The CQP Query Language Tutorial

Stefan Evert
evert@ims.uni-stuttgart.de

24 Dec 2003

Contents

1 Introduction

1.1 The IMS Corpus Workbench (CWB)

Project framework

- Tool development
  - 1993 – 1996: Project on Text Corpora and Exploration Tools (Land Baden-Württemberg)
  - 1998 – 2003: Continued in-house development. CWB v3.0 to be released in 01/2004
  - 1994 – 98: EAGLES: Morphosyntax, Tagset building, EU Programme LRE/LE
  - 1994 – 96: DECIDE (**D**esiging and evaluating **E**xtraction Tools for **C**ollocations **i**n **D**ictionari**e**s and Corpora), EU programme MLAP-93
  - Since 1996: Construction of a subcategorization lexicon for German (Land Baden-Württemberg)
  - Since 1996: Applications in terminology extraction and dictionary updates.

Technical aspects

- CWB uses proprietary format for corpus storage.
  - binary encoding ⇒ fast access
  - full index ⇒ fast look-up of word forms
  - specialised data compression algorithms
  - corpus size: up to ≈ 300 million words
  - corpus cannot be modified after encoding
- Supported operating systems:
  - SUN Solaris 2.8 (Sparc)
  - Linux 2.4 (Intel i386)
  - Data format is platform-independent

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
- GUI components in development (based on networking client-server interface)
- CWB/Perl interface for automation and web interfaces

1.2 From text to CWB corpus

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

2. **Text with XML markup**
   ```
   <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**
   ```
   <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**
   ```
   <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**
   See Figure 1.

1.3 Corpora used in the tutorial

English corpus

- a collection of novels by *Charles Dickens*
- ca. 3.4 million tokens
- derived from Etext editions by Project Gutenberg
- 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
`book`	when text is subdivided into books
`chapter`	chapters
`title`	marks novel, book, and chapter titles
`p`	paragraphs
`s`	sentences
`np`	noun phrases
`pp`	prepositional phrases

corpus position	word form	ID	part of speech	ID	lemma	ID
(0)	<text> value = “id=42 lang="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>					

Figure 1: Text encoded as a CWB corpus.

German corpus

- a collection of freely available German law texts
- ca. 816,000 tokens
- Positional attributes (token annotations)

`word`	word forms ("plain text")
`pos`	part-of-speech tags (STTS tagset)
`lemma`	base forms (lemmata)
`agr`	noun agreement features (*feature set*)
`alemma`	ambiguous lemmatisation (*feature set*)

- Structural attributes (XML tags)

`gesetz`	individual laws
`s`	sentences
`np`	noun phrases
`pp`	prepositional phrases
`vc`	verbal complexes
`ap`	adjectival phrases (within NPs)
`advp`	adverbial phrases

2 Introduction to CQP

2.1 Getting started

- start CQP by typing “`cqp -e`” in a shell window
- every CQP command must be terminated with a semicolon (`;`)
- list available corpora

  ```
  > show corpora;
  ```

- activate corpus

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

- get information about corpus

  ```
  > info;
  ```

- list attributes (“context descriptor”)

  ```
  > show cd;
  ```

2.2 Searching for words

- search single word form (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*

- special characters have to be “escaped” with backslash (`\`)

  `"("` fails / `"."` → *. , ! ? - a b ...* / `"\."` → .

- 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”, see Figure 2)
- if query results do not fit on screen, they will be displayed one page at a time
- press `SPC` to see next page, `RET` for next line, and `q` to return to CQP

```
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
```

Figure 2: Query results displayed in KWIC format.

- some pagers support `u` for previous page, or use of the cursor keys to move up and down
- at the command prompt, use cursor keys to edit input and repeat previous commands (↑ and ↓)
- start CQP with “`cqp -eC`” to use the experimental colour mode (may not work on some terminals)
- change context size

  ```
  > set Context 20;        (20 characters)
  > set Context 5 words;   (5 tokens)
  > set Context 1 s;       (entire sentence)
  ```

- type “`cat;`” to redisplay matches
- show/hide annotations

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

- overview of selected display options:

  ```
  > 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 Accessing linguistic annotations

- specify attribute/value pairs (brackets required)

