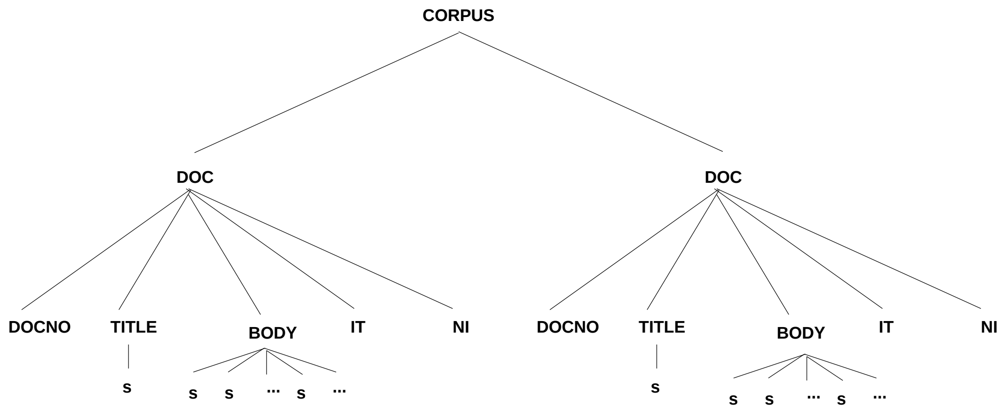

Figure 3 The hierarchical structure of an example document.

The query:

CORPUS/DOC/TITLE/s

means all s elements directly under TITLE's directly under DOC. This is shown graphically in figure 4. The NSL query functions return the indicated items one by one until the set denoted by the query is exhausted.

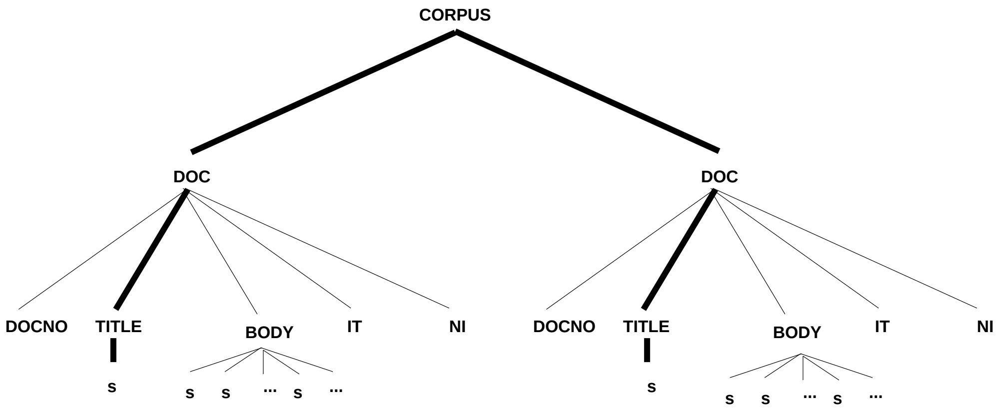

Figure 4 CORPUS/DOC/TITLE/s

CORPUS/DOC/./s

means all s's directly under anything directly under DOC, as shown in figure 5.

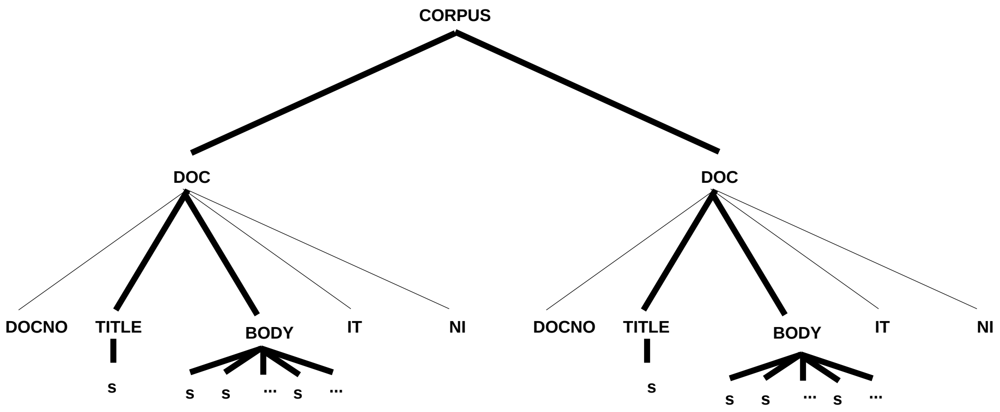

Figure 5 CORPUS/DOC/.*/s

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 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

./.[1]/.[2]/.[0]

means the 1st element below the 3rd element below the 2nd element in a stream of elements. as shown in figure 6.

