The IMS Open Corpus Workbench (CWB)
CQP Interface and Query Language Manual

— CWB Version 3.5 —

Stephanie Evert & The CWB Development Team
http://cwb.sourceforge.net/

July 2022

Contents

1 Introduction

1.1 The IMS Open Corpus Workbench (CWB)

History and framework

- Tool development
  - 1993 – 1996: Project on Text Corpora and Exploration Tools
    (financed by the *Land Baden-Württemberg*)
  - 1998 – 2004: Continued in-house development
    (partly financed by various research and industrial projects)
- Related projects and applications at the IMS
  - 1994 – 1998: EAGLES project (EU programme LRE/LE)
    (morphosyntactic annotation, part-of-speech tagset, annotation tools)
  - 1994 – 1996: DECIDE[1] project (EU programme MLAP-93)
    (extraction of collocation candidates, macro processor `mp`)
  - 1996 – 1999: Construction of a subcategorization lexicon for German
    (PhD thesis Eckle-Kohler, financed by the *Land Baden-Württemberg*)
  - Since 1996: Various commercial and research applications
    (terminology extraction, dictionary updates)
  - 1999 – 2000: DOT project (Databank Overheidsterminologie)
  - 1999 – 2003: Implementation of YAC chunk parser for German (PhD Kermes)
  - 2001 – 2003: Transferbereich 32 (financed by the DFG)
- Development as an open software project
  - 2005: Code released under GNU GPL by IMS, making CWB henceforth an open, public collaborative enterprise
  - 2001 – 2010: Work on first stable open version 3.0, released 2010
  - 2010 – 2022: Overlapping work on versions 3.1 (added Windows support), 3.2 (added Unicode support), 3.4 (misc. fixes and enhancements), and 3.5 new stable version)
  - v3.5 will be the final iteration of the present CWB; v4 will be a major rewrite
- Some external applications of the IMS Corpus Workbench
  (see `http://cwb.sourceforge.net/demos.php` for a longer list)
  - AC/DC project at the Linguateca centre (SINTEF, Oslo, Norway)
    (on-line access to a 180 M word corpus of Portuguese newspaper text)
    `http://www.linguateca.pt/ACDC/`
  - CorpusEye (user-friendly CQP) in the VISL project (SDU, Denmark)
    (on-line access to annotated corpora in various languages)
    `http://corp.hum.sdu.dk/`
  - SSLMIT Dev Online services (SSLMIT, University of Bologna, Italy)
    (on-line access to 380 M words of Italian newspaper text and other corpora)
    `http://sslmitdev-online.sslmit.unibo.it/corpora/corpora.php` ⟪*site no longer online*⟫ **TODO**

  - CucWeb project (UPF, Barcelona, Spain)
    (Google-style access to 208 million words of text from Catalan Web pages)
    http://ramsesii.upf.es/cucweb/ ⟪*site no longer online*⟫ **TODO**
  - BNCweb (CQP edition)
    (Web interface to the British National Corpus, ported from SARA to CQP)
    http://corpora.lancs.ac.uk/BNCweb/

Technical aspects

- CWB uses a bespoke token-based format for corpus storage:
  - binary encoding ⇒ fast access
  - full index ⇒ fast look-up of word forms and annotations
  - specialised data compression algorithms
  - corpus size: up to 2.1 billion words
  - text data and annotations cannot be modified after encoding
    (but it is possible to add new annotations or overwrite existing ones)
  - early versions assumed Latin-1 text encoding, later versions support multiple 8-bit character sets as well as UTF-8 for Unicode
- Typical compression ratios for a 100 million word corpus:
  - uncompressed text: ≈ 1 GByte (without index & annotations)
  - uncompressed CWB attributes: ≈ 790 MBytes (ratio: 1.3)
  - word forms & lexical attributes: ≈ 360 MBytes (ratio: 2.8)
  - categorical attributes (e.g. POS tags): ≈ 120 MBytes (ratio: 8.5)
  - binary attributes (yes/no): ≈ 50 MBytes (ratio: 20.5)
- Supported operating systems:
  - Linux
  - Mac OS
  - Microsoft Windows (64-bit)
  - SUN Solaris
  - Source code should compile on most recent Unix platforms (*BSD, Cygwin... etc.)
  - Corpus data format is platform-independent and compatible with all releases since 2001

Components of the CWB

- tools for encoding, indexing, compression, decoding, and frequency distributions
- global "registry" holds information about corpora (name, attributes, data path)
- corpus query processor (CQP):
  - fast corpus search (regular expression syntax)
  - use in interactive or batch mode
  - results displayed in terminal window
- CWB/Perl interface for post-processing, scripting and web interfaces
- CQPweb: a browser-based graphical interface to CWB/CQP, with extended analysis tools

[1] **D**esiging and evaluating **E**xtraction Tools for **C**ollocations **i**n **D**ictionarie**s** and Corpora

1.2 The CWB corpus data model

The following steps illustrate the transformation of textual data with some XML markup into the CWB data format.

1. **Formatted text** (as displayed on-screen or printed)

   An easy example. Another *very* easy example. **O**nly the **ea**siest ex**a**mples!

2. **Text with XML markup** (at the level of texts, words or characters)

   ```
   <text id=42 lang="English"> <s>An easy example.</s><s> Another <i>very</i> easy
   example.</s> <s><b>O</b>nly the <b>ea</b>siest ex<b>a</b>mples!</s> </text>
   ```

3. **Tokenised text** (character-level markup has to be removed)

   ```
   <text id=42 lang="English"> <s> An easy example . </s> <s> Another very
   easy example . </s> <s> Only the easiest examples ! </s> </text>
   ```

4. **Text with linguistic annotations** (annotations are added at token level)

   ```
   <text id=42 lang="English"> <s> An/DET/a easy/ADJ/easy example/NN/example
   ./PUN/. </s> <s> Another/DET/another very/ADV/very easy/ADJ/easy
   example/NN/example ./PUN/. </s> <s> Only/ADV/only the/DET/the
   easiest/ADJ/easy examples/NN/example !/PUN/! </s> </text>
   ```

5. **Text encoded as CWB corpus** (tabular format, similar to relational database)

   A schematic representation of the encoded corpus is shown in Figure 1. Each token (together with its annotations) corresponds to a row in the tabular format. The row numbers, starting from 0, uniquely identify each token and are referred to as *corpus positions*.

   Each (token-level) annotation layer corresponds to a column in the table, called a *positional attribute* or *p-attribute* (note that the original word forms are also treated as an attribute with the special name `word`). Annotations are always interpreted as character strings, which are collected in a separate lexicon for each positional attribute. The CWB data format uses lexicon `ID`s for compact storage and fast access.

   Matching pairs of XML start and end tags are encoded as token regions, identified by the corpus positions of the first token (immediately following the start tag) and the last token (immediately preceding the end tag) of the region. (Note how the corpus position of an XML tag in Figure 1 is identical to that of the following or preceding token, respecitvely.) Elements of the same name (e.g. `<s>...</s>` or `<text>...</text>`) are collected and referred to as a *structural attribute* or *s-attribute*. The corresponding regions must be *non-overlapping* and *non-recursive*. Different s-attributes are completely independent in the CWB: a hierarchical nesting of the XML elements is neither required nor can it be guaranteed.

   Key-value pairs in XML start tags can be stored as an annotation of the corresponding s-attribute region. All key-value pairs are treated as a single character string, which has to be "parsed" by a CQP query that needs access to individual values. In the recommended encoding procedure, an additional s-attribute (named *element_key*) is automatically created for each key and is directly annotated with the corresponding value (cf. `<text_id>` and `<text_lang>` in Figure 1).

6. **Recursive XML markup** (can be automatically renamed)

   Since s-attributes are non-recursive, XML markup such as

   ```
   <np>the man <pp>with <np>the telescope</np></pp> </np>
   ```


is not allowed in a CWB corpus (the embedded <np> region will automatically be dropped).[2] In the recommended encoding procedure, embedded regions (up to a pre-defined level of embedding) are automatically renamed by adding digits to the element name:

```
<np>the man <pp>with <np1>the telescope</np1></pp> </np>
```

corpus position	word form	ID	part of speech	ID	lemma	ID
(0)	<text> value = "id=42 lang="English""					
(0)	<text_id> value = "42"					
(0)	<text_lang> value = "English"					
(0)	<s>					
0	An	0	DET	0	a	0
1	easy	1	ADJ	1	easy	1
2	example	2	NN	2	example	2
3	.	3	PUN	3	.	3
(3)	</s>					
(4)	<s>					
4	Another	4	DET	0	another	4
5	very	5	ADV	4	very	5
6	easy	1	ADJ	1	easy	1
7	example	2	NN	2	example	2
8	.	3	PUN	3	.	3
(8)	</s>					
(9)	<s>					
9	Only	6	ADV	4	only	6
10	the	7	DET	0	the	7
11	easiest	8	ADJ	1	easy	1
12	examples	9	NN	2	example	2
13	!	10	PUN	3	!	8
(13)	</s>					
(13)	</text_lang>					
(13)	</text_id>					
(13)	</text>					

Figure 1: Sample text encoded as a CWB corpus.

[2] Recall that only the nesting of a <np> region within a larger <np> region constitues recursion in the CWB data model. The nesting of <pp> within <np> (and vice versa) is unproblematic, since these regions are encoded in two independent s-attributes (named pp and np).

1.3 Tutorial corpora referenced in this manual

Pre-encoded versions of the DICKENS and GERMAN-LAW corpora are distributed from the CWB website, http://cwb.sourceforge.net/download.php#corpora.

(Also available: a tool for encoding the *British National Corpus 1994*, http://cwb.sourceforge.net/download.php#import.)

English corpus: `DICKENS`

- a collection of novels by Charles Dickens
- ca. 3.4 million tokens
- derived from Etext editions (Project Gutenberg)
- document-structure markup added semi-automatically
- part-of-speech tagging and lemmatisation with TreeTagger
- recursive noun and prepositional phrases from Gramotron parser

German corpus: `GERMAN-LAW`

- a collection of freely available German law texts
- ca. 816,000 tokens
- part-of-speech tagging with TreeTagger
- morphosyntactic information and lemmatisation from IMSLex morphology
- partial syntactic analysis with YAC chunker

See Appendix A.3 for a detailed description of the token-level annotations and structural markup of the two tutorial corpora (positional and structural attributes).

2 Basic CQP features

2.1 Getting started

- start CQP by typing

  ```
  $ cqp -e
  ```

  in a shell window (the `$` indicates a shell prompt)

- `-e` flag activates command-line editing features[3]
- optional `-C` flag activates colour highlighting (experimental)
- every CQP command must be terminated with a semicolon (`;`)
- when command-line editing is activated, CQP will automatically add a semicolon at the end of each input line if necessary; explicit `;` characters are only necessary to separate multiple commands on a single line in this mode
- change the registry directory (where CQP will look for available corpora)

  ```
  > set Registry "/some/path/to/a/directory";
  ```

- list available corpora (in the current registry)

  ```
  > show corpora;
  ```

- get information about corpus (including corpus size in tokens)

  ```
  > info DICKENS;
  ```

  displays information file associated with the corpus, whose contents will vary (it is not created automatically)

- activate corpus for subsequent queries (use `TAB` key for name completion)

  ```
  [no corpus]> DICKENS;
  DICKENS>
  ```

  in the following examples, the CQP command prompt is indicated by a `>` character

- list attributes of activated corpus ("context descriptor")

  ```
  > show cd;
  ```

2.2 Searching for words

- search single word form (single or double quotes are required: `'...'` or `"..."`)

  ```
  > "interesting";
  ```

  → shows all occurrences of interesting

- the specified word is interpreted as a regular expression

  ```
  > "interest(s|(ed|ing)(ly)?)?";
  ```

  → *interest, interests, interested, interesting, interestedly, interestingly*

- see Appendix A.1 for an introduction to the regular expression syntax

[3] The `-e` mode is not enabled by default for reasons of backward compatibility. When command-line editing is active, multi-line commands are not allowed, even when the input is read from a pipe.

- the regular expression "flavour" used by CQP is *Perl Compatible Regular Expressions*, usually known as **PCRE**; lots of documentation and examples can be found on the WWW
- note that special characters have to be "escaped" with backslash (\)

  `"?"` fails; `"\?"` → *?*; `"."` → *. , ! ? a b c ...*; `"\$\."` → $.

  "critical" characters are: `. ? * + | ( ) [ ] { } ^ $`
- **CWB 3.0**: LaTeX-style escape sequences `\"`, `\'`, `\‘` and `\^`, followed by an appropriate ASCII letter, are used to represent characters with diacritics when they cannot be entered directly

  `"B\"ar"` → *Bär*; `"d\'ej\‘a"` → *déjà*

  NB: this feature is deprecated; it works only for the Latin-1 encoding and cannot be deactivated
- **CWB 3.0**: additional special escape sequences:

  `\"s` → ß; `\,c` → ç; `\,C` → Ç; `\~n` → ñ; `\~N` → Ñ;

  version 3.0.3 introduces two-digit hex escapes for inserting arbitrary byte values:

  `\xDF` → ß in a Latin1-encoded corpus; `\xC3\x9F` → ß in a UTF-8-encoded corpus
- **CWB 3.5**: full support for PCRE regular expressions, including two- and four-digit hex escapes
- use flags `%c` and `%d` after a regex to ignore case / diacritics

```
DICKENS> "interesting" %c;
GERMAN-LAW> "wahrung" %cd;
```

- if you need to match a word form containing single or double quotes (e.g. *'em* or *12"-screen*), there are two possibilities:
  - if the string does not contain both single and double quotes, simply pick an appropriate quote character: `"'em"` vs. `'12"-screen'`
  - otherwise, escape the quote characters with a backslash: `'\'em'` and `"12\"-screen"`
  - alternatively, double every occurrence of the quote character inside the string; our two examples could also be matched with the queries `'''em'` and `"12""-screen"`

2.3 Display options

- KWIC display ("key word in context")

```
15921: ry moment an <interesting> case of spo
17747:  appeared to <interest> the Spirit
20189: ge , with an <interest> he had neve
24026: rgetting the <interest> he had in w
35161: require . My <interest> in it , is
35490: require . My <interest> in it was s
35903: ken a lively <interest> in me sever
43031:  been deeply <interested> , for I rem
```

- if query results do not fit on screen, they will be displayed one page at a time using a pager program
- press `SPC` (space bar) to see next page, `RET` (return) for next line, and `q` to return to CQP

- some pagers support `b` or the backspace key to go to the previous page, as well as the use of the cursor keys, `PgUp`, `PgDn`, `Home`, and `End`.
- at the command prompt, use cursor keys to edit input (← and →, `Del`, backspace key) and repeat previous commands (↑ and ↓)
- change context size

```
> set Context 20;          (20 characters)
> set Context 5 words;     (5 tokens)
> set Context s;           (entire sentence)
> set Context 3 s;         (same, plus 2 sentences each on left and right)
```

- type "`cat;`" to redisplay matches
- display current context settings

```
> set Context;
```

- left and right context can be set independently

```
> set LeftContext 20;
> set RightContext s;
```

- all option names are case-insensitive; most options have abbreviations:
  `c` for `Context`, `lc` for `LeftContext`, `rc` for `RightContext`
  (shown in square brackets when current value is displayed)
- show/hide annotations

```
> show +pos +lemma;     (show)
> show -pos -lemma;     (hide)
```

- summary of selected display options (and available attributes):

```
> show cd;
```

- structural attributes are shown as XML tags

```
> show +s +np_h;
```

- hide annotations of XML tags

```
> set ShowTagAttributes off;
```

- hide corpus position

```
> show -cpos;
```

- show annotation of region(s) containing match

```
> set PrintStructures "np_h";
> set PrintStructures "novel_title, chapter_num";
> set PrintStructures "";
```

2.4 Useful options

- enter `set;` to display list of options (abbreviations shown in brackets)
- `set` *<option>*`;`
  to show current value of <*option*>

- `set ProgressBar (on|off);`
  to show progress of query execution
- `set Timing (on|off);`
  to show execution times of queries and some other commands
- `set PrintMode (ascii|sgml|html|latex);`
  to set output format for KWIC display and frequency distributions
- `set PrintOptions (hdr|nohdr|num|nonum|...);`
  to turn various formatting options on (`hdr`, `num`, ...) or off (`nohdr`, `nonum`, ...)
  type `set PrintOptions;` to display the current option settings
  useful options: `hdr` (display header), `num` (show line numbers), `tbl` (format as table in HTML and LaTeX modes), `bdr` (table with border lines)
- `set (LD|RD)` *<string>*`;`
  change left/right delimiter in KWIC display from the default < and > markers
- `set ShowTagAttributes (on|off);`
  to display key-value pairs in XML start tags (if annotated in the corpus)
- create `.cqprc` file in your home directory with your favourite settings
  (contains arbitrary CQP commands that will be read and executed during startup)
- for a persistent command history, add the lines

  ```
  set HistoryFile "<home>/.cqphistory";
  set WriteHistory yes;
  ```

  to your `.cqprc` file.

  NB: (i) history requires CQP to be run with `-e` option; (ii) the size of the history file is *not* limited automatically by CQP.
- `set AutoShow off;`
  no automatic KWIC display of query results (display must be explicitly invoked with `cat`)
- `set Optimize on;`
  enable experimental optimisations (sometimes included in beta versions)

2.5 Accessing token-level annotations

- specify p-attribute/value pairs (square brackets are required)

  ```
  > [pos = "JJ"];     (find adjectives)
  > [lemma = "go"];
  ```
- `"interesting"` is an abbreviation for `[word = "interesting"]`
- the implicit attribute in the abbreviated form can be changed with the `DefaultNonbrackAttr` option; for instance, enter

  ```
  > set DefaultNonbrackAttr lemma;
  ```

  to search for lemmatised words instead of surface forms
- the `%c` and `%d` flags can be used with any attribute/value pair

  ```
  > [lemma = "pole" %c];
  ```

- values are interpreted as regular expressions, which the annotation string must match; add `%l` flag to match literally:

  ```
  > [word = "?" %l];
  ```

- `!=` operator: annotation *must not* match regular expression

  `[pos != "N.*"]` → everything except nouns

- `[]` matches any token (⇒ *matchall* pattern)

- see Appendix A.2 for a list of useful part-of-speech tags and regular expressions

- or explore tagging with the `/codist[]` macro (more on macros in Sections 6.4 and 6.5):

  ```
  > /codist["whose", pos];
  ```

  → finds all occurrences of the word *whose* and computes frequency distribution of the part-of-speech tags assigned to it

- use a similar macro to find inflected forms of *go*:

  ```
  > /codist[lemma, "go", word];
  ```

  → finds all tokens whose lemma attribute has the value `go` and computes frequency distribution of the corresponding word forms

- abort query evaluation with `Ctrl-C`
  (does not always work, press twice to exit CQP immediately)

2.6 Combinations of attribute constraints: Boolean expressions

- operators: `&` (and), `|` (or), `!` (not), `->` (implication, cf. Section 4.1)

  ```
  > [lemma="under.+" & pos="V.*"];
  ```
  → verb with prefix *under...*

- attribute/attribute-pairs: compare attributes as strings

  ```
  > [lemma="under.+" & (word!=lemma];
  ```
  → inflected forms of lemmas with prefix *under...*

- complex expressions: parentheses (round brackets) group conditions and control order of evaluation

  ```
  > [(lemma="go") & !(word="went"%c | word="gone"%c)];
  ```

- any expression in square brackets (`[...]`) describes a single token (⇒ *pattern*)

2.7 Sequences of words: token-level regular expressions

- a sequence of words or patterns matches any corresponding sequence in the corpus

  ```
  > "on" "and "on|off";
  > "in" "any|every" [pos = "NN"];
  ```

- modelling of complex word sequences with regular expressions over *patterns* (i.e. tokens): every `[...]` expression is treated like a single character (or, more precisely, a character set) in a conventional regular expression

- token-level regular expressions use a subset of the POSIX syntax

- repetition operators:
  `?` (0 or 1), `*` (0 or more), `+` (1 or more), `{`n`}` (exactly n), `{`n`,`m`}` ($n \ldots m$)
- grouping with parentheses: `(...)`
- disjunction operator: `|` (separates alternatives)
- parentheses delimit scope of disjunction: `(` alt_1 `|` alt_2 `|` ... `)`
- Figure 2 shows simple queries matching prepositional phrases (PPs) in English and German. The query strings are spread over multiple lines to improve readability, but each one has to be entered on a single line in an interactive CQP session.

```
DICKENS>
  [pos = "IN"]                  "after"
  [pos = "DT"]?                 "a"
  (
    [pos = "RB"]?               "pretty"
    [pos = "JJ.*"]              "long"
  ) *
  [pos = "N.*"]+ ;              "pause"

GERMAN-LAW>
  (
    [pos = "APPR"] [pos = "ART"] "nach dem"
    |
    [pos = "APPRART"]           "zum"
  )
  (
    [pos = "ADJD|ADV"] ?        "wirklich"
    [pos = "ADJA"]              "ersten"
  )*
  [pos ="NN"];                  "Mal"
```

Figure 2: Simple queries matching PPs in English and German.

