The IMS Corpus Toolbox

# Corpus Administrator's Manual

Oliver Christ

Universität Stuttgart
Institut für maschinelle Sprachverarbeitung
– Computerlinguistik –
Azenbergstr. 12
D 70174 Stuttgart 1
oli@ims.uni-stuttgart.de

Last Modified: Wed Nov 9 14:33:27 1994 (oli)
Created: Thu Feb 24 10:34:11 1994 (oli)
Released: – not yet –
tc.bib entry: Christ:94b

# Contents

# Chapter 1

# Overview

## 1.1 Introduction

The IMS corpus toolbox is a set of tools for the efficient encoding, representation and querying of large text corpora. This manual describes how to encode a text corpus and how the various administration tools must be used to transform a text corpus into the representation used by the access tools. This manual does *not* describe the functionality of the query tools or the architecture of the toolbox in general. If the reader is not familiar with the overall architecture, we recommend a "top-down" reading through the more general papers, especially [Christ, 1994] for an overview of the system architecture as a whole.

The internal representation of a corpus consists of a set of files which represent the corpus data, the different "items" (i.e., words) used in the corpus and several index files for efficient lookup. To transform a text corpus from its textual representation to the internal representation used by the IMS toolbox, the following steps have to be performed:

1. transformation of the text file in one-word-per-line format;
2. encoding of the text file;
3. declaration of the corpus in a global "registry directory";
4. and building several file indices.

Steps 1 and 3 have to be done manually, for steps 2 and 4 there are tools within the toolbox.

The third step, the registration of a corpus, is necessary since almost all tools (but the one used in step 2 above) access a corpus via a symbolic name. When a symbolic name is passed to a tool, it is looked up in a central directory (called the "corpus registry"), where the tool expects to find a file with the very same name as the symbolic name of the corpus to be accessed. This file holds a description of the components of the corpus, mainly a list where the components are stored physically. So, a user does not have to know where a corpus is stored, he or she only has to know its symbolic name to access it.

After a corpus is transformed into its internal representation and registered, it can be used by the various tools of the toolbox, for example the query tools (XKWIC, CQP, `print-aligned`).

## 1.2 The role of the Corpus Administrator

Within the IMS corpus toolbox, the corpus administrator has the tasks to provide users with new corpora or to change existing corpora when some information has to be added or updated. Second, the administrator has to properly install the usually large corpus files in the filesystem and to find an "optimal" place with regard to backup policies, disk usage and access efficiency. Third, the corpus administrator has to care about access control, since corpora exist where copyrights or license agreements inhibit an unrestricted access, even within one institution. These tasks are similar to "standard" system administration. We therefore suggest that the corpus administration tasks are fulfilled by someone who is familiar with the standard Unix text processing tools, backup strategies, and security and access control, that is, your local system administrator.

## 1.3 Organization of this manual

This manual is organized as follows. The next chapter 2 explains the internal data structures which are used to store the corpus data. You may skip the entire chapter if you want, it is not necessary for the other chapters, but useful if you have problems with the tools or want to learn how to manipulate the data files. Chapter 3, then, describes the various steps which are necessary to transform a corpus from its textual representation into its internal representation. Chapter 4, then, describes in detail the registry directory and the format of the files which describe the physical attributes of a corpus. Chapter 5 describes how to set up the client-server-capabilities of the toolbox. Chapter 6 describes utilities and related tools which either add more (or other types of) information to a corpus or are useful for other purposes, for example for debugging of a corpus (during encoding). Another important point is access control for corpora, which is discussed in chapter 7. To check whether the tools can run at all on your system, you may refer to appendix A for hardware and operating system specific requirements of our tools.

## 1.4 Credits and Acknowledgements

The internal data structures we use in CQP, XKWIC and some other tools of the toolbox make use of integerized data files and reversed indices. Both of these techniques are well-known in the area of information processing for many decades, but to our knowledge the first who applied them to text and corpus processing in the linguistic area was KEN CHURCH. He deserves our greatest thanks for pointing us to these methods.

# Chapter 2

# Internal corpus representation

This chapter explains how corpus data is represented internally. When you understand the internal representation, you can use the tools of the toolbox to create, update or change corpus information without having to go back to the textual version and encoding the whole stuff again. You will also be able to figure out how to encode the internal representation for files which cannot be computed by the tools of the toolbox, for example due to memory problems, software bugs or limitations.

If you do not need to "hack" with the corpus data, you may skip the entire chapter. The understanding of the internal representation is not necessary for the other parts of this manual, but useful if you encounter problems with the tools.

## 2.1 Positional attributes

Within the IMS corpus toolbox, a corpus can have an arbitrary number of annotations of different types. In our system, a corpus is primarily regarded as a sequence of words (not as a sequence of characters). The words, then, are numbered, so that we can directly access the word at a certain corpus position $p$ (i.e., the first word in the corpus, or, in general, the $n$th word of the corpus). This leads to the more general notion of *positional attributes*, which is the most important annotation type. Attributes of this class have a (string) value at each corpus position.[1]

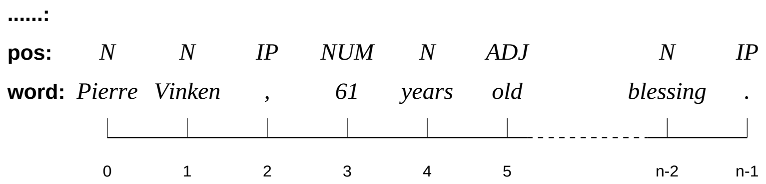


Figure 2.1: Corpus positions and values

[1]When the corpus is stored in a verticalized one-word-per-line format, a corpus position can also be regarded as the number of a line in this representation.

The corpus text falls within the class of positional attributes, since we can specify, for each corpus position, the word which occurs at that position. The positional attribute which holds the corpus text proper always has the predefined attribute name "word". Other positional attributes are, for example, part-of-speech tags, which are assigned to the words of the corpus. In our view, we regard POS-tags as assigned to a corpus position rather than to the word at that position. Then, the positional attributes "word" and "tag" do not differ very much any more: both have, for each corpus position, a value which is, in our case, a string (as illustrated in figure 2.1). We therefore use the same internal representation for the word sequence of the corpus (the corpus text) and the tag sequence (the associated POS-tags), as well as for other, additional positional attributes like lemmas, morphosyntactic tags, etc. In other words, a tagged corpus is in our view a set of two positional attributes of equal length, one of which captures the sequence of words, the other captures the sequence of tags. In the following, we therefore use the term "item" to abstract from the type of information encoded in such a positional attribute (here, word vs. tag). A corpus in general, then, is a collection of attributes of different types.

The question of the internal representation of corpora with multiple (positional) annotations can thus be reduced to the question of representing a single positional attribute (remember that the all positional attributes must be of equal length, that is, encode equal length item streams).

The two key concepts of the internal representation of a positional attributes are:

- *integerized representation:* items are encoded as integer numbers, where equal items (words, ...) get the same integer code. The sequence of items is then represented as a sequence of integer numbers;
- *inversed file indices*: for the sequence of numbers, an *inversed file* is created. The inversed file captures, for each item (better: item code) the set of occurrences of the item in the positional attribute.

For the construction of the integer code, you normally need a segmentation or tokenization tool, since `the,` and `the` are considered different and undesirably get different codes.

The advantages of the integer code is that the represented items have equal internal length (in the case of integers, 4 bytes on our machines). Since the length of the item sequence is known and the items are of equal length, the item sequence can be handled like an array of items, with the advantage of random access. The inversed file is needed for lookup: since it directly indexes the set of occurrences of a given item (code), the occurrences can be computed in a single step.

### 2.1.1 Integerized files

As said above, the first set of data structures is an integerized file of the textual representation of the item sequence. Then, the item sequence is represented as a sequence of integer codes.

It is obvious that at least two functions are needed to handle this encoding:

- first, a function to compute the integer code of a given item (a string);
- second, a function to retrieve the (character) string when the code is given.

These two functions require some auxiliary data structures to be efficiently computable.

The first data structure is the item list or *"lexicon"*: it captures the set of (different) items. Internally, this is the set of strings occurring in the item sequence, where a NULL character (octal `\000`) is padded at the end of each word. The file is *not* sorted (but it may be). A UNIX command to produce this file would be:

(2.1)
```
sort -u 1wpl-item-seq | tr '\n' '\0' > lexicon
```

where it is assumed that the input item sequence is in one-word-per-line format. In this example, the output would be sorted, but this is not necessary. The item list already defines the item code for each item, since it is assumed that the first item in the item list has code 0, the next one has code 1, and so on.

**Note** that "traditional" `tr`s delete ASCII 0 (
`0`) from the input stream, so that the example above will not work with "traditional tr". GNU's `tr` does not have this bug.

For the lookup of a string in this list, it is useful to have an index of starting positions of the strings in this file. This index gives, for each item code, a mapping from item codes to the file offsets (in bytes) in the item list. Thus, the starting position $s$ of the the string represented by the item code $c$ is computed in one step via this index, which we call the *item list index* or *lexicon index*. Since the strings in the item list are terminated with the NULL character (which must not occur in the items themselves), the string represented by an item code is everything between the starting position computed by the item index up to the next NULL character.

Again, this index can be computed by a UNIX command when the lexicon is already computed:

(2.2)
```
tr '\0' '\n' < lexicon |
  gawk 'BEGIN{pos=0}
        {print pos; pos= pos + length($1) + 1}' |
  atoi > lexicon.idx
```

`atoi` is a utility program included in the toolbox and maps numbers (represented textually as a sequence of digits) to their internal representation.

The next data structure supports the mapping from strings to their item codes. This could be done in a number of different ways – currently, binary search over a sorted string index is implemented. [2] For example, the same functionality could be achieved with TRIEs, which perhaps would need more space, but could compute the conversion in $n$ steps where $n$ is the length of the input string.

[2]The method currently implemented in the toolbox is very simple and could be sped up a lot, but since it is rarely used (all computations are done on the item codes, whenever possible, instead of strings), we didn't yet convert it to a more efficient method.

The binary search requires a sorted structure. For this purpose, we do not keep a sorted item list, but rather another index (denoted by $L_s$) which holds, for each position $p$ of an item in a "virtual" sorted item list (ranging from 0 to the number of encoded items minus 1), the item code at this position. So, $L_s(0)$ is the item code of the "smallest" item, and $L_s(1)$ is the code of the second-smallest item, etc. The sorted item list can thus be textually printed by the function

```
for (i = 0; i < "SizeOfItemSet"; i++) {
  code = SortIdx[i];
  s = LexIdx[code];
  print s;
}
```

Here, for each possible position in the sorted index, $i$, first the item code *code* at that position is computed. Then, through accessing the item index, the character string represented by *code* is determined, which is then printed.

