The CQP Query Language Tutorial

(CWB version 2.2.b90)

Stefan Evert
stefan.evert@uos.de

10 July 2005

Contents

1 Introduction **3**
1.1 The IMS Corpus Workbench (CWB) . . . 3
1.2 The CWB corpus data model . . . 5
1.3 Corpora used in the tutorial . . . 7

2 Basic CQP features **8**
2.1 Getting started . . . 8
2.2 Searching for words . . . 8
2.3 Display options . . . 9
2.4 Useful options . . . 10
2.5 Accessing token-level annotations . . . 11
2.6 Combinations of attribute constraints: Boolean expressions . . . 12
2.7 Sequences of words: token-level regular expressions . . . 12
2.8 Example: finding "nearby" words . . . 12
2.9 Sorting and counting . . . 14

3 Working with query results **15**
3.1 Named query results . . . 15
3.2 Saving data to disk . . . 15
3.3 Anchor points . . . 16
3.4 Frequency distributions . . . 18
3.5 Set operations with named query results . . . 18
3.6 The `set target` command . . . 19

4 Labels and structural attributes **21**
4.1 Using labels . . . 21
4.2 Structural attributes . . . 22
4.3 Structural attributes and XML . . . 23
4.4 XML document structure . . . 24

5 Advanced CQP features **26**
5.1 The matching strategy 26
5.2 Word lists 26
5.3 Subqueries 28
5.4 The CQP macro language 29
5.5 CQP macro examples 31
5.6 Feature set attributes (GERMAN-LAW) 32

6 Undocumented CQP **35**
6.1 Zero-width assertions 35
6.2 Labels and scope 36
6.3 Running CQP as a backend 37
6.4 Exchanging corpus positions with external programs 38
6.5 Generating frequency tables 40
6.6 Easter eggs 42

A Appendix **43**
A.1 Summary of regular expression syntax 43
A.2 Part-of-speech tags and useful regular expressions 44
A.3 Annotations of the tutorial corpora 45
A.4 Reserved words in the CQP language 47

A.4 Reserved words in the CQP language

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`

- 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)

1 Introduction

1.1 The IMS 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)
  - CWB version 3.0 to be released in early 2005 (pre-release versions have been shipped since October 2001)
- 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) (stand-alone system for extraction of Dutch legal terminology)
  - 1999 – 2003: Implementation of YAC chunk parser for German (PhD thesis Kermes, annotates results of CQP queries in the corpus)
  - 2001 – 2003: Transferbereich 32 (financed by the DFG) (applications in computational lexicography)
- Some external applications of the IMS Corpus Workbench
  - AC/DC project at the Linguateca centre (SINTEF, Oslo, Norway) (on-line access to a 180 M word corpus of Portuguese newspaper text) `http://acdc.linguateca.pt/cetempublico/`
  - CorpusEye (user-friendly CQP) in the VISL project (SDU, Denmark) (on-line access to annotated corpora in various languages) `http://corp.hum.sdu.dk/corpustop.html`
  - SSLMIT Dev Online services (SSLMIT, University of Bologna, Italy) (on-line access to 380 M words of Italian newspaper text) `http://sslmitdev-online.sslmit.unibo.it/`
  - CucWeb project (UPF, Barcelona, Spain) (Google-style access to 208 million words of text from Catalan Web pages) `http://ramsesii.upf.es/cucweb/`
  - Corpógrafo environment at the Linguateca centre (FLUP, Porto, Portugal) (an easy-to-use Web-based environment for corpus research) `http://www.linguateca.pt/corpografo/`

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

Technical aspects

- CWB uses proprietary 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 500 million words, depending on annotations
  - text data and annotations cannot be modified after encoding (but it is possible to add new annotations or overwrite existing ones)
  - assumes Latin-1 encoding, but compatible with other 8-bit ASCII extensions (Unicode text in UTF-8 encoding can be processed with some caveats)
- 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:
  - SUN Solaris 2.8 (Sparc processors)
  - Linux 2.4+ (Intel i386 and compatible processors)
  - Corpus data format is platform-independent
  - Source code should compile on most POSIX-compliant 32-bit platforms

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

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 5.6)
`agr`	noun agreement features (*feature set*, see examples in Section 5.6)

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`)

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

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 `IDs` 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 predefined 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`).

A Appendix

A.1 Summary of regular expression syntax

At the character level, CQP supports 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. Various books such as *Mastering Regular Expressions* give a gentle introduction to writing regular expressions and provide a lot of additional information.

- 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.>` → $word_1$, $word_2$, ...

  In many programming languages, it is customary to enclose regular expressions in slashes (`/`). CQP uses a different syntax where 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, goood*
    `[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

6.6 Easter eggs

- the pre-release versions of CQP v3.0 include a hidden *regular expression optimiser*; this optimiser detects simple expressions used for prefix, suffix or infix searches such as

  ```
  > "under.+";
  > ".+ment";
  > ".+time.+";
  ```

  and replaces the regexp engine with a highly efficient Boyer-Moore search algorithm

- the regular expression optimiser is activated with the command

  ```
  > set Optimize on;
  ```

  you can watch the optimiser at work by setting

  ```
  > set CLDebug on;
  ```

- the optimiser will be activated by default in the official v3.0 release

1.3 Corpora used in the tutorial

Pre-encoded versions of these corpora are distributed free of charge together with the IMS Corpus Workbench. Perl scripts for encoding the *British National Corpus* (World Edition) can be provided at request.