2.8 Example: finding "nearby" words

- insert optional matchall patterns between words

  ```
  > "right" []? "left";
  ```

- repeated matchall for longer distances

  ```
  > "no" "sooner" []* "than";
  ```

- use the range operator `{,}` to restrict number of intervening tokens

  ```
  > "as" []{1,3} "as";
  ```

- avoid crossing sentence boundaries by adding `within s` to the query

  ```
  > "no" "sooner" []* "than" within s;
  ```

- order-independent search

```
> "left" "to" "right"
          | "right" "to" "left";
```

2.9 Sorting and counting

- sort matches alphabetically (and re-displays query results)

```
> [pos = "IN"] "any|every" [pos = "NN"];
> sort by word;
```

- add `%c` and `%d` flags to ignore case and/or diacritics when sorting

```
> sort by word %cd;
```

- matches can be sorted by any positional attribute; just type

```
> sort;
```

  without an attribute name to restore the natural ordering by corpus position

- query results can also be sorted in random order (to avoid looking only at matches from the first part of a corpus when beginning to page through query results):

```
> sort randomize;
```

  more on random sorting and an important application in Section 3.6

- select descending order with `desc(ending)`, or sort matches by suffix with `reverse`; note the ordering when the two options are combined:

```
> sort by word descending reverse;
```

- sort by right or left context (especially useful for keyword searches)

```
> "interesting";
> sort by word %cd on matchend[1] .. matchend[42]; (right context)
> sort by word %cd on match[-1] .. match[-42]; (left context, by words)
> sort by word %cd on match[-42] .. match[-1] reverse; (same by characters)
```

- compute frequency distribution of matching word sequences (or annotations)

```
> count by word;
> count by lemma;
```

- `%c` and `%d` flags normalise case and/or diacritics before counting

```
> count by word %cd;
```

- set frequency threshold with `cut` option

```
> count by lemma cut 10;
```

- `descending` option affects ordering of word sequences with the same frequency; use `reverse` for some amusing effects (note that these keywords go before the `cut` option)

- see Sections 3.2 and 3.3 for an explanation of the syntax used in these examples and more information about the `sort` and `count` commands

2.10 CQP scripts

- CQP commands do not have to be entered interactively at the prompt, they can also be collected in a text file (a *CQP script*)
- every command in a CQP script *must* be terminated with ; (whereas the terminator is optional in interactive CQP with command-line editing enabled)
- consider a text file `script.txt` containing the lines below

```
## use comment lines to structure and document the script
DICKENS; # activate corpus
set Context 30; # 30 chars of left/right context
[lemma = "dog" & pos = "NN.*"]; # find noun DOG
sort by word %c on matchend[1] .. matchend[42]; # sort by right context
```

- you can execute this script from the command line; it will print out a concordance for the noun *dog* in corpus order, then sorted by right context

```
$ cqp -f script.txt
```

- new in CQP v3.4.22: CQP scripts can also be executed from an interactive session

```
> source "script.txt";
```

- always keep in mind that the script does not run in a localized environment; any changes made by the script will persist in the interactive sessions
- the `source` command can also be executed in a CQP script; this can be used to structure complex scripts into sub-modules in separate script files
- scripts do not accept arguments, but they can often be emulated with the help of CQP macros (see Sec. 6.4)

3 Working with query results

3.1 Named query results

- store query result in memory under specified name (should begin with capital letter)

  ```
  > Go = [lemma = "go"] "and" [];
  ```

  note that query results are *not* automatically displayed in this case

- list **named query results** (or **NQR** for short)

  ```
  > show named;
  ```

- the NQR specifiers shown in the output of this command are fully qualified with the CWB name of the corpus, e.g. `DICKENS:Go` for the query above. As a consequence, there can be multiple NQRs with the same name for different corpora.

- if NQR names are used without a prefixed corpus in CQP commands (as in the examples below), they are automatically prefixed with the currently activated corpus

- the result of the most recent (*last*) query is automatically stored in memory and named `Last`

- commands such as `cat`, `sort`, and `count` operate on `Last` by default

- `Last` is always temporary and will be overwritten when a new query is executed (or a `subset` command, see Section 3.5)

- display number of results

  ```
  > size Go;
  ```

- (full or partial) KWIC display

  ```
  > cat Go;
  > cat Go 5 9;    (6th – 10th match)
  ```

- sorting a named query result automatically re-displays the matches

  ```
  > sort Go by word %cd;
  ```

- the `count` command also sorts the named query on which it operates, that is,

  ```
  > count Go by lemma cut 5;
  ```

  implicitly executes the command `sort Go by lemma;`

- this has the advantage that identical word sequences now appear on adjacent lines in the KWIC display and can easily be printed with a single `cat` command; the respective line numbers are shown in square brackets at the end of each line in the frequency listing

  ```
  13      go and see  [#128-#140]
  10      go and sit  [#144-#153]
  9       go and do  [#29-#37]
  7       go and fetch  [#42-#48]
  7       go and look  [#87-#93]
  7       go and play  [#107-#113]
  ```

  to display occurrences of *go and see*, enter

  ```
  > cat Go 128 140;
  ```

- if a fully qualified NQR is used, a query result can be accessed even if its corpus isn't currently activated, so that

  ```
  > size DICKENS:Go;
  > count DICKENS:Go by lemma cut 5;
  ```

  will work regardless of the current corpus.

- Due to a long-standing bug in CQP, this feature should not be used with `cat` or any other command that generates KWIC output (such as `sort`). Doing so will corrupt the context descriptor, which holds information about all available attributes, those selected for printing, and the KWIC context size.

  ```
  > GERMAN-LAW;
  > show cd;
  > cat DICKENS:Time;
  > show cd;
  ```

- The context descriptor can only be repaired by temporarily activating a different corpus and then re-activating the desired corpus.

  ```
  > DICKENS;
  > GERMAN-LAW;
  ```

3.2 Saving data to disk

- named query results can be stored on disk in the `DataDirectory`

  ```
  > set DataDirectory ".";
  > DICKENS;
  ```

  NB: you need to re-activate your working corpus after setting the `DataDirctory` option

- save named query to disk (in a platform-dependent uncompressed binary format)

  ```
  > save Go;
  ```

- `md*` flags show whether a named query is loaded in memory (`m`), saved on disk (`d`), or has been modified from the version saved on disk (`*`)

  ```
  > show named;
  ```

- discard a named query result to free memory

  ```
  > discard Go;
  ```

- set `DataDirectory` to load previously-saved named queries from disk
  (after discarding, or in a new CQP session)

  ```
  > set DataDirectory ".";
  > show named;
  > cat Go;
  ```

  note that the actual data is only read into memory when the query results are accessed

- write KWIC output to text file (use `TAB` key for filename completion)

  ```
  > cat Go > "go.txt";
  ```

  use `set PrintOptions hdr;` to add header with information about the corpus and the query (previous CQP versions did this automatically)

- if the filename ends in `.gz` or `.bz2`, the file will automatically be compressed (this requires the respective command-line utilities `gzip` and `bzip2` to be available)[4]
- append to an existing file with `>>`; this also works for compressed files

```
> cat Go >> "go.txt";
```

- you can also write to a pipe[5] (this example saves only matches that occur in questions, i.e. sentences ending in *?*)

```
> set Context 1 s;
> cat Go > "| grep '\?$' > go2.txt";
```

- set `PrintMode` and `PrintOptions` for HTML output and other formats (still supported, but not recommended: see Section 2.4)
- frequency counts for matches can also be written to a text file

```
> count Go by lemma cut 5 > "go.cnt";
```

- new in CQP v3.4.14: `cat` can also be used to print an arbitrary string or redirect it to a file; within an entered string, escape sequences `\t` (TAB), `\r` (CR) and `\n` (LF) are interpreted, all other backslashes are passed through verbatim; note that the string is *not* automatically terminated with a newline

```
> cat "Just another\n\tCQP hacker.\n";
```

- the new functionality can be combined with output redirection, which is particularly convenient for adding header rows to tabular output files, e.g.

```
> cat "f\tmatch  [results]\n" > "go.cnt";
> count Go by lemma cut 5 >> "go.cnt";
```

3.3 Anchor points

- the result of a (complex) query is a list of token sequences of variable length (⇒ *matches*)
- each match is represented by two *anchor points*:
  `match` (corpus position of first token) and `matchend` (corpus position of last token)
- set an additional `target` anchor with `@` marker in query (prepended to a pattern)

```
> "in" @[pos="DT"] [lemma="case"];
```

  → shown in bold font in KWIC display

- only a single token can be marked as `target`; if multiple `@` markers are used (or if the marker is in the scope of a repetition operator such a `+`), only the earliest matching token[6] will be marked

```
> [pos="DT"] (@[pos="JJ.*"] ","?){2,} [pos="NNS?"];
```

- when `target`ed pattern is optional, check how many matches have target anchor set

```
> A = [pos="DT"] @[pos="JJ"]? [pos="NNS?"];
> size A;
> size A target;
```

[4] You can alternatively use 7-zip to handle these file formats, by setting the environment variable CWB_USE_7Z (CWB v3.4.35+). 7-zip is somewhat easier to install on Windows in particular than `gzip` and `bzip2` (from https://www.7-zip.org/), but note that you still need to make sure that the `7z` program is findable your environment's PATH.

[5] But see the notes on pipes in Sec. 3.3.

[6] Earliest here refers to corpus position, not to the position of the token pattern in the query string

- new in CQP v3.4.16: A second anchor position called `keyword` can also be set. The default notation is `@1`, but this can be changed with a user option (see Sec. 8.6 for details).

  ```
  > "in" @[pos="DT"] @1[pos="J.*"]? [lemma="case"];
  ```
  → `keyword` is underlined in KWIC display

- each token pattern in a query can only be marked with *one* of the two anchors

- anchor points allow a flexible specification of sort keys with the general form

  > `sort by` *attribute* `on` *start point* `..` *end point* `;`

  both *start point* and *end point* are specified as an anchor, plus an optional offset in square brackets; for instance, `match[-1]` refers to the token before the start of the match, `matchend` to the last token of the match, `matchend[1]` to the first token after the match, and `target[-2]` to a position two tokens after the `target` anchor

  NB: the `target` anchor should only be used in the sort key when it is always defined

- example: sort noun phrases by adjectives between determiner and noun

  ```
  > [pos="DT"] [pos="JJ"]{2,} [pos="NNS?"];
  > sort by word %cd on match[1] .. matchend[-1];
  ```

- if *end point* refers to a corpus position before *start point*, the tokens in the sort keys are compared from right to left; e.g. sort on the left context of the match *by token*:

  ```
  > sort by word %cd on match[-1] .. match[-42];
  ```

  whereas the `reverse` option sorts on the left context *by character*:

  ```
  > sort by word %cd on match[-42] .. match[-1] reverse;
  ```

- complex sort operations can sometimes be sped up by using an external helper program (on Unix, the standard `sort` tool)[7]

  ```
  > sort by word %cd;
  > set ExternalSort on;
  > sort by word %cd;
  > set ExternalSort off;
  ```

- the `count` command accepts the same specification for the strings to be counted

  ```
  > count by lemma on match[1] .. matchend[-1];
  ```

- display corpus positions of all anchor points in tabular format

  ```
  > A = "behind" @[pos="JJ"]? [pos="NNS?"];
  > dump A;
  > dump A 9 14;    (10th – 15th match)
  ```

  the four columns correspond to the `match`, `matchend`, `target` and `keyword` (see Section 3.7) anchors; a value of `-1` means that the anchor has not been set:

  ```
  1019887 1019888 -1      -1
  1924977 1924979 1924978 -1
  1986623 1986624 -1      -1
  2086708 2086710 2086709 -1
  2087618 2087619 -1      -1
  2122565 2122566 -1      -1
  ```

[7] External sorting may also allow language-specific sort order (*collation*) if supported by the system's `sort` command. To achieve this on Unix, set the `LC_COLLATE` or `LC_ALL` environment variable to an appropriate locale before running CQP. You should not use the `%c` and `%d` flags in this case.

note that any prior `sort` or `count` command affects the ordering of the rows (so that the n-th row corresponds to the n-th line in a KWIC display obtained with `cat`)

- the output of a `dump` command can be written (`>`) or appended (`>>`) to a file, if the first character of the filename is `|`, the ouput is sent to a pipe consisting of the command(s) that follow the `|`
- pipe commands can only use programs that are (a) available on your operating system, and (b) accessible via you environment's PATH (or, named with their full filesystem location); while Linux, Mac OS, WSL[8]etc. have the standard set of Unix tools, including `sort`/`gawk`/`uniq` used in the trick discussed below, Windows does not - unless you take special measures to install them; commands involving pipes to programs you don't have will, of course, fail
- use the following trick to display the distribution of match lengths in a named query result `A`:

```
> A = [pos="DT"] [pos="JJ.*"]* [pos="NNS?"];
> dump A > "| gawk '{print $2 - $1 + 1}' | sort -nr | uniq -c | less";
```

- see Section 7.2 for an opposite to the `dump` command, which may be useful for certain tasks such as locating a specific corpus position

3.4 Frequency distributions

- show frequency distribution of tokens (or their annotations) at anchor points

```
> group Go matchend pos;
```

set cutoff threshold with `cut` option to reduce size of frequency table

```
> NP = [pos="DT"] @[pos="JJ"]? [pos="NNS?"];
> group NP target lemma cut 50;
```

- add optional offset to anchor point, e.g. distribution of words preceding matches

```
> group NP match[-1] lemma cut 100;
```

- frequencies of token/annotation pairs (using different attributes or anchor points)

```
> group NP matchend word by target lemma;
> group Go matchend lemma by matchend pos;
```

Despite what the command syntax and output format suggest, results are sorted by pair frequencies (*not* grouped by the second item). The order of the two items in the output is opposite to the order in the `group` command.

- you can write the output of the `group` command to a text file (or pipe)

```
> group NP target lemma cut 10 > "adjectives.go";
```

(in CQP v.3.4.11 and newer, the file is automatically compressed if it ends in `.gz` or `.bz2`; see Sec. 3.1 above)

- new in CQP v3.4.9: use `group by` instead of `by` for nested frequency counts

```
> group Go matchend lemma group by matchend pos;
```

where an optional `cut` clause applies to the individual pairs

[8] *Windows Subsystem for Linux*; for purposes of running CWB, WSL is just another Unix.

- new in CQP v3.4.26: Compute document frequencies based on s-attribute regions rather than token frequencies by adding the `within` keyword (before `cut`). The example below counts the number of novels in which each distinct lemma occurs in the *go and X* construction rather than its overall frequency.

  ```
  > group Go matchend lemma within novel cut 3;
  ```

- Any items outside regions of the selected s-attribute are silently discarded in the frequency counts. The same happens for undefined anchor points in a simple grouping, because they cannot be assigned to any region. Notice that the top entry `(none)` is no longer present in the the paragraph-frequency count below.

  ```
  > group NP target lemma within p cut 50;
  ```

  The second example counts head nouns in chapter and novel titles, silently discarding all other occurrences. Keep in mind that repetitions within the same title will be counted only once; add a `within` constraint to the initial CQP query if you want a token frequency count within titles.

  ```
  > group NP matchend lemma within title cut 5;
  ```

- In the case of a `group ... by`, both elements must be contained in the same s-attribute region; otherwise the pair is silently discarded. It is valid for one of the anchors to be undefined, so the output of the commands below still includes `(none)` entries for NPs without adjective:

  ```
  > group NP target lemma group by matchend lemma within novel cut 10;
  > group NP matchend lemma by target lemma within novel cut 10;
  ```

- Computation of document frequencies is only possible if the s-attribute regions are traversed in corpus order by the query result. This will usually be the case and is guaranteed for anchors set in a CQP query with matching `within` constraint. However, a `set target` operation with a large search context can sometimes result in out-of-order anchors. In this case, the frequency count will abort with an error message.

  ```
  > set NP keyword nearest [pos="JJ.*"] within s;
  > group NP keyword lemma within np; # keyword anchors traverse NPs out of order
  ```

3.5 Set operations with named query results

- named queries can be copied, especially before destructive modification (see below)

  ```
  > B = A;
  > C = Last;
  ```

- compute subset of named query result by constraint on one of the anchor points

  ```
  > PP = [pos="IN"] [pos="JJ"]+ [pos="NNS?"];
  > group PP matchend lemma by match word;
  ```

  ```
  > PP1 = subset PP where match: "in";
  > PP2 = subset PP1 where matchend: [lemma = "time"];
  ```

  → `PP2` contains instances of *in ... time(s)*

- set operations on named query results

  ```
  > A = intersection B C;
  ```
  $A = B \cap C$

  ```
  > A = union B C;
  ```
  $A = B \cup C$

  ```
  > A = difference B C;
  ```
  $A = B \setminus C$

  `intersection` (or `inter`) yields matches common to `B` and `C`; `union` (or `join`) matches from either `B` or `C`; `difference` (or `diff`) matches from `B` that are not in `C`

- cut query result to first n matches

  `> cut A 50;` *(first 50 matches)*

  or select a range of matches (as with the restricted `cat` command)

  `> cut A 50 99;` *(51^{st} – 100^{th} match)*

  NB: `cut A 50;` is exactly the same as `cut A 0 49;`

- The modifier `cut` n can als be appended to a query:

  `> "time" cut 50;`

  The main purpose of this usage is to reduce memory consumption and processing time in Web interfaces and similar applications by stopping query execution early if a sufficient number of matches has been found. For internal reasons, this optimization cannot be applied to queries with alignment constraints (see Sec. 5.2); but the `cut` modifier still guarantees that only the first n matches will be returned.