As you can imagine, this file can easily produced by a UNIX command:

```
(2.3) tr '\0' '\n' < lexicon |
        gawk '{print NR-1 "\t" $1}' |
        sort +1 |
        gawk '{print $1}' |
        atoi > lexicon.srt
```

The first line computes the strings from the item list, which are then prefixed by their code (which is the "position" in the item list), beginning with code 0 for the first word. This list of code/value pairs is then sorted by the values, which occur in the second column. The output of the sorting is then filtered, so that only the codes are printed. The code sequence is then transformed into the internal format and written to the index file.

**Note:** One of the reasons we do *not* use these UNIX commands to create the data structures is that the UNIX `sort` command sometimes handles the order of 8bit-characters differently from other programs (it works with signed characters, whereas internally we work with unsigned characters). So the UNIX commands which use `sort` only create the same files when the standard 7bit ASCII character set is used. The internal functions which map from items to item codes will not work properly otherwise.

The data structures used to represent the encoded item sequence and the associated auxiliary data structures which facilitate the necessary mappings are illustrated in figure 2.2.

The only file for which we didn't yet give a UNIX command is the item sequence (or better, the sequence of encoded items). `gawk`'s arrays can be used for this purpose. One possiblity is to read an already existing item list file, which may be produced by the commands above. Another possiblity is to produce the item list and the indices in a single `gawk` run. The script below can be used for this second purpose, but it assigns other item codes:

```
(2.4) BEGIN{
        maxcode = 0;
        position = 0;
      }
```

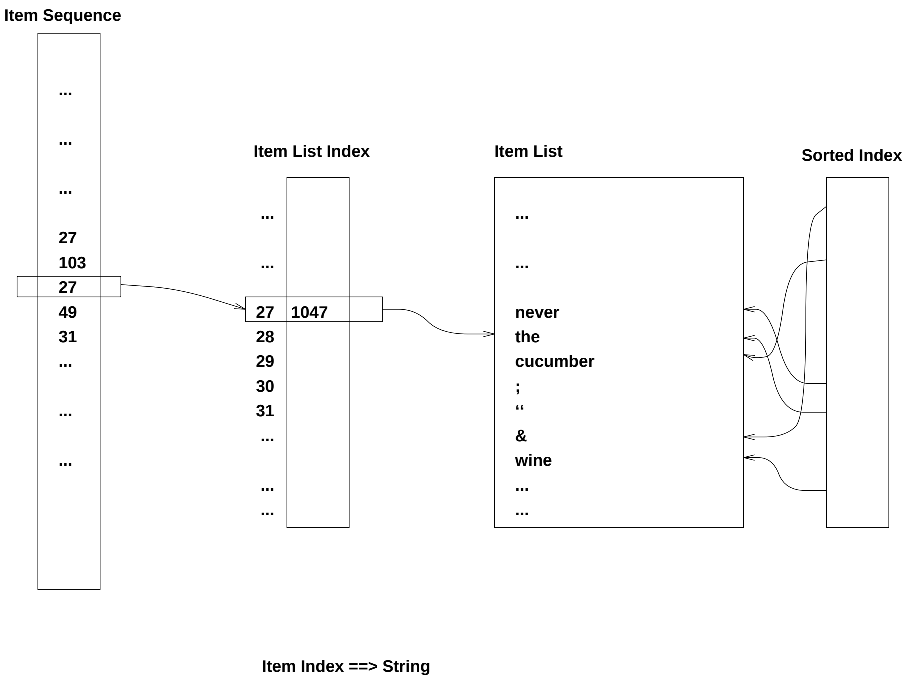


Figure 2.2: Integerized items and associated data structures

```
{ if (!($1 in itemlist)) {
    itemlist[$1] = maxcode;
    print $1 > "lexicon.asc"
    print position > "lexicon.idx.asc"
    position = position + length($1) + 1;
    code = maxcode;
    maxcode++;
  }
  else
    code = itemlist[$1];
  print code > "corpus.asc"
}
```

After the code is executed with a text file as input, the ASCII representations have to be converted into the internal format (this could be done via pipes in the `gawk` script also, but we left that out here for the sake of clarity):

(2.5)
```
atoi < corpus.asc > corpus
tr '\n' '\0' < lexicon.asc > lexicon
atoi < lexicon.idx.asc > lexicon.idx
rm -f *.asc
```

After that, command 2.3 can be used to produce the sorted item list index.

### 2.1.2 Inversed item sequence

The second set of data structures concerns the inversed file index associated with the item sequence. This inversed file holds, for each item code, the set of positions in the item sequence where the item code occurs. Through the mapping functions introduced in the last section, we can also regard the inversed item sequence as a list of corpus positions where a certain word or part-of-speech tag occurs.

The inversed file is represented by a set of three files:

- first, the inversed file itself, which contains a set of corpus positions;
- second, an index into this file. This index returns, for each item code, the start point of the associated occurrences in the inversed file;
- third, a table of item code frequencies, which gives, for each item code, the number of occurrences of the code in the corpus (which is, of course, equal to the size of the set of occurrences).

The three files can also be computed by UNIX commands. First, the reversed sequence is produced by the following command:

(2.6)
```
itoa corpus |
  gawk '{print $i "\t" NR-1}' |
  sort -ns |
  gawk '{print $2}' |
  atoi > corpus.rev
```

First, the internal representation of the item sequence is converted into readable numbers. This number sequence is then suffixed with its position in the corpus, which is then sorted by the code, so that we get code/position pairs. From this sequence, the position is stripped off, so that we only get the sequence of positions, which exactly is the inversed file.

The frequencies can already be computed in the `gawk` encode script (2.4), but another possibility is a slightly modified version of the script above:

(2.7)
```
itoa corpus |
  gawk '{print $i "\t" NR-1}' |
  sort -ns |
  gawk '{print $1}' |
  uniq -c |
  gawk '{print $1}' |
  atoi > corpus.cnt
```

Here, we keep the code sequence of the code/position pairs. This sequence of codes appears in sorted order. By the call to the `uniq` utility, equal subsequent lines (here: codes) are collapsed into only a single line and counted. These counts are stripped off and converted to internal format.[3]

The last file, the index into the inversed file, can simply be computed from the frequencies by summing them up:

(2.8)
```
itoa corpus.cnt |
  gawk 'BEGIN{pos=0}{print pos; pos+=$1}' |
  atoi > corpus.rdx
```

[3] It would be more efficient to use a `gawk` array to hold the item code counts, since the `sort` step could be omitted. The version here is just for clarity.

Now, the whole set of seven files representing the data of a positional attribute (which we call the seven *components* of a positional attribute) have been created. In the toolbox, there are tools which perform these steps much faster than the shell scripts presented here. But in some cases, the utilities of the toolbox run into memory problems, and then these scripts may help to produce the encoded version of a corpus.

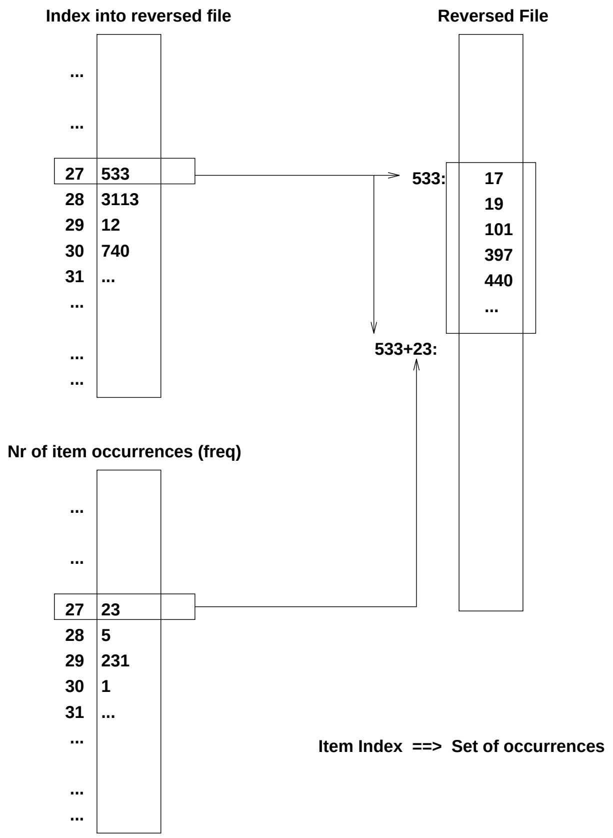


Figure 2.3: Reversed file indices

The components associated with the inversed file and their meanings are illustrated in figure 2.3. The next section will show the single steps which are taken when a simple search for an item, a word for example, is performed.

### 2.1.3 Example: a simple word search

After the internal data structures have been introduced, we can compute the concordance for a single item, for example the word *the*. Most datastructures can be treated as an array, so we use the symbols

- $C$ for the item sequence (accessed by $C[i]$ where $i$ is a corpus position). The elements of $C$ are item codes;
- $R$ for the reversed item sequence (accessed by $R[i]$ where $i$ is an index into this sequence, computed from $I_R$ below). The elements of $R$ are corpus positions;

