CQP

A practical guide

v.0.2

Draft

Susanne Flach
Freie Universität Berlin

susanne.flach@fu-berlin.de
bit.ly/sflach

23. April 2015

Table of Contents

How to use this tutorial

This tutorial is primarily written for users of cqp@fu, especially for students in linguistics classes (from semester 2 onwards). FU members can access cqp@fu with their user accounts. If you do not have access to cqp@fu or wish to work with CQP on your own machine, see the infobox to the right.

The tutorial assumes no prior knowledge of either corpus linguistics or CQP syntax, nor does it presuppose particular computer skills or even technology savvyness. It is suitable even for those who tend to avoid stuff that smells remotely nerdy. You do not need to know anything in advance—all that's required is the willingness and openness to learn a new skill.

The tutorial is split into units (one or two pages each) that have a general topic with explanations and code examples. I use the units as the basis for illustrations and in-class activities of about 15–20 minutes. In most cases, you will use them as revision of in-class activities. The tutorial is also very useful after attending one of our workshops, but can be used for self-study.

While going through this tutorial, you should sit in front of a terminal window. Codes and explanations only make sense if you put them into practice—they feel like gibberish otherwise. I wrote the tutorial as a 'narrated cookbook', which won't be of much use if you do not practice.

All units contain exercises that are sufficiently general, so they don't assume a particular course, theory, or field of analysis (such as morphology, syntax, semantics, or historical linguistics). There is a solution guide at the end that provides suggestions; run these codes for additional practice. Some of the exercises and codes might not always make a lot of sense linguistically, but they give you a very good idea of what CQP can do. There is nothing that should stop you from doing your own case studies—keep practicing!

There is also a dense CheatSheet available from my website. Once you master the principle of CQP, the CheatSheet will be of enormous help. I initially wrote the CheatSheet for myself and I still consult it occasionally. So relax—you are not expected to know (or memorize) all this by heart. Keep the tutorial and the CheatSheet as companions in your endeavours of corpus linguistics with CQP.

What this tutorial is not

It's not an introduction to corpus linguistics. There are tons of introductory books on any number of topics, such as concept of a corpus, issues of corpus composition, strengths and weaknesses of the corpus-based approach, case studies, applications, or statistical analyses. There are also dozens of manuals and video tutorials on specific corpus software.

This tutorial aims to be a student-friendly guide to learn a powerful tool for corpus exploitation. And although it not about the linguistic and methodological issues of corpus linguistics, I will occasionally point out to problems and potential pitfalls in the contexts where they bridge a gap between conceptual, technical and practical issues. But you're strongly advised to consult textbooks, handbooks, and corpus-based research for the issues at hand.

Typographic conventions

Code to be entered in CQP is in **`bold code font`**, and parts of code that you need not enter (because it's already there) in `normal code font`. Variables and concepts are in SMALL CAPS (such as a USERNAME where your own username appears). Strings of words are given in *italics* and lemmas and categories in *CAPITAL ITALICS*.

Infobox

The primary corpus used in this tutorial is the BNC-BABY, a four million token subset of the British National Corpus. Unless instructed otherwise, use BNC-BABY for tutorial and exercises.

CWB/CQP can be installed locally, although this requires expertise that the standard computer user usually lacks. For info on the CWB and how to install, go to http://cwb.sourceforge.net/.

The website also has demo corpora that could be used with this tutorial, but it might be worthwhile to obtain and convert the BNC and BNC-BABY or other corpora.

The website also has a CQP manual. So once you are finished with this practical guide, you find more advanced functions in the CQP query language tutorial.

Both the BNC and BNC-Baby are available from the Oxford Text Archive (http://ota.ox.ac.uk/) and can be converted to CQP using the BNC encoding tool, which is available from the CWB website. For documentation of the BNC consult the reference manual: http://www.natcorp.ox.ac.uk/docs/URG.

Thanks

This tutorial is based on a number of workshops I held over recent months, for students and colleagues, which included users of varied levels of literacy in corpus linguistics. Thanks for the helpful feedback—even if you didn't realise that your reaction to CQP was in fact feedback. Thanks to all who wrote emails of joy and frustration alike. Feedback is of course still welcome for future versions and extensions.

Thanks to the developers of CWB for making it freely available, and to Stefan Evert & colleagues for the detailed query language and encoding manuals.

Thanks especially to Berit Johannsen and Christine Reichhardt for helpful comments and valuable suggestions on earlier draft versions of this guide, to Anatol Stefanowitsch for advice and encouragement during all stages of the cqp@fu project, and to our students who continue to be the best nerdhood trainees one could wish for.

Connecting to cqp@fu

cqp@fu uses the Corpus Workbench (CWB), of which CQP (*Corpus Query Processor*) is part. CWB has been installed on the FU's login server. To use cqp@fu, you need to access the server via an ssh connection ('secure shell') using your ZEDAT details. Ssh is like a tunnel you crawl through to use a program on a remote server. cqp@fu can be used with any device capable of establishing ssh connections (even with smartphones).

Windows. To connect, download PuTTY from https://www.zedat.fu-berlin.de/tip4u_03.pdf (the PDF contains a link to the program) and save it to a convenient location. PuTTY doesn't need to be installed, simply double-click to start. Enter the information of the server you want to connect to (*Host Name*: login.fu-berlin.de, *Port* 22, *Type*: SSH — see tip box for how to save settings more permanently). Connect with *Open*. If it's your first connection, you will be asked to accept a *Host Key*.

You will see `login as:`, enter your ZEDAT user name and hit enter (mine is `flach`, see screenshot). You will be asked to enter your password. The password is not shown when you enter it.

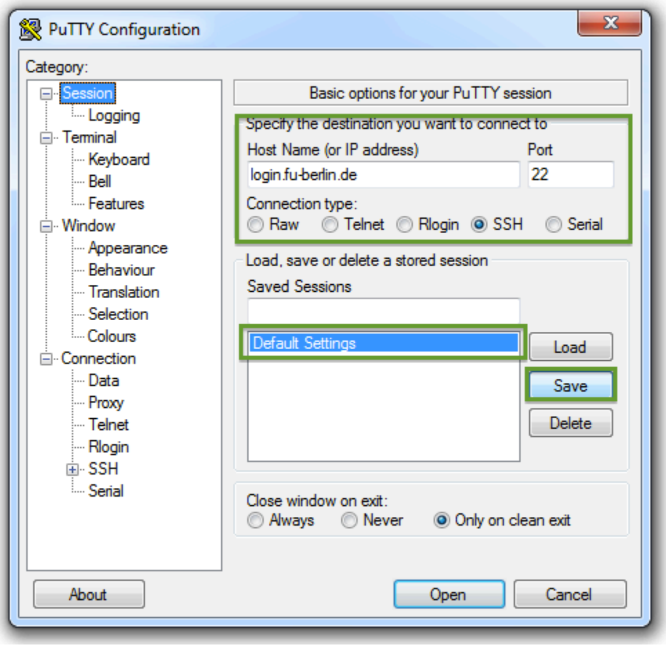
Mac/Linux. Open the program Terminal. This is pre-installed on Mac/Linux systems. To connect to the server via ssh, type `ssh USERNAME@login.fu-berlin.de`. Accept the host key during your first attempt by typing `yes` when prompted. Enter your password, which will not be shown as you enter it.

After connecting. Once `USERNAME@login:~$` is on your screen (the 'prompt'), you are on the server working in a 'shell'. (From now on it doesn't make a difference whether you work from Windows, Mac, tablet, or phone.) I refer to this place as the 'server room', which is like a virtual office with shelves (we will export data files to the server room while working with CQP).

Settings. Before you can work with CQP, you need to run a script that imports settings and tells your shell where to find CQP (if you enter `cqp` before running the script, you get a *command not found* message). You only need to run the script once (i.e. not every time you connect to cqp@fu).

To run, type `sh /home/s/structeng/cqp.sh` (note: *sh* is different to *ssh* and the space between *sh* and the file path is vital). If nothing appears to have happened, that's good news!

Activate settings. Entering `cqp` still gives you an error message. Log off from the server (`exit`) and log back on to activate and load settings. From now on, typing `cqp` after connecting will start CQP.



Specify the destination you want to connect to

Host Name (or IP address): login.fu-berlin.de

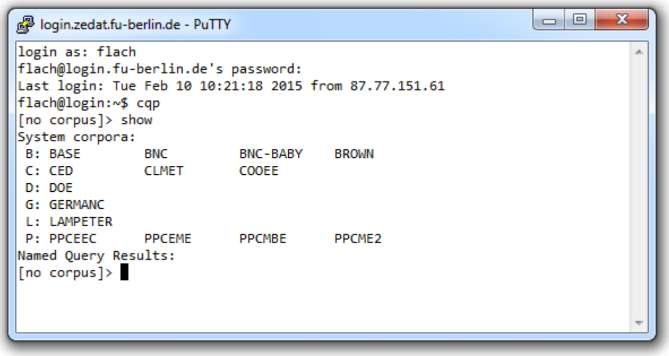
Port: 22

Connection type: ○ Raw ○ Telnet ○ Rlogin ● SSH ○ Serial

Load, save or delete a stored session

Saved Sessions: Default Settings

Close window on exit: ○ Always ○ Never ● Only on clean exit


login.zedat.fu-berlin.de - PuTTY

```
login as: flach
flach@login.fu-berlin.de's password:
Last login: Tue Feb 10 10:21:18 2015 from 87.77.151.61
flach@login:~$ cqp
[no corpus]> show
System corpora:
 B: BASE        BNC         BNC-BABY    BROWN
 C: CED         CLMET       COOEE
 D: DOE
 G: GERMANC
 L: LAMPETER
 P: PPCEEC      PPCEME      PPCMBE      PPCME2
Named Query Results:
[no corpus]> █
```

Connection: cqp@fu on tablets

To work with cqp@fu on tablets, download an ssh app. We recommend *JuiceSSH* (Android), *Serverauditor* (iOS), *The SSH Client* or *Remote Terminal* (Windows). Set up a connection profile: USER NAME goes in *identity* or *account* field; login.fu-berlin.de in *address* or *host name* field. (Connection profiles also have names; don't confuse that with *identity* or *account*.)

Tips: PuTTY

If you don't want to type the host name every time you start PuTTY, you can save the settings: enter the information and, before you connect, select *Default Settings* and then click *Save*.

You can change the colour scheme in the section *Window*. For settings like above, set white for background (*Default Background*), black for text (*Default Foreground*), and a black cursor (*Cursor Colour*). Remember to save the colour changes under *Default Settings* before you connect.

Tips: web access to your user space

You have three options to manage files: if you know standard unix commands, you can manage files in the 'server room'. You can also use WinSCP or scp (if you know what they are).

Most users will prefer to use the FU web interface to access the user space ('server room') for file management. Go to http://zedat.fu-berlin.de/, and log in to *Datenablage*.