3.6 Random subsets

- when there are a lot of matches, e.g.

  ```
  > A = "time";
  > size A;
  ```

  it is often desirable to look at a random selection to get a quick overview (rather than just seeing matches from the first part of the corpus); one possibility is to do a `sort randomize` and then go through the first few pages of random matches:

  `> sort A randomize;`

  however, this cannot be combined with other sort options such as alphabetical sorting on match or left/right context; it also doesn't speed up frequency lists, `set target` and other post-processing operations

- as an alternative to randomized ordering, the `reduce` command randomly selects a given number or proportion of matches, deleting all other matches from the named query; since this operation is destructive, it may be necessary to make a copy of the original query result first (see above)

  ```
  > reduce A to 10%;
  > size A;
  > sort A by word %cd on match .. matchend[42];
  ```

  ```
  > reduce A to 100;
  > size A;
  > sort A by word %cd on match .. matchend[42];
  ```

  this allows arbitrary further operations to be carried out on a representative sample rather than the full query result

- set random number generator seed before `reduce` for reproducible selection

  `> randomize 42;` *(use any positive integer as seed)*

- a second method for obtaining a random subset of a named query result is to sort the matches in random order, and then take the first n matches from the sorted query; the example below has the same effect as `reduce A to 100;` (though it will not select exactly the same matches)

  ```
  > sort A randomize;
  > cut A 100; (NB: this restores corpus order, as with the reduce command)
  ```

reproducible subsets can be obtained with a suitable `randomize` command before the `sort`; the main difference from the `reduce` command is that `cut` cannot be used to select a percentage of matches (i.e., you have to determine the number of matches in the desired subset yourself)

- the most important advantage of the second method is that it can produce *stable* and *incremental* random samples
- for a stable random ordering, specify a positive seed value directly in the sort command:

  ```
  > sort A randomize 42;
  ```

  different seeds give different, reproducible orderings; if you randomize a subset of `A` with the same seed value, the matches will appear exactly in the same order as in the randomized version of `A`:

  ```
  > A = "interesting" cut 20; (just for illustration)
  > B = A;
  > reduce B to 10; (an arbitrary subset of A)
  > sort A randomize 42;
  > sort B randomize 42;
  ```

- in order to build incremental random samples from a query result, sort it randomly (but with a fixed seed value to ensure reproducibility) and then take the first n matches as sample #1, the next n matches as sample #2, etc.; unlike two subsets generated with `reduce`, the first two samples are disjoint and together form a random sample of size $2n$:

  ```
  > A = "time";
  > sort A randomize 7;
  > Sample1 = A;
  > cut Sample1 0 99; (random sample of 100 matches)
  > Sample2 = A;
  > cut Sample2 100 199; (random sample of 100 matches)
  ```

  note that the `cut` removes the randomized ordering; you can reapply the stable randomization to achieve full correspondence to the randomized query result `A`:

  ```
  > sort Sample2 randomize 7;
  > cat Sample2;
  > cat A 100 199;
  ```

- stability of the randomization ensures that random samples are reproducible even after the initial query has been refined or spurious matches have been deleted manually

3.7 The `set target` command

- the additional `keyword` anchor can be set *after* query execution by searching for a token that matches a given *search pattern* (see Figure 3)
- example: find noun near adjective *modern*

  ```
  > A = [(pos="JJ") & (lemma="modern")];
  > set A keyword nearest [pos="NNS?"] within right 5 words from match;
  ```

- keyword should be underlined in KWIC display (may not work on some terminals)
- search starts from the given anchor point (excluding the anchored token itself), or from the opening and closing boundaries of the match if `match` is specified
- with `inclusive`, search includes the anchored token, or the entire match, respectively

```
set <named query>
    (keyword | target)          (anchor to set)
    (leftmost | rightmost |
     nearest | farthest)        (search strategy)
    [<pattern>]                 (search pattern)
    within
    (left | right)?             (search direction)
    <n> (words | s | ...)       (window)
    from (match | matchend | keyword | target)
    (inclusive)? ;              (include start token in search)
```

Figure 3: The `set target` command.

- `from match` is the default and can be omitted
- the `match` and `matchend` anchors can also be set, modifying the actual matches[9]
- Anchor positions can also be copied, possibly modifying the matching ranges. In line with the complex form of the `set target` command described above, the first anchor is the destination and the second the source:

  ```
  set A target match;
  set A matchend keyword;
  ```

- or they can be deleted from the named query result (`keyword` and `target` only, of course):

  ```
  set A keyword NULL;
  set A target NULL;
  ```

- an important use case is adjustments to the matching range if a query needs to include additional context as a filter. For example, this query attempts to identify noun phrases in object position (cf. Sec. 4.3), then uses `set` to cut off the context filter before the NP[10]:

  ```
  > NPobj = [pos="V.*"] [pos="RB"]* <np> @[] []* </np>;
  > set NPobj match target;
  > set NPobj target NULL;
  ```

- new in CQP v3.4.31: the undocument and somewhat inconsistent behaviour of the `copy` operation in the case of undefined or invalid anchor positions [11] has been consolidated and is described precisely below. The reimplementation also provides enhanced functionality: an offset can be added to shift anchors to the left or right.
- Offsets are convenient if a fixed number of extra tokens of context have been matched and need to be cut off, or to shift a target anchor that could not easily be set at the desired position. A typical example is an anchor on the last token of a group of alternatives `(...|...|...)`, which would have to be set in each branch of the group (and possibly multiple times if there are optional

[9] The `keyword` and `target` anchors are set to undefined (`-1`) when no match is found for the search pattern, while the `match` and `matchend` anchors retain their previous values. In this way, a `set match` or `set matchend` command may only modify some of the matches in a named query result.

[10] Keep in mind that you have to type `cat NPobj;` in order to display the result, because the implicit NQR `Last` is just a copy that will not have been modified.

[11] This issue arises if the `match` or `matchend` anchor is modified: What should the algorithm do if the source anchor is undefined? What if the update would create an invalid matching range with `matchend` < `match`? Up to v3.4.30, CQP would leave the match unmodified in the first case, but shorten it to a single token in the second case.

elements at the end of a branch). It is much easier to set the anchor on the following token (possibly with a zero-width marker `@[::]`, cf. Sec. 8.1), and then to shift it one token to the left afterwards:

```
> set A target target[-1];
```

- The new implementation performs a *conditional update* of the destination anchor by default. If the source anchor is undefined (i.e. `-1`) or if the offset puts it outside the valid corpus range, the destination remains unmodified.[12] The same happens if the `match` or `matchend` anchor is modified and would result in an invalid matching range (with `matchend` < `match`).

- the example below extends the match *elephant(s)* to start from the `keyword` anchor, but only if it is defined and not to the right of the match:

```
> Elephants = [lemma = "elephant"];
> set Elephants keyword leftmost [pos="JJ.*"] within 3 words;
> set Elephants match keyword;
```

- append an exclamation mark `!` for a *forced update*, in which an undefined (or out-of-range) source anchor always overwrites the destination; if the destination is `match` or `matchend`, the corresponding item is dropped from the query result. Retry the commands above with

```
> set Elephants match keyword !;
```

- as a consequence of these rules, creating a query result that consists of the single `target` token requires a sequence of three commands in order to work reliably:

```
> Elephants = [lemma = "elephant"];
> set Elephants target nearest [pos="JJ.*"] within 3 words;
> set Elephants match target;       # S1
> set Elephants matchend target;    # S2
> set Elephants match target !;     # S3
```

  Quiz: Can you explain why without looking up the solution in the footnote?[13]

[12] If the destination anchor is newly created by the command, it is initialised to undefined values.

[13] If `target` is to the left of the match, S1 extends the start of the match to `target`, then S2 sets `matchend` to the same position. If `target` is to the right of the match, S1 is a no-op (because it would cross over `match` with `matchend`). S2 then extends the end of the match to `target`, and S3 can set `match` to the same position. In either case, S3 deletes matches for which `target` is undefined. Display the query using `cat Elephants;` after each step for an illustration of this process.

4 Labels and structural attributes

4.1 Using labels

- patterns can be labelled (similar to the target marker `@`)

  ```
  > adj:[pos = "JJ.*"] ... ;
  ```

  the label `adj` then refers to the corresponding token (i.e. its corpus position)

- label references are usually evaluated within the *global constraint* introduced by `::`

  ```
  > adj:[pos = "ADJ."] :: adj < 500;
  ```
  → adjectives among the first 500 tokens

- annotations of the referenced token can be accessed as `adj.word`, `adj.lemma`, etc.

- labels are not part of the query result and must be used within the query expression (otherwise, CQP will abort with an error message)

- labels set to optional patterns may be undefined

  ```
  > [pos="DT"] a:[pos="JJ"]? [pos="NNS?"] :: a;
  ```
  → global constraint `a` is true iff match contains an adjective

- to avoid error messages, test whether a label is defined before accessing its attributes

  ```
  > [pos="DT"] a:[]? [pos="NNS?"] :: a -> a.pos="JJ";
  ```

  (`->` is the logical implication operator →, cf. Section 2.6)

- labels are used to specify additional constraints that are beyond the scope of ordinary regular expressions

  ```
  > a:[] "and" b:[] :: a.word = b.word;
  ```

- labels allow modelling of long-distance dependencies

  ```
  > a:[pos="PP"] []{0,5} b:[pos = "VB.*"]
       :: b.pos = "VBZ" -> a.lemma = "he|she|it";
  ```

  (this query ensures that the pronoun preceding a 3rd-person singular verb form is *he*, *she* or *it*; an additional constraint could exclude these pronouns for other verb forms)

- labels can be used within patterns as well

  ```
  > a:[] [pos = a.pos]{3};
  ```
  → sequences of four identical part-of-speech tags

- however, a label cannot be used within the pattern it refers to; use the special *this* label represented by a single underscore (`_`) instead to refer to the current corpus position

  `[_.pos = "NPS"]` ⟺ `[pos = "NPS"]`

- the *this* label can also be used to constrain tokens to a certain range of corpus positions without explicit labels, e.g.

  ```
  > [pos = "ADJ." & _ < 500];
  ```

  such constraints are not allowed in query-initial position, so queries such as `[_ >= 666];` and `[_ < 500 & pos = "ADJ."];` will be rejected as invalid

- new in CQP v3.4.17: as a special case, the pattern

  ```
  > [_ = 666];
  ```

  can be used to look up a known corpus position efficiently

- the built-in functions `distance()` and `distabs()` compute the (absolute) distance between 2 tokens (referenced by labels)

  ```
  > a:[pos="DT"] [pos="JJ"]* b:[pos="NNS?"] :: distabs(a,b) >= 5;
  ```
  → simple NPs containing 6 or more tokens

- the standard anchor points (`match`, `matchend`, and `target`) are also available as labels (with the same names)

  ```
  > [pos="DT"] [pos="JJ"]* [pos="NNS?"] :: distabs(match, matchend) >= 5;
  ```

- various other built-in functions have been added in recent versions of CQP and can be used with label references or directly with attribute values; see Sec. 8.3 for a complete list

- new in CQP v3.4.17: use function `strlen()` to filter by word length, e.g. to find particularly long words:

  ```
  > [word = ".*ment" & strlen(word) >= 16];
  ```

- NB: inequality comparisons (`>`, `>=`, `<`, `<=`) are only allowed for integers (corpus positions, string lengths, etc.), but not for strings and regular expressions; CQP versions before v3.4.17 used to silently accept and misinterpret such inequality comparisons

4.2 Structural attributes

- XML tags match start/end of s-attribute region (shown as XML tags in Figure 1)

  ```
  > <s> [pos = "VBG"];
  > [pos = "VBG"] [pos = "SENT"]? </s>;
  ```
  → present participle at start or end of sentence

- pairs of start/end tags enclose single region (if `StrictRegions` option is enabled)

  ```
  > <np> []* ([pos="JJ.*"] []*){3,} </np>;
  ```
  → NP containing at least 3 adverbs

  (when `StrictRegions` are switched off, XML tags match any region boundaries and may skip intervening boundaries as well as material outside the corresponding regions)

- `/region[]` macro matches entire region

  `/region[np];` ⟺ `<np> []* </np>;`

- different tags can be mixed

  ```
  > <s><np>[]*</np> []* <np>[]*</np></s>;
  ```
  → sentence that starts and ends with a noun phrase (NP)

- the name of a structural attribute (e.g. `np`) used within a pattern evaluates to *true* iff the corresponding token is contained in a region of this attribute (here, a `<np>` region)

  ```
  > [(pos = "NNS?") & !np];
  ```
  → noun that is *not* contained in a noun phrase (NP)

- built-in functions `lbound()` and `rbound()` test for start/end of a region

  ```
  > [(pos = "VBG") & lbound(s)];
  ```
  → present participle at start of sentence

- new in CQP v3.4.13: Built-in functions `lbound_of()` and `rbound_of()` return the corpus positions of the start/end of a region. Because of technical limitations, the anchor position has to be specified explicitly as a second argument, which will often be the *this* label:

  ```
  > [(word = "\d+") & (lbound_of(s, _) = lbound_of(chapter, _))];
  ```
  → a number in the first sentence of a chapter

  The same query could also be written with an explicit label or anchor reference in a global constraint (which is perhaps easier to read):

  ```
  > "\d+" :: lbound_of(s, match) = lbound_of(chapter, match);
  ```

  If the referenced position is not contained in a suitable s-attribute region, the functions return an undefined value, which evaluates to false in most contexts (in particular, all comparisons with this value will be false).

- The `lbound_of()` and `rbound_of()` functions are mainly used in connection with `distance()` or `distabs()`. For example, to find occurrences of the word *end* within the first 40 tokens of a chapter:

  ```
  > [word = "end"%c & distabs(_, lbound_of(chapter, _)) < 40];
  ```

- use `within` to restrict matches of a query to a single region

  ```
  > [pos="NN"] []* [pos="NN"] within np;
  ```
  → sequence of two singular nouns within the same NP

- most linguistic queries should include the restriction `within s` to avoid crossing sentence boundaries; note, however, that only a single `within` clause may be specified

- query matches can be expanded to containing regions of s-attributes

  ```
  > A = [pos="JJ.*"] ([]* [pos="JJ.*"]){2} within np;
  > B = A expand to np;
  ```

  one-sided expansion is selected with the optional `left` or `right` keyword

  ```
  > C = B expand left to s;
  ```

- the expansion can be combined with a query, following all other modifiers

  ```
  > [pos="JJ.*"] ([]* [pos="JJ.*"]){2} within np cut 20 expand to np;
  ```

4.3 Structural attributes and XML

- XML markup of NPs and PPs in the `DICKENS` corpus (cf. Appendix A.3)

```
<s len=9>
  <np h="it" len=1> It </np>
  is
  <np h="story" len=6> the story
    <pp h="of" len=4> of
       <np h="man" len=3> an old man </np>
    </pp>
  </np>
  .
</s>
```

- key-value pairs within XML start tags are accessible in CQP as additional s-attributes with annotated values (marked `[A]` in the `show cd;` listing): `s_len`, `np_h`, `np_len`, `pp_h`, `pp_len` (cf. Section 1.2)
- s-attribute values can be accessed through label references

  ```
  > <np> a:[] []* </np> :: a.np_h = "bank";
  ```
  → NPs with head lemma *bank*

  an equivalent, but shorter version:

  ```
  > /region[np,a] :: a.np_h="bank";
  ```

  or use the `match` anchor label automatically set to the first token of the match

  ```
  > <np> []* </np> :: match.np_h="bank";
  ```
- constraints on key-value pairs can also directly be tested in start tags, using the appropriate auto-generated s-attribute (make sure to use a matching end tag)

  ```
  > <np_h = "bank"> []* </np_h>;
  ```

  comparison operators `=` and `!=` are supported, together with the `%c` and `%d` flags;
  `=` is the default and may be omitted
- constraints on multiple key-value pairs require multiple start tags

  ```
  > <np_h="bank"><np_len="[1-6]"> []* </np_len></np_h>;
  ```

  (or access the value of `np_len` through a label reference)
- `<np>` and `<pp>` tags are usually shown without XML attribute values;
  they can be displayed explicitly as `<np_h>`, `<np_len>`, ... tags:

  ```
  > show +np +np_h +np_len;
  > cat;
  ```

  (other corpora may show XML attributes in start tags)
- use *this* label for direct access to s-attribute values within pattern

  ```
  > [(pos="NNS?") & (lemma = _.np_h)];
  ```

  (recall that `np_h` would merely return an integer value indicating whether the current token is contained in a `<np>` region, not the desired annotation string)
- typecast numbers to `int()` for numerical comparison

  ```
  > /region[np,a] :: int(a.np_len) > 30;
  ```
- NB: s-attribute annotations can *only* be accessed with label references:

  ```
  > [np_h="bank"];
  ```
  does not work!
- regions of structural attributes are non-recursive
  ⇒ embedded XML regions are renamed at time of indexing to `<np1>`, `<np2>`, ... `<pp1>`, `<pp2>`, ...
- embedding level must be explicitly specified in the query:

  ```
  > [pos="CC"] <np1> []* </np1>;
  ```

  will only find NPs contained in *exactly one* larger NP
  (use `show +np +np1 +np2;` to experiment)

- regions representing the attributes in XML start tags are renamed as well:
  ⇒ `<np_h1>`, `<np_h2>`, ..., `<pp_len1>`, `<pp_len2>`, ...

  ```
  > /region[np1, a] :: a.np_h1 = a.np_h  within np;
  ```

- CQP queries typically use *maximal* NP and PP regions (e.g. to model clauses)
- find *any* NP (regardless of embedding level):

  ```
  > (<np>|<np1>|<np2>) []* (</np2>|</np1>|</np>);
  ```

  CQP ensures that a matching pair of start and end tag is picked from the alternatives
- observe how results depend on matching strategy (see Section 6.1 for details)

  ```
  > set MatchingStrategy shortest;
  > set MatchingStrategy longest;
  > set MatchingStrategy standard;
  ```

  (re-run the previous query after each `set` and watch out for "duplicate" matches)
- when the query expression shown above is embedded in a longer query, the matching strategy usually has no influence
- annotations of a region at an arbitrary embedding level can only be accessed through constraints on key-value pairs in the start tags:

  ```
  > (<np_h "bank">|<np_h1 "bank">|<np_h2 "bank">) []*
        (</np_h2>|</np_h1>|</np_h>);
  ```

4.4 XML document structure

- XML document structure of `DICKENS`:

```
<novel title="A Tale of Two Cities">
  <tilepage> ... </titlepage>
  <book num=1>
    <chapter num=1 title="The Period">
       ...
    </chapter>
    ...
  </book>
  ...
</novel>
```

- use `set PrintStructures` command to display novel, chapter, ... for each match

  ```
  > set PrintStructures "novel_title, chapter_num";
  > A = [lemma = "ghost"];
  > cat A;
  ```

- find matches in a particular novel

  ```
  > B = [pos = "NP"] [pos = "NP"] ::
        match.novel_title = "David Copperfield";
  > group B matchend lemma by match lemma;
  ```

  (note that `<novel_title = "...">` cannot be used in this case because the XML start tag of the respective `<novel>` region will usually be far away from the match)
- frequency distributions can also be computed for s-attribute values

  ```
  > group A match novel_title;
  ```

4.5 Match selectors

- new in CQP v3.4.32: **match selectors** allow a subpart of query to be returned as the final match
- More sophisticated CQP queries often include additional material before and after the actual matching range of interest as "context filters". Consider e.g. an *it*-extraposition construction such as *it is the time that have changed*, which can be matched by the query

  ```
  > It = "it"%c [lemma="be"] [pos="DT"] @[pos="NNS?"] "that"%c [pos="V.*"];
  ```

- We may only be interested in the nouns that are emphasised in this way, but have to include the entire construction in the match to identify relevant contexts. In an interactive CQP session, we can use the target marker to obtain a frequency count with `group` (Sec. 3.4) or adjust the matching range with `set` (Sec. 3.7). However, this is not possible if CQP queries are executed via a Web interface such as CQPweb.
- A match selector is introduced by the keyword `show` and allows a subpart of the query to be extracted as the matching range, based on labels marking the first and last token of this range. In our example, this also frees the target marker to be set on the verb (which will no longer be part of the match).

  ```
  > "it"%c [lemma="be"] [pos="DT"] noun:[pos="NNS?"] "that"%c @[pos="V.*"]
    show noun .. noun;
  ```

- Specify `match` as the first label in order to leave the start of the match unmodified and/or `matchend` as the second label in order to leave the end unmodified. No other anchor names are allowed in the range specification. The trivial match selector `show match..matchend` has no effect, whereas `show matchend..matchend` is not accepted by the query parser.
- If one of the specified labels is undefined, the corresponding match will be discarded. The query

  ```
  > [pos="DT"] adj:[pos="JJ.*"]? [lemma="nail"] show adj .. matchend;
  ```

  will only return matches that include the optional adjective. Similarly, any invalid ranges (where the specified end token precedes the start token) will be discarded. The following query produces an empty result:

  ```
  > start:[pos="DT"] [pos="JJ.*"]? end:[lemma="nail"] show end .. start;
  ```

- Make sure that labels used in the match selector are always defined, especially in queries with disjunctions (each label must be set in all of the alternative branches). If you forget to set `stop:` in one of the branches below, it will effectively be erased from the query.

  ```
  > "from" (stop:[pos="PP"] | [pos="DT"] stop:[pos="NNS?"]) "to" show match..stop;
  ```

- Optionally, an offset can be added to the start and end position of the selector. This is convenient in order to shift the start and end of the matched by a fixed number of tokens, e.g.

  ```
  > "from" ([pos="PP"] | [pos="DT"] [pos="NNS?"]) "to" show match[1]..matchend[-1];
  ```

  It is also convenient to adjust a label that cannot easily be set on the desired position, especially on the last token of an s-attribute region or a group of alternatives. The first version of this query can be rewritten in a less error-prone way as

  ```
  > "from" ([pos="PP"] | [pos="DT"] [pos="NNS?"]) stop:"to" show match .. stop[-1];
  ```

  If the offset puts the start or end point outside the valid corpus range, the match is discarded.