  `> [pos = "JJ"];` (find adjectives)
  `> [lemma = "go"];`

- `"interesting"` is an abbreviation for `[word = "interesting"]`

- flags can be used with any attribute/value pair

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

- attribute/value pairs use regular expressions (add `%l` flag to avoid this)

- `!=` operator: “does *not* match regular expression”
  `[pos != "N.*"]` → *everything except nouns*

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

- see Appendix A.2 for list of useful part-of-speech tags

- or find out with the `/codist[]` macro:

  ```
  > /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)

2.5 Combinations of attributes: Boolean expressions

- operators: & (and), | (or), ! (not)

  ```
  > [(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*)

2.6 Sequences of words

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

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

- regular expressions over *patterns* (i.e. tokens): every `[...]` expression is treated like a single character in conventional regular expressions

- repetition operators:
  `?` (0 or 1), `*` (0 or more), `+` (1 or more)
  arbitrary repetition: `[...]{3,5}`, `[...]{2,}`

- grouping with parentheses: `(...)`

- disjunction operator: `|` (separates alternatives)

- Figure 3 shows a simple query matching prepositional phrases (PP)

```
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 3: Simple queries matching PPs in English and German.

2.7 Named query results

- assign name to query result

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

  (query names should begin with capital letter)

- list named query results

  ```
  > show named;
  ```

- result of *last* query is implicitly named `Last`
- display number of results

  ```
  > size Go;
  ```

- (full or partial) KWIC display

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

- named query results can be stored on disk in the `DataDirectory` (in binary format)

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

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

- save named query to disk

  ```
  > save Go;
  ```

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

  ```
  > show named;
  ```

- discard named query 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;
  ```

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

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

- you can also write to a pipe

  ```
  > cat Go > "| perl my_prg.pl > go2.txt";
  ```

- set `PrintMode` and `PrintOptions` options for HTML output and other formats (see below)

2.8 Anchor points

- result of a complex query is a list of token sequences (⇒ *matches*)
- each match has two anchor points: `match` (first token) and `matchend` (last token)
- set additional `target` anchor with `@` marker

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

  → shown in **bold** font in KWIC display
- if `target`ed pattern is optional, check how many matches have target anchor

  ```
  > size Last target;
  ```

2.9 Frequency distributions

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

  ```
  > group Last target lemma; > group Go matchend pos;
  ```
- frequencies of token/annotation pairs (using different attributes or anchor points)

  ```
  > group Last matchend word
            by target lemma;
  > group Go matchend lemma
            by matchend pos;
  ```
- you can write the output of the `group` command to a text file (or pipe)

  ```
  > group Last target lemma > "lst.go";
  ```

2.10 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";
  ```

3 Advanced CQP features

3.1 Using labels

- patterns can be labelled

  ```
  > 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

- 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 →)

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

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

- 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

  `[_.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;
  ```

3.2 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 and group commands
- `set Timing (on|off);` to show execution times of queries and groupings
- `set PrintMode (ascii|sgml|html|latex);`
  to set output format for KWIC display and frequency distributions
- `set PrintOptions (hdr|nohdr|...);`
  to turn various formatting options on / off
  (`set PrintOptions;` shows current status)
- `set (LD|RD) <`*`string`*`>;`
  change left/right delimiter in KWIC display from the default `"<"` and `">"` markers
- create `.cqprc` file in your home directory listing your favourite settings
  (will be read on 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: size of history file is not limited by CQP
- `set AutoShow off;`
  no automatic KWIC display of query results

3.3 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 all adjectives modifying a noun within a noun phrase)
- Figure 4 shows examples for all four matching strategies

3.4 Structural attributes

- XML tags match start/end of s-attribute region

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

- pairs of start/end tags enclose single region

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

- `/region[]` macro matches entire region

  `/region[np];` ⟺ `<np> []* </np>;`

- different tags can be mixed

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

- a structural attribute `np` within a pattern evaluates to *true* iff the corresponding token is within an `<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 match to single region

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

search pattern:

```
DET? ADJ* NN (PREP DET? ADJ* NN)*
```

input:									
	the	old	book	on	the	table	in	the	room
shortest match strategy									
●			book						
●						table			
●									room
longest match strategy									
●	the	old	book	on	the	table	in	the	room
standard match strategy									
●	the	old	book						
●					the	table			
●								the	room
traditional strategy									
●	the	old	book						
●		old	book						
●			book						
●					the	table			
●						table			
●								the	room
●									room

Figure 4: CQP matching strategies.

3.5 Structural attributes and XML

- XML markup of NPs and PPs in DICKENS:

```
<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>
```

- attributes within XML start tags can be made accessible to CQP in the form of additional s-attributes with annotations:
  s_len, np_h, np_len, pp_h, pp_len (marked [A] in the show cd; listing)

- access annotations 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 at the beginning of a query

  ```
  > <np> []* </np> :: match.np_h="bank";
  ```

- <np> and <pp> tags are usually shown without XML attribute values; these can be displayed explicitly as <np_h>, … tags:

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

  (other corpora may show XML attributes in start tags)

- use *this* label for direct access within pattern

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

- 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 specified explicitly in query:

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

  will only find NPs contained in *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;
  ```

- most CQP queries will use *maximal* regions

- find *any* NP (regardless of embedding level):

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

- observe how results depend on matching strategy

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

  (watch out for “duplicate” matches)

- when the above is embedded in a longer query, the matching strategy usually has no influence

- there is no easy way of accessing the XML attributes of a region at an arbitrary embedding level

3.6 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 ... ;` to show in which novel, chapter, . . . matches were found

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

- find matches in a particular novel

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

- frequency distributions do *not* work for s-attributes

  `> group A match chapter_title;` fails!

- solution: create new *positional* attribute which annotates each token with desired values; e.g. `nbc` (novel-book-chapter) in `DICKENS`:

  ```
  > show +nbc;
  ```

- compute frequency distribution of `nbc` values:

  ```
  > group A match nbc;
  ```

3.7 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];
  ```

- 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`)

- `%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];
  ```

```
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 5: The `set target` command.

3.8 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 5)
- 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 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
- the `match` and `matchend` anchors can also be set, modifying the actual matches
- anchors can be copied:

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

3.9 Set operations with named query results

- compute subset of named query result by condition 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 = union B C;
  > A = difference B C;
  ```

  $A = B \cap C$
  $A = B \cup C$
  $A = B \setminus C$

  `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`

3.10 Subqueries

- queries can be limited to the matching regions of a previous query (⇒ *subqueries*)
- activate named query instead of system corpus

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

- the matches of `First` now define a temporary *virtual* structural attribute `match` on the corpus `DICKENS`
- all following queries are evaluated with an *implicit* `within match` clause
- 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 *starting* with an anchor tag is evaluated very efficiently
- appending `!` (*keep*) character to subquery returns entire matches from activated query result (an implicit `expand to match`)

3.11 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

  ```
  > <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
- 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
- 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)

```
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 6: A sample macro definition file.

- 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 6)

3.12 CQP macro examples

- use macros for easier access to embedded noun phrases (NP)
- write and load the following macro definition file shown in Figure 7
- 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[];
  ```

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

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

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

Figure 7: Macro definition file for accessing embedded noun phrases.

- CQP ensures that the “generalised” start and end tags nest properly (if the `StrictRegions` option is enabled)
- 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];
  ```

“*case*:*gender*:*number*:*def/indef*”

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

Figure 8: Annotation of noun agreement features in the `GERMAN-LAW` corpus.

3.13 Feature set attributes (GERMAN-LAW)

- feature set attributes use special notation, separating set members by `|` characters
- e.g. for the `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

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

- test non-membership with `not contains`

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

- used to annotate phrases with 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> [_.$0_f contains "$1"] []* </$0>';
  ```

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

- noun agreement features (`agr` attribute) use the pattern shown in Figure 8 (see Figure 9 for an example)

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

- match set members against regular expression