1 Basics I: navigation

This unit covers the basics of starting CQP, loading and switching corpora, navigation, and basic queries.

1.1 Starting CQP

After connecting to the login server, you'll see `USERNAME@login~$` and a cursor (which is usually _, ▯, or █); this is called the 'prompt' (Eingabeaufforderung). You are in your user space on the server, which can be thought of as a virtual 'office' with files and folders. CQP runs in its own 'room', which is entered from the server room. To enter the cqp room, type **`cqp`** and hit enter. (To leave CQP and return to the server room, enter **`exit`**. To leave the server, enter **`exit`** again.)

1.2 Loading corpora

You will see `[no corpus]>`, which means that the CQP program is running, but that no corpus has been loaded yet. To see available corpora, type **`show corpora`** and hit enter. To load a corpus, type the name of the corpus you would like to access (`[no corpus]>` **`BNC-BABY`**). You must enter the names of corpora in capital letters as everything in CQP is case-sensitive.

1.3 Switching corpora

You can switch corpora easily by entering the name of the new corpus. You can access the list of available corpora anytime with **`show corpora`** (or simply **`show`**).

1.4 Command line view

What you see when you start CQP or load a corpus (i.e. when you see `BNC-BABY>_` in your terminal window), is called the 'command line view' (Befehlszeilenansicht). This simply means that you can enter commands (**`show`** is such a command). I use 'command line view' to set this view off from the 'concordance line view' (see §1.6).

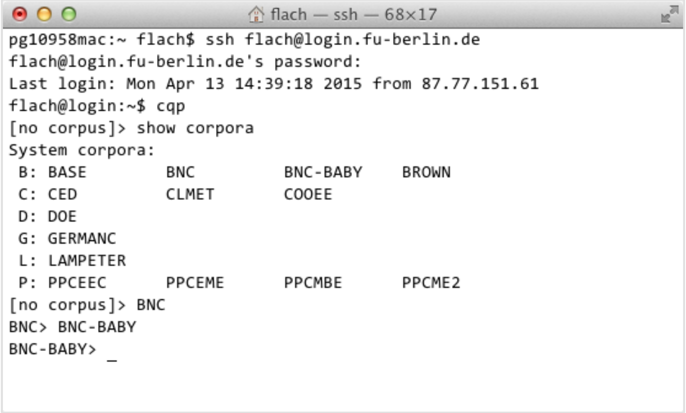
1.5 First query

Once a corpus is loaded, you can run a query. All queries have to be in quotes: if you want to search for the string *fantastic*, type **`"fantastic"`** and hit enter.

You will see a list of hits for *fantastic* in your terminal (the 'concordance'), and a header with information on corpus, number of hits, query entered. The keyword is set off from its left and right context by `[[[` and `]]]`.

1.6 Concordance line view

If a query has more hits than can be displayed on the screen, you will see **`:_`** below the list. This is the 'concordance line view'. It means you can 'browse' the concordance with the arrow keys ↓↑ (line by line) or the space bar/**`w`** (page by page). Leave the concordance line view by hitting **`q`**. This will get you back to the command line view. To redisplay you previous query (e.g., if you accidentally left it), type **`cat Last`** (§2.3).

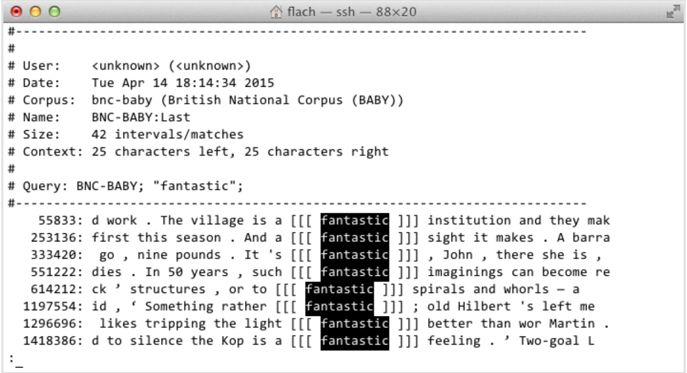


```
pg10958mac:~ flach$ ssh flach@login.fu-berlin.de
flach@login.fu-berlin.de's password:
Last login: Mon Apr 13 14:39:18 2015 from 87.77.151.61
flach@login:~$ cqp
[no corpus]> show corpora
System corpora:
 B: BASE        BNC         BNC-BABY    BROWN
 C: CED         CLMET       COOEE
 D: DOE
 G: GERMANC
 L: LAMPETER
 P: PPCEEC      PPCEME      PPCMBE      PPCME2
[no corpus]> BNC
BNC> BNC-BABY
BNC-BABY> _
```

Tip: saving typing effort I

Unix systems have 'command completion': if you start typing a command and hit TAB, the full command appears (if it's unambiguous). If there are several commands that start like this, hitting the TAB key *twice* shows options.

Try this with BNC and BNC-BABY: type **`BN`** and hit TAB. You will see BNC and BNC-BABY as available options. If you type **`BNC-`** and hit TAB, the full string **`BNC-BABY`** will appear. Then press enter to use (or, here, to load corpus).



```
#---------------------------------------------------------------------------
#
# User:    <unknown> (<unknown>)
# Date:    Tue Apr 14 18:14:34 2015
# Corpus:  bnc-baby (British National Corpus (BABY))
# Name:    BNC-BABY:Last
# Size:    42 intervals/matches
# Context: 25 characters left, 25 characters right
#
# Query: BNC-BABY; "fantastic";
#---------------------------------------------------------------------------
    55833: d work . The village is a [[[ fantastic ]]] institution and they mak
   253136: first this season . And a [[[ fantastic ]]] sight it makes . A barra
   333420:  go , nine pounds . It 's [[[ fantastic ]]] , John , there she is ,
   551222: dies . In 50 years , such [[[ fantastic ]]] imaginings can become re
   614212: ck ' structures , or to [[[ fantastic ]]] spirals and whorls – a
  1197554: id , ' Something rather [[[ fantastic ]]] ; old Hilbert 's left me
  1296696:  likes tripping the light [[[ fantastic ]]] better than wor Martin .
  1418386: d to silence the Kop is a [[[ fantastic ]]] feeling . ' Two-goal L
:_
```

Tip: saving typing effort II

CQP has a 'query history': use the arrow keys ↓↑ to access previous queries while in command line view. Use ← and DELETE key to modify a query. You can ignore (or delete) the semicolon **`;`**, but if you keep it, make sure it's at the end of your query string (CQP requires **`;`**, but cqp@fu settings automatically insert it with each query anyway).

Tip: getting error messages

If you get 'CQP syntax error', CQP is telling you that it doesn't understand what you want. Most of these messages occur when you forget things like **`"`** or when **`;`** isn't where it should be, i.e. when you get the syntax wrong.

Exercises §1

i. Query these strings in BNC-BABY: *nice*, *oscillate*, *get*, *count*, and *yank*. How many matches do you find?

ii. Look at the results by browsing through them. What is found? What is *not* found? List your ideas.

iii. If you need a hint, query the above strings in capital letters and compare the results and number of hits.

iv. For a second hint, query *got*, *counting* and *nicer*.

v. Think about the nature and consequence of this problem. What would you need to be able to do?

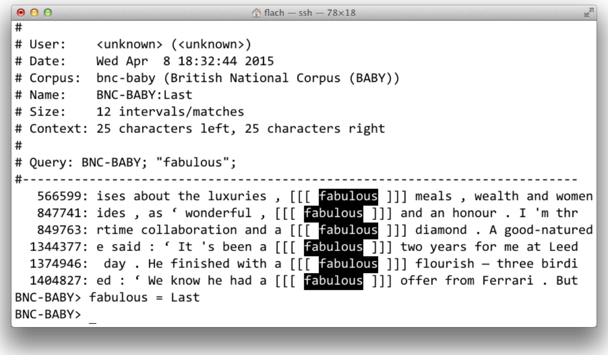
2 Basics II: managing queries

This unit covers some technical basics of saving, printing, and exporting queries.

2.1 Naming queries

Every query is temporarily saved in a variable `Last` — variables are like drawers in your office. Each new query overwrites the previous one (which is saved as `Last`).

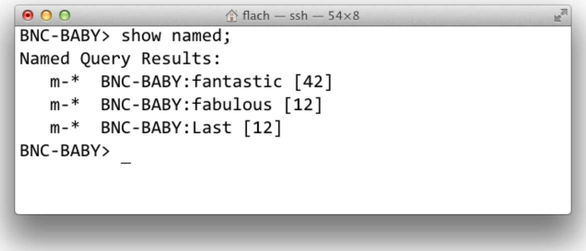
You can name queries to save them from being overwritten: `BNC-BABY>` **`fabulous = Last`** (i.e. if you run a concordance of **`"fabulous"`** and like to keep it). As CQP is case-sensitive, it can only recognize **`Last`** if entered as instructed; **`X = Last`** means 'save content of **`Last`** into drawer **`X`**'. The name given for a query can be anything, provided it's not a predefined CQP command (like **`show`** or **`cat`**).



```
#
# User:    <unknown> (<unknown>)
# Date:    Wed Apr  8 18:32:44 2015
# Corpus:  bnc-baby (British National Corpus (BABY))
# Name:    BNC-BABY:Last
# Size:    12 intervals/matches
# Context: 25 characters left, 25 characters right
#
# Query: BNC-BABY; "fabulous";
#---------------------------------------------------------------------------
   566599: ises about the luxuries , [[[ fabulous ]]] meals , wealth and women
   847741: ides , as ‘ wonderful , [[[ fabulous ]]] and an honour . I 'm thr
   849763: rtime collaboration and a [[[ fabulous ]]] diamond . A good-natured
  1344377: e said : ‘ It 's been a [[[ fabulous ]]] two years for me at Leed
  1374946:  day . He finished with a [[[ fabulous ]]] flourish – three birdi
  1404827: ed : ‘ We know he had a [[[ fabulous ]]] offer from Ferrari . But
BNC-BABY> fabulous = Last
BNC-BABY> _
```

2.2 Listing named queries

Similar to the **`show corpora`** command above, you can view a list of saved/named queries with **`show named`**. (Simply **`show`** gives you a list of corpora *and* named queries.) The number in square brackets gives the number of hits. This screenshot shows two queries—before you read on, think about why two, not three:



```
BNC-BABY> show named;
Named Query Results:
   m-*  BNC-BABY:fantastic [42]
   m-*  BNC-BABY:fabulous [12]
   m-*  BNC-BABY:Last [12]
BNC-BABY> _
```

`Last` is identical to the query `fabulous` (12 hits each) as the latter was just saved from `Last`.