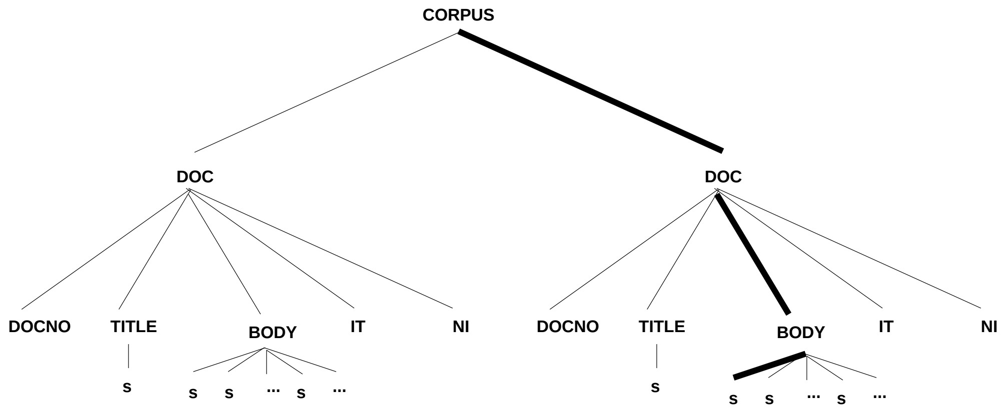

Figure 6 ./.[1]/.[2]/.[0]

This is also the referent of

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.

.*/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 7.

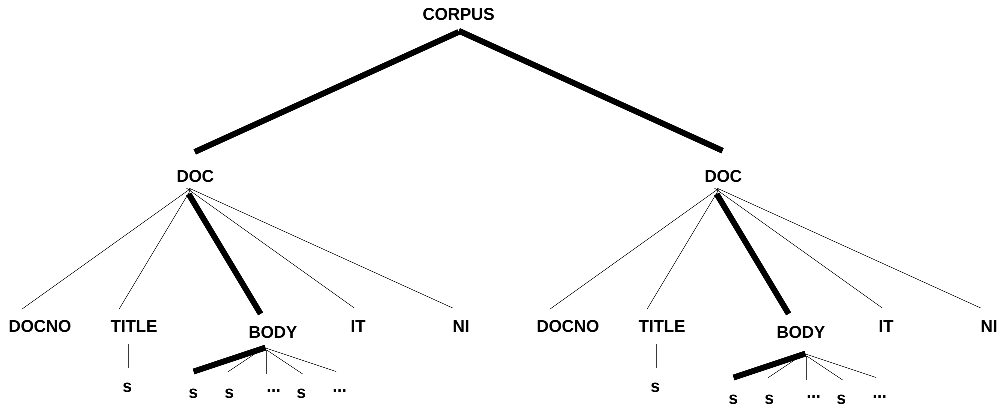

Figure 7 .*/BODY/s[0]

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.

8 simpleq.c - a model NSL application using queries

The following program simpleq.c shows how the NSL API query functions can be used.

Include header file for character functions.

```
#include <ctype.h>
```

Include header file for string functions.

```
#include <string.h>
```

Include the header file for the NSL API.

```
#include "nsl.h"
```

Main program.

```
void main(int argc, char **argv) {
  NSL_File inf, outf;
  NSL_Doctype dct=NULL;
  NSL_Query qu;
  NSL_Item *item;
  char *ptr, *tagVal=NULL,buf[100],qustr[100];
  int ac=1,len;

  /* defaults for command line arguments */
  /* Name of attribute carrying tag -- set with -t */
  char* tagAttr=(char*)"TYPE";
  /* Name of word element -- set with -w */
  char* wordTag=(char*)"W";
  /* Format string for word, tag -- set with -f */
  char* textFormat=(char*)"%s/%s";
```