  ```
  > [ (pos = "NN") & (agr matches ".*:Pl:.*") ];
  ```

  → nouns which are uniquely identified as plurals

- both the `contains` and the `matches` operator use regular expressions and accept `%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

A Appendix

A.1 Regular expression syntax

- Regular expressions provide concise descriptions for certain groups of words. Regular expressions "match" the words they describe.

  Notation: `/<reg.exp.>/` → *word*$_1$, *word*$_2$, …

- Syntax of regular expressions

  - letters and digits are matched literally
`/word/` → *word*, `/C3PO/` → *C3PO*
  - `.` matches any single character ("matchall")
`/r.ng/` → *ring, rung, rang, rkng, r3ng, …*
  - list of characters: `[...]`
`/moderni[sz]e/` → *modernise, modernize*
`/[a-c5-9]/` → *a, b, c, 5, 6, 7, 8, 9*
  - repetition of the preceding character:
`?` (0 or 1), `*` (0 or more), `+` (1 or more)
`/colou?r/` → *color, colour*
`/[A-Z][a-z]+/` → *"regular" capitalised word*
  - grouping with parentheses: `(...)`
`/(bla)+/` → *bla, blabla, blablabla, …*
`/(school)?bus(es)?/` → *bus, buses, schoolbus, schoolbuses*
  - `|` separates alternatives
`/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*
`/sa(y(s|ing)?|id)/` → *say, says, saying, said*

- \ "escapes" special characters, i.e. forces them to match literally
`gen\.\*` → *gen.**,   `\(\)` → *()*

- Differences in regular expression syntax
⇒ POSIX standard.

A.2 Useful part-of-speech tags

The PENN tagset (`DICKENS`)

`NN`	Noun, singular or mass
`NNS`	Noun, plural
`NP`, `NPS`	Proper noun, singular/plural
`JJ`	Adjective
`JJR`, `JJS`	Adjective, comparative/superlative
`VB.*`	Any verb form
`VBG`, `VGN`	Present/past participle
`RB`	Adverb
`RBR`, `RBS`	Adverb, comparative/superlative
`MD`	Modal
`DT`	Determiner
`PDT`	Predeterminer
`IN`	Preposition, subord. conjunction
`CC`	Coordinating conjunction
`TO`	“to”
`RP`	Particle
`WP`	Wh-pronoun
`WDT`	Wh-determiner
`SENT`	Sentence-final punctuation

The STTS tagset (`GERMAN-LAW`)

`NN`	Common noun
`NE`	Proper noun
`ADJA`	Attributive adjective
`ADJD`	Predicative adjective (may be used as adverb)
`VV.*`	Any full verb
`VA.*`	Any auxilliary verb
`VM.*`	Any modal verb
`ADV`	Adverb
`ART`	Determiner
`APPR`	Preposition
`APPRART`	Combination of preposition+article
`KO.*`	Any conjunction
`P.*`	Pro-form (rough approximation)
`TRUNC`	Truncated word (“unter-”)
`\$\.`	Sentence-final punctuation
`\$,`	Sentence-internal punctuation

A.3 German tutorial corpus: standard annotations

- Positional attributes (token annotations)

`word`	word forms (surface forms)
`pos`	part-of-speech tag (STTS tagset)
`lemma`	base forms (lemmatised forms)
`agr`	noun agreement features (*feature set*)
`alemma`	ambiguous lemmatisation (*feature set*)

- Structural attributes (XML tags)

`<s>`	sentences
`<pp>`	prepositional phrases
`<np>`	noun phrases
`<ap>`	adjectival phrases
`<advp>`	adverbial phrases
`<vc>`	verbal complexes
`<cl>`	subclauses

- XML element attributes

```
<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

- phrase properties (`f` element attribute)

`<np_f>`	`norm`, `rel`, `wh`, `pron`, `ne` (named entity), `nodet`, `refl`, `es`, `sich`, `numb` (list item), `quot` (in quotes), `brac` (in parentheses), `trunc` (contains truncated nouns), `temp` (temporal), `card`, `date`, `year`, `meas` (measure noun), `street` (address), `tel` (telephone), `news` (news agency)
`<pp_f>`	same as `<np_f>` (projected from NP) + `nogen` (no genitive modifier)
`<ap_f>`	`norm`, `quot`, `hypo` (hypothetical), `invar` (invariant), `vder` (deverbal), `pp` (contains PP), `pred` (predicative)
`<advp_f>`	`norm`, `temp`, `loc` (locative), `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)