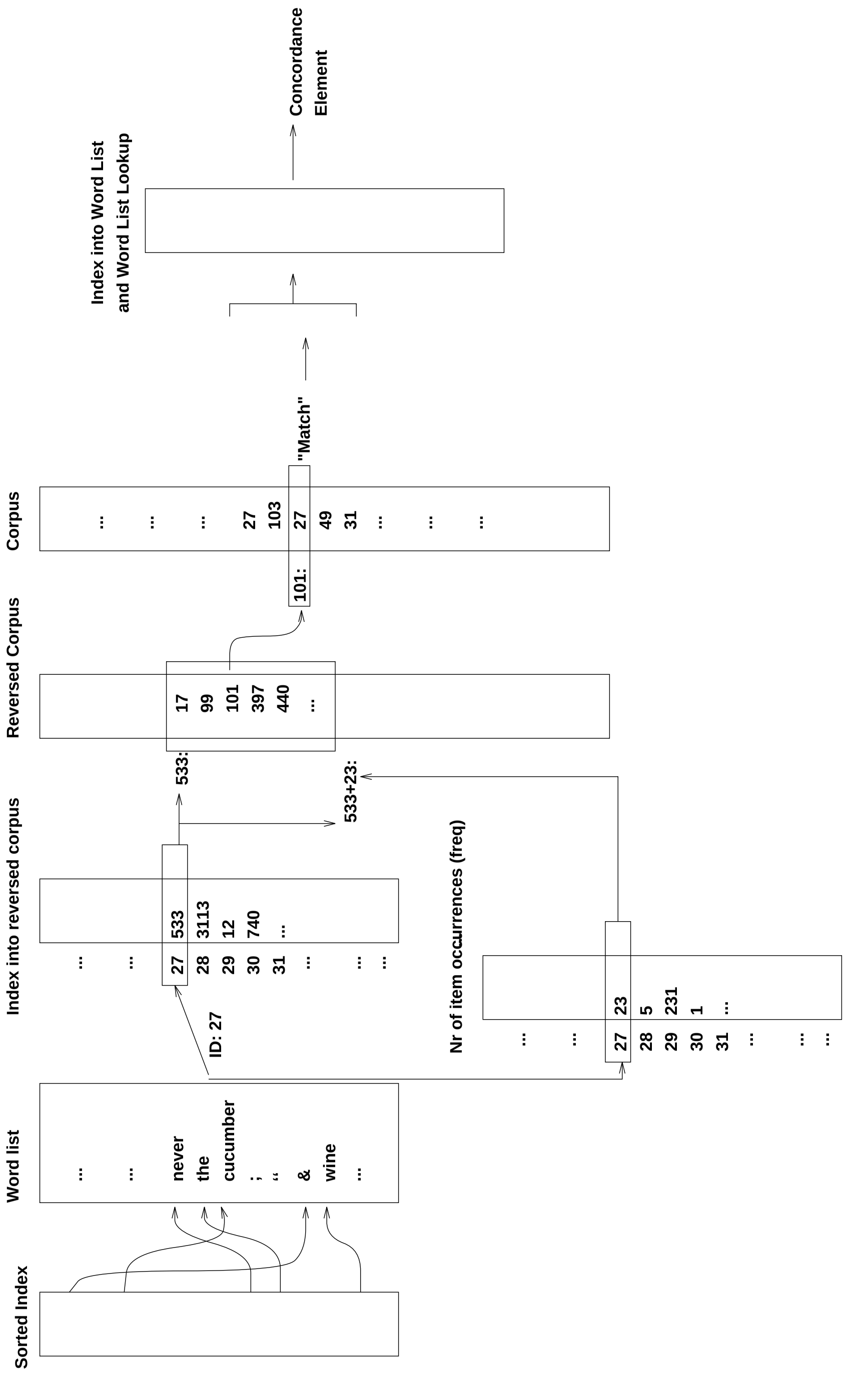


Figure 2.4: A simple word search

- $I_L$ for the item list index (accessed by $I_L[c]$ where $c$ is an item code). The elements of $I_L$ are byte offsets into the item list;
- $F$ for the item frequency table (accessed by $F[c]$ where $c$ is an item code). The elements of $F$ are item frequencies in $C$;
- $I_R$ for the reversed item sequence index (accessed by $I_R[c]$ where $c$ is an item code). The elements of $I_R$ are pointers (offsets) into $R$;
- $L$ for the item list (an array of characters, only accessed by offsets of $I_L$);
- $S_L$ for the sorted item list index (accessed by $S_L[i]$ where $i$ is a position in the "virtual" sorted item list). The elements of $S_L$ are item codes.

These seven arrays are the *components* of a positional attribute.

For computing the set of occurrences of a textual item in $C$, the following steps have to be taken (also illustrated in figure 2.4 for the word "the"):

- first, the *item code* $c(i)$ of item $i$ has to be determined. For this purpose, the *sorted item index* $S_L$ is consulted and searched with binary search until the item code is found;
- if the item code could be determined, the *reversed item sequence index* is consulted to determine the starting position $r_s(i) = I_R[c(i)]$ of the position set associated with $i$ in the reversed item sequence;
- second, the *item frequency list* is accessed to compute the "length" of the position set $f(i) = F[c(i)]$;
- then, the set of occurrences $P(i)$ is the set of positions stored in the *reversed item sequence* $R$ starting at $r_s(i)$ with length $f(i)$ $(R[r_s(i)] \ldots R[r_s(i) + f(i) - 1])$.

The task of computing the set of occurrences of $i$ in the item sequence is then completed. Note that the item sequence itself didn't have to be accessed.

For computing the concordance and printing it, though, the item sequence $C$ must be consulted. When $c_l$ is the left display context (in terms of items) and $c_r$ is the right context, for each $p \in P(i)$ the "subsequence" between $[p - c_l, p + c_r]$ in $C$ must be computed (in the bounds $(0, |C| - 1)$. For each item $k$ in this subsequence, the associated (textual) item must be determined by computing the start position $t_s(k) = I_L[k]$ of $k$ in the item list index. Then, the item list can be consulted to get the string $s(k)$, which then is printed.

### 2.1.4 The set of positional attributes

The IMS Corpus Toolbox supports an arbitrary number of positional attributes. Each positional attribute has its own set of components. For each positional attribute, the length of $C$ and $R$ are equal (see also figure 2.5):

$$|C| = |R|$$

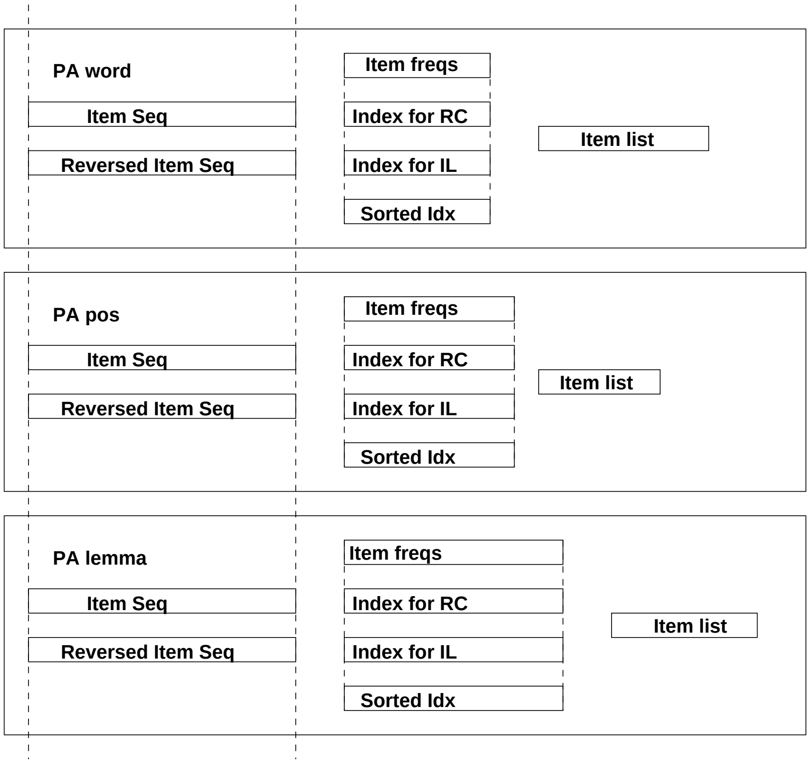


Figure 2.5: The set of positional attributes

and the lengths of $I_L, I_R, S_L$, and $F$ are equal:

$$|I_L| = |I_R| = |S_L| = |F|$$

Furthermore, between all positional attributes associated with a corpus, the lengths of the item sequences of these attributes must be equal:

$$|C^{word}| = |C^{lemma}| = |C^{pos}| = |C^{syn}| = ...$$

of course, no such condition usually holds between the other components of two positional attributes.

## 2.2 Other attribute types

### 2.2.1 Structural attributes

Structural attributes capture information about boundaries of sentences, paragraphs, phrases, or other entities. Internally, these structures are represented as *intervals of corpus positions*, which are the start and end point (inclusive) of the structure. Such an interval is a pair of corpus positions. Therefore, each structural item needs 8 bytes of storage internally (4 bytes, the size of an integer, for each of the two positions).

Currently, there are two limitations with respect to structural attributes:

- first, the intervals must not be recursive (for example, embedded NPs in NPs);
- and they must not be overlapping.

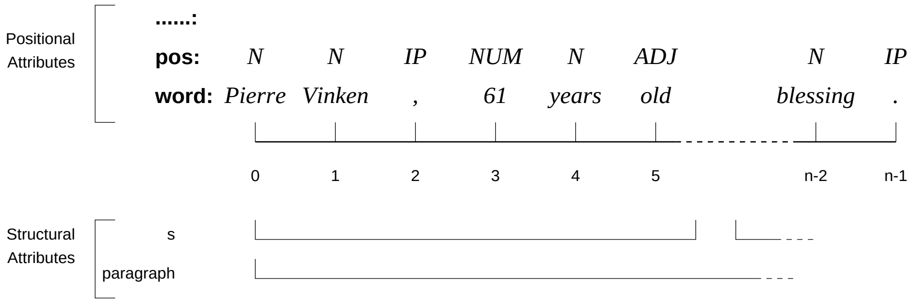


Figure 2.6: Structural attributes

Figure 2.6 illustrates the representation of structural attributes. The number of structural attributes associated with a corpus is not limited.

### Creating structural attribute data

Unlike positional attributes, the data for a positional attribute is stored in a single file, $S$. Normally, the structural attribute data is created with the `encode` utility. But in some cases, it is useful to manipulate or create the files through other utilities. The data file $S$ is an array of integer pairs, where $|S|$ is the number of intervals. The file size of $S$ is then $4 * 2 * |S|$, since for each interval, two integer numbers have to be stored, each of which needs 4 bytes.

If the files are constructed manually (without the help of `encode`, for example, after the corpus has been encoded), a simple `awk` script can help. You must, however, be aware of the internal representation of positional attributes and the "logics" of corpus positions; second, some pitfalls have to be circumvented.

Let's assume that a one-word-per-line input file with marked sentence boundaries (in SGML-style, like <s>...</s>) is available. Then, the intervals can be extracted by the following `awk` script (the output of which has to be converted into internal integer format with `atoi`):

(2.9)
```
BEGIN{
  position = 0;
  open=0;

  structure = "s"

  opentag = "<" structure ">"
  closetag = "</" structure ">"
}
{ if ($1 == closetag) {
    # closing tag, don't increment position.
    if (open) {
      print position - 1; # thanks to a3@wsserv.vdl.nl (Adri Verhoef)
```

```
      open = 0;
    }
    else {
      print "Closing non-open group at line " NR ": " $0 >> "/dev/stderr"
      exit
    }
  }
  else if ($1 == opentag) {
    # tag, don't increment position then
    if (open) {
      # forgot to close group, which we don't consider an error
      print position-1
    }
    print position
    open = 1
  }
  else if ($1 ~ /<\/?[a-zA-Z]+>/) {
    # do nothing, other structural tag?
  }
  else
    position++;
}
END{
  if (open)
    print position-1
}
```

First, care must be taken when groups are closed which are not open. The other case, reopening open groups, is not considered an error, since closing tags are optional. Additionally, when structure tags are used in the text, the line number (position) must not be incremented. But even then, this `awk` program may yield errors. So, at least check whether the size of the resulting file can be divided by 8.