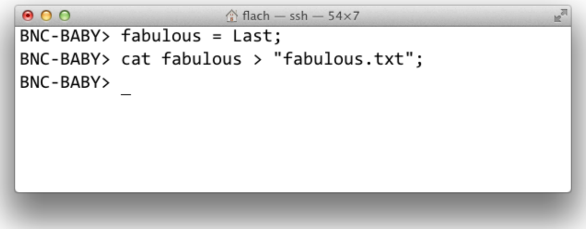
2.3 Accessing named queries

If you want to access a named query, or redisplay a query you left with **`q`**, you need to 'print' it to the screen again. Type **`cat Last`** or **`cat fabulous`** (*cat* is the standard unix command for 'printing' and short for *catenate*). All named queries are deleted when you exit CQP; you should save important queries as files (§2.4).

2.4 Exporting queries

You will not need the information in this column in the initial stages, so you can ignore this for now. But it involves 'printing' queries, so it belongs here.

Since **`cat Last`** (or **`cat NAMEOFQUERY`**) prints the contents of the drawer `Last` to the screen, you need to 'redirect' this output to a text file:



```
BNC-BABY> fabulous = Last;
BNC-BABY> cat fabulous > "fabulous.txt";
BNC-BABY> _
```

The **`>`** is a re-routing operator, telling CQP to redirect the contents of **`fabulous`** to a file called **`fabulous.txt`**. Note that the quotes here are necessary.

2.5 Accessing exported files

If you are working with cqp@fu, the file is saved in your user space. That's the 'room' you entered when you accessed the login server and from which you entered the cqp room.

If you leave the cqp room (`BNC-BABY>` **`exit`**), you return to the server room. You can see your saved file by typing **`ls`** or **`ls -l`**. (for *list*). You should also see the file **`CWBclean.pl`** (see §2.6).

2.6 Cleaning exported files

`CWBclean.pl` is a script which cleans CQP output. It also inserts tabs to your concordance, which is very handy for working in spreadsheet programs (see §10.5).

To use the script, type the following command while you are in the server room (don't type `~$`; this is just to indicate that you are not in the cqp room anymore!):

`~$ perl CWBclean.pl INPUT.txt > OUTPUT.txt`

ATTENTION: (i) spaces are really important; (ii) the output file is not enclosed in quotes (different to **`cat`** in CQP); (iii) the output file needs to have different name than the input file, but can be any name you find useful.

2.7 Downloading files

Download output.txt from http://zedat.fu-berlin.de → *Datenablage*. You'll download a zip file. You can also delete old, unnecessary files here (your quota is 8GB).

Exercises §2

i. Re-run the queries in the exercise box in §1 and name the queries to save them temporarily. Use the strings you query as names for the drawers. (You should get an error message for one of the items. Any clue why?)

ii. Re-print the contents of some of some queries to practice.

3 Simple queries: word forms

This unit extends the problems discussed in §1 and introduces regular expressions to improve queries.

3.1 Regular expressions

One of the problems discussed in §1 above was that queries such as `"fabulous"` will only find hits that match the string between quotes exactly. So `"get"` will not match *gets*, *getting*, *got*, or *gotten* (we ignore capital letters in strings such as *GET* or *Get* for the moment).

A first step towards finding these is to use so-called regular expressions (regex). Some are like wildcards, some group characters together, and some repeat stuff.

Let's look at an example: to find third-person forms of *improve*, you need to add *s* to *improve*, but make it optional with `?`: `BNC-BABY>"improves?"`—this matches strings *improve* and *improves*, as `?` tells CQP that *s* can be there, but doesn't have to. You should get 171 hits (`?` goes *inside* the quotes, behind what's optional.)

Yet, this does not find *improved*. One solution is to remind yourself that the only difference between *improves* and *improved* is *d* instead of *s*, so we can add the *d* and make that optional, too: `"improves?d?"`. CQP will now match *improve*, *improves*, and *improved* —if it doesn't find *s*, it will look for *d* (but of course will also return tokens where it doesn't find either). This should return 312 hits—browse the results!

But we haven't found *improving*. One solution is to add the characters *i*, *n*, and *g* and make these optional, too: `"improves?d?i?n?g?"`. This still only finds 312 hits; and browsing the results does not yield a token of *improving*. Before you read on: look at the query and make sure you understand why this is the case!

It is, of course, because *improving* does not contain <e>, but our query wants one. One solution would be to make <e> optional, too: `"improve?s?d?i?n?g?"`—this will now match *improving* to give us 362 hits.

3.2 Grouping characters

By now our query is rather clumsy—luckily, regular expressions can simplify things a lot by grouping characters together. There are two different types of grouping: the two letters that give us *improves* and *improved* both occur as alternatives in the *same* position in the string, whereas the *ing* is where the characters occur in *different* (i.e. subsequent) positions—so these two groups require different grouping strategies.

The first case, *s* and *d*, can be grouped with `[]`, which groups a *class* of characters; *ing* can be grouped with `()`, which groups a *sequence*. Try `"improve?[sd]?(ing)?"` and make sure you understand the principle and why it gives you the same result as `"improve?s?d?i?n?g?"`. This should also give you 362 hits. Note that the groupings still require the operator `?`.

3.3 Wildcards

There is a match-all wildcard, the period `.`. It matches *any* character in the position you put it in. See what the following does to your query, if you swap the grouped class `[sd]` with `.`: `"improve?.?(ing)?"`.

Well, instead of asking for an optional *s* or *d* in this position, you want *any* character (not just *s* and *d*), which is also optional. It is *almost* the same query—but it finds two more hits (both of which are *improver*). Make sure you understand why.

3.4 Combining regular expressions

Being able to use regular expressions in CQP gives you an extremely powerful range of options and there are almost always several ways to achieve a desired result. Compare `"improve(s|d)?"` and `"improve[sd]?"`—they return the same set of results. Can you see why?

Well, we said that `()` groups *sequences*, while `[]` groups *alternatives*. `(s|d)` still matches a sequence, but by using the OR operator `|`, it matches a 'sequence' of one character, which here is either *s* OR *d*.

3.5 Repetition operators

There are two very handy operators, `*` and `+`. They are *not* wildcards like `.` (online applications often use `*` as a wildcard, so don't confuse them!). Instead, they tell CQP how many instances of the *preceding* character or group you want repeated: `+` repeats it 1 or more times, `*` repeats it 0 or more times.

If this seems superfluous to you, run the following examples to check whether you understand why they are not identical: `"improves*"` returns 171 hits, but `"improves+"` only 17 (look at *what* they return!).

3.6 Case-insensitive matches

Our queries above did not match strings with capital letters. We could include these, but luckily CQP already has an operator: adding `%c` to your query outside the quotes will ignore case. Thus, `"improve[sd]?" %c` will also find *Improves* or *IMPROVED*. Run the queries with and without `%c` and browse through the results to appreciate the difference and importance of `%c`!

Exercises §3

i. Find all forms of *SNOW*, *RAIN*, *SLEEP*, *HOUSE*, and *ROUND*. Start with basic forms, browse your results to determine what your query finds (but what it doesn't!). Then improve & simplify the query.
ii. Search for all forms of *SNOW* and *RAIN* in one query.
iii. Formulate a query for *realize*. What do you notice?
iv. Find all forms of *NICE*, *OLD*, and *FIT*.
v. Find all forms for *GET*.
vi. Find strings that (a) begin with *un-* (e.g. *unfriendly*), (b) end in *-ment* (e.g. *movement*), and (c) begin with *un-* AND end in *-ment* (e.g. *unemployment*).

4 Accessing token annotation

This unit expands simple CQP queries to the more powerful "CQP principle" and shows how to access token-level annotation (e.g. lemmas or part-of-speech tags).

4.1 The CQP principle

Up to now, we enclosed our queries in quotes, `"STRING"`. This accesses the string *exactly* as it appears in a corpus text. We have very powerful options once we use and combine regular expressions. For many purposes, this is sufficient, and in some cases, i.e. in plain text or unannotated corpora, this is the only way.

However, many corpora come with rich token-level linguistic annotation, such as part-of-speech or lemma information. Such annotation identifies a string like <*improves*> as 3rd-person singular present tense of *IMPROVE* with the pos tag VVZ (in the BNC). The information can be accessed, but we need a different notation to tell CQP which information we want.

The key to the "CQP principle" is to understand that CQP converts `"STRING"` to its standard CQP notation (this happens in the background). This 'real' notation requires that every token definition is in square brackets, stating precisely which *level* of annotation CQP has to go to in order to find what we want.

So `"STRING"` is actually short for `[word="STRING"]`, which tells CQP to accesses the `word`-level (which is the string as it occurs in the text). If we want the `lemma`-level of a token, we have to use `[lemma="LEMMA"]` (there is no shorthand for the `lemma` or any other level).

The true power of this notation becomes obvious once you realize that the expression inside the square brackets contains conditions for matching tokens and that these conditions can be combined in almost unlimited complexity. Let's look at an example.

4.2 Finding verbs

One of the exercises in §3 asked you to find word forms of *ROUND*. You were also asked to list properties about the hits that your query found (and what it didn't). One thing you should have noticed is that while you can formulate queries to find all forms of *ROUND*, i.e. *round*, *rounds*, *rounded*, and *rounding* (and those with capital letters), you actually have no way of separating the verb *round(s)* from *round(s)* as noun, or *rounded* as a past verb from *rounded* as an adjective.

You probably queried `"rounds?(ed)?(ing)?"%c` (or similar), which CQP converted to its 'real' notation, i.e. `[word="rounds?(ed)?(ing)?"%c]`. Now, suppose you only wanted to find *ROUND* as a verb, you have to instruct CQP to restrict the query to find only verbs. You have to expand the notation to contain *another* condition, i.e. an instruction to match only instances that also satisfy *that* condition (here: verbs). You add conditions on a token by using the AND operator `&`:

```
[word="rounds?(ed)?(ing)?"%c & class="VERB"]
```

4.3 Using part-of-speech information

There is no real limit on the number of conditions you can put into the brackets (at least I have not come across one in my daily business with CQP). Seriously, don't underestimate the power of this property! To illustrate, let's search for adverb uses of *ROUND*. This is a little easier for now, as we don't have to worry about inflectional forms at the moment.

We saw above that conditions can be formulated on the `word` and `class`-levels; part-of-speech is on the `pos`-level. To search for adverb uses of *ROUND*, you need the string *round* in the `word`-condition and the tag for adverbs in the `pos`-condition ('AV0' in the BNC): `[word="round"%c & pos="AV0"]`. Another option would be the `class`-level; adverbs have 'ADV' as a value here: `[word="round"%c & class="ADV"]`.

The difference between `pos` and `class` is that `pos` is more detailed than `class`. This is not really relevant for adverbs, but it has some major advantages in the case of nouns and verbs, because 4 noun tags are subsumed under `class="SUBST"`, and 25 verb tags are included in `class="VERB"`. Take a look at the list of tags for the BNC. Make sure you understand the principle!