Initialise the NSL SGML API.

```
  NSLInit();
```

Read the command line options. The optional -d option allows one to specify a .ddb file externally. If not given, then we assume that the input file contains a
<?NSL DDB ... > line.

```
while (ac<argc) {
  if (STREQ(argv[ac], "-d")) {
    dct=DoctypeFromDdb(argv[++ac]);
    ac++;
  }
  else if (STREQ(argv[ac], "-t")) {
    ptr=tagAttr=argv[++ac];
    ac++;
    /* need upper case for attribute comparison */
    while (*ptr) {
      *ptr=toupper(*ptr);
      ptr++;
    };
  }
  else if (STREQ(argv[ac], "-w")) {
    ptr=wordTag=argv[++ac];
    ac++;
    /* need upper case for tag comparison */
    while (*ptr) {
      *ptr=toupper(*ptr);
      ptr++;
    };
  }
  else if (STREQ(argv[ac], "-f")) {
    textFormat=argv[++ac];
    ac++;
  }
  else {
    break;
  };
};
```

Open the input NSGML file, passing in the doctype declaration `dct` if any has been specified by a -d option. If the doctype declaration was not specified by a -d option, then it will be set by reading from the input file on opening.

```
inf=SFFopen(stdin, dct, NSL_read);
dct=DoctypeFromFile(inf);
```

Open the output NSGML stream using the same doctype declaration.

```
outf=SFFopen(stdout, dct, NSL_write_normal);
```

Construct a query, which looks for words anywhere inside paragraphs anywhere inside a text.

```
  strcpy(qustr,".*/TEXT/.*/P/.*/");
  strcat(qustr,wordTag);
  qu=ParseQuery(qustr);
```

Read items of the SGML input text. When we find an item which matches the query we execute the body of the while loop. Items which do not match are written to the output stream by `GetNextQueryItem`. Each call of `GetNextQueryItem` creates a new item, which it is the responsibility of the programmer to free once it has been used.

```
    while( ( item=GetNextQueryItem(inf, qu, outf ) ) ) {
```

If we are inside the text of a word element, then strip off trailing whitespace (in the inner while loop), and write the word and its part of speech in a user defined format. Note that we use `PrintItem` to write the modified item to the output file. This is in order to keep the NSGML output state up-to-date. Note that the code here assumes that each word element contains only text and no embedded SGML markup. More complex code could cope with this more complex possibility.

```
        len=strlen(item->data->first);
        while (strchr((char*)" \t\n",item->data->first[len-1])) {
          item->data->first[--len]='\000';
        };
        tagVal=GetAttrStringVal(item,tagAttr);
        sprintf(buf,textFormat,item->data->first,tagVal);
        item->data->first = buf;
        PrintItem(outf, item);
```

NSL_Items are fixed but their contents are not freed unless we do it ourselves.

```
        FreeItem(item);
  }; /* end while */
```

At the very end we need to close the output NSGML stream.

```
  SFclose(outf);
}
```

We don't bother to explicitly free the NSL_Query since there was only one created.

---

9 Datatypes defined by the NSL API

The following C data structures are defined by the NSL API. They are included in the C header file `nsl.h`.

9.1 Char

The type `Char` is introduced for 8-bit characters, i.e. for characters which can appear in SGML text. Element and attribute names are of type `char*` since we use the reference concrete syntax.

```
typedef unsigned char Char;
```

9.2 NSL_Item

The NSL_Item type describes an SGML element and all its contents in a document, i.e. it represents a complete subtree of the document structure.

```
typedef enum {NSL_inchoate, NSL_non_empty, NSL_empty, NSL_free} NSL_ItemType;

/* Tree node corresponding to an SGML element */
typedef struct NSL_Item {
    char *label; /* The SGML element name */
    NSL_ItemType type;
    struct NSL_Data *data; /* The SGML element content */
    NSL_ElementSummary defn; /* The NSL summary definition of
                                the element type */
    struct NSL_Attr *attr; /* Linked list of actual (as opposed to defaulted)
                              attributes for the element */
    struct NSL_Data *in; /* dominating NSL_Data if any */
} NSL_Item;
```

The `label` field contains the name of the SGML element which this item is describing, e.g. “P”. This field is constant across all occurrences of this element type.