- The `show` clause has to be written in a specific position: after a global constraint (`::`) and `within` clause, but before any alignment constraints (Sec. 5.2), `cut` limit or `expand` clause.

5 Working with aligned corpora

All examples in this section are based on EuroParl v3, a parallel corpus of debates of the European Parliament that can be downloaded in pre-indexed form from the CWB website.

5.1 Displaying aligned sentences

- CWB can encode information about sentence-level alignment between parallel corpora in its index. For each pair of source and target corpus, only a single alignment may be defined; the name of the corresponding alignment attribute (**a-attribute**) is a lowercase version of the CWB name of the target corpus.
- For example, the English component of the EuroParl corpus

  ```
  > EUROPARL-EN;
  ```

  is aligned to the French (`EUROPARL-FR`) and German (`EUROPARL-DE`) components *inter alia*.
- The available alignment attributes are listed as "Aligned Corpora:" in the output of `show cd;`.
- One or more alignments can be displayed in the KWIC output produced by `cat`. For example, in order to find out how the idiom *take the biscuit* can be expressed in Franch and German, we activate the corresponding a-attributes:

  ```
  > show +europarl-fr +europarl-de;
  > [lemma="take"] "the" "biscuit";
  ```

  the target languages are always printed in the same order as in the `show cd;` output; in this example, the German translation will be shown first, followed by the French translation.
- It is recommended to set the KWIC context for the source language to a full sentence

  ```
  > set Context 1 s;
  ```

  However, translations are always displayed as complete **alignment beads**, which can be confusing if multiple sentences in the source language are translated into a single target sentence (a 2:1 bead) or divided in a different way in the target language (a 2:2 bead). For example, the single hit of the query

  ```
  > "price-tag";
  ```

  is translated 1:1 into French, but combined with the previous sentence in the German translation (resulting in a 2:1 bead).
- An unofficial feature allows setting the KWIC context to an a-attribute, ensuring that a complete alignment bead (for the selected target corpus) is displayed.

  ```
  > set Context europarl-de;
  > cat;
  ```

  Keep in mind that this has been implemented as a special case: a-attributes cannot be used as context specifiers elsewhere (e.g. in a `within` clause).
- You will find that some sentences have no translation into the target language, e.g.

  ```
  > "cats" cut 6;
  ```

  Notice that the alignment KWIC context shows only the matching string without surrounding words in this case.

- In order to exclude matches outside alignment beads (i.e. without a translation), you can add a trivial alignment constraint to the query (see Sec. 5.2). The example below shows that out of 49 occurrences of *cats*, only 46 have a translation into French:

```
> AllCats = "cats";
> size AllCats;
> GoodCats = "cats" :EUROPARL-FR [];
> size GoodCats;
```

5.2 Querying aligned corpora

This section explains how alignment information can be used as a filter in CQP queries.

- As a first example, let us consider the word *nuclear power*, which can be translated into German as *Kernkraft*, *Kernenergie*, *Atomkraft* or *Atomenergie*.

```
> Nuke = "nuclear"%c "power"%c;
```

- Instead of manually perusing the translations of all 1417 hits, we can directly search for hits that contain one of the relevant words in the German translation.

```
> "nuclear"%c "power"%c :EUROPARL-DE [lemma = "Kernkraft"];
> "nuclear"%c "power"%c :EUROPARL-DE [lemma = "Kernenergie "];
```

etc.

An alignment constraint consists of the marker `:TARGET-CORPUS` followed by an arbitrary query expression. CQP will scan the region aligned to each match of the main query and keep only those for which a match to the alignment constraint is found.

- The alignment constraint must be specified after the main query (including the `within` clause), but before a `cut` statement (which applies to the filtered query results). Multiple alignment constraints can be chained, in which case, they must all be satisfied.

- Alignment constraints can be negated by placing `!` immediately after the marker. In this case, only those matches are kept for which the alignment constraint is *not* satisfied.

```
> Other = "nuclear"%c "power"%c :EUROPARL-DE ! "(Kern|Atom).*";
```

Can you figure out how *nuclear power* is translated in these examples?

- By chaining negated constraints, we can identify cases where the French translation is also different from the expected *nucléaire*:

```
> "nuclear"%c "power"%c :EUROPARL-DE! "(Kern|Atom).*" :EUROPARL-FR! "nucleaire.*" %cd;
```

- An alignment constraint can never be satisfied for a match that has no translation in the target corpus, regardless of whether it is negated or not. Therefore,

```
> "cats" :EUROPARL-FR ! [];
```

returns no results at all. If you need to find unaligned instances of *cats*, you can only do so in a two-step process. Using the NQR *AllCats* and `GoodCats` (with translation) from above:

```
> BadCats = diff AllCats GoodCats;
> size BadCats;
```

- Alignment constraints can only be added to regular CQP queries, not to MU queries (see Sec. 8.4).

5.3 "Translating" query results

- A named query result can be "translated" to an aligned corpus, which allows more flexible display of the aligned regions, access to metadata, etc. (new in CQP v3.4.9).
- Consider the following example:

```
> EUROPARL-DE;
> set Context 1 s;
> Zeit = [lemma = "Zeit"];
```

- The NQR `Zeit` now contains all occurrences of the German word for *time* in the German part of EuroParl. The following command "translates" the NQR to the English part of EuroParl, i.e. it replaces each match by the complete aligned region in the target corpus (as would be displayed with `show +europarl-en;`.

```
> Time = from Zeit to EUROPARL-EN;
```

- This creates a new NQR `EUROPARL-EN:Time` containing the aligned regions. You can now e.g. tabulate or count metadata:

```
> tabulate EUROPARL-EN:Time match text_date;
> group EUROPARL-EN:Time match text_date;
```

- The somewhat arcane syntax of the command avoids introduction of a new reserved keyword
  - while it looks similar to a corpus query or set operation, the assignment to a new NQR is mandatory (otherwise the parser won't accept the syntax)
  - note that the new NQR must be specified as a short name; the name of the target corpus is implied and added automatically with the assignment
- Some important details:
  - matching ranges that are not aligned to the target corpus are silently discarded; you cannot expect the new NQR to contain the same number of hits as the original NQR
  - if there are multiple matches in the same alignment bead, they will *not* be collapsed in the target corpus; i.e. the new NQR will contain several identical ranges
  - in order to collate source matches with the aligned regions, make sure to discard unaligned hits from the original NQR first:

    ```
    > Zeit = [lemma = "Zeit"] :EUROPARL-EN [];
    ```

    or post-hoc as a subquery filter

    ```
    > Zeit;
    > ZeitAligned = <match> [] :EUROPARL-EN [] !;
    ```

- Do not `cat` the translated query directly (`cat EUROPARL-EN:Time;`) without first activating the target corpus, as this would corrupt the context descriptor (see Sec. 3.1). The correct procedure is

```
> EUROPARL-EN;
> cat Time;
```

  You can now customize the KWIC display as desired.

- But it is safe to apply `dump`, `tabulate`, `group`, `count` and similar operations. Only commands that auto-print the NQR (including a bare *sort* or a set operation) will trigger the bug.
- The problem is mentioned in this section because users are most likely to be tempted to do this when working with a set of aligned corpora.

- As a second example, we will return to German translations of *nuclear power*.

```
> EUROPARL-DE;
> Other = from EUROPARL-EN:Other to EUROPARL-DE;
```

- We can now run a subquery on the aligned regions in the German part of EuroParl in order search for possible translations other than *Kern-* and *Atom-*. One possibility is that *nuclear power plant* has been translated into the acronym *AKW* (for *Atomkraftwerk*).

```
> Other;
> [lemma = "AKW"];
```

- Further translation candidates can be found by computing a frequency breakdown of all nouns in the aligned sentences:

```
> N = [pos = "N.*"];
> group N match word;
```

- We could have applied the same strategy to the NQR `Nuke` in order to determine the frequencies of different translation equivalents:

```
> Nuke = from EUROPARL-EN:Nuke to EUROPARL-DE;
> Nuke;
> TEs = "(Atom|Kern|AKW).*";
> group TEs match lemma;
```


6 Advanced CQP features

6.1 The matching strategy

- `set MatchingStrategy (shortest | standard | longest);`
- In `shortest` mode, `?`, `*` and `+` operators match the smallest number of tokens possible (refers to regular expressions at token level)
  ⇒ finds *shortest* sequence matching query,
  ⇒ optional elements at the start or end of the query will *never* be included
- In `longest` mode, `?`, `*` and `+` operators match as many tokens as possible;
- In the default `standard` mode, CQP uses an "early match" strategy: optional elements at the start of the query are included, while those at the end are not.
- The somewhat inconsistent matching strategy of earlier CQP versions is still available in the `traditional` mode, and can sometimes be useful (e.g. to extract cooccurrences between multiple adjectives in a noun phrase and the head noun). The example below only gives the intended frequency counts in `traditional` mode.

```
> [pos="JJ"]+ [pos="NNS?"];
> group Last matchend lemma by match lemma;
```

- Figure 4 shows examples of all four matching strategies for the CQP query

```
> [pos="DT"]? [pos="JJ.*"]* [pos="NNS?"]
  ( [pos="IN|TO"] [pos="DT"]? [pos="JJ.*"]* [pos="NNS?"] )*
```

search pattern:
`DET? ADJ* NN (PREP DET? ADJ* NN)*`

input:

the	old	book	on	the	table	in	the	room

shortest match strategy: (3 matches)

▷		book						
▷					table			
▷								room

longest match strategy: (1 match)

▷	the	old	book	on	the	table	in	the	room

standard matching strategy: (3 matches)

▷	the	old	book					
▷					the	table		
▷							the	room

traditional matching strategy: (7 overlapping matches)

▷	the	old	book					
▷		old	book					
▷			book					
▷					the	table		
▷						table		
▷							the	room
▷								room

Figure 4: CQP matching strategies.

- Standard queries in CQP are executed by first looking up potential starting points with the help of available index data structures, and then scanning for a match of the query from each candidate starting point (by non-deterministic simulation of a finite-state automaton (FSA), as is commonly done for regular languages). If the query contains a top-level disjunction of multiple alternatives and/or there are optional elements in query initial position (as is the case in Fig. 4), the simulation is carried out in multiple passes, one for each query element that could be a starting point (`DET?`, `ADJ*` and `NN` in Fig. 4).

- During a FSA simulation run for a given start position, all matching strategies except `longest` will stop as soon as a successful match has been found, while `longest` continues until there are no active states left or the search boundary has been reached. This makes `longest` matching potentially expensive, especially for unspecific queries involving `[]*`.

- The other three matching strategies only differ in how nested matches from multiple passes are integrated into the final query result:

  - `longest` discards any match that is fully contained in a longer match;
  - `shortest` discards any match that fully contains a shorter match;
  - `standard` picks the *shortest* one of multiple matches with the same start point, and then the *longest* one of multiple matches with the same end point;[14]
  - `traditional` simply keeps all nested matches from the individual passes.

  All matching strategies remove exact duplicates from different passes, even if they have different `target` or `keyword` anchors. However, overlapping matches (that do not nest) are never discarded.

- new in CQP v3.4.12: The matching strategy can be set temporarily with an embedded modifier at the start of a CQP query, e.g.

  ```
  > (?longest) [pos = "NP.*"]+;
  ```

  Currently, only these four modifiers are supported: `(?shortest)`, `(?standard)`, `(?longest)` and `(?traditional)`. Embedded modifiers are particularly useful for Web interfaces that do not give users direct control over the matching strategy. Since they are part of the CQP query syntax, no modifications to existing Web interfaces are required.

- The matching strategy only applies to standard queries, not to TAB queries (Sec. 8.5) or MU queries (Sec. 8.4).

6.2 Word lists

- word lists can be stored in *variables*

  ```
  > define $week =
           "Monday Tuesday Wednesday Thursday Friday";
  ```

  and used instead of regular expressions in the attribute/value pairs

  ```
  > [lemma = $week];
  ```

  (word lists are not allowed in XML start tags, though)

- add/delete words with `+=` and `-=`

  ```
  > define $week += "Saturday Sunday";
  ```

[14] These rules are designed to produce the effect described above, i.e. optional elements at the start of a query are included, but those at the end are excluded. Note that `standard` does allow nested matches provided they are *properly* nested, i.e. have neither the same start point nor the same end point.

- show list of words stored in variable

  ```
  > show $week;
  ```

  use `show var;` to see all variables

- read word list from file (one-word-per-line format)

  ```
  > define $week < "/home/weekdays.txt";
  ```

  new in CQP v3.4.11: files ending in `.gz` or `.bz2` are automatically decompressed (see Sec. 3.1), and word lists can be read from a shell pipe indicated by a `|` character at the start of the filename

  for example, to read a file with whitespace-delimited words (and multiple entries per line):

  ```
  > define $week < "| perl -pe 's/\s+/\n/g' words.txt";
  ```

- use `TAB` key to complete variable names (e.g. type "`show $we`" + `TAB`)

- word lists can be used to simulate type hierarchies, e.g. for part-of-speech tags

  ```
  > define $common_noun = "NN NNS";
  > define $proper_noun = "NP NPS";

  > define $noun = $common_noun;
  > define $noun += $proper_noun;
  ```

- `%c` and `%d` flags can *not* be used with word lists

- use lists of regular expressions with the `RE()` operator (*compile regex*)

  ```
  > define $pref="under.+ over.+";
  > [(lemma=RE($pref)) & (pos="VBG")];
  ```

- flags can be appended to the `RE()` operator

  ```
  > [word = RE($pref) %cd];
  ```

6.3 Subqueries

- queries can be limited to the matching regions of a previous query (⇒ *subqueries*)

- activate named query instead of system corpus (here: sentences containing *interest*)

  ```
  DICKENS> First = [lemma = "interest"] expand to s;
  DICKENS> First;
  DICKENS:First[624]>
  ```

  NB: matches of the activated query must be non-overlapping[15]

- the matches of the named query `First` now define a *virtual* structural attribute on the corpus `DICKENS` with the special name `match`

- all following queries are evaluated with an *implicit* `within match` clause
  (an additional explicit `within` clause may be specified as well)

- re-activate system corpus to exit subquery mode

  ```
  DICKENS:First[624]> DICKENS;
  DICKENS>
  ```

[15] Overlapping matches may result from the `traditional` matching strategy, set operations, or modification of the matching word sequences with `expand`, `set match`, or `set matchend`. When a named query with overlapping matches is activated, a warning message is issued and some of the matches will be automatically deleted.

- XML tag notation can also be used for the temporary `match` regions

  ```
  > <match> [pos = "W.*"];
  ```

  to find tokens matching the given pattern at the *start* of an activated region

- if `target`/`keyword` anchors are set in the activated query result, corresponding XML tags (`<target>`, `<keyword>`, ...) can be used, too

  ```
  > </target> []* </match>;
  ```

  → range from the `target` anchor to end of match, but excluding `target`

  `<target>` and `<keyword>` regions always have length 1 !

- a subquery that *starts* with an anchor tag can be evaluated very efficiently

- appending the *keep* operator `!` turns the subquery into a filter, i.e. it returns all ranges from the activated query result that contain a match of the subquery (equivalent to an implicit `expand to match`)

Subqueries can serve a range of different purposes, especially for advanced users. The examples below illustrate three typical applications.