The advantage in CQP is that you can combine these levels to refine your queries by including or excluding aspects you want (or don't want). Suppose you wanted verbal *rounded*, but only past tense, not past participles, you could set restrictions as follows:

```
[word="rounded" & class="VERB" & pos!="VVN"]
```

(You could formulate your query to match verb pos tags except VVN, but that's a fair bit of typing even with regular expressions, as you need to exclude VVN.)

The `!=` operator means IS NOT, so the above query instructs CQP to 'find all (lower-case) instances of *rounded*, which are verbs, but are not past participles' (VVD is the tag for 'past tense' and VVN for 'paste participle' in the BNC). So while `[word="rounded" & pos="VVD"]` is shorter for this purpose, `!=` illustrates the power of the CQP principle and its `[]`-annotation of formulating conditions on tokens.

> CAUTION!
>
> Part-of-speech information almost always comes from automatic taggers, meaning that a computer program annotated the data based on algorithms and probabilities.
>
> While some taggers claim to be up to 97% accurate—though many corpus linguists doubt this—it still means that about 1 in 30 words is erroneously tagged. Be aware of this and treat tagging with healthy scepticism.
>
> You should make it a habit to critically review initial search results: What is found? What is not found? What should have been found? It should not scare you to repeat, reformulate, and refine queries several times (and use different strategies) before you export and analyse data.
>
> Treat this as part of the learning experience: you'll actually learn a lot about the nature of your data and the fascinating complexity of language. We've all been there (and still are!).

4.4 Accessing lemma information

So far we met the **word**, **pos**, and **class** annotation levels, with the latter two referring to the same level, just with different focus on detail.

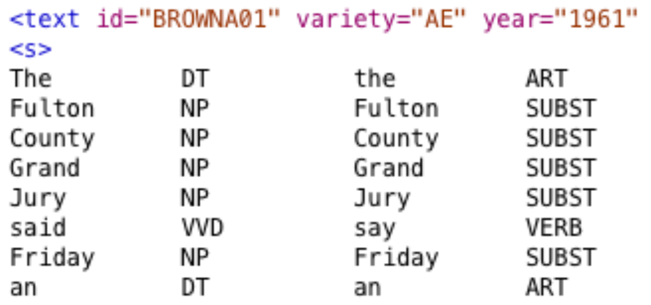
One of the most widespread types of annotation is the lemma of a token, i.e. the corpus forms *is*, *was*, *were*, or *been*, are variants of the abstract lemma *BE*.

Thus, if you want to avoid the tedious typing of the **word**-level string with regular expressions, it is often desirable and useful to resort to lemma information (if available). The usual disclaimers apply: lemmatization is added automatically, too, and may not always be correct (though rarely so), so keep this in mind.

In most corpora, the name for the lemma level is **lemma**. In the CQP versions of the BNC(-BABY), **lemma** is **hw**, for *head word* (for irrelevant technical reasons).

4.5 Excursion: CQP data model

It helps to understand what CQP does if you know the underlying data model. The corpus files that go into CQP look like this (vertical format; BROWN corpus):



```
<text id="BROWNA01" variety="AE" year="1961"
<s>
The         DT      the         ART
Fulton      NP      Fulton      SUBST
County      NP      County      SUBST
Grand       NP      Grand       SUBST
Jury        NP      Jury        SUBST
said        VVD     say         VERB
Friday      NP      Friday      SUBST
an          DT      an          ART
```

The levels of annotation are arranged in columns: the corpus text **word** in the first, **pos** in the second, **lemma** or **hw** in the third, and **class** in the fourth (though the order is irrelevant). So when you instruct CQP to find **[word="said" %c & pos="VVD"]**, it will look for and return all instances of *said* in the text where the **pos**-column also contains VVD (rather than JJ for *said* as an adjective). Similarly, if you query **[class="SUBST"]**, it will return *all* noun tokens in the corpus, because their **class**-column contains SUBST, regardless of what is in the other columns (from the screenshot above it would return *Fulton*, *County*, *Grand*, *Jury*, and *Friday*).

4.6 CQP principle revisited

Let's summarize the CQP principle: you state a query in square brackets, in which you formulate (combinations of) conditions that tell CQP which tokens you want returned. The pattern is: **[attribute="VALUE" & …]**.

The 'levels' of annotation are the *attributes*, which have *values*. Attributes are CQP syntax (largely identical across corpora), but the values are corpus-specific (and can vary considerably across corpora). The colours in the box (right) illustrate the difference. You can use regular expressions on values, but not on attributes.

Know your corpus!

Searching for part-of-speech information requires knowledge both of the annotation scheme of a corpus and how it is represented in the corpus' CQP version. There are several tagsets out there and knowing which tagset is used on which corpus is essential (see CheatSheet for the tagsets used in cqp@fu, i.e. compare PENN vs. CLAWS).

The CQP syntax is, for the most part, identical between corpora, such as the column and level names or the type of operators you can use (the major difference you have met so far is that **lemma** is **hw** in the BNC). But the *values* to be inserted between quotes can be very different.

Knowing your corpus is absolutely vital in CQP (as it would be in *any* other system, for that matter). You can usually access additional information by typing **info CORPUSNAME** in CQP. Most info files contain the names and availability of attributes and their values.

Tip: displaying annotation in concordances

Token-level annotation (column values) can be displayed in concordances. This can be very useful if you would like to see how a particular word is tagged, either if you're wondering what part-of-speech a word is, or, more on the technical side, if you need a reminder of the tagset ('values of attributes') used on your current corpus.

To display token annotation, use **show**—you know the command from **show corpora** / **show named**. To display pos tags, type **show +pos**, if you want class, type **show +class** (no brownie points for guessing how to show lemma or hw!). This will print the information in the concordance next time you run it (or use **cat Last**), set off by a slash: for *the* it prints **the/AT0**. To turn off, type **show -pos**. You can combine on and off commands in one line: **show +pos +class -hw**.

[attribute$_1$="VALUE" & attribute$_2$="VA.(UE)?"]

CQP syntax	**corpus-specific data**
no regular expressions	*regular expressions possible*
usually identical across corpora	*variable across corpora*

Exercise §4

i. In the exercises in §3, you searched for word forms of SNOW, RAIN, and SLEEP. Now determine the number of uses as nouns vs. verbs in BNC-BABY.

ii. For ROUND, determine the word-class distribution by the **class** attribute. Use the strategy suggested in the tips box above to determine **class** values for *round*.

iii. The BNC has so-called 'ambiguity tags'. These occur where the tagger was unsure of the part-of-speech; i.e. whether a token is adverb or preposition, noun or verb, etc. Many tokens thus can have pos tags like **AVP-PRP**, **NN1-AJ0**, or **VVD-VVN** (unsure which verb). This is a serious problem when working with the BNC. How can you query all adverb, all verb uses, all noun uses of *round* by **pos**, including tokens the tagger could not decide on?

iv. How many tokens in the BNC-BABY have ambiguity tags? How much (in per cent) of the corpus is that?

5 Multi-token queries

This unit covers searching for patterns of more than one token, i.e. multi-word units of fixed and variable length.

5.1 Multiple tokens: principle

If the CQP principle is *one* square bracket for *one* token definition (`[TOKEN]`), then querying a *sequence* of tokens is easy: you'll need *one* square bracket for *each* of the tokens in a string (`[TOKEN`$_1$`] [TOKEN`$_2$`]`). To search for *the house*, you define a pattern for *the* and one for *house*.

The handy thing is that each token can get its own definition independent of the other. Also, the shorthand notation with quotes (`"STRING"`) can be combined with the square bracket notation. Let's learn to appreciate this flexibility!

5.2 Bigrams & n-grams

N-grams are strings of *n* orthographic words (strings separated by spaces). A bigram is 2, a trigram is 3, etc.

You can search for a bigram like *the house(s)* as `"the"%c [hw="house"]`; and for trigrams like *the blue house*, the query `"the" "blue" "house"` will actually already do the trick (case-sensitive string). You probably already see the huge number of options by formulating precise conditions on each of the tokens.

A simple example: suppose you want to retrieve *car(s)* in noun phrases headed by *the* or *a(n)* and modified by an adjective (DET ADJ NOUN), you can try:

`"the|an?"%c [class="ADJ"] [hw="car"%c]`.

5.3 Flexible n-grams

What if you don't know (or want to impose a priori) the length of your string? What if you want to identify noun phrases with one *or more* modifying adjectives in a single query? Think about this for a second—you actually already know the operators to do this!

You can use repetition operators. If the adjective in your pattern can occur *once or more*, you can use the *once or more* operator `+` (see §3). You put it *outside* the square brackets (so it applies to the entire `[]`):

`"the|an?"%c [class="ADJ"]+ [hw="car"%c]`

(Think about what the operators `?` and `*` instead of `+` do to your query, then run the queries to check.)

There is another useful expression, which allows you to specify the *extent* of repetitions: if you want NPs with a variable, but limited number of adverbs or adjectives, say a minimum of 2, but a maximum of 4 such tokens, you can use `"the"%c []{2,4} [class="SUBST"]`. (Note, though, that many results are not at all the NPs we expected! See exercise box for discussion.)

This 'range' expression `{MIN,MAX}` means 'match *x* repeated MIN to MAX times'. If you use `{1,}`, this matches one *or more* tokens (identical to `+`), `{,2}` says to match 'up to two', and `{3}` means 'exactly three'. You can use `{}` inside quotes, too, to specify the range of repeated characters, classes or sequences.

Tip: escaping characters

What happens if you want to find ?, i.e. a real question mark? Or a real period? There is obviously a major problem because if you query `"?"`, CQP returns an error message ('illegal regular expression'); if you query `"."`, it returns *all* one-letter tokens (including, but not limited to, punctuation). Both shouldn't surprise you now that you're familiar with regular expressions and what they do.

So what you need is to tell CQP that you don't want the regular expression function of these symbols. You have to 'escape' their technical use. You do this with the backslash directly preceding the symbol: `"\?"` will find real ?s. Now it's easy to match real periods, brackets, or asterisks without using `pos` or `class`.

Tip: spaces

You can, but don't have to use spaces in your queries. CQP ignores them, as long as they don't occur when matching *values*. So `"the"%c` is the same as `"the" %c` and `[hw="house"]` is the same as `[hw = "house"]`. Spaces can make your query easier to read (which is why I am using them).

The important exception is that you can't use spaces for whatever goes *between* quotes, i.e. you can't use them on values: `[hw=" house "]` will return 0 matches. Can you think of why?

It's because CQP will try to match the space in this case, i.e. search for a lemma/hw that has spaces around it. (And there aren't such lemmas, primarily for technical reasons—in other words, CQP versions cannot match spaces in text.)