The `type` field describes what sort of item this is. NSL_Items of type `NSL_inchoate` describe an SGML start tag e.g. <P type=indented>, but not the contents of this element. NSL_Items are initialised with this type value. If NSL decides, after checking the DTD, that this item corresponds to an empty SGML tag (i.e. one with an EMPTY content model), then the type field is set to `NSL_empty`. If non-empty, then once NSL has read the entire contents of this SGML element (i.e. as far as the corresponding end tag), then the type is set to `NSL_non_empty`. Finally, NSL_Items that have been 'Freed', by `FreeItem` will have type `NSL_free`.

The field `data` is a pointer to the representation of the content of this SGML element, i.e. a linked list of NSL_Data structures.

The field `defn` is an internal representation of the nature and details of the DTD definition of the SGML element.

The field `attr` is a pointer to a linked list of `NSL_Attr` structures which describes the attributes explicitly attached to this element, i.e. not defaulted attributes.

The field `in` is a pointer to the `NSL_Data` structure which contains this item, i.e. the content of the SGML element which contains this element.

9.3 NSL_Data

The `NSL_Data` data structure represents a chunk of SGML element content, i.e. either an SGML element or a piece of text without element structure. They are organised into a linked list of mixed NSL_Items and text in mixed content, or all NSL_Items in element-only content.

```
typedef enum {NSL_undefined,NSL_text_data,NSL_item_data} NSL_Data_Type;

typedef struct NSL_Data {
    int ref;
    NSL_Data_Type type; /* type pointed to by first */
    struct NSL_Data *next; /* next NSL_Data or NULL */
    void *first; /* could be either text or an NSL_Item per type field */
    NSL_Item *in; /* dominating item */
} NSL_Data;
```

The field `ref` is used internally and should not be modified by user code.

The field `type` tells us whether this `NSL_Data` refers to an SGML element (NSL_item_data) or to some character data (NSL_text_data).

The field `next` points to the next document chunk which is at the same level of nesting in the SGML document structure. It is NULL if this is the last daughter of its `in` node.

The `first` field points to a string (a 0-terminated char array) or to an `NSL_Item` which is the content of this data element.

The `in` field points to the `NSL_Item` which contains this data element.

9.4 NSL_Bit

`NSL_Bits` describe the basic chunks of an SGML document as follows:

NSL_bad - You should never see this value.

NSL_start_bit - An SGML start tag, e.g. <P type=indent>, for non-empty elements.

NSL_end_bit - An SGML end tag, e.g. </P>

NSL_empty_bit - An SGML milestone tag, e.g. <pb>

NSL_eof_bit - Only set when we have reached end of file. You should never see this value.

NSL_text_bit - A piece of text with no SGML element content.

NSL_pi_bit - An SGML processing instruction e.g. <?tabset tab=8>

```
typedef enum {NSL_bad, NSL_start_bit, NSL_end_bit, NSL_empty_bit,
              NSL_eof_bit, NSL_text_bit, NSL_pi_bit} NSL_Bit_Type;

/* Container for passing structure to user */
typedef struct NSL_Bit {
    NSL_Bit_Type type;
    union {
      NSL_Item *item; /* type NSL_start_bit */
      Char *body; /* text, pi */
    } value;
```

```
    char *label; /* valid only if type NSL_start_bit or NSL_end_bit */
} NSL_Bit;
```

An `NSL_Bit` points to either `Char` data (type NSL_text_bit or NSL_pi_bit) or to an `NSL_Item` (type NSL_start_bit).

The `label` field (when defined, i.e. for NSL_start_bit, NSL_empty_bit or NSL_end_bit) is the name of the corresponding SGML element.

9.5 NSL_Attr

Attributes attached to SGML start tags are described by a linked list of `NSL_Attr` data structures.

```
typedef enum {NSL_attr_num, NSL_attr_nums, NSL_attr_id,
              NSL_attr_refid, NSL_attr_refids, NSL_attr_string,
              NSL_attr_entity, NSL_attr_entities, NSL_attr_float,
              NSL_attr_strings} NSL_AVType;

typedef enum {NSL_defval_optional, NSL_defval_implied, NSL_defval_current,
                  NSL_defval_required, NSL_defval_value} NSL_ADefType;

typedef struct NSL_Attr {
    NSL_AVType valuetype;           /* type of value */
    NSL_ADefType deft;               /* type of default value */
    char *name;                     /* name of attribute */
    union {
        Char *string;               /* NAME, STRING */
    } value;                        /* actual value */
    struct NSL_Attr *next;         /* list link */
} NSL_Attr;
```

The `valuetype` field describes the type of the attribute's value, see SGML documentation for a description of the meaning of the different attribute value types.