Searching a subcorpus

- select entire texts (or suitable sub-text regions) based on metadata annotation to define a subcorpus, making sure to `expand` matches appropriately

  ```
  > HardTimes = <novel_title = "Hard Times"> [] expand to novel;
  ```

  ☞ combine multiple queries with set operators (`union`, `diff`, `intersect`) for complex metadata restrictions

- after activating the named query, all following queries will be restricted to the subcorpus

  ```
  > HardTimes;
  DICKENS:HardTimes[1]> [lemma = "hard"];
  ```

- we can also define a subcorpus by content, e.g. all paragraphs that mention horses

  ```
  > HorseSubcorpus = [lemma = "horse" expand to p];
  > HorseSubcorpus;
  ```

Iterative refinement of queries

- start with a fairly general query, e.g. for a prepositional phrase with a particular head noun

  ```
  > A = [pos = "IN"] [pos != "[NP].*"]{0,6} [lemma = "dog"] within s;
  > cat A;
  ```

- use subqueries as filters (i.e. with the *keep* operator `!`) to apply further constraints to the matches; this is often easier than working all constraints into the original query

- e.g. limit to PPs containing an adjective

  ```
  > A;
  DICKENS:A[127]> B = [pos = "JJ.*"] !;
  DICKENS:A[127]> cat B;
  ```

- activate new query result **B** to apply further filters; this can also be used to exclude false positives (FP) from the matches

- e.g. remove false positives that contain punctuation (by specifying the Unicode property escape sequence `\pP`[16], as a full or partial token) or that begin with *that* or *as*

```
DICKENS:A[127]> B;
DICKENS:B[35]> FP = <match> "that|as"%c | ".*\pP.*" !;
DICKENS:B[35]> C = diff B FP;
DICKENS:B[35]> cat C;
```

Pre-filtering complex queries

- a well-known deficit of CQP is that complex queries with a small result set may still run very slowly on large corpora if highly specific constraints appear only near the end of the query; this is exacerbated by many optional elements at the start of the query
- a typical example is searching a noun phrase with a specific head noun, e.g.

```
> set Timing on;
> Horses = [pos="DT"]? ([pos="RB"]? [pos="JJ.*"])* [lemma="horse"];
```

- since (correct) matches must occur within sentences, we can speed up the search by restricting it to sentences that contain the lemma *horse*

```
> Cand = [lemma = "horse"] expand to s;
> Cand;
DICKENS:Cand[545]> H2 = [pos="DT"]? ([pos="RB"]? [pos="JJ.*"])* [lemma="horse"];
```

- the pre-filtered query should be executed 10 to 15 times more quickly, in this example
- you may want to verify that both have exactly the same results:

```
> diff Horses H2;
> diff H2 Horses;
```

6.4 The CQP macro language

- complex queries (or parts of queries) can be stored as macros and re-used
- define macros in a text file (e.g. `macros.txt`):

```
# this is a comment and will be ignored
MACRO np(0)
  [pos = "DT"]    # another comment
  ([pos = "RB.*"]? [pos = "JJ.*"])*
  [pos = "NNS?"]
;
```

  (the above defines macro "`np`" with no arguments)

- load macro definitions from file

```
> define macro < "macros.txt";
```

- macro invocation as part of a CQP command
  (use `TAB` key for macro name completion)

```
> <s> /np[] @[pos="VB.*"] /np[];
```

[16] See https://www.pcre.org/original/doc/html/pcrepattern.html#SEC5

- list all defined macros or those with a given prefix

```
> show macro;
> show macro region;
```

- show macro definition
  (you must specify the number of arguments)

```
> show macro np(0);
```

- (re-)define macro interactively
  (must be written as a single line)

```
> define macro np(0) '[pos="DT"] [pos="JJ.*"]+ [pos="NNS?"]';
```

  or re-load macro definition file

```
> define macro < "macros.txt";
```

- macros are interpolated as plain strings (*not* as elements of a query expression), and may have to be enclosed in parentheses for proper scoping

```
> <s> (/np[])+ [pos="VB.*"];
```

- it is safest to put parentheses around macro definitions:

```
MACRO np(0)
(
  [pos = "DT"]
  ([pos = "RB.*"]? [pos = "JJ.*"])*
  [pos = "NNS?"]
)
;
```

  NB: The start (`MACRO ...`) and end (`;`) markers must be on *separate lines* in a macro definition file.

- macros accept up to 10 arguments; in the macro definition, the number of arguments must be specified in parentheses after the macro name

- in the macro body, each occurrence of `$0`, `$1`, ... is replaced by the corresponding argument value (escapes such as `\$1` will not be recognised)

- e.g. a simple PP macro with 2 arguments: the initial preposition and the number of adjectives in the embedded noun phrase

```
MACRO pp(2)
  [(pos = "IN") & (word="$0")]
  [pos = "DT"]
  [pos = "JJ.*"]{$1}
  [pos = "NNS?"]
;
```

- invoking macros with arguments

```
> /pp["under", 2];
> /pp["in", 3];
```

- macro arguments are character strings and must be enclosed in (single or double) quotes; the quotes may be omitted around numbers and simple identifiers

- the quotes are *not* part of the argument value and hence will not be interpolated into the macro body; nested macro invocations will have to specify additional quotes
- define macro with prototype ⇒ named arguments

```
MACRO pp ($0=Prep $1=N_Adj)
...
;
```

- argument names serve as reminders; they are used by the `show` command and the macro name completion function (`TAB` key)
- argument names are *not* used during macro definition and evaluation
- in interactive definitions, prototypes must be quoted

```
> define macro pp('$0=Prep $1=N_Adj') ... ;
```

- CQP macros can be overloaded by the number of arguments (i.e. there can be several macros with the same name, but with different numbers of arguments)
- this feature is often used for unspecified or "default" values, e.g.

```
MACRO pp($0=Prep, $1=N_Adj)
...
MACRO pp($0=Prep)      (any number of adjectives)
...
MACRO pp()             (any preposition, any number of adjs)
...
```

- macro calls can be nested (non-recursively) ⇒ thus the macro file defines a context-free grammar (CFG) without recursion (see Figure 5)
- Macro definition files can import other macro definition files using statements of the form

  `IMPORT other_macros.txt`

  Each import statement must be written on a separate line. It is recommended (but not required) to collect all `IMPORT`s at the top of the file. File paths are interpreted relative to the CQP working directory, *not* the location of the current macro file.
- string arguments need to be quoted when they are passed to nested macros (since quotes from the original invocation are stripped before interpolating an argument)
- single or double quote characters in macro arguments should be avoided whenever possible; while the string *'s* can be enclosed in double quotes (`"'s"`) in the macro invocation, the macro body may interpolate the value between single quotes, leading to a parse error
- in macro definitions, it's normally better to use double quotes, which are less likely to occur in argument values

6.5 CQP macro examples

- use macros for easier access to embedded noun phrases (NP)
- write and load the macro definition file shown in Figure 6

```
MACRO adjp()
  [pos = "RB.*"]?
  [pos = "JJ.*"]
;

MACRO np($0=N_Adj)
  [pos = "DT"]
  ( /adjp[] ){$0}
  [pos = "NNS?"]
;

MACRO np($0=Noun $1=N_Adj)
  [pos = "DT"]
  ( /adjp[] ){$1}
  [(pos = "NN") & (lemma = "$0")]
;

MACRO pp($0=Prep $1=N_Adj)
  [(word = "$0") & (pos = "IN|TO")]
  /np[$1]
;
```

Figure 5: A sample macro definition file.

```
MACRO np_start()
  (<np>|<np1>|<np2>)
;

MACRO np_end()
(</np2>|</np1>|</np>)
;

MACRO np()
( /np_start[] []* /np_end[] )
;
```

Figure 6: Macro definition file for accessing embedded noun phrases.

- then use `/np_start[]` and `/np_end[]` instead of `<np>` and `</np>` tags in CQP queries, as well as `/np[]` instead of `/region[np]`

  ```
  > /np_start[] /np[] "and" /np[] /np_end[];
  ```

- CQP ensures that the "generalised" start and end tags nest properly
  (if the `StrictRegions` option is enabled, cf. Sections 4.2 and 4.3)

- extending built-in macros: view definitions

  ```
  > show macro region(1);
  > show macro codist(3);
  ```

- extend `/region[]` macro to embedded regions:

  ```
  MACRO anyregion($0=Tag)
    (<$0>|<$01>|<$02>)
    []*
    (</$02>|</$01>|</$0>)
  ;
  ```

- extend `/codist[]` macro to two constraints:

  ```
  MACRO codist($0=Att1 $1=V1 $2=Att2 $3=V2 $4=Att3)
    _Results = [($0 = "$1") & ($2 = "$3")];
    group _Results match $4;
    discard _Results;
  ;
  ```

- usage examples:

  ```
  > "man" /anyregion[pp];
  > /codist[lemma, "go", pos, "V.*", word];
  ```

- the simple string interpolation of macros allows some neat tricks,
  e.g. a `region` macro with constraints on key-value pairs in the start tag

  ```
  MACRO region($0=Att $1=Key $2=Val)
    <$0_$1 = "$2"> []* </$0_$1>
  ;

  MACRO region($0=Att $1=Key1 $2=Val1 $3=Key2 $4=Val2)
    <$0_$1 = "$2"><$0_$3 = "$4"> []* </$0_$3></$0_$1>
  ;
  ```

6.6 Feature set attributes (GERMAN-LAW)

- feature set attributes use special notation, separating set members by `|` characters (both in the input files and in the indexed corpus itself)

- they can be used to represent annotations that can be ambiguous and/or multi-valued

- e.g. for an `alemma` (ambiguous lemma) attribute

`|Zeug|Zeuge|Zeugen|`	(three elements)
`|Baum|`	(unique lemma)
`|`	(not in lexicon)

- `ambiguity()` function yields number of elements in set (its *cardinality*)

  ```
  > [ambiguity(alemma) > 3];
  ```

- use `contains` operator to test for membership in the set

  ```
  > [alemma contains "Zeuge"];
  ```
  → words which *can be* lemmatised as *Zeuge*

  equivalent to `[alemma = ".*\|Zeuge\|.*"]`

- test non-membership with `not contains`

  ```
  (alemma not contains "Zeuge")
  ⟺ !(alemma contains "Zeuge")
  ```

- also used to annotate phrases with sets of properties

  ```
  > /region[np, a] :: a.np_f contains "quot";
  ```

- see Appendix A.3 for lists of properties annotated in the `GERMAN-LAW` corpus

- define macro for easy experimentation with property features

  ```
  > define macro find('$0=Tag $1=Property')
                '<$0_f contains "$1"> []* </$0_f>';
  ```

  ```
  > /find[np, brac];
  > /find[advp, temp];
  ```
  etc.

- nominal agreement features of determiners, adjectives and nouns are stored in the `agr` attribute, using the pattern shown in Figure 7 (see Figure 8 for an example)

  "*case* : *gender* : *number* : *determination*"

case	`Nom`, `Gen`, `Dat`, `Akk`
gender	`M`, `F`, `N`
number	`Sg`, `Pl`
determination	`Def`, `Ind`, `Nil`

Figure 7: Annotation of noun agreement features in the `GERMAN-LAW` corpus.

```
der     |Dat:F:Sg:Def|Gen:F:Pl:Def|Gen:F:Sg:Def
        |Gen:M:Pl:Def|Gen:N:Pl:Def|Nom:M:Sg:Def|
Stoffe  |Akk:M:Pl:Def|Dat:M:Sg:Def|Gen:M:Pl:Def|Nom:M:Pl:Def
        |Akk:M:Pl:Ind|Dat:M:Sg:Ind|Gen:M:Pl:Ind|Nom:M:Pl:Ind
        |Akk:M:Pl:Nil|Dat:M:Sg:Nil|Gen:M:Pl:Nil|Nom:M:Pl:Nil|
```

Figure 8: An example of noun agreement features in the `GERMAN-LAW` corpus

- require all set members to match a regular expression

  ```
  > [ (pos = "NN") & (agr matches ".*:Pl:.*") ];
  ```
  → nouns which are uniquely identified as plurals

- both `contains` and `matches` use regular expressions and accept the `%c` and `%d` flags

- unification of agreement features $\Longleftrightarrow$ intersection of feature sets
- use built-in `/unify[]` macro:

  ```
  /unify[agr, <label1>, <label2>, ...]
  ```

- undefined labels will automatically be ignored

  ```
  > a:[pos="ART"] b:[pos="ADJA"]? c:[pos="NN"]
       :: /unify[agr, a,b,c] matches "Gen:.*";
  ```

  → (simple) NPs uniquely identified as genitive

  ```
  > a:[pos="ART"] b:[pos="ADJA"]? c:[pos="NN"]
       :: /unify[agr, a,b,c] contains "Dat:.:Sg:.*";
  ```

  → NPs which *might* be dative singular

- use `ambiguity()` function to find number of possible analyses

  ```
  > ... :: ambiguity(/unify[agr, a,b,c]) >= 1;
  ```

  → to check agreement within NP

- in the `GERMAN-LAW` corpus, NPs and other phrases are annotated with partially disambiguated agreement information; these features sets can also be tested with the `contains` and `matches` operators, either indirectly through label references or directly in XML start tags

  ```
  > /region[np, a] :: a.np_agr matches "Dat:.:Pl:.*";
  > <np_agr matches "Dat:.:Pl:.*"> []* </np_agr>;
  ```

- for computation speed, `/unify[]` expects features sets in *canonical format*, with members sorted according to CWB's internal sort order; this is usually ensured with the `-m` option to `cwb-s-encode`
- even if an attribute hasn't explicitly been defined as a feature set (and converted to canonical format), `ambiguity()`, `contains` and `matches` are guaranteed to work as long as the `|`-separated set notation is used correctly and consistently
- however, the `/unify[]` macro cannot be used *unless* the features within each set are sorted in the canonical format. Only if an attribute is explicitly declared as a feature set at indexing-time are the members of the sets sorted into the canonical order.
- thus, feature set attributes cannot encode *ordered* lists of values; if you need to distinguish between a first, second, ... alternative, you might add this information explicitly as a feature component, e.g.

  ```
  |1:Zeuge|2:Zeug|3:Zeugen|
  ```

7 Interfacing CQP with other software

7.1 Running CQP as a backend

- CQP is a useful tool for interactive work, but many tasks become tedious when they have to be carried out by hand; macros can be used as *templates*, providing some relief; however, full *scripting* is still desirable (and in some cases essential)
- similarly, the output of CQP requires post-processing at times: better formatting of KWIC lines (especially for HTML output), different sort options for frequency tables, frequency counts on normalised word forms (or other transformations of the values)
- for both purposes, an external scripting tool or programming language is required, which has to interact dynamically with CQP (which acts as a query engine)
- CQP provides some support for such interfaces: when invoked with the `-c` flag, it switches to *child mode* (which could also be called "slave" mode):
  - the init file `.cqprc` (in the current user's home folder) is not automatically read at startup
  - CQP prints its version number after startup
  - all interactive features are deactivated (paged display and highlighting)
  - query results are not automatically displayed (`set AutoShow off;`)
  - after the execution of a command, CQP flushes output buffers (so that the interface will not hang waiting for output from the command)
  - in case of a syntax error, the string `PARSE ERROR` is printed on `stderr`
  - the special command `.EOL.;` can be used to insert the line

    `-::-EOL-::-`

    as a marker into CQP's output
  - when the `ProgressBar` option is activated, progress messages are not echoed in a single screen line (using carriage returns) on `stderr`, but rather printed in separate lines on `stdout`; these lines have the standardized format

    `-::-PROGRESS-::-` TAB *pass* TAB *no. of passes* TAB *progress message*
- the CWB/Perl interface makes use of all these features to provide an efficient and robust interface between a Perl script and the CQP backend
- these features are also used extensively by CQPweb and many other web interfaces
- the output of many CQP commands is neatly formatted for human readers; this *pretty printing* feature can be switched off with the command

  `> set PrettyPrint off;`

  the output of the `show`, `group` and `count` commands now has a simple and standardized format that can more easily be parsed by the invoking program; output formats for the different uses of the `show` command are documented below; see Section 7.3 for the output formats of `group` and `count`

  - `show corpora;` prints the names of all available corpora on separate lines, in alphabetical order
  - `show named;` lists all named query results on separate lines in the format

    flags TAB *query name* TAB *no. of matches*

flags is a three-character code representing the flags `m` = stored in memory, `d` = saved to disk, `*` = modified since last saved; flags that are not set are shown as `-` characters; *query name* is the long name of the query result, i.e. it has the form *corpus*`:`*name*; when a query result has not yet been loaded from disk, the *no. of matches* cannot be determined and is reported as 0

- `show;` concatenates the output of `show corpora;` and `show named;` without any separator; it is recommended to invoke the two commands separately when using CQP as a backend
- `show active;` prints the name of the currently active corpus on a line on its own (this is in fact available when using CQP interactively, albeit useless because the active corpus is displayed in the CQP command prompt!)
- `show cd;` lists all attributes that are defined for the currently active corpus; each attribute is printed on a separate line with the format

  attribute type `TAB` *attribute name* [`TAB` [`-V`]]

  attribute type is one of the strings `p-Att` (positional attribute), `s-Att` (structural attribute) or `a-Att` (alignment attribute), so the attribute type can easily be recognized from the first character of the output line; the third column is only printed for s-attributes and is either an empty string (no annotations) or `-V` (regions have annotated values)
- new in CQP v3.4.18: the output of `show cd;` now always prints four TAB-separated columns, i.e. lines of the form

  attribute type `TAB` *attribute name* `TAB` [`-V`] `TAB` [`*`]

  the third column is `-V` for an s-attribute with annotations and an empty string otherwise; the fourth column is `*` if the attribute is currently selected for display and an empty string otherwise

- the CWB/Perl interface automatically deactivates pretty printing
- running CQP as a backend can be a security risk, e.g. when queries submitted to a Web server are passed through to the CQP process unaltered; this may allow malicious users to execute arbitrary shell commands on the Web server
- as a safeguard against such attacks, CQP provides a *query lock* mode, which allows only queries to be executed, while all other commands (including `cat`, `sort`, `group`, *etc.*) are blocked
- the query lock mode is activated with the command

  `> set QueryLock` n`;`

  where n is a randomly chosen integer number; it can only be de-activated with

  `> unlock` n`;`

  using the same number n

- new in CQP v3.4.18: A frontend that intends to parse KWIC output from the `cat` command should normally change the default slash (`/`) separator between p-attributes to something less ambiguous. For example, a KWIC output line with `word` and `lemma` activated might look as follows:

  ⟹ `<s> Most/most CP/M/CP/M sytems/system ...`

  The new CQP option `AttributeSeparator` or (`sep`) allows users to override this default setting with an (almost) arbitrary string, e.g.

  ```
  > set AttributeSeparator "__";
  ⟹ <s> Most__most CP/M__CP/M sytems__system ...
  ```

- new in CQP v3.4.24: Likewise, a frontend might well prefer to use a different separator between tokens in the KWIC in place of the default space (since space is a legal character within p-attribute values); the new option `TokenSeparator` or `tok` parallels `AttributeSeparator` to override the default space with (almost) anything.
- The safest option for unambiguous attribute or token separator strings is to use a control code that is disallowed in attribute values, e.g. TAB (#9), BEL (#7) or ESC (#27).[17] Note that these controls have to be included as literal characters in the `set` command because CQP doesn't support escape sequences such as `\t` or `\x09`.
- An overriding `AttributeSeparator` or `TokenSeparator` can be turned off by setting the option in question to an empty string. This re-enables the default (slash or space respectively).

7.2 Exchanging corpus positions with external programs

- An important aspect of interfacing CQP with other software is passing back and forth lists of corpus positions of query matches (as well as `target` and `keyword` anchors). This is a prerequisite for extracting further information about the matches by direct corpus access, and it is the most efficient way of relating query matches to externally managed data structures (e.g. metadata held in a SQL database or spreadsheet application).
- The `dump` command (Section 3.3) prints the required information in a tabular ASCII format that can easily be parsed by other tools or read into a SQL database.[18] Each row of the resulting table corresponds to one match of the query, and the four columns give the corpus positions of the `match`, `matchend`, `target` and `keyword` anchors, respectively. The example below is reproduced from Section 3.3

```
1019887 1019888 -1      -1
1924977 1924979 1924978 -1
1986623 1986624 -1      -1
2086708 2086710 2086709 -1
2087618 2087619 -1      -1
2122565 2122566 -1      -1
```

  Undefined `target` anchors are represented by `-1` in the third column. Even though no `keyword` anchors were set for the query in question, the fourth column is included in the dump table, but with all values set to `-1`.
- The table created by the `dump` command is printed to standard output (`stdout`) by default, where it can be captured by a program running CQP as a backend (e.g. the CWB/Perl interface, cf. Sec. 7.1). The dump table can also be redirected to a file:

  ```
  > dump A > "dump.tbl";
  ```

  which is automatically compressed if the filename ends in `.gz` or `.bz2` (new in CQP v3.4.11; see Sec. 3.1).
- Alternatively, the output can also be redirected to a pipe, e.g. to create a dump file without the superfluous `keyword` column

  ```
  > dump A > "| cut -f 1-3 > dump.tbl";
  ```

  (Windows users should refer to the caveats on use of pipes in Sec. 3.3.)

[17] CR or LF is probably a very bad idea. As is use of the same string for both attribute and token separators.

[18] Since this command dumps the matches of a named query in their current sort order, the natural order should normally first be restored by calling `sort` without a `by` clause. One exception is if the dump is to be used for a KWIC display of the query results in their sorted order.