Exercises §5

i. Search for *X-and-X* coordination. What types do you find if you query them very schematically, i.e. with very few conditions? Browse through the results: which hits seem to be particularly interesting cases that would merit further investigation? And why do others seem to be less interesting?

ii. The query `"the"%c []{2,4} [class="SUBST"]` in 5.3 returns too many hits that are not instances of what we wanted (e.g. *the more luxurious the luncheon*): our query has a low precision rate. Make sure you understand why. Then reformulate your query, and do so by filling the middle slot `[]` with a condition that matches only adverbs or adjectives.

iii. Find verbs with the prefix *re-*. What appears to be the problem? Try to improve it with the `{}` operator.

iv. Most idioms in English are surprisingly flexible, lexically and morphologically. Formulate queries that match as many instances as possible for *speak one's mind*, *there is something X about Y*, and *sit through*. Start with very schematic queries, i.e. use very general (or no) restricting conditions for the slots that you suspect to vary. Then browse through the results to check what is found (and what is not) and determine where and how you can refine your queries to achieve better results.

v. Proverbs are usually considered to be the most rigidly fixed multi-word expressions, but they also tend to be rather infrequent. Think of examples and query them (like *kick the bucket*, *barking up the wrong tree* etc.)

6 Counting

This unit introduces the command to count hits by a number of values.

6.1 Counting one-grams

We know how many hits a query has from the information provided in the header. But counting in CQP is so much more powerful. While the header itself returns how many matches where found for the entire query, it does not give information on the distribution of these hits between word forms, lemmas, or pos tags.

Let's look at *RAIN* from the exercises in §3 and see how the uses are distributed between nouns and verbs. You could query `[hw="rain" & class="VERB"]`, then note down the number of matches and re-run the same for `SUBST`. This is tedious, though. And in some cases you might not be aware of the different classes a word is tagged with. Plus, if you tried to do the same by pos tags, you'd have to do this for four noun tags and 25 verb tags (even if you used regular expressions for pos tags!) to find out that *RAIN* is tagged 3 times as past tense (VVD), out of 338 hits. How do I know?

Remember from the error message in the exercise in §2 that `count` is a command in CQP that expects being followed by something else than `=`? It expects two things at least: (i) the name of the query you want to count things *in*, and (ii) the name of the attribute you want to count *by*. So to count the distribution by `class` for your query of *RAIN*, type `count Last by class`. The output should tell you that the lemma *RAIN* is tagged 238 times as a noun and 100 times as a verb. Similarly, `count Last by pos` lists the distribution of the pos tags.

Now: what will `count Last by word` do and what is the difference to `count Last by word %c`? Think for a second, then run both commands to check your suspicion. What happened? The former counts the frequencies of the *exact* strings (e.g., 237 for *rain* and 9 for *Rain*), while `%c` ignores case and adds the two figures (to give 237+9=246 for case-insensitive *rain*).

6.2 Counting n-grams

If you have a multi-token pattern, the basic version `count Last by word` will count the types of the full pattern. Let's look at *ADV-and-ADV* coordination: query `[class="ADV"] "and"%c [class="ADV"]`, then run `count Last by word`.

The phrase *up and down* is the most frequent (90 hits), followed by *in and out* and *now and then*. (The frequency list is longer than can be displayed, so browse with arrows/spacebar, exit with `q`). `%c` will combine/add string counts. This is not immediately obvious here as one of the first strings where this shows is quite far down. (It's somewhat indicative of the British fascination for the Australian soap *Home and Away*—see how case-sensitivity *can* occasionally be revealing!). Remember: it always pays to browse before conclude.

6.3 Counting by specific positions

What if you wanted to know which adverb occurs most frequently in the first position in *ADV-and-ADV*? Remember how `count` needs two arguments (required), i.e. the drawer and the attribute to count by? `count` further allows *optional* arguments: the *position* in the match to count by. So you can tell `count` that you want to count by position: `count Last by word on match` — *up* is still the most frequent (133).

By default, `match` selects the *first* position in a match. It's short for `count Last by word on match[0]` (because most computer programs start counting at 0). Think what `count Last by word on match[1]` will do with *X-and-X*? Run to check your suspicion.

Of course, it will return a frequency list of only a few lines: as many orthographic variants *and* occurs in (*and*, *And*, *AND*, etc.). If you set `%c` after `word`, your frequency list only has one line. So if `match[1]` selects the second position, `match[2]` selects the third, etc.

The cool thing is: there is `matchend`, too. It counts from the *end* of your search pattern. This is more helpful than you may think at first: suppose you have a multi-word pattern of variable length (because some tokens in your query are optional), but want to know which words occur most frequently in the last position. There is no way to do this with `match`, because `match[2]` would simply look for and count the third tokens in every hit (regardless of whether that hit has three or four words). So here it helps to use `matchend[0]`, which selects the last token in a hit.

6.4 Counting beyond patterns

Now the *really* cool thing is that the numbers in square brackets for `match` can be set to select and count items *beyond* the query. `match[-1]` will count the tokens one position to the *left* of a pattern (as `[0]` is the beginning of a pattern). By the same logic, `matchend[1]` counts one position to the *right* of a pattern (as `matchend[0]` is the end of a pattern). `match[-2]` counts two positions to the left etc.; `match[NUMBER]` works like an anchor or reference point for CQP to know where to look.

Exercises §6

i. Make verb frequency lists (word-form and lemma/hw). Browse the list and look at the first 25 verbs or so. What types do you find? Does it make sense to group them? How? Take a look at the BNC pos tag set and then make different frequency lists for verb types you find useful.

ii. Make a frequency list of adverbs (query by `class`). What's the most frequent? If you are surprised (even if not), create a frequency list without the first two items on your initial list.

iii. In your *ADV-and-ADV* query, what's the most frequent item to the left and right of the pattern?

iv. In the sequence *the X car*, what's the most frequent adjective modifying *car*? Is this also the most frequent *string* between *the* and *car*?

7 Sorting & randomizing

This unit covers useful functions for working with concordances, i.e. sorting, randomizing and sampling.

7.1 Corpus positions

By default, the concordance list is presented with the hits in corpus order. That's what the number on the far left means: the only hit for *oscillate* in BNC-BABY occurs as the 2,763,551st token in the database (as if all sentences in the corpus were written in one long string, ordered here by file name). In this unit, we'll look at some of the functions CQP provides for sorting and randomizing, pointing out issues along the way.

7.2 Sorting queries

There is a command called `sort` and its syntax is identical to that of `count` (§6): you need arguments to say what you want to sort, by what, on which position.

Let's illustrate this with an example: suppose you wanted to study the behaviour of the suffix *-ity*. We use a strategy to exclude the very frequent term *city*, and query `[word=".{2,}ity"%c]`. You could use `count` to determine the distribution by types. But say you wanted to browse through actual concordances, you need to sort the results to do that (as the output is in the order the matches occur in the corpus). Know the command?

If you use `sort Last by word`—what happens? The bulk of the hits comprise *Authority*, then *Christianity*, then *Community*, and so on. Notice what happens and why? CQP first sorts by capital letters if you don't use `%c` on `word` (it doesn't do this for sorting by hw because lemmas usually don't contain capital letters). See the exercise box for an example of multi-word tokens.

What will `sort Last by word desc` do, if `desc` means 'descending'? It will list your results [Z–A].

7.3 Sorting context

The more useful and more frequent application of `sort` is that of sorting the context of your search pattern, because that's really where you need context, i.e. to investigate a word's phraseologies.

Let's illustrate this: query `"interested"` and sort by context on the right (you should immediately notice the high frequency of the preposition *in* at R1). Since the syntax is identical to `count`, you should be able to formulate the command.

Run `sort Last by word %c on match[1]`. If you only have a single token, `match[1]` and `matchend[1]` are identical (i.e. you only have one anchor which is beginning and end of match at the same time). Note: `%c` may be desirable, even if you only queried a case-sensitive string, here *interested*. Why? Because `%c` will apply case-insensitivity to `match[1]` and will sort capital letters first.

7.4 Randomizing

Randomizing is a very important issue in corpus linguistics, for a number of reasons. While the text files in most corpora are randomly named, they are usually not randomly ordered, but grouped by genre (the BROWN family of corpora, for instance, has the press category first, then religion, then skills, trades, hobbies, and so on). So unless you are going to use all tokens of a query, you will need to randomize (or sample a random subset). But even if you only want to get a first glance, always bear in mind that results at the top of a list may not be representative for the phenomenon.

`sort` can also randomize (i.e. sorting in random order): instead of instructing it to sort by an attribute, tell CQP to randomize with `sort Last randomize` (you'll see that the numbers that indicate corpus positions now appear mixed). You can undo this and return to the corpus ordering with `sort Last`.

7.5 Sampling

It is often useful (or required) to look at a subsample of your query, say if it is too frequent to look at all of them. If you randomized your query, you can take the first 30, 50, 200 tokens (depending on what you're after). But you can also let CQP draw a random subsample for you with `reduce`.

`reduce` is the command, and you can sample either a fixed number of hits or reduce the query to a percentage share of the original result: `reduce A to 200` reduces the query `A` to 200 hits, whereas `reduce A to 15%` samples 15% of the original query `A`.

I didn't use `Last` in this case. This is because `reduce` only keeps the sampled data—and you can't get back the original. So it's advisable to 'copy' a query to a new drawer. If you want to reduce a query `A`, you should copy it to `B` before reducing (using `B = A`, see §2.1).

7.6 Excursion

You don't necessarily have to sample and randomize in CQP before exporting, as this job can also be handled in spreadsheet programs (§XX). But randomizing can be useful even for browsing through a few concordances.

CQP's sorting capabilities are far superior to sorting options in spreadsheet programs at this level of complexity. If you want to export the sorting in your concordance, sort to your liking and then export with `cat Last` and the re-direct operator (see §2.4, §10).

Exercises §7

i. `sort Last by word %c on matchend[2] desc reverse` — try and decipher this command. What could it mean? What will it do? Run the command for any query, e.g. `"interested"`. What did it do?

ii. How many hits does a 1% sample of the article *the* have?

8 Meta information

This unit introduces to accessing information on the text level, such as genre, register, or mode.

8.1 General

We dealt with access to token-level annotation up to now. But almost all corpora also come with metadata on the text level, i.e. information that applies to all tokens in a file or longer stretches of text within a file. Such metadata, for instance, can tell you whether a text contains language from the spoken or written medium (if that distinction is made in a corpus), or from a press or fiction subsection, or if it's academic text from the natural or the social sciences, or if it was written or uttered by men or women, and so on.

CQP calls this type of metadata *structural attributes* (as opposed to *positional attributes* for the token-level annotation like `pos` or `class`). This section will introduce you to the principle of how to access this information in CQP, *not* what it means conceptually. Corpora vary in the amount and detail of metadata, so you need to consult the official documentation. To see what's available in CQP, `show cd` ('context descriptor') or `info CORPUSNAME` provide overviews.

8.2 Query principle

As with accessing token-level annotation, CQP also distinguishes CQP syntax and corpus-specific data, although it is more complicated, because corpora vary considerably on the names of structural attributes.