The `deft` field describes the kind of default value for this attribute, see SGML documentation for a description of the meaning of the different default value types.

The `name` field gives the name of this attribute.

The `value.string` field gives the value of this attribute as a string (0-terminated Char array). This is a union type because in a latter release we want to introduce typed attribute values.

The `next` field points to another `NSL_Attr` structure for the next attribute defined on this start tag, or NULL if this is the last.

9.6 NSL_Query

An NSL_Query is a data structure which is the internal representation of a query. A query is a description of a path in the SGML document structure.

```
typedef struct NSL_Query_I* NSL_Query;
```

NSL_Query is defined as a pointer to a private data structure, i.e. all you can do with them is to pass them around between NSL functions.

The details of this and the following 'private' type declarations can be found in the internal header file nsllib.h. If you need their internal structure and know what you are doing, you should include this header file in your C program instead of the public nsl.h.

9.7 NSL_File

An NSL_File describes an nSGML file or document.

```
typedef struct NSL_File_I* NSL_File;
```

NSL_File is defined as a pointer to a private data structure, i.e. all you can do with them is to pass them around between NSL functions.

The following enumerated type gives the allowed kinds of NSL_File, see section 10.2 below for details of what they mean.

```
typedef enum {NSL_read, NSL_write_minimal, NSL_write_normal,
              NSL_write_pretty, NSL_write_normal_nd} NSL_FType;
```

9.8 NSL_Doctype

An NSL_Doctype describes an SGML Document Type Description (DTD).

```
typedef struct NSL_Doctype_I* NSL_Doctype;
```

NSL_Doctype is defined as a pointer to a private data structure, i.e. all you can do with them is to pass them around between NSL functions.

9.9 NSL_ElementSummary

The NSL_ElementSummary data structure describes the structure of an SGML element as defined in the DTD.

```
typedef struct NSL_ElementSummary_I* NSL_ElementSummary;
```

NSL_ElementSummary is defined as a pointer to a private data structure, i.e. all you can do with them is to pass them around between NSL functions.

10 NSL Public function interface

10.1 Initialisation of NSL

extern void NSLInit(void);

This function initialises the NSL API library. It should be called once before any other NSL functions are called.

extern void ParseInit(void);

This function initialises some character arrays (defining what whitespace and name characters are) used to parse SGML. It is called by `NSLInit`, so should not normally need to be called.

10.2 SGML **Files**

extern NSL_File SFFopen(FILE *fp, NSL_Doctype dtype, SFType type);

Open an `NSL_File` 'stream' to an NSGML document connected to the C file stream `fp`. The mode of the stream is given by the `type` parameter as follows:

NSL_read - read the NSGML document.

NSL_write_minimal - write an NSGML document but print nothing but what is explicitly given, including no DDB PI at start.

NSL_write_normal - write an NSGML document adding RE before start-tags in element-only content.

NSL_write_pretty - write an NSGML document adding RE also at beginning and end of text content.

NSL_write_normal_nd - write an NSGML document as for NSL_write_normal but no DDB PI at start.

In the above, using standard SGML terminology, PI = processing instruction and RE = record end.

The `NSL_Doctype dtype` can either be an SGML DTD description explicitly given (needed if the file does not contain an explicit <?NSL DDB ... > processing instruction, in particular for output files), or NULL, in which case the DTD to be used will be read from the file and saved in the return value where it can be found by using `DoctypeFromFile`.