### 2.2.2 Alignment attributes

### 2.2.3 Bigram and mapping tables

## 2.3 External tools and dynamic attributes

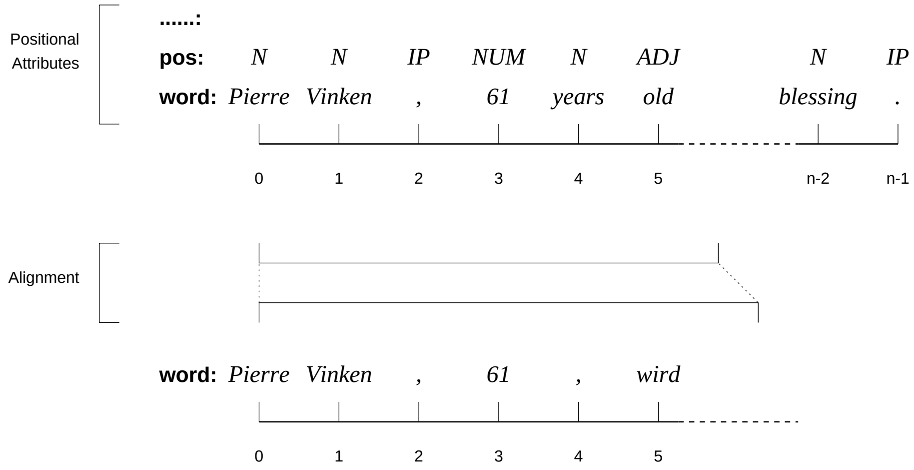


Figure 2.7: Alignment attributes

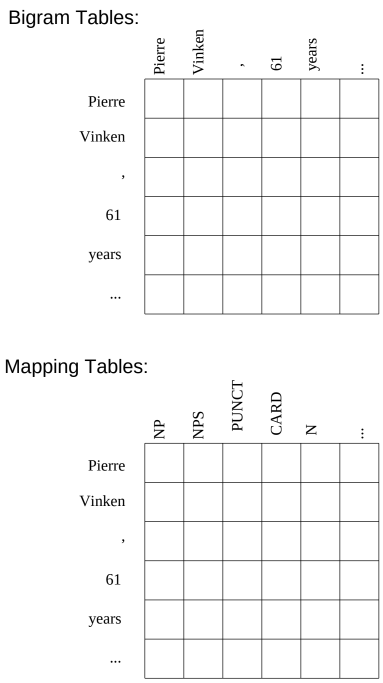


Figure 2.8: Bigram and mapping tables

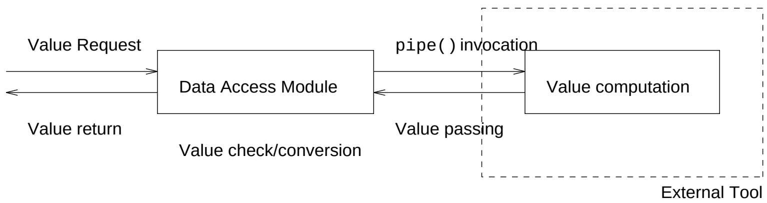


Figure 2.9: External (dynamic) attributes

# Chapter 3

# Encoding: Transforming a corpus into its internal representation

Within the IMS Corpus Toolbox, a corpus can have an arbitrary number of annotations of different types. In our system, a corpus is primarily regarded as a sequence of words (not as a sequence of characters). The words, then, are numbered, so that we can talk about the word at a certain corpus position $p$, the first word in the corpus, or, in general, the $n$th word of the corpus. This leads to the more general notion of *positional attributes*, which is the most important annotation type. Attributes of this class have a (string) value at each corpus position.[1]

The corpus text falls within the class of positional attributes, since we can specify, for each corpus position, the word which occurs at that position. The positional attribute which holds the corpus text proper always has the predefined attribute name "word". Other positional attributes are, for example, part-of-speech tags, which are assigned to the words of the corpus. In our view, we regard POS-tags as assigned to a corpus position rather than to the word at that position. Then, the positional attributes "word" and "tag" do not differ very much any more: both have, for each corpus position, a value which is, in our case, a string. We therefore use the same representation for the word sequence of the corpus (the corpus text) and the tag sequence (the associated POS-tags). In other words, a tagged corpus is in our view a set of two corpora of equal length, one of which captures the sequence of words, the other captures the sequence of tags. In the following, we therefore use the term "item" to abstract from the type of information encoded in such an attribute (here, word vs. tag). A corpus then, is a collection of attributes of different types.

In the following section 3.1, we shortly describe the internal representation of a corpus. Section 3.2 describes the steps which are necessary to prepare a textually represented corpus to be suitable as input for the encoding tools as well as the first of the two encoding tools, `encode`. Section 3.3 then describes the second encoding tool which is used to build the indices associated with a corpus.

[1] When the corpus is stored in a verticalized one-word-per-line format, a corpus position can also be regarded as the number of a line in this representation.

## 3.1 The internal representation of a corpus

After encoding, each item of a textual corpus is represented as a unique integer value[2]. For example, if the first item of a text corpus is "The", all "The"s in the text corpus will internally be represented as the integer number 0[3]. The corpus can then be physically represented as a sequence of integer numbers. To be able to get the item which is represented by an integer, another (indexed) file holds the mappings from integers to strings. Up to now, 3 files are necessary to hold the information: the corpus (consisting of a sequence of integers), the "lexicon" which holds, for each integer, the string it represents, and an index to the lexicon. These three files are those which are produced within the second step during the encoding of a corpus. The tool which performs this task is called `encode` and is described below in section 3.2.

Two additional files are built next: the first holds information about the sorted sequence of the items in the lexicon, which is necessary to efficiently compute the integer code of an item, given the string. The other file holds, for each item, the number of times it occurs in the corpus.

For efficient lookup, a *reversed file* or *reversed index* has to be built. This index holds, for each item in the corpus, the corpus positions where the item occurs. This index is indexed itself, leading to another file. In summary, we have seven files so far which represent the information of one positional attribute. The four additional files which are not built by the `encode` program are created with the `makeall` program described in section 3.3. But prior to the encoding of a corpus, it has to be transformed into a format which is suitable as input for the `encode` program, which is described in the next section.

## 3.2 The `encode` program

When a corpus consists of several positional attributes (for example, a POS attribute additionally to the standard "word" attribute), it can either be encoded in one single step (provided that it is in a suitable textual input format for `encode`) or the various positional attributes can be encoded one after another and be added to an already existing corpus. This latter way is also useful when one of the positional attributes has been changed, for example, when a tagset has been changed or a better tagger was available to produce more accurate tag assignments.

In both cases, the input format is a *one-word-per-line* format, where each item which is to be encoded occurs in a single line. This line may contain blanks, providing a way to encode adjacent multi-word lexemes, if desired. But care should be taken to avoid blanks at the end of an item, since, for example, "`the`" and "`the `" are different strings and therefore are encoded with different codes which can lead to undesired effects when not all occurrences of `the` are found in a text due to a blank at the end of some.

In the case of the corpus text, the input may look as follows:

[2] The internal corpus representation we use is highly inspired by an – unfortunately – unpublished draft paper by Ken W. Church, "A Set of Unix Tools for Processing Large Text Corpora".

[3] Of course, "the" will get another code than "The", since "The" and "the" textually differ and are not considered equal.

```
Pierre
Vinken
,
61
years
old
,
will
join
the
board
as
a
nonexecutive
director
Nov.
29
.
```

Another file, then, may hold the sequence of assigned tags (in which case both files must hold the same number of lines). The input format can, for example, be produced out of a raw text file with the `tr` command[4]:

```
tr ' ' '\n' < text_file > 1wpl-file
```

This command replaces all blanks in the input file with line breaks. The `encode` program then takes a one-word-per-line input file (or reads that format from stdin) and creates the three files `corpus`, `lexicon` and `lexicon.idx` in the current directory:

```
encode -t 1wpl-file
```

The `-t` option instructs `encode` to read its input from the file given as an argument of the option instead of reading the standard input. To do both the transformation and the encoding in a single step, one could enter the following pipe:

```
tr ' ' '\n' < text_file | encode
```

or, if one wants to hold the text in a compressed format:

```
zcat text_file.gz | tr ' ' '\n' | encode
```

In general, the one-word-per-line format can be produced by any program you like. In the simple `tr` examples above, it is already assumed that the corpus has been tokenized (special characters have been separated from the words), perhaps even a sentence boundary detector must be run on a raw corpus to produce the appropriate input file. The simple example only shows how `encode` processes its input, in general, this method is not applicable to new, raw corpora which first may have to be preprocessed by other tools.

Now, `encode` may also take an annotated corpus with several positional attributes in a single file. In this case, each line of the input format consists of a number of attribute values, separated by tabulator characters. Thus, the input file consists of several columns, each denotes one positional attribute. A POS-tagged text then may look as follows:

[4]The `tr` command is a standard command available on many platforms and is not part of this tool set.

```
Pierre<tab>NP
Vinken<tab>NP
,<tab>,
61<tab>CD
years<tab>NNS
old<tab>JJ
,<tab>,
will<tab>MD
join<tab>VB
the<tab>DT
board<tab>NN
as<tab>IN
a<tab>DT
nonexecutive<tab>JJ
director<tab>NN
Nov.<tab>NP
29<tab>CD
.<tab>SENT
```

where <tab> denotes a single tabulator character (ASCII value 9)[5]. `encode` must then know which positional attribute is represented in the other columns of the file and how they shall be named. This is done with the `-P` option:

```
encode -t <input-file> -P pos
```

Here, the option `-P pos` ("P" for "positional attribute") instructs encode to treat the second column in the file as the sequence of values of the `pos` attribute. The files associated with the `pos` attribute have their names prefixed with `pos` (which leads to `pos.corpus`, `pos.lexicon` and so on). The order in which the `-P` options are given is relevant, since the first `-P` option denotes the name of the attribute represented in the second column in the input file, the second `-P` option denotes the third column etc. By default, the first column is treated as the word sequence and therefore gets the prefix `word`, but this can be overridden with the `-p` option. Please refer to the manual page of `encode` for details.

Up to 32 positional attributes can currently be encoded in a single step. If large amounts of text are to be encoded, try to determine the disk space the corpus needs after encoding in advance and look for a filesystem where enough space is available. Some hints on the expected size are given in section 3.4. As explained in chapter 4, it is possible to split the files of a corpus between several filesystems in case there isn't enough space on a single disk. If all else fails, you may have to encode the set of positional attributes in several runs of `encode`, each with the appropriate prefix passed with the `-p` option.