Say you want only tokens of *shit* from the spoken part of a corpus. What you need to know is the attribute name for spoken data (one of which in the BNC is `text_mode`, also see `text_text_type`!), and the syntax CQP uses to match the *values* of that attribute. Try this:

```
[word="shit"%c] :: match.text_mode="spoken"
```

You add your restriction on metadata by adding two colons `::` to the query (again, spaces are not required), followed by `match.` and the name of the attribute (`text_mode`). These then equal one of the values of `text_mode` (*spoken* or *written*). As the values, *spoken* and *written*, are corpus-specific, they have to be enclosed in quotes `"VALUE"`, a principle parallel to the access to token-level annotation above.

8.3 Using regular expressions

As with conditions on tokens, you can use regular expressions on values, but not on attributes. To illustrate: if you want to compare uses of *significant* in academic texts from humanities vs. natural sciences:

```
"significant"::match.text_genre="W:ac:(hum|nat).*"
```

This looks for text that is *written* (`W`), academic (`ac`), either humanities/arts (`hum`) OR natural sciences (`nat`). Note: the colons inside the quotes are *not* CQP syntax here—it's how the builders of the BNC happened to name the values; see CheatSheet for attribute-value list.

8.4 Combining values

You can combine metadata. If you want to restrict by two *different* attributes, this is the only way (if you restrict on the *same* attribute, as *written* vs. *spoken* in `text_mode`, you can use regular expressions; see §8.3).

After the semicolon, you need `match.` for every piece of metadata; combine them with the operators `&` (AND) or `|` (OR). To find tokens that are either *spoken* or from *written-to-be-spoken* (which is *written*): `::match.text_text_type="written-to-be-spoken" | match.text_mode="spoken"`. Note that `&` returns 0 matches here—why? Because it looks for tokens that are from written *and* spoken mode at the same time (that's impossible, as text is classified *either* spoken *or* written).

8.5 Group

The `group` command works on any query result list. But it groups by text-level annotation, so it belongs here.

Say you want to know quickly how many times *THE* `[word="the"%c]` occurs in spoken vs. written texts, run the query, then display the distribution by the values of `text_mode` with `group Last match text_mode`. Also try `group Last match text_text_type`.

```
[QUERY]::match.ATTRIBUTE_1="VA.+" & match.ATTRIBUTE_2="VA.+"
```

CQP syntax	corpus-specific data
no regular expressions	*regular expressions possible*
usually identical across corpora	*variable across corpora*

Know your corpus!

Always consult the corpus documentation! Don't assume you know what's behind the names of metadata. Would you know just like that what the difference is between *spoken_demographic* and *spoken_context*? Or between `text_context`, `text_domain`, and `text_medium`? Most of the time you will not reach this level of detail, but if you do, make sure you know what the data source is. Plus, it's important that you know what text went into the corpus to assess whether the corpus is suitable to your question.

Note that quoted speech (e.g., in fiction or press), is *not* classified spoken. If the *text* is classified as *W:SOMETHING*, it is classified as written (because that's what *W:* texts are) — regardless of whether it contains transcriptions of (real or artificial) spoken language.

Exercises §8

i. The following restrictions for the string `"lovely"%c` do not return the same number of hits. Ideas why? `::match.u_sex="(male|female)"` `::match.u_sex=".*"`; try to solve with `group`.

ii. Look at the CheatSheet for structural attributes. Find all tokens for *PLAY* by males & females. Make sure you see why `u` and `text` attributes influence the result.

iii. Think about what the distribution of *THE* in BNC-BABY for *written* (183,444) vs. *spoken* (27,704) means. Does it make sense that *THE* is 7 times more frequent in written language? Why not? Can you find out why? And what you should do to compare the difference?

9 Settings & displays

This unit covers the technical principles about displaying options and changing settings with `set`.

9.1 Context

The standard cqp@fu settings for left and right context is 30 characters left and right of the keyword. If you need more context, use the command `set Context` plus the type and length of context you wish to set.

Characters. The default argument of `set Context` is the number of characters. If you want 90 characters on either side of the keyword, use `set Context 90`. If you want different numbers of characters on either side, use `set LeftContext 20` for 20 characters to the left, and `set RightContext 50` for 50 characters to the right.

Words. If you want ten words on either side, use `set Context 10 words`. Note that for CQP a 'word' is anything between spaces, so *girl friend* is two words.

Sentences. You can also use metadata of the file structure. Most corpora mark sentence boundaries (see the screenshot of the file that went into CQP in §4, where there is an `<s>` before the first word; this is a sentence tag). To display the entire sentence of your hit, use `set Context 1 s`, regardless of the length of the sentence. `set Context 3 s` returns the sentence with the keyword plus two sentences on either side.

Paragraphs & co. Some corpora have additional structural annotation like `<p>` for paragraphs, so you can use them to display the entire paragraph where your item occurs in. Some historical corpora have `<lb>` (for 'line break'), e.g. in SHAKESPEARE, so you can set the context as `set Context 3 lb` to have two lines above and below the search item.

9.2 Display metadata in output

One of the most useful things about `set` is that you can display metadata in concordances. What you should *always* do, for instance, because you need a reference when citing corpus examples in your work, is the name of the text file where your example occurs. But it's extremely useful with other data, too.

The pattern is `set PrintStructures 'ATTRIBUTE'` stating which structure(s) you want printed. To display the name of the text file (which is often *text_id*), use: `set PrintStructures 'text_id'`. To combine corpus file name *and* sentence id, call both, separated by a comma: `set PrintStructures 'text_id, s_n'`.

So if you want *text mode* displayed in your output, set it as `set PrintStructures 'text_mode'`. Now every subsequent query prints out the values *written* or *spoken*. Note: every new call of `set PrintStructures` sets exactly the specified settings (it 'deletes' previous settings). So the call we just made with *text_mode* overruled the settings we did with *text_id*. If you need all of them: `set PrintStructures 'text_id, s_n, text_mode'`. All settings are cleared on exiting CQP.

9.3 Restricting pattern by context

Suppose you want to check whether your school-book rule of *Do not use would in if-sentences!* is accurate for real English, you would not want to specify the number of tokens between *if* and *would*. But if you query: `[word="if"%c] []+ [word="would"%c]`, you get results that are really long—because CQP looks for *if* and then returns everything until it hits *would* somewhere (for some hits, this means jumping sentence boundaries or entire paragraphs). To restrict the query for searches within sentences (`s`, or `lb`, or `p` etc.):

```
[word="if"%c] []+ [word="would"%c] within s
```

This will still give you large stretches of text (and not all are counterexamples to the school rule), but it has the advantage that you can restrict the context without settling on a range (as you also would with `{min,max}`).

Tips: more on structural attributes

Some structural attributes really only make sense to be used in displays, than in queries with `::`. This is especially the case for attributes whose values have too many levels to be grouped or queried by sensibly. For example, the values of `text_id` have the names of the files. Since there are more than 4,000 files in the full BNC, it would not really make sense to group tokens by corpus files. On the other hand, it makes a lot of sense to display values of `text_id` so you know where your example comes from. (Though it *can* occasionally make sense to look for tokens only in files that start in A or to know how many tokens a file holds.) It depends on what you need the output for.

Tips: different corpora, different attributes

When switching between corpora, some settings may carry over, other might not. Generally, what is identical across corpora will carry over, so the `set Context` settings will (because it's CQP syntax). The obvious 'copying' of settings for PrintStructures occurs if two corpora have the same attributes (regardless of whether they differ in the respective values). So if you have set `text_id` and `s_n` for BNC-BABY, switching to the BNC will 'copy' this. Switching to BROWN copies `text_id` and ignores `s_n` (because BROWN doesn't have `s_n`).

To know what's available for a corpus, use `show cd` or read the more detailed info file (`info CORPUSNAME`). For example, the year in historical corpora is coded as `text_year` in CLMET, but as `letter_date` in PPCEEC. Many historical corpora also have time periods, which is `source_subperiod` in CED and `text_period` in CLMET. It will take a little practice to get displayed what you want, but it's worth it—and it always follows the CQP principle.

Exercises §9

i. To get a feel for the difference between `text_mode` and `text_text_type`, switch to the BNC, set the attributes to be displayed, then search for *lovely*. As all hits are displayed in corpus order, the values for these attributes are identical for the first few hits, randomize the output (`sort Last randomize`). Browse.

ii. Do the same for `text_mode` and `text_genre`, to give you a feeling for how values are represented (see §8).

10 Exporting & cleaning

This unit covers the export of data as files and how to clean the output for the import in spreadsheet software.

10.1 Export output as files

Remember from §2 that you can save the output of a CQP query in a file. Just as you print the last or a saved query with `cat NameOfQuery`, you can use that same command, but you need to specify the output channel. By default, the output channel is the screen, so if you can tell CQP you want it redirected somewhere else, you use the 'redirect' operator `>`. So to save a query to a file called *myQuery.txt*, print and redirect the contents as follows: `cat Last > "myQuery.txt"`.

Note that the quotes here are essential, but what you put inside the quotes as the name of the file is entirely up to you (although it should be something sensible and it should also contain a file extension such as *.txt*). Nothing seems to have happened, which is a good sign (if something did happen, it's probably a CQP syntax error message that you got the syntax wrong).

Where does the file end up? Well, remember how you work in the CQP room that you accessed from the server room, which I introduced as a virtual office with shelves and folders? That's where the file ended up. It's been written to your server room (or user space). You can also see the file if you login to the webinterface (*Datenablage*) on the ZEDAT website.

10.2 Cleaning output

The output of queries looks rather messy. But what you want to work with during manual annotation and analysis is tab-separated data, so that keywords and metadata occur in their own columns.

For this purpose, I wrote a script that cleans the data. While this step is not essential, it will make things easier. The clean-up script is called `CWBclean.pl`—and it was copied to your virtual office in the server room when you ran the settings script before your very first contact with CQP. It's a perl script that can be executed in your user space (after you leave CQP with `exit`).

To run (don't type `~$`, it indicates the server room):

```
~$ perl CWBclean.pl INPUT.txt > OUTPUT.txt
```

Where the inputfile is to the left and the name of the outputfile is to the right (using the 'redirect' operator `>`, meaning you write the result of the command to a file). You need to use a different name for the output file. If you have *lovely.txt* as input, use, e.g. *lovely_clean.txt* for the outputfile. Also note: different to the export within CQP, you do not use quotes around filenames here.

10.3 Download

You download your file in a zip folder from Datenablage on the ZEDAT website (the folder will be named with your user name). Unzip the folder (called 'Extrahieren' in Windows).