A standard technique for opening an NSGML file, processing it and writing a modified version of it, can be initialised as follows (see also `simple.c` and `simpleq.c`):

```
NSL_File inf, outf;
NSL_Doctype dct=NULL;
inf=SFFopen(stdin, dct, NSL_read);
dct=DoctypeFromFile(inf);
outf=SFFopen(stdout, dct, NSL_write_normal);
```

This code opens an NSGML file for reading which sets the DTD that is being used. This DTD is then passed to `SFFopen` for the output stream, which will create an NSGML document stream using the same DTD as the input document.

extern NSL_File SFopen(char *fname, NSL_Doctype dtype, SFType type);

This is the same as the previous function, except that instead of a C file stream, we give it a file name.

extern void SFclose(NSL_File f);

Close the `NSL_File f` . This should be done explicitly for every output file opened by your program.

extern NSL_Doctype DoctypeFromDdb(char * filename);

Read a description of an SGML DTD which is contained in the .ddb file given by the file name `filename`.

extern NSL_Doctype DoctypeFromFile(NSL_File file);

Return a description of the SGML DTD which is used by an `NSL_File`.

10.3 Reading SGML

There are two methods of reading NSGML files. The first is to use `GetNextBit` or `GetNextItem` which are described below. The other is to use the NSL query facilities which are described in the next section.

extern NSL_Bit * GetNextBit(NSL_File sgmlfile);

Read and return the next `NSL_Bit` from the `NSL_File`, which must be open for reading. NULL is returned at end of file. At the moment it skips over processing instructions, it probably should not.

extern NSL_Item * ItemParse(NSL_File sgmlfile, NSL_Item *item);

Given an `NSL_Item` of type `NSL_inchoate`, (i.e. one which only refers to an SGML start tag), read the SGML file `sgmlfile` (up to the matching end tag) and fill in the contents of this `NSL_Item`.

extern NSL_Bit * GetNextItem(NSL_File sgmlfile);

Read and return the next `NSL_Item` from `sgmlfile`. If the current position in the file is before an SGML start tag, the entire contents of this element is returned. If before an SGML empty tag, then this is returned. NULL is returned at end of file. Processing instructions are ignored and the next element read and returned. It is an error if the current file position is not before SGML markup, i.e. before text or a close tag.

10.4 SGML Queries

extern NSL_Query ParseQuery(char* qu);

This function takes a string containing a query and converts it into an internal form which can be used as a argument to `GetNextQueryItem` or `RetrieveQueryItem`. A query is a pattern which defines which SGML elements to select.

extern NSL_Item * GetNextQueryItem(NSL_File infile, NSL_Query q, NSL_File ofile);

Given am NSGML document `infile` (open for reading) and a NSL_Query `query` return the next complete SGML element which matches this query as an `NSL_Item`.

Parts of the input document which occur between the present position and the matching item are written to the `ofile` output document. If `ofile` is NULL then input which does not match the query is read and discarded.

Subsequent calls to GetNextQueryItem return subsequent matches. In nested elements (for example <P> inside <P>) only the outer element is returned.

GetNextQueryItem returns a pointer to the matching item or NULL if end of file was reached.

A standard paradigm for using `GetNextQueryItem` is thus:

```
while( ( item=GetNextQueryItem(infile, query, outfile ) ) ) {
     process_item(item);
     PrintItem(outfile, item);
     FreeItem(item);
};
```

where `process_item` is your code that modifies `item`.

extern NSL_Item * RetrieveQueryItem(NSL_Item *uitem, NSL_Query query , NSL_Item *from);

This function does a depth first search of `uitem` to find all items inside it which match the query `query`. On the first call, `from` should be NULL. On subsequent calls, if `from` is set to the previous match (return value) then subsequent matches will be returned. NULL is returned if there are no (more) matches.

10.5 Printing

The following functions can be used to write to `NSL_File` outputs. Writing to `NSL_Files` using `printf` or other such functions is not recommended since that could result in the generation of invalid SGML .

extern void PrintBit(NSL_File fp, NSL_Bit *bit);

extern void PrintEndTag(NSL_File fp, char *label);

Equivalent to `PrintBit` on an item of type `NSL_end_bit`. `Label` is the name of the SGML element which is being closed.

extern void PrintItem(NSL_File fp, NSL_Item *item);

extern void PrintText(NSL_File fp, Char *text);

If `text` contains SGML markup, it is your responsibility to ensure that it is properly nested inside `text`.

extern void ForceNewline(NSL_File fp);

Writes a newline to the output stream `NSL_File`, but taking account of NSGML state and printing mode.