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 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 (`;`)

- list available corpora

  `> show corpora;`

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

  `> info DICKENS;`

  displays information file associated with the corpus, whose contents may vary; ideally, this should give a description of the corpus composition, a summary of the positional and structural annotations, and a brief overview of annotation codes such as the part-of-speech tagset used

- 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

- note that special characters have to be "escaped" with backslash (\)

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

  "critical" characters are: `. ? * + | ( ) [ ] { } ^ $`

[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.

- 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 the `sort` and `count` commands; without an attribute name, this will print the corpus positions for the selected anchor:

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

  produces exactly the same output as `dump A;` when targets and 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 by the empty string; the same happens when an anchor point is not defined or outside the corpus range (because of an offset)

- a range between to 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 of 5 tokens to the left and right of each match, which can be used to identify collocates of the matching string(s)

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

  note that 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

- 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 `sort` and `count`, a range is considered empty when the end point lies *before* the start point and will always be printed as an empty string

6.5 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 6.3), 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 (cf. Section 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 these two commands 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)
- 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

- 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 works only for the Latin-1 encoding and cannot be deactivated

- additional special escape sequences:

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

- use flags `%c` and `%d` to ignore case / diacritics

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

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
- 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`, and `PgDn`
- 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>`*`;` shows current value
- `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)


```
2591586 2591591
2591593 2591598
```

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 only when command-line editing support is enabled with the `-e` switch, so CWB/Perl and smilar interfaces have to create a temporary file for the undump table; try out the example above with

```
> undump B;
```

further columns for the `target` and `keyword` anchors (in this order) can be selected by adding `with target` or `with target keyword` to the command:

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

- when the rows of the undump table are not sorted in the 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 automatically remembered and will be used by the `cat` and `dump` commands (until it is modified by a new `sort`); 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` (after adjusting the file format), `A` and `B` will be sorted identically
- in many cases, overlapping or unsorted matches are not intentional but rather errors in an automatically generated undump 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";
```

  will abort with an error message (indicating the row number where the error occurred) unless the match ranges in `mydump.tbl` are non-overlapping and sorted in corpus order
- a typical example for the use of `dump` and `undump` is linking CQP queries to corpus metadata stored in an external SQL database; assume that a corpus consists of a large collection of transcribed dialogues, which are represented by `<dialogue>` regions; a rich amount of 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;[8] the following commands generate a dump table with the required information, which can easily be loaded into the database (ignoring the third and fourth column)

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

  corpus queries will often be restricted to a subcorpus by specifying constraints on the metadata; when the metadata constraints have been resolved in the SQL database, they can be translated to the corresponding regions in the corpus (again represented by start and end corpus position); after sorting these regions in ascending order and saving them to a suitable undump file, they are loaded into CQP with an `undump` command; the resulting query result can then be activated as a subcorpus for the ensuing query

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

[8] Of course, it is also possible to establish an indirect link through document IDs, which are annotated as `<dialogue id=XXXX> .. </dialogue>`. Especially when there are a very large number of dialogues in the corpus, the direct link can be much more efficient.

6.4 Exchanging corpus positions with external programs

- an important aspect of interfacing CQP with external tools is access to the corpus positions of query matches (as well as `target` and `keyword` anchors); this is a prerequisite for the extraction of further information about the matches by direct corpus access, and it is the most efficient way to relate query matches to external data structures (e.g. in a SQL database)
- 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;[7] 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 keywords were set for the query, the fourth column is included in the dump table with all values set to `-1`
- the table created by the `dump` command is printed on `stdout` by default (where it is parsed by the CWB/Perl interface); it can also be redirected to a file or pipe; use the following command to create a compressed dump file:

  `> 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; this functionality is provided by the `undump` command, which reads its input from a file or pipe

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

  this command creates a new named query result `B` for the currently activated corpus, or overwrites an existing one
- the format for undump tables is almost identical to the dump format, with two exceptions: the first line must contain a single number specifying the total number of rows in the table (i.e. the number of matches), and the following table has only two columns (for the `match` and `matchend` anchors) by default; the example below is a valid undump file for the `DICKENS` corpus, creating a query result with 5 matches

```
5
20681   20687
379735  379741
1915978 1915983
```

[7] since this command respects the current sort order of the query results, the natural order should first be restored by calling `sort` without a `by` clause One exception is a formatter for a CGI interface to CQP, which usually displays matches in their sorted order.

- 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 (if CQP is run with `-e` option)
  NB: the size of the history file is *not* limited automatically by CQP
- `set AutoShow off;`
  no automatic KWIC display of query results
- `set Optimize on;`
  enable experimental optimisations (sometimes included in beta versions)

2.5 Accessing token-level annotations

- specify p-attribute/value pairs (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
- `%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 find out with the `/codist[]` macro (more on macros in Sections 5.4 and 5.5):

  `> /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:

  ```
  > [(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 conventional regular expressions

- 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.