10.4 Software: text editors

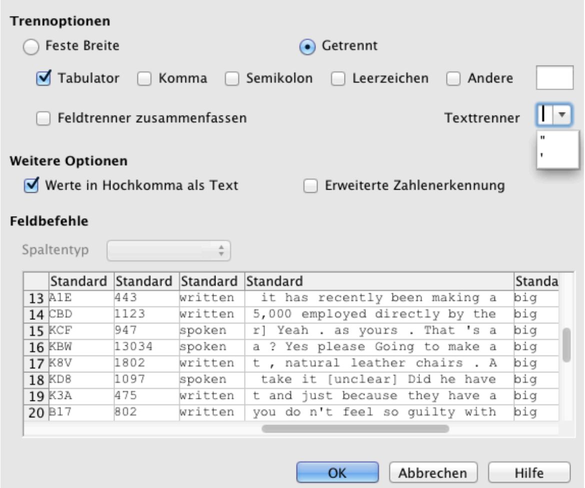
If you want to open and work with *.txt* files, you should always use a *real* text editor—MS Word, Apple Pages, or OfficeWriter are *word processors*, not text editors. Notepad (on Windows) or TextEdit (on Mac) are closer, but not close enough for many purposes. We recommend Notepad++ (Windows) and TextWrangler (Mac), both of which are free software.

10.5 Software: spreadsheet

Spreadsheet software (*Tabellenkalkulationsprogramme*) are used for most types of manual annotation and simple data analyses. We recommend OpenOffice or LibreOffice, though Excel can be used.

LibreOffice. Open an empty spreadsheet document (*Datei > Neu > Tabellendokument*). Then go *Insert > Table from file* (*Einfügen > Tabelle aus Datei*). A dialogue pops up. Navigate to the file you want to open. Select the (unzipped) file. A new dialogue window pops up: pay attention to *Field separator* (*Trennoptionen*). Tick *Tab* (*Tabulator*) and make sure the field for text separator is empty (*Texttrenner*)! Delete " or ' if it contains either of the two. Click OK.

OpenOffice now puts all info neatly in different columns. In the example below it has *text_id* in one, *s_n* in the next, *text_mode* in the third, left context in the fourth, the keyword in the fifth, and so on.



Trennoptionen

○ Feste Breite ● Getrennt

☑ Tabulator □ Komma □ Semikolon □ Leerzeichen □ Andere

□ Feldtrenner zusammenfassen

Texttrenner

Weitere Optionen

☑ Werte in Hochkomma als Text □ Erweiterte Zahlenerkennung

Feldbefehle

Spaltentyp

	Standard	Standard	Standard	Standard	Standa
13	A1E	443	written	it has recently been making a	big
14	CBD	1123	written	5,000 employed directly by the	big
15	KCF	947	spoken	r] Yeah . as yours . That 's a	big
16	KBW	13034	spoken	a ? Yes please Going to make a	big
17	K8V	1802	written	t , natural leather chairs . A	big
18	KD8	1097	spoken	take it [unclear] Did he have	big
19	K3A	475	written	t and just because they have a	big
20	B17	802	written	you do n't feel so guilty with	big

Excel. Excel deals very badly with encoding and other issues. So to avoid the major problems, open the *.txt* file in a text editor first, select and copy all text (STRG+A, then STRG+C), open an Excel document and paste the text (STRG+V). You see what I mean by 'dealing badly with encoding' when you notice funny symbols.

10.6 Filters

The row directly above rows with corpus data should definitely always be a header describing the information the column holds (e.g. TEXT_ID, MODE, KEYWORD, etc.). To insert, select the first row with corpus data, then *Insert/Einfügen > Row/Zeile*. Select the header row, set a filter and explore (*Data > Filter > Auto filter*).

11 Solution guide to exercises

11.1 Unit 1

i. `"nice"` (1,465), `"oscillate"` (1), `"get"` (6,888), `"count"` (220), and `"yank"` (0). Note that message "0 matches" means that there are no matches of that string (this is something different to "CQP syntax error" that you get if you forget the closing quotes, for example).

ii. You'll find hits for the words *exactly* as they appear in the corpus text. Thus, what you do *not* find in (i) are inflected forms (e.g., *gets* or *nicer*) or strings that have capital letters (such as ***Get*** *help*!).

iii./iv. E.g. `"Fantastic"` returns 3 hits and `"GET"` 6.

iv. You would need to formulate your query such that it ignores case (small and capital letters). To find inflected forms, you would need a solution to query word forms or lemmas—CQP would not be of much use if you had to query `"get"`, `"Get"`, `"GEt"`, `"GET"`, and so on (not to mention all of the inflectional forms and small and capital letters!). That is, simply, you'd need either wildcards or, if your corpus has linguistic annotation, you'd need to know how to access that information (but see §3 & §4).

11.2 Unit 2

i. First, run a query for a form. Then save that query with `nice = Last` for querying `"nice"` before you run the next query (as new queries override the contents in your temporary drawer `Last`). Note that CQP does not 'check' whether the name for your query actually makes sense (so you could save the `"get"` query as `oscillate`). If you used the strings as names for saving queries, you should have received an error message while trying to save the hits for *count* (`count = Last`). The error message says, among other things, `unexpected '='`—CQP did not expect `=`. Why? Because `count` is a command in CQP, like `show`, and commands have their own argument structure, i.e. `count` expects the *name* of a drawer with contents to be *counted* (see §6 for counting concordances).

11.3 Unit 3

i. These are all only suggestions. There are usually multiple ways to achieve the desired result, sometimes with slight differences. `"snows?(ed)?(ing)?"` (145; 120 without `%c`), `"rains?(ed)?(ing)?"` (314/342).

ii. `"(snow|rain)s?(ed)?(ing)?"%c` (487), `"(snow|rain)(s|ed|ing)?"%c` also works. An alternative, with more typing and slightly more clumsy: `"(snows?(ed)?(ing)?|rains?(ed)?(ing)?)"%c`. In the more compact first query, you put all that varies in `()` varies between forms, in the second you have compact, but full queries on either side of the OR operator `|`.

iii. The issue is orthography (<s> vs. <z>). Just because *realise* tends to be the preferred British variant and the BNC contains British English, doesn't mean you should assume there are no <z>-variants in the BNC. Both `"reali[sz]e"` and `"reali(s|z)e"` fix this (here: 252 case-sensitive hits). Note that you do not need `?` here as *s* or *z* are not optional (but it won't make a difference if you use it).

iv. These will find *nice*, *nicer*, *nicest*: `"nice[rs]?t?"`, `"nicer?(st)?"`, `"nicer?s?t?"`—1,633 if you used `%c`, and 1,500 if not. The first query may seem a little strange, because it groups two characters together that occur in different forms of *NICE*, i.e. *r* and *s* (*nicer* and *nicest*), but they can be grouped here because they occur in that position, i.e. as the fifth character in their respective cases and `[rs]` is in the fifth place.

Queries for *old*, *older* and *oldest* include `"old(er)?(est)?"%c` or `"olde?[rs]?t?"%c`. They have slightly different outputs (2,491 vs. 2,506), because the latter also finds *olde* and *olds*.

v. `"g[eo]t[ts]?(en|ing)?"%c`, 18,191 hits. This will work, too: `"(get|gets|getting|got|gotten)"%c` but it is worth grasping the principle of the first.

vi. (a) `"un.+"%c`. If you use `*` instead of `+`, this will also match *UNO*, *uns*, *une* etc. (you find *only* these if you query `"un."%c`. (b) `".*ment"%c` or `".+ment"%c`. If you want the plural forms of *-ment* words too, include *s* as optional: `".+ments?"%c`.

(c) `"un.*ments?"%c`— the majority of hits are indeed *unemployment*, but there is, e.g. *under-achievement*, *undernourishment*, and *understatement*. So see how `*` and `+` also match the hyphen -.

11.4 Unit 4

i. You can query each word with values for verbs and nouns, respectively. Doing it this way is easier than using pos information, since you want *all* noun or verb tags (and there are a lot of verb tags). But see the next exercise for a different strategy. (Also see §6 for the `count` command.)

```
[hw="snow" & class="VERB"]
[hw="snow" & class="SUBST"]
```

ii. If there are 1,920 hits for `[hw="round"]`, but only 175 of them are nouns and only 53 verbs, you could use `show +class` to display class information in the concordance (then run the `[hw="round"]` query again or use `cat Last` if it was your last query). Values for `class` so identified and queried are `ADV` (936 hits), `PREP` (669), and `ADJ` (94). (To switch off, type `show -class`, re-show your 'clean' concordance with `cat Last`).

iii. The solution is actually fairly simple (and it saves you the trouble of going through all 25 verb pos tags): you can use regular expressions on pos tags, too (because you use them on *values*). To find all verb uses of *round*, including those with ambiguity tags, query `[hw="round" & pos="V.+"]`: this means 'find all instances of the lemma *round* that have a pos tag starting in V (this includes tags with more than three-letters). The uses of adverbs and nouns are identical, just with A and N, respectively.

iv. Since ambiguity tags contain a hyphen `-`, and if you want all of the tokens that have a tag with a hyphen, you can formulate a pattern that matches a hyphen surrounded by 3 characters on either side, i.e. `[pos=".+-.+"]` or `[pos=".{3}-.{3}"]` (you will meet the `{}` expression in the next unit). This translates to 'in the pos-column, find any character, which is repeated more than once {three times} until you hit a hyphen, followed by any character which is repeated more than once {three} times. This will give you 144,916 tokens in the BNC-BABY where the tagger was unsure—that's 3.6% of all non-punctuation tokens!

But how do you know that's 3.6%? Well, if you want that figure, you would need to know how many hits there are in total, and the way to find that is to think that you want every token, i.e. that you do not specify any condition in `[]`. If you query `[]`, this give you 4,644,834 tokens. Now that's actually 'only' 3.1% (144916/4644834=0.0312). But querying `[]` actually also matches every instance of punctuation in the BNC-BABY. So to measure the ratio of ambiguity tags to total word tokens more realistically is to compare it to the number of word tokens in the BNC-BABY.

The way to find this out is to *exclude* punctuation in your 'match-all' query, i.e. formulate an IS NOT condition. Look up the `class` tag for punctuation (which is STOP) and run `[class!="STOP"]` to find only word tokens. This gives you 4,024,537 hits—and 4024537/4644834=0.036, i.e. 3.6%. This should really tell you to always use regular expressions on pos tags in the BNC.