A corpus (or an arbitrary positional attribute) may be assigned another type of information which can be encoded with the `encode` program, namely structural information which can be used to represent article, sentence or paragraph boundaries. This kind of information is represented in the input file with SGML-like markers:

```
<article>
<s>
Pierre<tab>NP
Vinken<tab>NP
,<tab>,
```

[5] There are no general hints on how to produce this input format. In general, it is a good idea to use standard tools like `awk` and `sed`.

```
...
29<tab>CD
.<tab>SENT
</s>
<s>
...
</s>
</article>
```

Of course, structural information can be encoded independently of additional positional attributes in the file. Some points have to be noted:

- the end tags may be omitted. In that case, a structure spans all items until the start of the next structure or the end of file;
- in a line with a structure marker (s, article), no values of positional attributes may occur;
- if a structure marker line, everything after a blank or after the closing angle bracket (>) of the tag is neglected;
- structures must not be recursive or overlapping, that is, trees cannot be represented.

In the above example, the call for `encode` would look like this:

```
encode -t <input-file> -P pos -S article -S s
```

Here, the two encoded structural attributes are each declared with the `-S` (for "structural attribute") option. The order in which the structural attributes are declared does not matter.

Again, up to 32 structural attributes can currently be encoded in a single step.

Be careful to declare all structural attributes in the `encode` call, since undeclared structural attributes are considered as simple attribute values in the first column and therefore are treated literally. If you are encoding several positional attributes at once, an error will occur since lines with undeclared structural attributes in the first column in general do not have tabulator-separated columns after them. The rule is simple: either a line consists of a structure attribute designator in angle brackets or is line of a fixed numer of tabulator-separated columns. Errors may occur if this rule is not obeyed.

`encode` accepts a number of further options:

- `-p <prefix>` has the effect that the files belonging to the positional attribute in the first column of the input file will get the prefix "`prefix.`". Note that the dot at the end of the prefix is added automatically and must not be part of the option value. Normally, the files get the name `corpus[.cnt,.rev,.rdx]` and `lexicon[.idx,.srt]`. This will lead to problems if a positional attribute is encoded *after* a corpus is already present in the directory the data is written to, since the names of the corpus files may collide with the those of the new attribute. Therefore, when the prefix is not given, data may be overwritten and lost. This option affects only the names of the files for the positional attribute in the first column of the input file, the other positional attributes – if present – will get the prefix given with the `-P` option;

- `-d <path>` lets the user specify the directory in which the data shall be written. The path should *not* end with a slash. Default is to write all output files to the current directory;
- `-s` instructs encode to skip empty lines (lines with no characters – not even blanks – in it) during encoding.

`encode` accepts a number of further options. Please refer to the `encode` manual page for the most recent description of the program.

Please be aware that `encode` writes it output files into the current directory unless the `-D` option is specified. Be careful when you add (or update) a positional attribute to a corpus after encoding the corpus itself: a loss of data may occur during the encoding of the positional attribute, since it tries to write the data to the same files in which the corpus data may already be stored. Either you should pass the `-p` option to prefix the files belonging to the first column or put each positional attribute in a directory of itself to prevent `encode` from overwriting important data. It is a *very* good idea to change the file access mode of all files belonging to a positional attribute to non-writeable for anyone after encoding, in order to prevent accidential overwriting.

## 3.3 The `makeall` program

After encoding a corpus, each positional attribute has to be declared in the corpus registry. Please refer to chapter 4 for a detailed description of how to do this. The `makeall` program, which constructs the second set of files during the encoding process, will not work on undeclared positional attributes or corpora.

After a positional attribute is declared in the registry, `makeall` must be run to construct the necessary index files. There are no options, and the only argument `makeall` accepts is the symbolic name of the positional attribute for which the index files shall be produced:

```
makeall treebank
```

This will produce all missing files for all positional attributes declared for the corpus `treebank`.

If you only want to produce the files for a single positional attribute, give the name of the attribute as an additional argument:

```
makeall treebank pos
```

(given that these symbolic names are those of the respective positional attributes). Note that this call can be issued from any point in the filesystem, since `makeall` looks up in the registry to find the corpus data. Data is only written to the directory specified in the registry description file. It is therefore a good idea to run `makeall` either on a very fast machine or on a machine which locally holds the disks the corpus is stored on (to reduce network load in case of NFS-mounted filesystems).

**Note:** "makeall" will currently try to create non-compressed files for attributes which already have the complete data in compressed form. This is a bug and will be fixed in a future release.

`makeall` currently produces a lot of debugging output. This output is only important when errors occur or when the program has to be debugged.

After running `makeall` on all positional attributes of a corpus, the corpus is ready for use.

When trying to encode really big corpora (20 million words and above), `encode` and `makeall` may have problems due to memory or swapspace limitations and yield error messages like "can't allocate memory" or "not enough memory". These problems can only be solved by providing more swapspace to the machine the program is running on (or to run `encode/makeall` on another machine at your site which has enough swapspace). The available swapspace is shown with the `pstat -s` command and should be enough to hold the whole corpus data (see below in section 3.4). Ask your system administrator for further help.

## 3.4 Space requirements

This section gives some hints on how much space will be used by an encoded attributes. Currently, we only cover positional and structural attributes here.

### 3.4.1 Positional attributes

Let $A$ be the sequence of items captured by the positional attribute. The length of this sequence, $|A|$, therefore is the number of elements of this sequence. Further, let $D$ be the set of distinct strings encoded in $A$ (the list of different words, for example). Then, $|D|$ is the number of distinct words.

Two numbers are important:

- the number of items in the input file ($|A|$) which is equal to the number of lines in the one-word-per-line input file for `encode` (minus the number of structural annotation markers, if present). This number can, for example, be computed with the `wc -l` command (maybe pre-piped with a `grep -v` command to get rid of the structural annotation markers);
- the number of *distinct* items in the input file ($|D|$) (this number can be computed by running the pipe `sort -u | wc   -l` over the respective column in the one-word-per-line input text file).

Another important number is the space needed for the one-time representation of all different items, which here is denoted $S$. This is the sum of the lengths of each different word plus one:

$$S = \sum_{s \in D} (strlen(s) + 1) = |D| + \sum_{s \in D} strlen(s)$$

The adding of 1 is necessary since a null character (`'\0'`) is added to each string. The number is given by running the pipe `sort -u | wc -c` over the respective column in the input file.[6]

Now, the size of one positional attribute (in bytes) can be computed as follows:

$$M_p = 2(|A| * 4) + S + 4 * (4 * |D|) = 8|A| + 16|D| + S$$

The size of the input text file does not go into this formula, since it is not needed any more after encoding.

For each positional attribute, this formula has to be evaluated again, since the number of different items in the attribute ($|D|$) and the space needed to represent them once ($S$) usually differs between several positional attributes ($|A|$ must be constant for any two positional attributes of a corpus). Since positional attributes but the word attribute usually have a small number of distict values, the space needed to represent a positional attribute mainly depends on its length, which is not very surprising. A less accurate number of space requirement can be roughly estimated by multiplying the size of the uncompressed input text file(s) by 2.

### 3.4.2 Structural attributes

The data of a structural attribute, for example sentence boundaries, is stored in a single file, as an ordered sequence of corpus intervals (that is, pairs of corpus positions). So, the computation of the space needed to represent the information of one structural attribute $S$ is very simple: let $S$ be the structural attribute. Then, $|S|$ is the number of intervals stored in this attribute ("the number of sentences"), each being a pair of two corpus positions. Since each corpus position is stored as a 4-byte integer number, the space can be computed as follows:

$$M_s = (|S| * 4 * 2) = 8 * |S|$$

So, if you want to represent 1 000 sentences, you need 8 000 bytes to store the data.

[6]The columns can be extracted from a multi-column file with the `cut` program which is contained within GNU's/FSF's set of text utilities, or with the `awk` utility.

# Chapter 4

# The corpus registry

The corpus registry holds, for each corpus being processed by the toolbox, a file which describes where in the filesystem the various files which build the corpus are stored. Additionally, the registry holds a file which describes who can access a local corpus from remote hosts and a file which captures a log of all remote connections to local corpora. This chapter only describes the description files for local corpora and the description files for remote corpora, the two additional files which are necessary for remote connections are described in chapter 5.

The registry is simply a globally accessible directory, called the *registry directory*, and holds, for each corpus, a file with the same name as the corpus name, in lowercase letters.[1] Such a file is called the *registry file* of the corpus. The contents of the file, described in section 4.2 below, define which annotations are associated with the corpus and where the data is stored. An annotation which is not defined in the registry file of a corpus cannot be accessed by any of the tools. Similarly, when an attribute is defined in the registry, it is supposed to be accessible by all tools. It is within the responsibility of the corpus administrator (you!) to assure that all annotations defined in all registry files are accessible, and that only those attributes are defined which are in fact accessible.

Currently, there are some rules how to name the attributes of a corpus and how to name the respective registry files. These rules are described in the following section.

As already mentioned in section 1.2 above, corpora exist where access has to be controlled. This issue is discussed in chapter 7 below.

At the end of this chapter, section 4.5 summarizes the single steps which you should follow when preparing and registering a new corpus.

## 4.1 Some remarks about nomenclature

Two simple rules have to be obeyed for the definition of names for corpora and attributes:

[1] In CQP, all corpus names are entered in upper case, but they are converted to lower case to load the correct corpus.

corpus and attribute names must begin with a lowercase letter, and may be followed by an arbitrary long sequence of lowercase letters or digits.

By default, the tools expect the registry directory to be `/corpora/c1/registry`. Since at your site this directory most probably does not exist, the default value can be overridden by the environment variable `CORPUS_REGISTRY`. Please do not add a slash at the end of the value of this variable. We suggest that either each user of the tools sets this environment variable in his or her `.tcshrc` or `.cshrc` shell initialization file in his/her home directory or that it is set in of the global shell initialization files, which usually reside in `/etc` and are only writeable by the system administrator or the superuser. Please ask your system administrator for further help in case you shouldn't know where or how to set the variable. Additionally, almost all tools of the toolbox take the `-r` command line option to specify a non-default registry directory.

## 4.2 The contents of a registry file

In the registry file, all attributes of a corpus are declared. Additionally, some "global" variables are set.

A comment begins with a hash mark (`#`), everything up to the end of the line is not read. A registry file may contain empty lines.

The format of a registry file is:

*<Header information and global variables>*
*<Attribute definitions>*

This order has to be kept in all registry files. In the attribute definition section, the attributes may be declared in any order.

### 4.2.1 The header

The header consists of 4 declarations of values, each of which is preceded by the field name. The field names (keywords) are all uppercase.

The field names are

- a short (one-line) description of the corpus (keyword `NAME`). The field value is a string enclosed in double quotes;
- a unique identifier (keyword `ID`). The field value is a symbol. Usually, the field value should be the same as the file name of the registry file;
- optionally, the "home directory" of the corpus (keyword `HOME`). The field value is a path (not enclosed in double quotes);

- optionally, the path of the "info file" of the corpus (keyword `INFO`). This text file should contain a description of the corpus, its annotations, perhaps administrative information, etc. It is also a good idea to include a description of the tagset of the part-of-speech annotation there, if the corpus is tagged.