10.6 Freeing NSL data structures

The following functions will reclaim space used by NSL data structures of various kinds. It is the responsibility of the programmer to free structures which are no longer needed, garbage collection is not supported. The following functions are recursive however, in that they free the other data structures pointed to by the structures explicitly freed. You can free these `NSL_Items` twice without ill effects (the second free being a no-op, although an warning message will be written to stderr). Freeing any of the others twice will have undefined and probably catastrophic effects.

extern void FreeAttr(NSL_Attr *attr);

extern void FreeBit(NSL_Bit *bit);

extern void FreeData(NSL_Data *data);

extern void FreeItem(NSL_Item *item);

extern void FreeQuery(NSL_Query q);

10.7 Attributes

extern Char * GetAttrStringVal(NSL_Item *item, char *name);

Return the value of attribute `name` associated with the `NSL_Item item` as a string (0-terminated Char). Defaulted as well as explicitly given attributes can be accessed using this function. There is no direct way to tell if you are getting explicit or default values.

extern void * GetAttrVal(NSL_Item *item, char *name);

Return the value of attribute `name` associated with the `NSL_Item item` as an untyped pointer. Defaulted as well as explicitly given attributes can be accessed using this function. In the present release, this function is the same at `GetAttrStringVal`. In future, it is planned that we will introduce typed attributes, in which case this function will return a pointer to the typed value of the attribute, (for example a pointer to an int if the attribute is of type `NSL_attr_num`).

extern boolean PutAttrVal(NSL_Item *item, const char *aname, char *newval);

This function changes an existing attribute if present, adds a new one otherwise, and returns TRUE or FALSE depending on whether a new one was added. The type `boolean` is defined (in `nsl.h`) as `#define boolean int`.

extern void NewAttrVal(NSL_Item *item,const char *aname,char *newval);

Add a new attribute called `aname` to the `NSL_Item item` with the string `newval` as value. Even if there is already such an attribute present, another one is added. For this reason the use of `PutAttrVal` is recommended.

The following functions manipulate NSL_Attrs directly, and hence are at a lower level of detail than those above.

extern void SetAttrValue(NSL_Attr *refvar, Char *valin);

Set the value of an `NSL_Attr` to a string.

extern void * GetAttrValue(NSL_Attr *refvar);

Get the string value of an `NSL_Attr`.

extern NSL_Attr * CopyAttr(NSL_Item *item, NSL_Attr *attr);

Copy an `NSL_Attr`. This copies the whole linked list of `NSL_Attrs` pointed to by `attr`. `item` does not appear to be used.

extern NSL_Attr * FindAttr(NSL_Attr *attr, char *name);

Search down the linked list of `NSL_Attrs` starting at `attr` to find one which has the given `name`.

10.8 Creating NSL data structures

extern NSL_Data * NewNullNSLData(void);

Create a new empty `NSL_Data` structure (with type `NSL_undefined`).

extern NSL_Item * NewNullNSLItem(char *name, int len);

Create a new `NSL_Item` (with type set to `NSL_inchoate`) which refers to an SGML element with name `name`. `Len` is the length of the name.

10.9 Copying NSL structures

extern NSL_Data * CopyData(NSL_Data *data, NSL_Item *item);

This function makes a copy of the `NSL_Data` structure `data` and returns the copy. The new `NSL_Data` will be placed inside `item`, (i.e. its `in` pointer will point at `item`).

extern NSL_Item * CopyItem(NSL_Item *item);

This function makes a copy of the `NSL_Item` structure `item` and returns the copy.

extern void CopyBit(NSL_Bit *dest, NSL_Bit *source);

This function copies `source` to `dest`. It does not allocate space or return a useful result.

10.10 Other functions

extern NSL_Item * AddPCdata(NSL_Item *uitem, Char *pcdata, char *path);

Given an `NSL_Item uitem` and a string query `path` which describes a location of an element relative to `uitem`, we add `pcdata` at this location. If an item is not found which matches this query, then a matching item (and necessary intermediate structure) is created (assumes presumably that there are no . or .* in the query). The `pcdata` is then added to the matching item as the last data daughter.

extern NSL_Item * NextDFSNoChildren(NSL_Item *item);

Given an `NSL_Item` return the next `NSL_Item`. The returned `NSL_Item` corresponds to the next SGML element occuring textually after `item` in the document. However, this function does not read from a file, i.e. item must already be inside a tree.

extern NSL_Item * ObtainData(NSL_Data *data, char *itemname, int len);