11.5 Unit 5

i. To query X-and-X coordination, you could simply use `[]"and"%c []`. To find 'cases that merit further investigation' is actually not easy to spot at all—but think about it: *, and 1992*, *base and the*, or *rated and is* don't really seem to be all that revealing. Note how such 'uninteresting' cases have different word classes on either side of *and*. So, possibly, more interesting types are to be found if the items on either side are of the same word class. Find *ADJ-and-ADJ* or *ADV-and-ADV*-coordination (or any other class to practice): `[class="ADV"] "and"%c [class="ADV"]`, browse the results and think about how many of these types could merit further investigation (think along the lines of 'non-random' coordination or juxtaposition).

ii. `"the"%c [class="(ADV|ADJ)"]{2,4} [class="SUBST"]`

iii. `[word="re.+"%c`] also gives you *read*, *real*, or *rest*, i.e. words that start with *re-*, but where *re-* is not a prefix. One, but problematic, solution is to think that *re-* prefixes are probably followed by more than two characters, so `[word="re.{3,}"%c]` may solve this. To include (the actually very infrequent) *redo*, you could combine this with an OR condition: `[word="re(.{3,}|do)"%c]`. This is still far from perfect, but it's a very good illustration of why you must look at what's found and think about what's not (but what should've), and how you need to refine your query continually to strike the best balance between PRECISION (how much of what's found is an instance) and RECALL (how much of what's an instance is actually found).

iv. For typical proverbs, BNC-BABY is usually too small a corpus, and there are no hits for *kick the bucket*, and only two for *barking up the wrong tree* (and it would be surprising if you found a sizable amount for your own examples of proverbs). One solution is to switch to a larger corpus (BNC).

kick the bucket: there are only 13 instances of *kick the bucket*—and if you investigate the results, it turns out that some are 'literal' uses and some others are meta uses, i.e. that people talk *about* its meaning. One further instance can be retrieved if you query for an adjective (or a match-all token) before *bucket*, which, again, is a meta use, not really an instance.

barking up the wrong tree: if you start simply with *bark* (either with the 'as-is' string, the *ing*-form, or the head word/lemma), you will have a hard time going through the concordances until you find actual instances of the idiom (there are 1,239 hits for hw bark in BNC, both noun and verb hits). If you start to expand your query the entire string, this will obviously only give you invariant instances (`[hw="bark"] "up"%c "the"%c "wrong"%c "tree"%c`). But, if you suspect that the slot occupied by *barking* is variable, you can query `[hw="bark"] "up"%c` to alleviate some of the problems. As it turns out, there seems to be no real variability in the *barking* slot. But what else is interesting? Two things, at least: first, the sequence *barking up* seems to *only* occur in instances of the proverb, and that, second, the string *the wrong tree* shows some interesting variability (*yet another wrong tree*, *the right tree*), but that it is generally in the form of the idiom's schema.

11.6 Unit 6

i. `[class="VERB"]`—806,574 verbs; `count Last by word %c` to create case-insensitive word-form frequency list and `count Last by hw` to create lemma list (these commands might take a few seconds, the program is counting and summing up a lot of tokens!). The lemma list is slightly more revealing for a quick glance of 20 to 25 verbs, but what's obvious is that forms of the auxiliaries *BE*, *HAVE*, *DO* and the modal verbs (*WILL*, *CAN*, *COULD*) are very frequent classes of verbs, with some of the so-called 'lexical verbs' (*SAY*, *GET*, *GO*)—although some of the lexical verbs are arguably also often used as auxiliaries (especially *GO*). So if you were to come up with three major classes, auxiliaries, modals, and lexical verbs could be a plausible grouping.

Incidentally, they also tend to have their own types of pos tags: all forms of *BE* have pos tags that start in VB (VBI for infinitive *be*, VBG for *being*, etc.), the tags for *HAVE* tags start in VH, and those for *DO* in VD. Modals have their own tag (VM0). All remaining verbs have tags starting in VV (VVI, VVG, etc.). So you could make frequency lists (then use `count`) by querying them `[pos="V(B|H|D).+"]` for auxiliaries, `[pos="VM0.+"]` for modals, or `[pos="VV.+"]` for lexical verbs.

Note that this works for the BNC and the BNC-BABY. The tagset is CLAWS. If you worked on BROWN or CLMET, you'd have to take the different tagset into account: the PENN tagset does not have as many fine-grained distinctions in the verbal area (which I initially thought of as being a major drawback). But CQP can fix the problem with its powerful query syntax: if you wanted to differentiate between the auxiliaries *BE*, *HAVE*, and *DO*, and lexical verbs such as *SEE*, *GIVE*, and *GET*, which all have tags starting in VB (modals have their own tag, MD): for a frequency list of auxiliaries, you can query the VB-tags and include a condition on lemmas: `[pos="VB.+" & lemma="(be|have|do)"]`, for lexical verbs, i.e. not *BE*, *HAVE*, or *DO*, *exclude* auxiliaries: `[pos="VB.+" & lemma!="(be|have|do)"]`

ii. `[class="ADV"]` — `count Last by hw`. The most frequent item is *not* (*not* and *n't* if you counted by `word`). This might be slightly surprising, but is due to the fact that while *not* is pos tagged as XX0, its class value is ADV, same as for adverbs (which have AV0 as a pos tag). To exclude *not* from a frequency list of adverbs, you can query `[class="ADV" & hw!="not"]`, or `[class="ADV" & pos!="XX0"]`, then count again.

Take-home message: know you corpus, know the tagset! (And by now hopefully appreciate CQP's combinatorial powers on formulating conditions and the control you actually have over it!)

iii. `[class="ADV"] "and"%c [class="ADV"]`— `count Last by word %c on match[-1]`, so the comma is the most frequent item to the left; `count Last by word %c on matchend[1]`, where the period is the most frequent item to the right. Now think about what this could mean more generally about the distribution of *ADV-and-ADV*-coordination.

iv. `"the"%c [class="ADJ"] [word="car"%c]` to find instances of *the ADJ car*; `count Last by hw on match[1]`—*good* is most frequently modifying *car*. To check the most frequent *string* that comes between *the* and *car*, remind yourself that this does not necessarily have to be an adjective (even if you cannot think what else could come in a prototypical NP), so the strategy is to impose no condition on the second token: `"the"%c [] [word="car"%c]`, then `count Last by word %c on match[1]`—and the most frequent string here is *police*, quite obviously from the compound *police car*. And this is very generally an issue with English data in particular—finding noun phrases in their entirety can be a pain and the distinction between compounds and *ADJ-NOUN* sequences is always a problem, not just because taggers are unreliable.

11.7 Unit 7

i. `sort Last by word %c on matchend[2] desc reverse` — this sorts by word on R2, i.e. the second token in the right context in descending order, in reverse order. So for `"interested"` this first lists the tokens that have punctuation at R2, the sorting principle becomes obvious a few lines further down, where you have *in community*, *him very*, . *Very*, *in any*, … *in money*, *in you*, …, *in it*, . *Let*, …, *in concerts*, until the last line which does not have punctuation in R2, is *in a*. Notice how it's sorted in descending order (Z–A) by the last letter ('reverse') of R2 (*communit**y***, *ver**y***, *an**y***, *yo**u***, *in i**t***, *concert**s***, ***a***). This may seem absolutely useless—but you realise the power of what CQP commands can really do (and who knows what you can use it for one day!).

ii. Query `"the"%c` (211,148), `reduce Last to 1%`. To establish the number of hits, `cat Last` or `show named` will give you the 1% sample as 2,111 tokens.

11.8 Unit 8

i. `"lovely"%c::match.u_sex="(male|female)"` returns 400, while `"lovely"%c::match.u_sex=".*"`: returns 436 hits. Now this indicates that there must be hits that are neither classified as coming from males or females (since the `.*` between quotes matches more than *male* of *female*). So if you group the query (the one you restricted with the regular expression) by the attribute `u_sex` (`group Last match u_sex`), you find your 'missing' 36 hits classified as *unknown*. Check with the list of structural attributes in the CheatSheet: `u_sex` can have the values, *male*, *female*, *unknown* and ---, meaning that the data holds utterances by speakers whose sex is unknown, could not be determined, or where speakers chose not to state it.

ii. The BNC has a spoken and a written part, and `u_sex` refers to utterances in the spoken component, while `text_author_sex` refers to written text. (`u` attributes are spoken part only, with `text` attributes it's a bit more difficult; they refer to texts from both written and spoken part). So to find all hits for *PLAY* from both males and females in the entire BNC-BABY, query `[hw="play"]::match.u_sex="(male|female)" | match.text_author_sex="(female|male)"`—this will of course give you all noun and verb uses and if you wanted to specify that, you could do so of course in the token query.

iii. Surely it is not really plausible to assume that *THE* is so much more frequent in the written part than in the spoken part, although some differences are to be expected. The issue is, of course, that you may have different amounts of text in the different components so that the comparison of raw frequencies is useless. So if you have more written texts, chances are, of course, that items are also more frequent there. So you would either have to know how many tokens there are in the different components or know how many tokens in the different components are not the to make any kind of comparison.

Knowing the total number of tokens in a (sub-)corpus allows you to calculate a relative frequency (generally given as per-million-words; knowing the total number of tokens not identical to your search word allows you to create contingency tables (see §XX).

To find out the total of words per subcorpus, you can query `[]`, i.e. match all tokens, and then group by `text_mode`: `group Last match text_mode`, so that gives you 3,434,900 tokens for the written and 1,209,934 for the spoken part. So the relative frequency of *THE* per-million words comes out as follows:

(183,444 x 1,000,000) / 3,434,900 = 53,405.9 per million words for the written component and

(27,704 x 1,000,000) / 1,209,934 = 22,897.1 pmw for the spoken component. So *THE* indeed seems much more frequent in the written language. It depends on the interpretation of the reasons to assess whether (and how) this is linguistically meaningful.

You also often see relative frequencies calculated per 10,000 words, so you can substitute that in the formula. The formula to calculate is F_{word} * BASELINE divided by F_{corpus}, where F_{corpus} is the total amount of tokens in the corpus (here: all tokens, including punctuation tokens). For our example, this would be 534.1 and 228.9, respectively.

A different way to represent frequency distributions is in a so-called contingency table, which is the basis of many statistical tests. You fill the table with the numbers of, e.g., the variable LEXICAL ITEM (rows, ¬ means 'is not') cross-tabulated with the variable TEXT MODE (colums):

	WRITTEN	SPOKEN	Total
THE	183,444	27,704	211,148
¬*THE*	3,251,456	1182230	4,433,686
Total	3,434,900	1,209,934	4,644,834

Both strategies of normalization make it possible to compare frequencies across corpora of different sizes. Consult textbooks on corpus statistics for background, explanations, and examples.

11.9 Unit 9

i. Load the BNC, set the metadata *text_mode* and *text_text_type* to be displayed in the concordance: `set PrintStructures 'text_mode, text_text_type'`, query *lovely* and `sort Last randomize` to randomize the list. You should see that text_mode is more coarse-grained than text_text_type; the latter is a bit more of a continuum. Consult the corpus documentation for the BNC for what the continuum is.

You can group to get a summary of distribution: `group Last match text_mode` or `group Last match text_text_type`.

Note that the more PrintStructures you display, the messier the output gets. This will be taken care of by the cleaning script that you can run before downloading the data file (§10).

ii. Try out a few things to get a feel for the technology and the representation of values: for example, `set PrintStructures 'text_mode, text_domain'` or `set PrintStructures 'text_domain'`, then `cat Last` or run a new query for lovely (or any other item).

Group the attributes to see a summary of distribution: `group Last match text_domain` or `group Last match text_mode` etc.