If the `HOME` field is missing, you have to specify the path for each attribute, so it is more convenient to define this field when all corpus-related data files are kept in a single directory.

If the `INFO` field is missing, no corpus information can be displayed in CQP or XKWIC (in CQP, this is done with the `info` command, in XKWIC, this file is displayed when the Info button is selected in the corpus list).

After the header, the set of corpus attributes (annotations) is declared. The different types of annotations are

- positional attributes (section 4.2.2);
- structural attributes (section 4.2.3);
- mapping tables (section 4.2.4);
- ngram tables (section 4.2.5);
- alignment information (section 4.2.6);
- dynamic attributes (section 4.2.7);

As said before, all annotations may be declared in almost any order (but see the notes in the case of mapping tables and bigram tables).

### 4.2.2 Positional attributes

A positional attribute is encoded as a set of seven files, which have been described above in section 3.2. These files are called the *components* of a positional attribute. A registry file defines, for each component of a positional attribute, the file name in which the component is stored. It is not necessary to manually set the names of all components since there are default rules how to compute undefined component file names from the defined ones. In fact, we suggest to rely on the default names which `encode` assigns to the files. Then, you do not have to declare component paths at all.

The declaration of a positional attribute looks as follows:

`ATTRIBUTE` *Name OptBody*

where *Name* is the identifier for the attribute (like word, pos, lemma, . . . ). The *OptBody* is optional and is only needed when you have to define non-default file names.

When you use the body, it can be one of the following:

- a definition of the component paths, *CompPathSpec*;

- or the declaration that the attribute is found on a remote host, *RemoteSpec*;
- or a path which overwrites the `HOME` field of the corpus and says that all components are stored at a different place, *AltPath*.

The *RemoteSpec* currently is not supported, since in the actual distribution, access to remotely stored corpora is disabled. When the *AltPath* declaration is used, it declares a path different of the corpus path for this special attribute.

**The component path specification**

The component path specification, *CompPathSpec*, must be enclosed in braces { ...}. Within these braces, a sequence of component name/path specification pairs is listed. Each component name may only occur once:

{
*ComponentID PathSpec*
...
}

One component is "virtual" (`DIR`) and doesn't describe filenames but rather denotes the "home directory" of the attribute. Two other virtual components are `ANAME`, the name of the attribute just being defined, and `APATH`, which is the "home directory" of the corpus, which defaults to the value of the value of the `HOME` field of the header (if present).

The *PathSpec* is a standard path, in which *Macros* may be used. Such a macro is started with a dollar sign `$` and directly followed by a component name. For example, the macro `$ANAME` represents the value of the attribute name. Macro values may be used in path specifications. For example,

```
$APATH/$ANAME.f
```

stands for the concatenation of the value of the `APATH` variable (usually the corpus home directory), followed by a slash, followed by the value of the `ANAME` variable, and then followed by `.f`.

In general, it is possible to refer to the value of any component by prefixing its component identifier with a dollar sign (`$`). Thus, when a component value is defined, it is possible to use the values of previously defined components in the definition. It is an error when you refer to a component value which is not yet defined.

The following table lists the components, the component identifiers and the default value or macro through which the (default) value is computed:

| Component | Component ID | Default Rule/Value |
|---|---|---|
| Directory | `DIR` | `$APATH` |
| Corpus | `CORPUS` | `$DIR/$ANAME.corpus` |
| Reversed Corpus | `REVCORP` | `$CORPUS.rev` |
| RevCorpusIdx | `REVCIDX` | `$CORPUS.rdx` |
| CorpusFreqs | `FREQS` | `$CORPUS.cnt` |
| Lexicon | `LEXICON` | `$DIR/$ANAME.lexicon` |
| Lexicon Index | `LEXIDX` | `$LEXICON.idx` |
| Lexicon Sortindex | `LEXSRT` | `$LEXICON.srt` |

This table also shows the component path values when there is no component path specification at all. The `APATH` field defaults to the `HOME` directory of the corpus (the root directory, `/`, is assumed if this `HOME` specification is missing in the header). The name of the attribute being defined, `ANAME`, is always known. Then, the other components get their path values by the rules listed in the table.

You can, of course, set all component path values as you like, but it is strongly recommended to rely on the default values.

**Note** that the `word` attribute always must be defined, and that an attribute with the name `word` *must* be present in every corpus declaraion (registry file).

Let's look at a small example. The following registry file only defines the positional attribute `word`:

```
NAME "Penn Treebank"
ID     up
HOME /corpora/kwic/up

ATTRIBUTE word
```

Due to the default rules, the paths of the `word` attribute components have the following values:

| Component | Value |
|---|---|
| Directory | `$APATH = /corpora/kwic/up` |
| Corpus | `$DIR/$ANAME.corpus = /corpora/kwic/up/word.corpus` |
| Reversed Corpus | `$CORPUS.rev = /corpora/kwic/up/word.corpus.rev` |
| RevCorpusIdx | `$CORPUS.rdx = /corpora/kwic/up/word.corpus.rdx` |
| CorpusFreqs | `$CORPUS.cnt = /corpora/kwic/up/word.corpus.cnt` |
| Lexicon | `$DIR/$ANAME.lexicon = /corpora/kwic/up/word.lexicon` |
| Lexicon Index | `$LEXICON.idx = /corpora/kwic/up/word.lexicon.idx` |
| Lexicon Sortindex | `$LEXICON.srt = /corpora/kwic/up/word.lexicon.srt` |

Please note that no checking is done with respect to file name collisions. If you use the same component path for different components (or for components of different attributes of different corpora etc.), the system will crash certainly, and also data damage may result.

The following example shows a declaration of a corpus with the "old" naming conventions:

```
NAME "Getaggter BZK (``Neues Deutschland'')"
ID     bzk
HOME /corpora/kwic/bzk-tagged
```

```
ATTRIBUTE word
{ CORPUS $APATH/corpus
  LEXICON $APATH/lexicon
}

ATTRIBUTE pos
{ CORPUS $APATH/corpus.pos
  LEXICON $APATH/pos.lexicon
}
```

Here, the paths of the `word` attribute components have the following values:

| Component | Value |
|---|---|
| Directory | `/corpora/kwic/bzk` |
| Corpus | `/corpora/kwic/bzk/corpus` |
| Reversed Corpus | `/corpora/kwic/bzk/corpus.rev` |
| RevCorpusIdx | `/corpora/kwic/bzk/corpus.rdx` |
| CorpusFreqs | `/corpora/kwic/bzk/corpus.cnt` |
| Lexicon | `/corpora/kwic/bzk/lexicon` |
| Lexicon Index | `/corpora/kwic/bzk/lexicon.idx` |
| Lexicon Sortindex | `/corpora/kwic/bzk/lexicon.srt` |

The paths of the `pos` attribute components have the following values:

| Component | Value |
|---|---|
| Directory | `/corpora/kwic/bzk` |
| Corpus | `/corpora/kwic/bzk/corpus.pos` |
| Reversed Corpus | `/corpora/kwic/bzk/corpus.pos.rev` |
| RevCorpusIdx | `/corpora/kwic/bzk/corpus.pos.rdx` |
| CorpusFreqs | `/corpora/kwic/bzk/corpus.pos.cnt` |
| Lexicon | `/corpora/kwic/bzk/pos.lexicon` |
| Lexicon Index | `/corpora/kwic/bzk/pos.lexicon.idx` |
| Lexicon Sortindex | `/corpora/kwic/bzk/pos.lexicon.srt` |

### The alternate path specification

In the alternate attribute path specification, *AltPath*, you "overwrite" the default `APATH` value. The component path values are computed by the default rules. With the *AltPath* specification, you "move" the whole set of attribute components to another directory, while keeping the standard name conventions.

Example:

```
NAME "Wall Street Journal, very large"
ID wsj
HOME /corpora/kwic/wsj

ATTRIBUTE word
ATTRIBUTE pos /var/space/kwic/wsj.pos
```

Here, the `word` attribute is on `/corpora/kwic/wsj`, but the set of components which belong to the `pos` attribute are stored on another file system, perhaps due to limitations of disk space.

**The remote specification**

*is currently disabled, so we do not have to write anything about it here...*

### 4.2.3 Structural attributes

Structural attributes also have components, but their values cannot be changed in the registry file. So, the declaration of a structural attribute is very simple:

`STRUCTURE` *Name OptStorageSpec*

`STRUCTURE` is the keyword, and *Name* is the name of the structure being defined. Optionally, this declaration may be followed by a storage specification, *OptStorageSpec*:

- this can either be a path, which "moves" the attribute data to another directory like the *AltPath* specification for positional attributes (the default is to store the data in the corpus `HOME` directory);
- or a remote declaration, which is currently disabled and therefore doesn't need to be covered here.

Normally, the declaration of a structural attribute simply is a sequence of the keyword `STRUCTURE`, followed by the name of the structure:

```
...
STRUCTURE s
STRUCTURE p
STRUCTURE article
STRUCTURE np
```

Structural attributes are stored in only one file, and the name of this file is derived by the rule `$APATH/$ANAME.rng` (where `$APATH` defaults to the value of `HOME`).

### 4.2.4 Mapping tables

The declaration of a mapping table looks as follows:

`MAPTABLE` *SourceName TargetName OptStorageSpec*

The *OptStorageSpec* is the same as in the case of structural attributes above, and has the same meaning.

The keyword `MAPTABLE` introduces the declaration of a mapping table. Then, the names of two positional attributes follow. These positional attributes must already be declared (so the declaration is not order-independent in this case). Currently, mapping tables are unidirectional, that is, you have to compute and declare mapping tables in both "directions" if you need them (for example, from pos to word as well as from word to pos).
Example declaration:

```
...
MAPTABLE word pos
MAPTABLE pos word
```

### 4.2.5 ngram tables

The declaration of a mapping table looks as follows:

`NGRAM` *PAName n OptStorageSpec*

The *OptStorageSpec* is the same as in the case of structural attributes above, and has the same meaning.

The keyword `NGRAM` introduces the declaration of a ngram table. Then, the name of a positional attribute follows, which already must be declared earlier in the registry file. The *PAName* is then followed by the "dimension" of the table, $n$. Currently, only bigram tables are supported, so that this value $n$ always must be 2.

Example declaration:

```
...
NGRAM word 2
NGRAM pos 2
```

### 4.2.6 Alignment attributes

With an alignment declaration, it is expressed that the corpus where this attribute is being declared, is aligned to another corpus which also has its registry file. The declaration is as follows:

`ALIGNED` *CorpusName OptStorageSpec*

The *OptStorageSpec* is the same as in the case of structural attributes above, and has the same meaning.

The *CorpusName* must be the name of an existing (registered) corpus.