2.8 Example: finding "nearby" words

- insert optional matchall patterns between words

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

- repeated matchall for longer distances

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


6.3 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` is not automatically read at startup
  - CQP prints its version number after intialisation
  - 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 prints an empty line & flushes 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.;` inserts the line

    ```
    -::-EOL-::-
    ```

    as a marker into CQP's output

- 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

- 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 format that can more easily be parsed by the invoking program; experiment with this setting in an interactive CQP session to learn more about the non-pretty output formats

- the CWB/Perl interfaces 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

6.2 Labels and scope

- returning to the `np_agr` macro from Section 6.1, 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;
  ```

- note that 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


```
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.

- 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 (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

- 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;
```

- 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)
- 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; (left c., by characters)
```

- 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

6 Undocumented CQP

6.1 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 5.5 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 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

- match set members against regular expression

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

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

- unification of agreement features ⟺ 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>;
  ```


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

  ```
  > show named;
  ```

- result of *last* query is implicitly named `Last`; commands such as `cat`, `sort`, and `count` operate on `Last` by default; note that `Last` is always temporary and will be overwritten when a new query is executed (or a `subset` command, cf. 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:

  ```
  > 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;
  ```

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 named query results to free memory

  ```
  > discard Go;
  ```

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

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

  note that the actual data are 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)

- you can also write to a pipe (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 (see Section 2.4)

- frequency counts for matches can also be written to a text file

  ```
  > 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 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 rightmost matching token will be marked

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


`|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

  ```
  > [alemma contains "Zeuge"];
  ```

  → words which *can be* lemmatised as *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 nound 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

```
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.

- 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>
;
```

5.6 Feature set attributes (GERMAN-LAW)

- feature set attributes use special notation, separating set members by `|` characters
- e.g. for the `alemma` (ambiguous lemma) attribute

- 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;
```

- 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 left from 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 speeded up by using an external helper program (the standard Unix `sort` tool)[4]

```
> 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 3.6) 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
```

[4] External sorting may also allow language-specific sort order (*collation*) if supported by the system's `sort` command. To achieve this, 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.

```
2087618 2087619 -1      -1
2122565 2122566 -1      -1
```

note that a previous `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 the pipe consisiting of the following command(s); use the following trick to display the distribution of match lengths in the query result `A`:

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

- see Section 6.4 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

- 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;
  ```

  NB: despite what the command syntax and output format suggest, results are sorted by pair frequencies (not grouped by the second item); also note that 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";
  ```

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)*


```
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.

- note that 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, use double quotes which are less likely to occur in argument values

5.5 CQP macro examples

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

- 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; they 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) ⇒ macro file defines a context-free grammar (CFG) without recursion (see Figure 5)

- set operations on named query results

```
> A = intersection B C;   A = B ∩ C
> A = union B C;          A = B ∪ C
> A = difference B C;     A = B \ 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`

- when there are a lot of matches, look at a random selection to get a quick overview

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

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 results 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];
```

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

```
> randomize 42;
```

- the modifier `cut` n can be appended to a query and performs a similar function: it returns approximately the first n results

```
> "time" cut 50;
```

since this is not a representative subset of the matches, the `cut` option should not be used for reducing large query results; its main purpose is to limit the number of query matches (and thus memory consumption) in Web interfaces and similar applications

3.6 The `set target` command

- 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 <u>underlined</u> in KWIC display (may not work on some terminals)
- search starts from the given anchor point (excluding the anchored token itself), or from the left and right boundaries of the match if `match` is specified
- with `inclusive`, search includes the anchored token, or the entire match, respectively
- `from match` is the default and can be omitted

```
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.

- the `match` and `matchend` anchors can also be set, modifying the actual matches[5]
- anchors can be copied:

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

[5] 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.

5.4 The CQP macro language

- complex queries (or parts of queries) can be stored as macros and re-used
- define macros in 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?"]
  ;
  ```

  (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[];
  ```
- list all defined macros or those with 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.

- %c and %d flags can *not* be used with word lists
- use lists of regular expressions with RE() operator (*compile regex*)

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

- flags can be appended to RE() operator

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

5.3 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[6]

- 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>
```

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

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

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

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

  → range from 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 is evaluated very efficiently
- appending the *keep* operator ! to a subquery returns full matches from the activated query result (equivalent to an implicit expand to match)

[6] 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.

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 label is defined before accessing 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 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;
```

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

- 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 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;
  ```


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.

5 Advanced CQP features

5.1 The matching strategy

- `set MatchingStrategy (shortest | standard | longest);`
- in `shortest` mode, `?`, `*` and `+` operators match 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 currently 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)

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

  only gives the intended frequency counts in `traditional` mode
- Figure 4 shows examples for all four matching strategies

5.2 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";
  ```
- 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";
  ```
- use `TAB` key to complete word list 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;
  ```


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 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 5.1 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;
  ```