- in versions of CQP prior to v3.4.11, a pipe was needed to compress a dump file on the fly

  ```
  > dump A > "| gzip > dump.tbl.gz";
  ```

- Sometimes it is desirable to reload a dump file into CQP after it has been modified by an external program (e.g. a database may have filtered the matches against a metadata table). The `undump` command creates a new named query result (`B` in the example below) for the currently activated corpus from a dump file (which may be a compressed file in CQP v3.4.11 and newer):

  ```
  > undump B < "mydump.tbl";
  ```

  Undumping data to an NQR (here, `B`) overwrites that NQR if it already exists, silently and without warning.

- The format of files to be undumped, here `mydump.tbl`, is almost identical to the output of `dump`, but it contains only two columns: for the `match` and `matchend` positions (in the default setting). The example below shows a valid dump file for the `DICKENS` corpus, which can be read with `undump` to create a query result containing 5 matches:

  ```
  20681   20687
  379735  379741
  1915978 1915983
  2591586 2591591
  2591593 2591598
  ```

  Save these lines to a text file named `dickens.tbl`, then enter the following commands:

  ```
  > DICKENS;
  > undump Twas < "dickens.tbl";
  > cat Twas;
  ```

- Further columns for the `target` and `keyword` anchors (in that order) can optionally be added. In this case, you must append the modifier `with target` or `with target keyword` to the `undump` command:

  ```
  > undump B with target keyword < "mydump.tbl";
  ```

- Dump files can also be read from a pipe or from standard input. In the latter case the table of corpus positions has to be preceded by a header line that specifies the total number of matches:

  ```
  5
  20681   20687
  379735  379741
  1915978 1915983
  2591586 2591591
  2591593 2591598
  ```

  CQP uses this information to pre-allocate internal storage for the query result, as well as to validate the file format. This format can also be used as a more efficient alternative if the dump is read from a regular file. CQP automatically detects which of the two formats is used.

- Pipes can be used e.g. to read a dump table generated by another program. They are indicated by a pipe symbol (`|`) at the start of the filename (new in CQP v3.4.11) or at the end of the filename (earlier versions); see further the notes in Sec. 3.3.

- Before CQP v3.4.11, pipes were also needed to read a dump table from a compressed file:

  ```
  > undump B < "| gzip -cd mydump.tbl.gz";
  ```


- In an interactive CQP session, the input file can be omitted and the undump table can then be entered directly on the command line. This feature works best if command-line editing support is enabled with the `-e` switch.

- Since the dump table is read from standard input here, only the second format is allowed, i.e. you have to enter the total number of matches first. Try entering the example table above after typing

  ```
  > undump B;
  ```

- Without the `-e` switch, the standard-input format is a little counterintuitive. The initial `undump` command must be terminated by a semi-colon, which is followed *directly* by the header number - with no space between the semi-colon and the number!! The remaining lines are entered as usual.

  ```
  > undump In-Non-E-Mode;2
  1915978 1915983
  2591586 2591591
  ```

- If the rows of the undump table are not sorted in their natural order (i.e. by corpus position), they have to be re-ordered internally so that CQP can work with them. However, the original sort order is recorded automatically and will be used by the `cat` and `dump` commands (until it is reset by a new `sort` command). If you sort a query result `A`, save it with `dump` to a text file, and then read this file back in as named query `B`, then `A` and `B` will be sorted in exactly the same order.

- In many cases, overlapping or unsorted matches are not intentional but rather errors in an automatically generated dump table. In order to catch such errors, the additional keyword `ascending` (or `asc`) can be specified before the `<` character:

  ```
  > undump B with target ascending < "mydump.tbl";
  ```

  This command will abort with an error message (indicating the row number where the error occurred) unless the corpus matches in `mydump.tbl` are non-overlapping and sorted in corpus order.

- A typical use case for `dump` and `undump` is to link CQP queries to corpus metadata stored in an external database. Assume that a corpus consists of a large collection of transcribed dialogues, which are marked as `<dialogue>` regions. Assume further that rich metadata (about the speakers, setting, topic, etc.) is available in a SQL database. The database entries can be linked directly to the `<dialogue>` regions by recording their start and end corpus positions in the database.[19] The following commands generate a dump table with the required information, which can easily be loaded into the database (ignoring the third and fourth columns of the table):

  ```
  > A = <dialogue> [] expand to dialogue;
  > dump A > "dialogues.tbl";
  ```

  Corpus queries will often be restricted to a subcorpus by specifying constraints on the metadata. Having resolved the metadata constraints in the SQL database, they can be translated to the corresponding regions in the corpus (again represented by start and end corpus position). The positions are then sorted in ascending order and saved to a TAB-delimited text file. Now they can be loaded into CQP with the `undump` command, and the resulting query result can be activated as a subcorpus for following queries. It is recommended to specify the `ascending` option in order to ensure that the loaded query result forms a valid subcorpus:

[19] Of course, it is also possible to establish an indirect link through document IDs, annotated as `<dialogue id=XXXX> .. </dialogue>`. If the corpus contains a very large number of dialogues, the direct link approach is usually much more efficient, though.

```
> undump SubCorpus ascending < "subcorpus.tbl";
> SubCorpus;
Subcorpus[..]> A = ... ;
```

7.3 Generating frequency tables

- for many applications it is important to compute frequency tables for the matching strings, tokens in the immediate context, attribute values at different anchor points, different attributes for the same anchor, or various combinations thereof
- frequency tables for the matching strings, optionally normalised to lowercase and extended or reduced by an offset, can easily be computed with the `count` command (cf. Sections 2.9 and 3.3); when pretty-printing is deactivated (cf. Section 7.1), its output has the form

  frequency `TAB` *first line* `TAB` *string (type)*

- advantages of the `count` command:
  - strings of arbitrary length can be counted
  - frequency counts can be based on normalised strings (`%cd` flags)
  - the instances (tokens) for a given string type can easily be identified, since the underlying query result is automatically sorted by the `count` command, so that these instances appear as a block starting at match number *first line*
- an alternative solution is the `group` command (see Sec. 3.4), which computes frequency distributions over single tokens (i.e. attribute values at a given anchor position) or pairs of tokens (recall the counter-intuitive command syntax for this case); when pretty-printing is deactivated, its output has the form

  [*attribute value* `TAB`] *attribute value* `TAB` *frequency*

- advantages of the `group` command:
  - can compute joint frequencies for non-adjacent tokens
  - faster when there are relatively few different types to be counted
  - supports frequency distributions for the values of s-attributes
- the advantages of `group` and `count` are for the most part complementary (e.g. it is not possible to normalise the values of s-attributes, or to compute joint frequencies of two non-adjacent multi-token strings); in addition, they have some common weaknesses, such as relatively slow execution, no options for filtering and pooling data, and limitations on the types of frequency distributions that can be computed (only simple joint frequencies, no nested groupings)
- new in CQP v3.4.9: The `group` command has been re-implemented with a hash-based algorithm. It is very fast now, even for large frequency tables. The other limitations still apply, though.
- therefore, it is often necessary (and usually more efficient) to generate frequency tables with external programs such as dedicated software for statistical computing or a relational database; these tools need a *data table* as input, which lists the relevant feature values (at specified anchor positions) and/or multi-token strings for each match in the query result; such tables can often be created from the output of `cat` (using suitable `PrintOptions`, `Context` and `show` settings)

- this procedure involves a considerable amount of re-formatting (e.g. with Unix command-line tools or Perl scripts) and can easily break when there are unusual attribute values in the data; both `cat` output and the re-formatting operations are expensive, making this solution inefficient when there is a large number of matches

- in most situations, the `tabulate` command provides a more convenient, more robust and faster solution; the general form is

  > `tabulate A` *column spec*`,` *column spec*`, ... ;`

  this will print a `TAB`-separated table where each row corresponds to one match of the query result `A` and the columns are described by one or more *column spec(ification)s*

- just as with `dump` and `cat`, the table can be restricted to a contiguous range of matches, and the output can be redirected to a file or pipe

  > `tabulate A 100 119` *column spec*`,` *column spec*`, ... ;`
  > `tabulate A` *column spec*`,` *column spec*`, ... > "data.tbl";`

- each column specification consists of a single anchor (with optional offset) or a range between two anchors, using the same syntax as `sort` and `count`; without an attribute name, this will print the corpus positions for the selected anchor, so

  > `tabulate A match, matchend, target, keyword;`

  produces exactly the same output as `dump A;` when target and keyword anchors are defined for the query result `A`; otherwise, it will print an error message (and you need to leave out the column specs `target` and/or `keyword`)

- when an attribute name is given after the anchor, the values of this attribute for the selected anchor point will be printed; both positional and structural attributes with annotated values can be used; the following example prints a table of novel title, book number and chapter title for a query result from the `DICKENS` corpus

  > `tabulate A match novel_title, match book_num, match chapter_title;`

  note that undefined values (for the `book_num` and `chapter_title` attributes) are represented in the tabulation output by the empty string

- if an anchor point is undefined or falls outside the corpus (because of an offset), `tabulate` prints an empty string or the corpus position `-1` (correct behaviour implemented in v3.4.10)

- a range between two anchor points prints the values of the selected attribute for all tokens in the specified range; usually, this only makes sense for positional attributes; the following example prints the `lemma` values for 5 tokens before and after each match; this data can be used to identify collocates of the items matched by the query

  > `tabulate A match[-5]..match[-1] lemma, matchend[1]..matchend[5] lemma;`

  ☞the attribute values for tokens within each range are separated by blanks rather than `TAB`s, in order to avoid ambiguities in the resulting data table

- any items in an anchor-point range that fall outside the bounds of the corpus are printed as empty strings or corpus positions `-1`; if either the start or end of the range is an undefined anchor, a single empty string or cpos `-1` is printed for the entire range (correct behaviour implemented in v3.4.10)

- the end position of a range must not be smaller than its start position, so take care to order items properly and specify sensible offsets; in particular, a range specification such as `match .. target` must not be used if the target anchor might be to the left of the match; the behaviour of CQP in such cases is unspecified

- attribute values can be normalised with the flags `%c` (to lowercase) and `%d` (remove diacritics); the command below uses Unix shell commands to compute the same frequency distribution as `count A by word %c;` in a much more efficient manner

  ```
  > tabulate A match .. matchend word %c > "| sort | uniq -c | sort -nr";
  ```

- note that in contrast to the behaviour of `sort` and `count`, a range is considered empty when the end point lies *before* the start point; such a range will always be printed as an empty string

8 CQP for experts

8.1 Zero-width assertions

- constraints involving labels have to be tested either in the global constraint or in one of the token patterns; this means that macros cannot easily specify constraints on the labels they define: such a macro would have to be interpolated in two separate places (in the sequence of token patterns as well as in the global constraint)
- zero-width *assertions* allow constraints to be tested during query evaluation, i.e. at a specific point in the sequence of token patterns; an assertion uses the same Boolean expression syntax as a pattern, but is delimited by `[: ... :]` rather than simple square brackets (`[...]`); unlike an ordinary pattern, an assertion does not "consume" a token when it is matched; it can be thought of as a part of the global constraint that is tested in between two tokens
- with the help of assertions, NPs with agreement checks can be encapsulated in a macro

```
DEFINE MACRO np_agr(0)
  a:[pos="ART"]
  b:[pos="ADJA"]*
  c:[pos="NN"]
  [: ambiguity(/unify[agr, a,b,c]) >= 1 :]
;
```

  (in this simple case, the constraint could also have been added to the last pattern)
- when the *this* label (`_`) is used in an assertion, it refers to the corpus position of the *following* token; the same holds for direct references to attributes
- in this way, assertions can be used as look-ahead constraints, e.g. to match maximal sequences of tokens without activating `longest` match strategy

```
> [pos = "NNS?"]{2,} [:pos != "NNS?":];
```

- assertions also allow the independent combination of multiple constraints that are applied to a single token; for instance, the `region(5)` macro from Section 6.5 could also have been defined as

```
MACRO region($0=Att $1=Key1 $2=Val1 $3=Key2 $4=Val2)
  <$0> [: _.$0_$1="$2" :] [: _.$0_$3="$4" :] []* </$0>
;
```

- like the matchall pattern `[]`, the matchall assertion `[::]` is always satisfied; since it does not "consume" a token either, it is a no-op (an operation that does nothing) that can freely be inserted at any point in a query expression; in this way, a label or target marker can be added to positions which are otherwise not accessible, e.g. an XML tag or the start/end position of a disjunction

```
> ... @[::] /region[np] ... ;
> ... a:[::] ( ... | ... | ... ) b:[::] ...;
```

  starting a query with a matchall assertion is extremely inefficient: use the `match` anchor or the implicit `match` label instead

8.2 Labels and scope

- returning to the `np_agr` macro from Section 8.1, we note a problem with this query:

```
> A = /np_agr[] [pos = "VVFIN"] /np_agr[];
```

when the second NP does not contain any adjectives but the first does, the `b` label will still point to an adjective in the first NP; consequently, the agreement check may fail even if both NPs are really valid

- in order to solve this problem, the two NPs should use different labels; for his purpose, every macro has an implicit `$$` argument, which is set to a unique value for each interpolation of the macro; in this way, we can construct unique labels for each NP:

```
DEFINE MACRO np_agr(0)
  $$_a:[pos="ART"]
  $$_b:[pos="ADJA"]*
  $$_c:[pos="NN"]
  [: ambiguity(/unify[agr, $$_a,$$_b,$$_c]) >= 1 :]
;
```

  a comparison with the previous results shows that this version of the `/np_agr[]` macro finds additional matches that were incorrectly rejected by the first implementation

```
> B = /np_agr[] [pos = "VVFIN"] /np_agr[];
> diff B A;
```

- however, the problem still persists in queries where the macro is *interpolated* only once, but may be *matched* multiple times

```
> A = ( /np_agr[] ){3};
```

  here, a solution is only possible when the scope of labels can be limited to the body of the macro in which they are defined; i.e. the labels must be reset to undefined values at the end of the macro block; this can be achieved with the built-in `/undef[]` macro, which resets the labels passed as arguments and returns a true value

```
DEFINE MACRO np_agr(0)
  a:[pos="ART"]
  b:[pos="ADJA"]*
  c:[pos="NN"]
  [: ambiguity(/unify[agr, a,b,c]) >= 1 :]
  [: /undef[a,b,c] :]
;
```

```
> B = ( /np_agr[] ){3};
> diff B A;
```

- however, it may still be wise to construct unique label names (either in the form `np_agr_a` *etc.*, or with the implicit `$$` argument) in order to avoid conflicts with labels defined in other macros or in the top-level query

8.3 CQP built-in functions

The CQP query language offers a number of built-in functions that can be applied to attribute values within query constraints (but not anywhere else, e.g. in `group` or `tabulate` commands). The list below shows all built-in functions that are currently available.

f(*att*): frequency of the current value of p-attribute *att* (cannot be used with s-attributes or literal values); e.g. `[word = ".*able" & f(word) < 10]`

dist(*a*, *b*), distance(*a*, *b*): signed distance between two tokens referenced by labels *a* and *b*; explicit numeric corpus positions may be specified instead of labels; computes the difference $b - a$; e.g. `... :: dist(matchend, match) >= 10;`

distabs(*a*, *b*): unsigned distance between two tokens; e.g. `[dist(_, 1000) <= 10]` as an inefficient way to match 10 tokens to the left and right of corpus position 1000

int(*str*): cast *str* to a signed integer number so numeric comparisons can be made; raises an error if *str* is not a numeric string; e.g. `... :: int(match.text_year) <= 1900;`

lbound(*att*), rbound(*att*): evaluates to true if the current corpus position is the first or last token in a region of s-attribute *att*, respectively

lbound_of(*att*, *a*), rbound_of(*att*, *a*): returns the corpus position of the start or end of the region of s-attribute *att* containing the token referenced by label *a*, suitable for use with `dist()`;[20] if *a* is not within a region of *att*, an undefined value is returned, which evaluates to false in most contexts [new in v3.4.13]

unify(fs_1, fs_2): compute the intersection of two sorted feature sets specified as strings fs_1 and fs_2, corresponding to a unification of feature bundles; if the first argument is an undefined value, fs_2 is returned; see Sec. 6.6 for details

ambiguity(*fs*): compute the size of a feature set specified as string *fs*, i.e. the number of elements; if *fs* is an undefined value, a size of 0 is returned (same as for `|`); see Sec. 6.6 for details

add(*x*, *y*), sub(*x*, *y*), mul(*x*, *y*): simple arithmetic on integer values *x* and *y*, which can also be corpus positions specified as labels; when performing computations on corpus annotations, they have to be typecast with `int()` first

prefix(str_1, str_2): returns longest common prefix of strings str_1 and str_2; warning: this function operates on bytes and may return an incomplete UTF-8 character

is_prefix(str_1, str_2): returns true if string str_1 is a prefix of str_2; e.g. `[is_prefix(lemma, word)]`

minus(str_1, str_2): removes the longest common prefix of str_1 and str_2 from the string str_1 and returns the remaining suffix; warning: this function operates on bytes and may return an incomplete UTF-8 character

ignore(*a*): ignore the label *a* and always return true; for internal use by the `/undef[]` macro, see Sec. 8.2 for details

normalize(*str*, *flags*): apply case-folding and/or diacritic folding to the string *str* and return the normalized value; *flags* must be a literal string `"c"`, `"d"` or `"cd"` (with an optional `%`, e.g. `"%cd"`); e.g. `[normalize(word, "cd") != normalize(lemma, "cd")]` to find non-trivial differences between word form and lemma [new in v3.4.11]

strlen(*str*): returns the length of *str* in characters (if the active corpus is encoded in UTF-8) or bytes (for all other encodings) [new in v3.4.17]

[20] The second argument is necessitated by technical limitations of built-in functions. To locate the start of a sentence containing the current token, use the *this* label: `lbound_of(s, _)`.

8.4 MU queries

- CQP offers search-engine-like "Boolean" queries in a special **meet-union** (MU) notation. This feature goes back to the original developer of CWB, but was not supported officially before CWB v3.4.12. In particular, there was no precise specification of the semantics of MU queries, and the original implementation did not produce consistent results.

- new in v3.4.12: Recently, MU queries have found more widespread use as *proximity queries* in the CEQL "simple query" syntax of BNCweb and CQPweb, giving them a semi-official status. For this reason, the implementation was modified to ensure a consistent and well-defined behaviour, although it may not always correspond to what is desired intuitively. The new MU implementation is documented here.

- Warning: both the syntax and the semantics of MU queries are subject to fundamental revisions in the next major release of CWB (version 4.0), but are considered stable with long-term support for CWB 3.5.

- A meet-union query consists of nested `meet` and `union` operations forming a binary-branching tree that is written in LISP-like prefix notation. MU queries always start with the keyword `MU` and are completely separate from the standard CQP syntax, sharing only the system by which individual token patterns are specified

- The simplest form of a MU query specifies a single token pattern, which may also be given in shorthand notation if the default p-attribute is to be matched. These queries are fully equivalent to the corresponding standard queries (which would be the same, but without the leading `MU`).

  ```
  > MU [lemma = "light" & pos = "V.*"];
  > MU "lights" %c;
  ```

- A `meet` clause matches two token patterns within a specified distance of each other. More precisely, instances of the first pattern are filtered, keeping only those where the second pattern occurs within the specified window. For example, the following query finds nouns that co-occur with the adjective *lovely*:

  ```
  > MU(meet [pos = "NN.*"] [lemma = "lovely"] -2 2);
  ```

  This query returns all nouns for which *lovely* occurs within two tokens to the left (window starting at offset `-2`) or right (window ending at offset `+2`)). The adjective *lovely* is not included in the match, nor marked in any other way.