Alignent tables are unidirectional. So, if the other aligned corpus also is aligned to the corpus currently being defined, the alignment in the other direction must be declared in the registry file of the other corpus.

Example declaration:

```
...
ALIGNED hansard-f
...
```

### 4.2.7 Dynamic attributes

Dynamic attributes are declared like functions: they have a name, an argument list, and a return value type. Additionally, it must be declared how the value is computed from the external knowledge source.

The declaration of a dynamic attribute looks as follows:

`DYNAMIC` *Name* `(` *ArgList* `)` `:` *RVType ShellCall*

The *Name* is the name of the dynamic attribute, following the standard naming conventions. After the name, the argument list is written in parentheses. The argument list is a sequence of comma-separated type identifiers. Currently, the type identifiers `STRING` (for character string arguments), `INT` (for integer type arguments) and `POS` (for corpus-position arguments) are supported. The return value type, *RVType*, also must be one of these type identifiers.

The *ShellCall* is a string, enclosed in double quotes. It defines how the return value is computed by a call to an external tool. The value computed by the external tool (which is expected to occur on the `stdout` of the external tool after its termination) is coerced to the return value type, whether this makes sense or not. So it is your task to make sure that the external tool computes information which can be coerced to the return value type.

In the *ShellCall*, you refer to the arguments of the function with `$1` for the first argument, `$2` for the second, and so on. It is an error when argument numbers are used which are higher than the number of arguments declared. It is not an error, though, when declared arguments are not used in the *ShellCall*.

Example:

```
DYNAMIC wndist(STRING,STRING,STRING):INT "/usr/local/bin/wnreq -s '$1' '$2' '$3'"
```

Here, the dynamic attribute `wndist` takes three `STRING`-type arguments and returns an `INT`-type value. The value is computed by calling the external program `wnreq` with some parameters and the (actual) arguments glued into the shell call.

When the value of a dynamic attribute is requested, the actual parameters of the function (which must agree with the declared argument types) are glued into the *ShellCall* at the positions indicated by the argument references ($\$n$). Then, this shell call is evaluated (via the *pipe* mechanism of Unix). It can yield to errors when arguments are passed which contain characters which are interpreted by the shell. These should either be excluded from passing through constraints in the query, or you have to design your shell call carefully (for example, by surrounding the variable references with single quotes, although this only partially helps). In a future release of the external knowledge source interface, perhaps we will support argument passing to the `stdin` of the external tool, so that shell evaluation does not take place any more.

The pipe to the external program is not kept open (we will fix that perhaps in a later release), so that the external tool is invoked for each value request. Two things are important:

- first, the number of dynamic calls must be minimized – use external information in queries carefully! Batch querying should also be considered;
- the startup time of the external tool should be minimized. It should only load as little data as necessary, and the time necessary to compute the requested information should be small. This can best be achieved by designing a client-server architecture, so that the server only starts once, and small clients (like `wnreq` in the example above) do not have to load large amounts of data.

Please remember that the external tool interface as well as dynamic attributes in general are experimental tools, and are not optimized towards efficiency.

## 4.3 Registration of remote corpora

| Remote corpora are currently disabled. So just skip the crap in this section. |
|---|

When corpora are stored remotely and have to be accessed via the network, they are declared differently than local corpora. For each remote corpus, a registry file similar to a "standard" registry file has to be created. This file declares that a corpus (or a positional attribute) is stored remotely. It must only contain the fields `NAME`, `ID` and `REMOTE`. `NAME` and `ID` have the same meaning like described above and are optional. The value of the `REMOTE` "component" is the name of the remote host which shall be connected when the corpus is accessed. No other component values must appear in the registry file for remote corpora. Here is an example for a full and valid remote declaration:

```
REMOTE maple.gardeners.edu
```

## 4.4 A last example

We hope that the format of the registry file is simple enough to understand it without too many explanations after you are fairly familiar with it. So just have a look at this last example.

```
NAME "Hansard-Corpus (English Part)"
ID      hansard-e
HOME /corpora/kwic/hansard-e
INFO    /corpora/kwic/hansard-e/.info

ATTRIBUTE word
{
  CORPUS  $APATH/corpus
  LEXICON $APATH/lexicon
}

ALIGNED hansard-f

STRUCTURE s
STRUCTURE np

ATTRIBUTE pos
ATTRIBUTE lemma

MAPTABLE word pos
MAPTABLE pos word

NGRAM word 2
NGRAM pos 2

DYNAMIC wndist(STRING,STRING,STRING):INT
 "/corpora/bin/wnreq -s '$1' '$2' '$3'"
```

## 4.5 Steps to follow

Now, what are the steps to follow if you have to register a new corpus?

- First, try to estimate the space requirements of the encoded corpus and find a place for it in your file system. Consider splitting of attributes or single attribute components if you do not find enough place on a single file system;
- make a new directory for the data of the new corpus. If there is place enough, try to put all data in this directory. Then, you will only have to set the `HOME` field of the registry file, and all other file names are then computed by the default rules. Remember that `encode`, the first step during corpus preparation, puts all data files in a single directory, so at least for this step there must be enough space on the disk.
- If you do not find enough space, you have to split the (positional) attributes between several file systems. Then, you will have to encode the set of positional attributes in several steps (by using the `-p` option for encode and only encoding one column of the input file in a single run of `encode`).
- Run `encode`, and let it write all corpus data in the new directory, if there is enough place. You may also use `/tmp` as an intermediate place to hold the data, and then `mv`ing the data files to their proper destinations.
- Register the new corpus: give it a new, unique name and create the registry file. Declare all annotations which have been produced by the run of `encode`. Remember that annotations can be added (or removed) at any time.
- If the corpus is registered, run `describe-corpus` with the name of the new corpus. This will show you whether the registry file is syntactically correct, which attributes are declared, where they will be stored, and what their status is. Check this list carefully.
- After all positional attributes have been `encode`d (either in a single step or in several steps), run `makeall` once, which will create all components of all positional attributes which have not yet been produced by `encode`.
- create and register additional attributes (mapping tables, bigram tables, alignment data, ...) with the respective utilities or by other means;
- You may check your corpus again with `describe-corpus`.
- Check the file permissions of all files produced by `encode` and `makeall`. Data files and the registry file should have the permission 444, and the directories (the registry directory and the data directory) should be readable (security and access control is covered in chapter 7).
- Start CQP or XKWIC, and check whether the new corpus appears in the list of available corpora.

Good luck!

# Chapter 5

# Remote access – client and server setup

| |
|---|
| In the current distribution, the corpus data server is not included, so that the IMS corpus toolbox does not support remote corpus data access at the moment. It was too slow anyway... |

This chapter explains how corpora are prepared to allow remote access. To "export" a corpus, it must be stored locally and be accessible by local users (that is, it must be declared in the registry). Basically, on each machine where corpora are stored which are to be exported to remote users, a server process must run which waits for corpus data requests from "outside" and serves these requests, after checking whether the remote user is authorized to access the respective corpus. It should be clear that remote access to corpora is the most vulnerable point with respect to copyright and access control issues. We therefore suggest that remote access is only granted for either "free" resources or when you have fully understood how to control remote access.

Remote connections can be built only if a server is running on the host on which a corpus is stored. Almost all tools (but neither `encode` nor `makeall`) have built-in remote access capabilities without the need for special tools or setups. Another access criterion is whether the user trying to access a corpus remotely is allowed to access the corpus at all. The following section 5.1 describes how to grant access to corpora, and section 5.2 describes how to run the corpus data server.

## 5.1 The `.rat` and `.ratlog` files

Remote access to corpora which are stored physically (either on local disks or on NFS-mounted disks) on the same machine the server is running on is controlled by a file called `.rat` ("remote access table") in the registry directory. This file holds an arbitrary number of lines, each being a pair of regular expressions based on the POSIX EGREP syntax[1]. The

[1] A description of this syntax can be found, for examle, in the documentation of the FSF/GNU regex package.

first regular expression in each line describes symbolic corpus names as they occur in the registry, the second regular expression describes who has access to the positional attributes described in the first expression.

When a corpus is to be accessed remotely, the server goes through all lines in the remote access table and checks whether the name of the corpus matches the first expression of the line. If so, it is checked whether the name of the user at the remote host (`user@host`) is matched by the second regular expression. If so, access is granted and no more lines in the table are checked. If not, the next line is tried. If no line is matched, access is denied.

The `.rat` file could, for example, look as follows:

```
treebank.* (joe|mary|chris)@(rose|tulip)\.gardeners\.edu
treebank.* jack@.*
up.* (joe|jack)@.*\.gardeners\.edu
.* corpus_adm@maple\.gardeners\.edu
```

The first line grants access to all positional attributes defined for the corpus `treebank` to Joe, Mary and Chris when they try to access the corpus from one of the hosts `rose.gardeners.edu` or `tulip.gardeners.edu`. Note that the dot, if not prefixed by a slash, matches every character and has to be escaped with a slash when a literal dot is expected. The second line grants access to the same positional attributes for Jack, no matter from which host he tries to access the corpus. Such a line should `not` occur in your `.rat` file, since there are many Jacks out there and you surely don't want that all of them probably get access to your corpus data. You should always try to specify the users and hosts from which access is to be granted as precisely as possible. The third line grants access to all `up` attributes for Joe and Jack, no matter from which host in the `gardeners.edu` domain they are connecting. The fourth line, also a little bit too general to my taste, grants access to all corpora for the user `corpus_adm@maple.gardeners.edu`.

Again, a "#" in the first column of the `.rat` file indicates a comment which extends up to the end of the line.

When a corpus has other positional attributes beyond the "word" attribute, it is important to allow remote access to all positional attributes assigned to the corpus (if this access should be possible over the network). This is achieved through the `.*` at the end of the first expression of each line which describes the positional attributes for which access should be granted.

The `.ratlog` file – also residing in the registry directory – logs all requests for remote access to corpora and whether a connection request was granted or not. Entries in this file look, for example, as follows:

```
CDS on maple at Thu Jan 20 16:47:27 1994
  login request from joe@tulip.gardeners.edu for corpus treebank (granted)

CDS on maple at Thu Jan 20 16:52:03 1994
  login request from bill@tulip.gardeners.edu for corpus treebank (denied)
```

## 5.2 How to start the corpus data server

Starting the server is quite simple: you only have to start the program `cds` (corpus data server) as a background process. There are, however, some points to be respected:

- only one `cds` process must run on a host;
- the `cds` process must be started by the same person who owns the `.rat` and the `.ratlog` file. The file `.rat` must be present as described in the previous section. If the `.ratlog` file doesn't exist when `cds` is started, it can be created with the `touch` command (although `cds` tries to create the file in case it should not yet be present).

Remote access to corpora is slow. We only implemented it for testing purposes, so don't expect it to work as fast as the access to locally stored corpora.