This function returns a pointer to the first `NSL_Item` contained in the `NSL_Data data` which has the label (element name) `itemname`. `Len` is the length of the element name. NULL is returned if no such `NSL_Item` is found.

extern NSL_Item * ParentItem(NSL_Item *item);

Given a pointer to a `NSL_Item, item` , this function returns the `NSL_Item` which contains it, or NULL if it is not contained in any. It follows the `in` chain of pointers twice, in terms of Figure 2, the ParentItem of the <surname> item is the <name> item.

extern char * ElementUniqueName(NSL_Doctype doctype, char *name, int length);

Given a `name` of an SGML element and `length` (giving its length), return a unique string name for this element as defined in the DTD described by `doctype`. This unique name can then be used for comparison with the `label` field of `NSL_Items` by using the C operator `==`. NB `name` should be upper case.

extern NSL_ElementSummary FindElementAndName (NSL_Doctype doctype, char **name, int len);

Given a `name` of an SGML element and `len` (giving its length), return an `NSL_ElementSummary` describing the element definition in the DTD. `Name` is also overwritten with a unique string name for this element as defined in the DTD described by `doctype`.

11 Definition of NSGML

We define *semi-valid* SGML to mean a text file which is marked up using valid SGML according to a DOCTYPE. Semi-valid SGMLconsists of a sequence of text and or SGML elements, each of which is a valid subtree of the supplied DOCTYPE. For example, if the DOCTYPE defines an element <name>, then a file consisting of a sequence of <name> elements is semi-valid SGML, even though it may not be valid SGML. The reason for this increased flexibility is to allow the output of `sggrep` queries to be processed by subsequent NSL programs.

The file must not contain a <!DOCTYPE ... > statement. Instead of this it *may* contain an SGML processing instruction of the form <?NSL DDB *DDBFile* 0>, where *DDBFile* is the name of a file (called the DDB file) which contains a summary of the information contained in the DTD for this document. If the document does not contain such a processing instruction, then a *DDBFile* must be explicitly given when reading the NSGML file, see the definitions of `SFFopen` and `DoctypeFromDdb` in section 10.2.

In addition to the above, a file is in NSGML format if it satisfies the following conditions:

1. Document is semi-valid SGML, as defined above.
2. Document is coded using one of the ISO-LATIN character sets, with embedded character entities where necessary.
3. Reference concrete syntax is used.
4. No capacity/length restrictions.

5. No short refs or tag minimisation.
6. No SUBDOCs.
7. No marked sections.
8. All end-tags present (except for empty elements).
9. All entity references terminated with ";"
10. No entity references except (a) of type SDATA or (b) external references with explicit NOTATIONS.
11. All NAMEs (i.e. start and end tag GIs, attribute names, or attribute values of type NAME or NAMELIST) contain no lower case letters.
12. No CDATA or RCDATA element content.

12 Restrictions, Bugs and Features

This is unlikely to be the last release of the NSL package. This section notes some areas where the existing code could be improved.

1. Only one NSL_File can be open for reading at one time. We hope to remove this 'feature' in the next release, but for now we assume that each NSL application only reads one SGML file at a time.
2. It would be useful for the API to contain a function to read the content model of an SGML element from the DTD. Similarly, a function to read the allowed values of an attribute from the DTD. These functions would ease the writing of DTD independent code.
3. There is no function to remove attributes from SGML elements, i.e. to revert to the default attribute value. There may prove to be a need for such a function, which could be built easily enough.
4. NSL utility programs should be consistent in allowing input files to be specified as command line parameters, as well as input streams, as is common practice for UNIX programs.
5. This document does not say enough about SGML entities and how they are processed by NSL. It should.
6. SFopen will deal with compressed files for reading or writing. This should be documented and extended to deal with ".gz" as well as ".Z" file name extensions.
7. GetNextItem fails with an error if it finds a close tag. It would be more user friendly if it returned normally with an indication of what was wrong.
8. In general, errors should be handled in a less abrupt fashion.

References

[Kernighan 88] Brian W. Kernighan & Denis M. Ritchie, "The C Programming Language. Second Edition", Prentice Hall Software Series, 1988

[Goldfarb 90] Charles F. Goldfarb, "The SGML Handbook", Clarendon Press, 1990