- In order to match only prenominal adjectives, we can change the window to include only the three tokens preceding the noun (i.e. offsets `-3` . . . `-1`):

  ```
  > MU(meet [pos = "NN.*"] [lemma = "lovely"] -3 -1);
  ```

- Since a `meet` clause returns only occurrences of the first token pattern, we need to change the ordering in order to focus on the adjective rather than the nouns. Don't forget to adjust the window offsets accordingly!

  ```
  > MU(meet [lemma = "lovely"] [pos = "NN.*"] 1 3);
  ```

  Note that `meet` operations are not symmetric: this query returns fewer matches than the previous one (viz. those cases where multiple nouns occur near the same instance of *lovely*).

- Alternatively, we can search for co-occurrence within sentences or other s-attribute regions. Again, the ordering of the token constrains determines whether we focus on *tea* or *cakes*:

  ```
  > MU(meet "tea"%c "cakes"%c s);
  ```

- A `union` clause simply combines the matches of two token patterns into a set union, corresponding to a disjunction (logical *or*) of the constraints. The following three queries are fully equivalent:

  ```
  > MU(union "tea"%c "coffee"%c);
  > "tea"%c | "coffee"%c;
  > [(word = "tea" %c) | (word = "coffee" %c)];
  ```

- MU queries are relatively powerful because the two elements of a `meet` or `union` clause can themselves be complex clauses. For example, the trigram *in due course* can be found by nesting two `meet` conditions:

  ```
  > MU(meet (meet "in" "due" 1 1) "course" 2 2);
  ```

  The inner clause returns all instances of *in* that are immediately followed by *due*; the outer clause requires that the following token (the token at an offset of `+2` from *in*) must be *course*. We can obtain exactly the same result with this query:

  ```
  > MU(meet "in" (meet "due" "course" 1 1) 1 1);
  ```

  Now the inner clause determines all occurrences of the bigram *due course*, but returns only the corpus positions of *due*, which must appear immediately after *in*.

  Can you find two other MU formulations that produce exactly the same results?

- Keep in mind that the final result includes only the corpus positions of the leftmost-specified token pattern. If you want to find instances of *course* in this multiword expression, rewrite the query as

  ```
  > MU(meet (meet "course" "due" -1 -1) "in" -2 -2);
  ```

- new in v3.4.30: `meet` clauses in MU queries can now also be **negated**. When the `not` operator is given, CQP discards (rather than eclusively retaining) all matches for the first subclause that co-occur with a match of the second subclause. The following query finds all instances of *ground* that are *not* preceded by an article within a span of 3 tokens:

  ```
  > MU(meet "ground" not [pos = "DT"] -3 -1);
  ```

  Such negated `meet` clauses work for all context specifications and can be used in arbitrarily nested MU queries, allowing for even more complex co-occurrence filters.

- MU queries are less flexible than standard CQP queries, because they lack the capacity for token-level regular expressions. But MU queries can be much more efficient for determining co-occurrences at relatively large distances, and for finding sequences that consist of one or more very frequent elements followed by a rare item. For example,

  ```
  > MU(meet (meet [pos="NN.*"] "virtue" 2 2) "of" 1 1);
  ```

  is considerably faster than

  ```
  > [pos = "NN.*"] "of" "virtue";
  ```

- This query finds sentences that contain both *one hand* and *other hand*. The MU query returns only the position of *one*, which is then expanded to the complete sentence:

  ```
  > MU(meet (meet "one" "hand" 1 1) (meet "other" "hand" 1 1) s) expand to s;
  ```

- Combinations of `meet` and `union` clauses offer additional flexibility. The following query finds nouns occurring close to a superlative adjective, which can either be synthetic (*strangest*) or analytic (*most extravagant*).

  ```
  > MU(meet [pos="NNS?"] (union [pos="JJS"] (meet [pos="JJ"] "most" -1 -1)) -2 4);
  ```

- Like standard queries, MU queries can be used as subquery filters (followed by `!`) or combined with a `cut` and/or `expand` clause. However, other elements of standard queries are not supported: labels, target markers (`@`), zero-width assertions (obviously), global constraints (after `::`), alignment constraints and `within` clauses.

8.5 TAB queries

- new in v3.4.12: **tabular** (TAB) queries are an obscure and formerly undocumented feature of CQP. They were dysfunctional for a long time, but have now been resurrected. The implementation was originally considered experimental, but is considered stable with full long-term support for CWB 3.5.

- A TAB query starts with the keyword `TAB` and matches a sequence of one or more token patterns with optional flexible gaps. In its simplest form, it corresponds to a standard query matching a fixed sequence of tokens, but is often executed faster. Compare standard query

  ```
  > "in" "due" "course";
  ```

  with the much more efficient TAB query

  ```
  > TAB "in" "due" "course";
  ```

  This query is both simpler and faster than the MU version given in Sec. 8.4.

- The most substantial performance gains are achieved for sequences that start with very frequent items and end in a selective token pattern, e.g.

  ```
  > TAB [pos = "DT"] [pos = "JJ.*"] "tea";
  ```

  TAB queries cannot be used as a general optimization for standard queries, though, because the individual elements cannot be made optional or repeated with a quantifier. It is also not possible to specify alternatives (`|`) within a TAB query (but you can run multiple queries and take the `union` of the results).

- The main purpose of tabular queries is to match sequences with flexible gaps. The following two-word TAB query finds *cats* followed by *dogs* with a gap of up to two intervening tokens:

  ```
  > TAB "cats" {0,2} "dogs";
  ```

  It is equivalent to the standard query

  ```
  > "cats" []{0,2} "dogs";
  ```

  but keep in mind that `TAB "cats" []{0,2} "dogs";` would mean something entirely different![21]

- Gaps can be specified using any of the repetition operators familiar from standard queries

op.	*gap size*
`?`	0 or 1 token
`*`	0 or more tokens
`+`	1 or more tokens
`{`*n*`}`	exactly *n* tokens
`{`*n*`,`*k*`}`	between *k* and *n* tokens
`{`*n*`,}`	at least *n* tokens
`{,`*k*`}`	up to *k* tokens (same as `{0,`*k*`}`)

  All gap specifications behave as if the repetition operator had been applied to a matchall (`[]`) in a standard query.

[21] Namely, *cats* followed by an arbitrary token, followed by a gap of up to two tokens, followed by *dogs*. Entering this command will print an error message because matchall patterns are not allowed in TAB queries.

- TAB queries can additionally be restricted by a *within* clause. For example, the query

  ```
  > TAB "girl" {2} "girl";
  ```

  finds a repetition of the noun *girl* after exactly two intervening tokens, but many of the matches cross a sentence boundary. In order to discard these matches, change the query to

  ```
  > TAB "girl" {2} "girl" within s;
  ```

- TAB queries do not support different matching strategies, but always use an early match principle similar to the default setting of standard CQP queries (regardless of the value of the `MatchingStrategy` option) For example, the query

  ```
  > TAB [pos = "JJ"] {,5} [pos = "NN"];
  ```

  will only match the underlined part of the phrase *a small and very old train station*. It cannot be configured to return the shortest (*old train*) or longest (*small and very old train station*) match.

- TAB queries always return the full range of tokens containing the specified items. Individual items *cannot* be marked in any way (i.e. neither as target pattern nor with labels), due to limitations of the current CQP implementation.

- When composing more complex TAB queries, it is important to understand how their "greedy search"[22]approach works, since its results may be different from the corresponding standard CQP queries.

  - for every possible start position, i.e. each match of the first token pattern
  - scan for a match of the second token pattern within the specified range
  - greedily fix the first such match that is encountered
    (i.e. the search algorithm commits to this corpus position being part of the full match)
  - starting from this position, scan for a match of the third token pattern
  - greedily fix the first such match that is encountered
  - etc.

  If a complete match is found, CQP continues with the next possible start position, so there can be at most one match for each start position. In addition, nested matches are discarded as in standard CQP queries (hence *old train* above is actually matched by the algorithm, but then discarded as a nested match).

- Always keep in mind that CQP does not perform an expensive combinatorial search to consider other matches of token patterns that might also fall within the specified ranges! If a greedily selected match item does not lead to a complete match, but a later item for the same token pattern would have, the correct solution will not be found.

- As a concrete example, consider the sentence

  Fortunately , we had $\underline{time}_{\mathrm{A}}$ $\underline{for}_{\mathrm{B1}}$ *delicious pastries and* $\underline{for}_{\mathrm{B2}}$ $\underline{coffee}_{\mathrm{C}}$.

  The TAB query

  ```
  > TAB "time" * "for" ? "coffee" within s;
  ```

  would not match this sentence because `"for"` is greedily fixed to the first available token B1, for which C is not in range. The corresponding standard query

  ```
  > "time" []* @"for" []? "coffee" within s;
  ```

  considers both options, on the other hand, and matches the range *A*. . . *C*, with the target anchor (`@`) set to *B2*.

[22] Note that *greedy* is used in a different sense here than the "greedy matching" of regular expression quantifiers.

- There are two special cases in which TAB queries are guaranteed to find every early match that satisfies the gap specifications:

  1. All gaps have a fixed size (`{n}`), which can be different for each gap. This includes in particular the case where the token patterns are directly adjacent.

     ```
     > TAB "Mr" {1} "Mrs" [pos = "N.*"];
     > TAB "in" "due" "course";
     ```

  2. All gaps are specified as `*` and the search range is only restricted by a `within` clause. Note that `*` and fixed-size gaps (even direct adjacency) must not be mixed in this case.

     ```
     > TAB "one" * "two" * "three" within s;
     ```

8.6 Numbered target markers

The numbered target markers feature, introduced with CQP v3.4.16, provides a work-around solution for marking up to 10 anchor positions in a finite-state query (with support from wrapper scripts, as discussed below).

- In addition to the implicit `match` and `matchend` anchors, CQP queries allow a single additional token pattern to be marked with an `@` sign, setting the `target` anchor to the matching corpus position. If multiple target markers are specified, the one encountered last during query evaluation "wins".

- Users would quite often like to mark multiple positions, however. Consider the query below, which has three additional tokens of interest (adverb, first adjective, second adjective); only one of them can be marked with `@`.

  ```
  > [pos="DT"] [pos="RB"] [pos="JJ"] [pos="JJ"] [pos="N.*"];
  ```

- It is now possible to mark up to 10 **potential targets** with numbered markers `@0` ... `@9`. Only two of them are active at a given time, controlled by the user options `AnchorNumberTarget` (`ant`) and `AnchorNumberKeyword` (`ank`).

  ```
  > [pos="DT"] @0[pos="RB"] @1[pos="JJ"] @2[pos="JJ"] [pos="N.*"];
  ```

- The query above will mark the adverb position as `target` and the first adjective as `keyword`, because `@0` and `@1` are active by default. Re-run the query after changing `AnchorNumberTarget` in order to mark the second adjective as target, instead of the adverb.

  ```
  > set AnchorNumberTarget 2;
  > [pos="DT"] @0[pos="RB"] @1[pos="JJ"] @2[pos="JJ"] [pos="N.*"];
  ```

- It is invalid to map both anchors to the same numbered target, and the `set` command will reject such a change with an error message. Scripts therefore need to keep track of the current settings when making changes; it is recommended always to map the two anchors to non-overlapping pairs of anchors (e.g. `@0` and `@1`, `@2` and `@3`, etc.).

- The main purpose of the new feature is to enable wrapper scripts to simulate up to 10 target anchor positions in a way that is fully compatible with CQP macros and does not require any custom extensions, so queries can be tested in an interactive CQP session or in CQPweb.

- Since the wrapper cannot know which numbered target markers are used in a query (especially with nested macros), every query has to be run 5 times, collecting the `target` and `keyword` positions from each run and combining them into a single table at the end. In CQP v3.4.31+,

the extra runs can be executed as anchored queries in order to reduce the overhead for complex search patterns in large corpora.[23]

- Here is an example how a wrapper might process such a query:[24]

```
> set AnchorNumberTarget 0; set AnchorNumberKeyword 1;
> Result = [pos="DT"] @0[pos="RB"] @1[pos="JJ"] @2[pos="JJ"] [pos="N.*"];
> Temp = <<Result/>> ( [pos="DT"] @0[pos="RB"] @1[pos="JJ"] @2[pos="JJ"] [pos="N.*"] );
> dump Temp;      # obtain matching ranges and first two target anchors @0 and @1
> set AnchorNumberTarget 2; set AnchorNumberKeyword 3;
> Temp = <<Result/>> ( [pos="DT"] @0[pos="RB"] @1[pos="JJ"] @2[pos="JJ"] [pos="N.*"] );
> dump Temp;      # obtain next two target anchors @2 and @3
⋮
> set AnchorNumberTarget 8; set AnchorNumberKeyword 9;
> Temp = <<Result/>> ( [pos="DT"] @0[pos="RB"] @1[pos="JJ"] @2[pos="JJ"] [pos="N.*"] );
> dump Temp;      # obtain last two target anchors @8 and @9
```

- NB: The wrapper script should not forget to reset `AnchorNumberTarget` and `AnchorNumberTarget` to their previous or default settings (and to re-activate the main corpus if anchored subqueries are used before CQP v3.4.31).

- For backward compatibility, the plain `@` marker unconditionally sets the `target` anchor, regardless of the value of `AnchorNumberTarget`. Queries should never mix `@` with the numbered potential target markers.

- All target markers can be followed by an optional colon `:` similar to the notation used for labels (including `@:` for the unconditional target).

  ```
  > [pos="DT"] @0:[pos="RB"] @1:[pos="JJ"] @2:[pos="JJ"] [pos="N.*"];
  ```

  This simplifies CQP macros, which do not have to distinguish between label names and target markers passed as arguments.

- CQP macros intended to be embedded in more complex queries should always use parameterized target markers. Consider the following definition of a macro matching simple noun phrases:

```
MACRO np($0=MarkNoun $1=MarkAdj)
(
  [pos="DT"]? [pos="RB"]? $1[pos="JJ.*"]* $0[pos="NN.*"]
)
;
MACRO np($0=MarkNoun)
/np["$0", ""]
;
MACRO np(0)
```

[23] In earlier versions of CQP, anchored subqueries can be used by activating `Result` as a subcorpus and anchoring the additional queries with `<match>` in initial position. This is less convenient and can lead to issues if there are overlapping matches (which are not allowed for subcorpora).

[24] Cautious programmers might want to verify that the matching ranges of each `dump Temp;` are identical before discarding the first two columns of the `dump`. Alternatively, `tabulate Temp target, keyword;` can be used from the second iteration in order to avoid redundant information. It would seem to be more efficient to obtain the matching ranges and first two anchors from `dump Result;` and save one iteration. However, the result sets might not be identical if `Result` contains multiple matches starting at the same corpus position (typically from top-level alternatives in the query), which is not possible for the anchored queries. It is safe to use the faster solution if matching strategy is set to `shortest` or `longest`.

```
/np["", ""]
;
```

- Invocations of the above macro can now specify numbered target markers to be placed on the head noun and last adjective in the NP, respectively. The markers must be quoted as macro arguments; an empty string "" omits the corresponding marker. The two additional macros simulate default values (no marker) for both arguments.
- As an example, let us search for the pattern "NP Prep NP" and extract the head noun and adjective (if present) of the first NP, as well as the preposition and head noun of the PP (e.g. *this festive season of the year*).

  ```
  > /np["@1", "@0"] @2[pos="IN"] /np["@3"];
  ```

  This query applies `@0` to the adjective of the first NP, `@1` to its head noun, `@2` to the preposition and `@3` to the head noun of the PP.
- Note that pairs of markers can be used to mark the start and end of a sub-pattern of flexible length. It is often convenient to enclose the sub-pattern in parentheses (if necessary) and use zero-width assertions to set the markers. In order to extract multi-word noun compounds, we could change our NP pattern as follows:

  ```
  MACRO np($0=StartNoun $1=EndNoun $2=MarkAdj)
  (
    [pos="DT"]? [pos="RB"]? $2[pos="JJ.*"]* $0[::] $1[pos="NN.*"]+
  )
  ;
  ```

  Due to the matching rules, the marker indicated by `$1` will be set to the last noun in the sequence (which may be identical to the first noun marked by `$0`). Keep in mind that `/np[]` only matches a single noun token unless it is embedded in a larger query or the matching strategy is set to `longest`.

8.7 Region elements and ad-hoc annotation

- new in v3.4.31: CQP now has limited support for **ad-hoc structural annotation**, which does not need to be indexed in the form of an s-attribute.[25]
- The core element of the new syntax is `<<`*`name`*`>>` to match an entire token span, e.g. a region of the s-attribute *name*. We will therefore refer to this item as a **region element**.[26] The two queries below are fully equivalent:

  ```
  > "dine" [pos = "IN"] <<np>>;
  > "dine" [pos = "IN"] <np> []+ </np>;
  ```
- The implementation of `<<np>>` in the first of these queries is almost exactly the same as that used for the second query: CQP will step through all tokens of the region until it reaches its end, so both queries have very similar execution times. But a key difference is that `<<np>>` doesn't allow any constraints on the content of the region or on annotation values of the s-attribute; it also needs less trickery behind the scenes to match up start and end tags and is therefore less prone to issues with obscure corner cases of CQP queries.[27]

[25] The new feature is only available in standard "finite state" queries, of course, because both MU and TAB queries are entirely token-based and cannot integrate structural annotation.

[26] In contrast to the use in queries of actual XML tags, there can be whitespace between the brackets and the element name, e.g. `<< np >>`.

[27] See the document *Finite State Queries in CWB3*, available in the CWB subversion repository, for a discussion of these issues, as well as a sketch of the implementation of ad-hoc annotation.

- Structural annotation can sometimes be generated with a (usually rather complex) CQP query, e.g. noun phrase chunking with a query such as

```
> NP = (?longest) [pos = "DT"]?
       ( [pos = "JJ.*"] ([word=",|and|or" | pos="RB"]* [pos = "JJ.*"])* )?
       [pos = "NNS?"]+;
```

  It will often be desirable to reuse such noun chunks in other queries, and the customary recommendation is to define a suitable macro `/np[]` (cf. Sec. 6.4), which can then be integrated into queries for larger patterns such as "NP *about* NP":

```
> /np[] "about" /np[];
```

  This approach results in extremely complex and slow queries that are difficult to debug. It also makes it necessary to manage a macro library in order to use such query fragments across multiple CQP sessions (with different versions of the macro for different tagsets).

- One possibility – which has already been used to implement YAC, a cascaded finite-state chunk parser for German (see Sec. 1.1) – is to `dump` the query result and index it as a structural attribute in the corpus with `cwb-s-encode`. The query for "NP *about* NP" can then be executed more efficiently as

```
> <<np>> "about" <<np>>;
```

  in the new region element syntax. This approach has several drawbacks:

  – the end user must have write access to the corpus data directory and registry file, and a suitable wrapper infrastructure has to be implemented;
  – registry entries and `show cd;` may be cluttered with a large number of undocumented s-attributes if multiple users have write access to the corpus;
  – CQP has to be restarted in order to recognise the newly indexed s-attributes;
  – a query result may contain overlapping and even nested spans,[28] which are not allowed in s-attributes; care has to be taken to discard the problematic matches beforehand.