# Chapter 6

# Utilities and debugging tools

Additionally to the `encode` and `makeall` utilities introduced in chapter 3, there are some other utilities for the construction of the various attributes or the display of the information captured by these attributes. These tools are described in this section.

Unless otherwise indicated, for each of the tools mentioned in this chapter Unix manual pages have been written and should be available at your site.

| This chapter is "under construction", so please refer to the manual pages of the tools. |
|---|

## 6.1 Decoding of corpus and attribute information

### 6.1.1 Decoding of corpus information: `decode`

`decode` decodes (that is, prints encoded attribute values of) a registered corpus encoded with `encode` and `makeall` and prints the data textually on stdout. The user can select the attributes which are printed, as well as the start and end corpus positions. Alternatively, corpus positions may be piped into `decode` in order to print the values at these positions.

The attribute values are separated by a tabulator character and are preceded by the attribute name. The order in which the attribute values are printed is the same as the order of the corresponding command line arguments. At least one attribute must be specified with one of the options.

### 6.1.2 Decoding of word lists: `lexdecode`

`lexdecode` prints the values of a positional attribute of a registered corpus encoded with `encode` and `makeall` and prints the attribute values textually on stdout. If no positional attribute name is given via the `-P` option, the values of the standard positional attribute `word` are printed. Additionally, information about the absolute frequency of the values in `corpus` can be printed and/or the values can be printed in lexically ascending order.

## 6.2 Creation and Decoding of Bigram Tables

### 6.2.1 Creation of bigram tables: `gen-bigrams`

`gen-bigrams` computes bigram tables for a positional attribute of a corpus in a certain window.

`gen-bigrams` can use two different algorithms to compute the table: the sequential method (which can be selected through the `-S` option and is the default) sequentially shifts the corpus words and increments the counts, whereas the `reversed method` (selectable with the `-R` option) works via the reversed index.

When a bias is given, only bigram cells with a higher count than this bias are stored. When a minimal frequency is given, bigrams are only computed for those words which have at least the frequency `mfreq`. In this case, the computation is always done via the reversed corpus.

### 6.2.2 Decoding of bigram tables: `decode-bigrams`

`decode-bigrams` prints the contents of the bigram table associated with a positional attribute of a corpus in the window size. (which currently always is 2, but has to be given as an argument).

By default, the table is printed in the internal form, but with the `-t` option, a more readable, tabular output is produced. Then, the window width and height can be altered with the appropriate parameters.

## 6.3 Creation and Decoding of Mapping Tables

### 6.3.1 Creation of mapping tables: `gen-mapping-table`

`gen-mapping-table` computes mapping tables from the positional attribute `sourcePA` to the positional attribute `targetPA` of `corpus` The tables are direction-specific – if you want to have mapping tables in the other direction, you must create them, too.

`gen-mapping-table` can use three different algorithms to compute the table: the `standard method` (which can be selected through the `-S` option and is the default) allocates a large table and runs sequentially through the corpus and increments the counts. The tree computation (selected with the `-T` option) internally uses a tree-like structure, thus speeding up search. Third, the direct method (selected with the `-D` option) allocates a huge table with space for all possible cells, it can only be used for small tables (less than 10MB size), but should be the fastest method. The table size is computed by multiplying the number of values of the two attributes, times 4 (space for one integer).

### 6.3.2 Decoding of mapping tables: `decode-mapping-table`

`decode-mapping-table` prints the contents of the mapping table associated with the source positional attribute `sourcePA` and the target positional attribute `targetPA` of the corpus `corpus`

By default, the table is printed in the internal form, but with the `-t` option, a more readable, tabular output is produced. By default, the whole table is printed, but by using the `-p` option, `decode-mapping-table` reads the sets of values of the source attribute from stdin. Alternatively, a regular expression `pattern` can be given as the last argument. Then, the mappings are only displayed for those source values which match the pattern.

## 6.4 General utilities

### 6.4.1 Comparing word lists and corpora: `check-coverage`

`check-coverage` is a program which reads a list of words either from stdin (default) or from the file specified with the `-f` option. `check-coverage` then, for each word, looks in the list of corpora specified in the `corpus-names` list (maximum is 32 corpus names) whether the word occurs in the `word` attribute of these corpora. If not, the word is printed to stdout. If the word occurs in one of the corpora and if the `-s` option ("print success") is given, matches are also written to stdout.

### 6.4.2 Converting internal integers to readable numbers: `itoa`

`itoa` reads, from stdin or from each of the named files, a sequence of integers in the internal (machine-dependent) data format (4 byte integers) and writes each number textually to stdout, one number per line.

### 6.4.3 Converting readable numbers to internal integers: `atoi`

`atoi` reads, from stdin or from each of the named files, a sequence of numbers represented as digit sequences. The input must be in a one-number-per-line format and must only contain digits and newlines. The numbers (not the individual digits, of course) are then written to stdout as a sequence of 4-byte ints, in the internal (machine-dependent) data format. The function is similar to the atoi(3) function of the C library.

# Chapter 7

# Access control and security issues

This chapter describes how to control access to the corpus data. Access control is an important issue when corpora with restricted access are provided, for example the TIPSTER corpora available from the Linguistic Data Consortium (LDC).

## 7.1 Controlling local access to corpora

Access to corpus data currently can only be controlled by restricting the readability of the registry directory, the registry files in the registry directory and the directory the corpus data is stored in by setting the user and group IDs and the access permissions of the files and directories.

The easiest way to control access is by properly setting the read permissions of a registry file. If someone cannot read a registry file, he or she cannot access the corpus by way of its symbolic name. However, independent of the readability of the registry file, a user can figure out where the components of a corpus are stored and – with some knowledge of the internal corpus representation or with the help of the utilities mentioned in the previous chapter 6 – can reconstruct the corpus text from the encoded data unless the components are read-protected. We therefore recommend that the access restrictions of the directory the encoded data is stored in (user, group and r/w permissions) are the same as the access restrictions of the registry file. When a corpus is more or less "public" in the sense that its use is either free or restricted to your institution, access control is probably not necessary.

Further, we strongly recommend that

- the registry directory is owned by the corpus administrator;
- the registry directory is readable by everyone but writeable only for the corpus administrator (mode 755);
- the registry files in the registry directory are *only* writeable by the corpus administrator, and readable exactly by those users which may access the corpus;

- the directories the data of a positional attribute is stored in ("data directories") are writeable only by the corpus administrator;
- the data directories have the same group ID, owner ID and read permissions as the registry file which describes the attribue (plus the necessary "exec" bits);
- the data files (components) have mode 444 (are not writeable by anyone). For changes or updates, only the corpus administrator may change the access restrictions of a file or a directory.

Additionally, the read/exec permissions of tools other than the query tools (XKWIC, CQP, `print-aligned`) may be restricted to the corpus administrator. The utility `lexdecode`, however, should be executable by all corpus users since it produces useful frequency information which doesn't allow the reconstruction of the corpus text proper. We strongly recommend, however, that execution permission for the corpus data server (`cds`) is restricted to only the corpus administrator. None of the programs should be setuid.

Since the decoding utilities permit to textually decode a corpus with all its information, care should be taken that unauthorized users (guests with temporary accounts, for example) cannot use these tools. One idea, for example, is to set the setgid bit of the query tools (CQP, XKWIC) to the group id under which the corpus data, the registry files etc. are stored (the top directory suffices). The corpus data and the registry files (or the directories in which they are stored), then, should be readable only for members of this "corpus group", but not readable for members of other groups (`o-rX`). The decoding utilities must *not* be setgid for the corpus group. By this strategy, the corpus data can be accessed via the query tools, but not via the decoding utilities if the user does not belong to the group which has access to the corpus registry files by the normal group permissions.

## 7.2 Controlling remote access to corpora

As mentioned in chapter 5 above, remote access is a vulnerable point with respect to access control. With some knowledge about the internals of the tools, it would be possible for almost everyone to fake his or her identity and to gain illegal access to your corpora. Although this is not a trivial task, you should keep in mind that it is at least possible. The only way to prevent this is not to export corpora at all, that is, not to run a corpus data server (and to prevent other users to start it as well). Additionally, it is good policy to check the `.ratlog` file frequently in order to gain an overview which corpora are accessed by whom.

The `.rat` and the `.ratlog` files should be owned by the corpus administrator and have access mode 600, that is, readable and writeable only by the owner (i.e., the corpus administrator).

Remember that the corpus data server is usually run by the corpus administrator. Therefore, the server process has access to *all* registry files and to all positional attributes defined for all corpora. Access to corpora from "outside" is therefore only controlled by checking whether the corpus/user pair is contained in the remote access table (`.rat`).

Therefore, the entries in the `.rat` file must be carefully designed and no one but the corpus administrator should be able to read or write the `.rat` file. You should have a fair knowledge of regular expressions and take care that the expressions are not "overgenerating". If you are not sure about which user names or attributes are matched by your regular expressions, we suggest that you best enumerate all attribute/user pairs with their full names and not to use "wildcards" (.), the kleene star (*) or the plus construct (+).

And, lastly, don't broadcast that you permit access to your corpora too widely or to people who aren't in the list of intended users. Just to be on the safe side ...

# Appendix A

# Hardware and operating system requirements

The tools have been developed and tested on the following development platform:

- Compiler: gcc V2.5.7 on SunOS Release 4.1.3
- Window System: X Window System (tm), Version 11, Release 5 (X11R5)
- Widget set: OSF/Motif (tm) V1.2
- Hardware: Sun-4M/Sparcstation 3 (2 CPUs, 64MB memory, 50MHz)

We currently only deliver binary files for this platform. No other systems are supported. In order to run the tools, at least 32MB of memory are recommended. One CPU is of course sufficient.

# Appendix B

# Reused software packages and copyright notices

In the implementation of our system, we made use of the following software packages. We thank the providers of the software and include their disclaimers and copyright notices.

## B.1 The regular expression matcher by Henry Spencer

# Bibliography

[Christ, 1993] Oliver Christ. *The Xkwic User Manual*. Institut für maschinelle Sprachverarbeitung, Universität Stuttgart, 1993.

[Christ, 1994] Oliver Christ. 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)*, pages 23–32, Budapest, Hungary, 1994. CMP-LG archive id 9408005.

[Schulze and Christ, 1994] Bruno M. Schulze and Oliver Christ. *The CQP Users's Manual*. Institut für maschinelle Sprachverarbeitung, Universität Stuttgart, Version 1.0d, May 1994. (Revised October 1994).

[Schulze, 1994] Bruno M. Schulze. Entwurf und Implementierung eines Anfragesystems für Textcorpora. Diplomarbeit Nr. 1059, Universität Stuttgart, Institut für maschinelle Sprachverarbeitung (IMS) and Institut für Informatik, January 1994. (In German).