- Region elements provide a convenient and powerful alternative: `<<NP>>` interprets the named query result `NP` as structural annotation and matches any of its `match`...`matchend` spans.[29] We can thus immediately search for "NP *about* NP" with the query

```
> <<NP>> "about" <<NP>>;
```

  Named query results thus provide flexible **ad-hoc annotation**. They can directly be used (and updated) as temporary annotation in a running CQP session, but can also be persisted to disk (`save NP;`) and will be loaded on-demand in a new session. Ad-hoc annotation does not need special file access permissions and is private to each user, but can also easily be shared with other users (by copying the saved query results).

- If naming conventions are observed (lowercase for s-attribute names, CamelCase for named query results), there can be no conflict between the two uses of region elements. Otherwise, a query result always takes precedence, since it is the main application of the new feature.

- To search for spans between other anchors in the query result, make a copy of the NQR and use `set` to modify the matching span. Assuming that (for instance) a target anchor has been set on the first noun in `NP`,[30] we would type

[28] depending on the matching strategy used, and always for the `union` of query results

[29] For technical reasons, a similar approach involving XML tags `<NP>` and `</NP>` was not feasible; nor is it possible to select arbitrary spans (e.g. `target`...`matchend`) from the query result.

[30] It is left as an exercise to the reader to set this anchor in the query above, most conveniently with `@[::]`.

```
> PreNominal = NP;
> set PreNominal matchend target[-1] !;
```

in order to be able match the prenominal elements of a noun phrase with `<<PreNominal>>`. The `set ... !;` operation automatically discards any matches consisting only of nouns, because then `target[-1]` precedes the `match` anchor (resulting in an invalid span).[31]

- Of course, region elements can be repeated with quantifiers, included in a complex subexpression, etc. As a simple example, consider the query

```
> <<NP>> ( [pos = "IN|TO"] <<NP>> ){5};
```

- Let us now consider a query result `NamedEntity` that contains tentatively identified named entities in the corpus:

```
> NamedEntity = (?traditional)
                <np> []* [pos = "NPS?"] </np> | <np1> []* [pos = "NPS?"] </np1>;
```

where the `traditional` matching strategy enables nested matches for complex named entities such as the [*Lord Mayor of* [*London*]]. We can then find a preposition followed by a tentative named entity:

```
> (?traditional) [pos = "IN|TO"] <<NamedEntity>>;
```

taking all nested and/or overlapping NE candidates into account, which would not be possible with an s-attribute. Again, the `traditional` matching strategy enables nested matches (e.g. look for the line *in the presence of the Ghost of Christmas*).

- Such named entity candidates are often generated by an **external annotation tool** and would traditionally be encoded as an s-attribute. In order to allow for overlapping and nested NEs, they can be `undump`ed into a NQR instead. If there is an additional categorisation of NEs, multiple NQRs (`NEperson`, `NEloc`, `NEorg`, ...) can be created for the different classes.

- **Labels and target markers** can be set on the first and last token of the range matched by a region element, using the same notation as for token expressions. Those for the first token are inserted immediately after the opening `<<`, those for the last token immediately before the closing `>>`. It is recommended (and sometimes necessary) to separate them from the element name with whitespace in order to avoid parsing errors.

- The example below uses labels to ensure that the second NP is at least 3 tokens long and it sets target markers on the end of the first NP and the start of the second (with the other boundaries implicity given by the `match` and `matchend` anchors).

```
> <<NP @0>> "after" <<@1 a: NP b:>> :: distabs(a, b) >= 2;
```

- **Zero-width** region elements are indicated by a slash after the element name, e.g. `<<NP/>>`. They match at the *start* of a suitable span, but do not consume any tokens. Because there might be multiple spans at the same position, a label or target marker can only be set on the first token of the span. Such zero-width region elements are of little use for s-attributes: `<<np/>>` behaves almost exactly like the XML start tag `<np>`, except that it allows a label to be set and does not match up with a corresponding end tag `</np>`.

- The main purpose of zero-width region elements is to enable **anchored queries**, which take their potential starting point from a previous query result. This is similar to the earlier strategy of running a subquery starting with a `<match>` anchor (see Sec. 6.3), but much more convenient; moreover it allows for overlapping matches.

[31] This did not work correctly before CQP v3.4.31, which also introduced support for offsets.

- A query with a highly selective element near the end such as

  ```
  > "the"%c [pos="JJ.*"]{3,} [lemma="creature"];
  ```

  can now conveniently be sped up[32] by a pre-filtering strategy:

  ```
  > Cand = MU(meet "the"%c [lemma="creature"] 1 5);
  > <<Cand/>> "the"%c [pos="JJ.*"]{3,} [lemma="creature"];
  ```

8.8 Easter eggs

- Starting with version 3.0 of the Corpus Workbench, CQP comes with a built-in *regular expression optimiser*; this optimiser detects simple regular expressions commonly used for prefix, suffix or infix searches such as

  ```
  > "under.+";
  > ".+ment";
  > ".+time.+";
  ```

  and replaces the normal regexp evaluation with a highly efficient Boyer-Moore search algorithm.

- the optimiser will also recognise some slightly more complex regular expressions; if you want to test whether a given expression can be optimised or not, switch on debugging output with

  ```
  > set CLDebug on;
  ```

- Some beta releases of CQP may contain hidden optimisations and/or other features that are disabled by default because they have not been tested thoroughly; such hidden features will usually be documented in the release notes and can be activated with the option

  ```
  > set Optimize on;
  ```

- The official LTS releases v3.0 and v3.5 of CQP have *no* hidden features.

[32] More than 5 times faster on this author's laptop computer.

A Appendix

A.1 Summary of regular expression syntax

At the character level, CQP supports regular expressions using one of two regex libraries:

CWB 3.0: Uses POSIX 1003.2 regular expressions (as provided by the system libraries). A full description of the regular expression syntax can be found on the *regex(7)* manpage.

CWB 3.5: Uses PCRE (*Perl Compatible Regular Expressions*). A full description of the regular expression syntax can be found on the *pcrepattern(3)* manpage; see also http://www.pcre.org/.

Various books such as *Mastering Regular Expressions* give a gentle introduction to writing regular expressions and provide a lot of additional information. There are also many tutorials to be found online using Your Favourite Web Search Engine™.

- A regular expression is a concise descriptions of a set of character strings (which are called *words* in formal language theory). Note that only certain sets of words with a relatively simple structure can be represented in such a way. Regular expressions are said to *match* the words they describe. The following examples use the notation:

  `<reg.exp.>` → $word_1$, $word_2$, ...

  to indicate that the regular expression before the arrow matches the word or words after the arrow. In many programming languages, it is customary to enclose regular expressions in forward slashes (`/`). CQP uses a different syntax: regular expressions are written as (single- or double-quoted) strings. The examples below omit any delimiters.

- Basic syntax of regular expressions

  - letters and digits are matched literally (including all non-ASCII characters)
    `word` → *word*; `C3PO` → *C3PO*; `déjà` → *déjà*
  - `.` matches any single character ("matchall")
    `r.ng` → *ring, rung, rang, rkng, r3ng, ...*
  - character set: `[...]` matches any of the characters listed
    `moderni[sz]e` → *modernise, modernize*
    `[a-c5-9]` → *a, b, c, 5, 6, 7, 8, 9*
    `[^aeiou]` → *b, c, d, f, ..., 1, 2, 3, ..., ä, à, á, ...*
  - repetition of the preceding element (character or group):
    `?` (0 or 1), `*` (0 or more), `+` (1 or more), `{`n`}` (exactly n), `{`n`,`m`}` ($n \ldots m$)
    `colou?r` → *color, colour*; `go{2,4}d` → *good, goood, gooood*
    `[A-Z][a-z]+` → "regular" capitalised word such as *British*
  - grouping with parentheses: `(...)`
    `(bla)+` → *bla, blabla, blablabla, ...*
    `(school)?bus(es)?` → *bus, buses, schoolbus, schoolbuses*
  - `|` separates alternatives (use parentheses to limit scope)
    `mouse|mice` → *mouse, mice*; `corp(us|ora)` → *corpus, corpora*

- Complex regular expressions can be used to model (regular) inflection:

  - `ask(s|ed|ing)?` → *ask, asks, asked, asking*
    (equivalent to the less compact expression `ask|asks|asked|asking`)
  - `sa(y(s|ing)?|id)` → *say, says, saying, said*

  - `[a-z]+i[sz](e[sd]?|ing)` → any form of a verb with *-ise* or *-ize* suffix

- Backslash (\) "escapes" special characters, i.e. forces them to match literally
  - `\?` → *?*; `\(\)` → *()*; `.{3}` → ...; `\$\.` → *$.*
  - `\^` and `\$` must be escaped although `^` and `$` anchors are not useful in CQP

A.2 Part-of-speech tags and useful regular expressions

The English PENN tagset (DICKENS)

NN	Common noun, singular or mass noun
NNS	Common noun, plural
NP, NPS	Proper noun, singular/plural
N.*	Matches any common or proper noun
PP.*	Matches any pronoun (personal or possessive)
JJ	Adjective
JJR, JJS	Adjective, comparative/superlative
VB.*	Matches any verbal form
VBG, VGN	Present/past participle
RB	Adverb
RBR, RBS	Adverb, comparative/superlative
MD	Modal
DT	Determiner
PDT	Predeterminer
IN	Preposition, subordinating conjunction
CC	Coordinating conjunction
TO	Any use of "to"
RP	Particle
WP	Wh-pronoun
WDT	Wh-determiner
SENT	Sentence-final punctuation

The German STTS tagset (GERMAN-LAW)

NN	Common noun (singular or pluarl)
NE	Proper noun (singular or plural)
N.	Matches any nominal form
PP.*	Matches any pronoun (personal or possessive)
ADJA	Attributive adjective
ADJD	Predicative adjective (also when used adverbially)
ADJ.	Matches any adjectival form
VV.*	Matches any full verb
VA.*	Matches any auxilliary verb
VM.*	Matches any modal verb
V.*	Matches any verbal form
ADV	Adverb
ART	Determiner
APPR	Preposition
APPRART	Fused preposition and determiner
KO.*	Matches any conjunction
TRUNC	Truncated word (e.g. "unter-")
\$\.	Sentence-final punctuation
\$,	Sentence-internal punctuation

A.3 Annotations of the tutorial corpora

English corpus: `DICKENS`

- Positional attributes (token annotations)

`word`	word forms ("plain text")
`pos`	part-of-speech tags (Penn Treebank tagset)
`lemma`	base forms (lemmata)

- Structural attributes (XML tags)

`novel`	individual novels
`novel_title`	title of the novel
`book`	when text is subdivided into books
`book_num`	number of the book
`chapter`	chapters
`chapter_num`	number of the chapter
`chapter_title`	optional title of the chapter
`title`	encloses title strings of novels, books, and chapters
`p`	paragraphs
`p_len`	length of the paragraph (in words)
`s`	sentences
`s_len`	length of the sentence (in words)
`np`	noun phrases
`np_h`	head lemma of the noun phrase
`np_len`	length of the noun phrase (in words)
`pp`	prepositional phrases
`pp_h`	functional head of the PP (preposition)
`pp_len`	length of the PP (in words)

German corpus: `GERMAN-LAW`

- Positional attributes (token annotations)

`word`	word forms ("plain text")
`pos`	part-of-speech tag (STTS tagset)
`lemma`	base forms (lemmatised forms)
`alemma`	ambiguous lemmatisation (*feature set*, see examples in Section 6.6)
`agr`	noun agreement features (*feature set*, see examples in Section 6.6)

Each agreement feature has the form *ccc*:*g*:*nn*:*ddd* with

ccc	= case	(`Nom`, `Gen`, `Dat`, `Akk`)
g	= gender	(`M`, `F`, `N`)
nn	= number	(`Sg`, `Pl`)
ddd	= determination	(`Def`, `Ind`, `Nil`)

- XML elements representing syntactic structure

`<s>`	sentences
`<pp>`	prepositional phrases
`<np>`	noun phrases
`<ap>`	adjectival phrases
`<advp>`	adverbial phrases
`<vc>`	verbal complexes
`<cl>`	subclauses

- Key-value pairs in XML start tags

```
<s len="..">
<pp f=".." h=".." agr=".." len="..">
<np f=".." h=".." agr=".." len="..">
<ap f=".." h=".." agr=".." len="..">
<advp f=".." len="..">
<vc f=".." len="..">
<cl f=".." h=".." vlem=".." len="..">
```

  `len` = length of region (in tokens)
  `f` = properties (feature set, see next page)
  `h` = lexical head of phrase (`<pp_h>`: "*prep* : *noun*")
  `agr` = nominal agreement features (feature set, partially disambiguated)
  `vlem` = lemma of main verb

- Properties of syntactic structures (`f` key in start tags)

`<np_f>`	`norm` ("normal" NP), `ne` (named entity), `rel` (relative pronoun), `wh` (wh-pronoun), `pron` (pronoun), `refl` (reflexive pronoun), `es` (*es*), `sich` (*sich*), `nodet` (no determiner), `quot` (in quotes), `brac` (in parentheses), `numb` (list item), `trunc` (contains truncated nouns), `card` (cardinal number), `date` (date string), `year` (specifies year), `temp` (temporal), `meas` (measure noun), `street` (address), `tel` (telephone number), `news` (news agency)
`<pp_f>`	same as `<np_f>` (features are projected from NP) + `nogen` (no genitive modifier)
`<ap_f>`	`norm` ("normal" AP), `pred` (predicative AP), `invar` (invariant adjective), `vder` (deverbal adjective), `quot` (in quotes), `pp` (contains PP complement), `hypo` (uncertain, AP was conjectured by chunker)
`<advp_f>`	`norm`, `temp` (temporal adverbial), `loc` (locative adverbial), `dirfrom` (directional source), `dirto` (directional path)
`<vc_f>`	`norm`, `inf` (infinitive), `zu` (*zu*-infinitive)
`<cl_f>`	`rel` (relative clause), `subord` (subordinate clause), `fin` (finite), `inf` (infinitive), `comp` (comparative clause)

A.4 Reserved words in the CQP language

Reserved words cannot be used as identifiers (i.e. corpus handles, attribute names, query names or labels) in CQP queries and interactive commands.

- new in CQP v3.4.13: Reserved words can now be used as identifiers if they are quoted between backticks whenever mentioned.

```
> show +‘no‘;’
> ‘MU‘ = [lemma = "meeting|union"];
> group ‘MU‘ match lemma;
> [‘size‘ = "\d{5,}"];
```

- Quotes are neither required nor allowed in label definitions and qualified label references.

```
> left: [pos = "NN"] "after" right: [pos = "NN"] :: left.lemma = right.lemma;
```

- The usual rules for identifiers still apply, so e.g. `size ‘007‘;` will not be accepted.

The full list of reserved words is>:

```
a:  asc ascending
b:  by
c:  cat cd collocate contains cut
d:  def define delete desc descending diff difference discard dump
e:  exclusive exit expand
f:  farthest foreach
g:  group
h:  host
i:  inclusive info inter intersect intersection
j:  join
k:  keyword
l:  left leftmost
m:  macro maximal match matchend matches meet MU
n:  nearest no not NULL
o:  off on
r:  randomize reduce RE reverse right rightmost
s:  save set show size sleep sort source subset
t:  TAB tabulate target target[0-9] to
u:  undump union unlock user
w:  where with within without
y:  yes
```

A.5 Full list of CQP options

This appendix lists all the CQP options that can be changed using the `set` command during a CQP session. There are many more configurable settings, but they cannot be set by the user during a session. Instead, they must be set when CQP is invoked (see `cqp -h` for more).

A.5.1 Boolean options

Boolean (true/false) options are set as `on` or `off`, or alternatively as `yes` or `no`. Their present value is always displayed as `yes` or `no`.

Abbr.	Option	Summary
	AutoSave	Automatically save subcorpora/query results to disk
as	AutoShow	Automatically display query results
sub	AutoSubquery	Automatically enter subquery mode by activating new subcorpus/query result on creation
col	Colour	Enable colour highlighting
es	ExternalSort	Use external helper program to sort queries
h	Highlighting	Highlight hits (and target/keyword anchors) within KWIC output
o	Optimize	Enable experimental optimisations
p	Paging	Use external pager program to display KWIC
pp	PrettyPrint	Format output neatly for human readers
pb	ProgressBar	Show the progress of query execution
	SaveOnExit	Save all unsaved subcorpora/query results when CQP exits
sta	ShowTagAttributes	Display key-value pairs in XML tags (in KWIC)
st	ShowTargets	Print identifier numbers for *target*(0) and *keyword*(1) in KWIC; same effect as `show +targets`
sr	StrictRegions	Make XML start/end tags within query match a single region
	Timing	Print time taken to execute queries
wh	WriteHistory	Write all commands entered to a history file

A.5.2 Integer options

Integer options are set to a numeric value (a whole number). The valid range is usually restricted and will be checked when the option is set. Keep in mind that numeric values must *not* be enclosed in quotation marks.

Abbr.	Option	Summary
ant	AnchorNumberTarget *new in v3.4.17*	Which numeric target marker is currently active as the referent of `target` (see Sec. 8.6); valid range is 0 . . . 9, default value: 0 (i.e. the marker `@0`)
ank	AnchorNumberKeyword *new in v3.4.17*	Which numeric target marker is currently active as the referent of `keyword` (see Sec. 8.6); valid range is 0 . . . 9, default value: 1 (i.e. the marker `@1`)

A.5.3 String options

String options contain a line of data that has some effect on or role in CQP's operation. When setting a string option, it must be enclosed in quote marks - which will *not* form part of the actual option value.

Abbr.	Option	Summary
	AttributeSeparator	Override the default separator / between KWIC token annotations
	new in v3.4.18	with a user-defined string (set to "" to return to default)
dd	DataDirectory	Directory to be used for saving/loading query results
da	DefaultNonbrackAttr	P-attribute used to match regular expressions which appear alone outside [...]
esc	ExternalSortCommand	Shell command to invoke external sort program
hf	HistoryFile	Location where command history will be saved
ld	LeftKWICDelim	Delimiter before the "hit" on the KWIC line
pg	Pager	Shell command to invoke external pager program
ps	PrintStructures	List of s-attributes to print within KWIC (delimited by space, comma, or full stop)
r	Registry	Directory from which registry files are loaded, determining what corpora are presently available
rd	RightKWICDelim	Delimiter before the "hit" on the KWIC line
	StructureDelimiter	Delimiter which appears before and after all s-attribute
	new in v3.4.25	tags in the concordance (default is empty string)
	TokenSeparator	Override the default separator (space) between KWIC tokens
	new in v3.4.24	with a user-defined string (set to "" to return to default)

A.5.4 Enumerated options

Enumerated options - a sub-type of string - can only be set to one of a fixed list of values. In the case of `PrintOptions`, each item on the list sets one of a set of Boolean output formatting options either on or off, and some items are synonyms.

Abbr.	Option	Summary
ms	MatchingStrategy	Match strategy used for token-level regular expressions. One of: *traditional, shortest, standard, longest.*
pm	PrintMode	Print format used to display KWIC ouput. One of: *ascii, sgml, html, latex.*
po	PrintOptions	Operation to perform on the print options setup. One of: *wrap, nowrap, table/tbl, notable/notbl, header/hdr, noheader/nohdr, border/bdr, noborder/nobdr, number/num, nonumber/nonum.*

A.5.5 Context options

Context options - a sub-type of string - set the width of the left or right context in the KWIC display (or both) in units of characters, words, or s-attribute regions.

Abbr.	Option	Summary
c	Context	Pseudo-option to set LeftContext and RightContext to the same value
lc	LeftContext	Set width of left context
rc	RightContext	Set width of right context