CQP, R, and rcqp in a Nutshell

Felix Bildhauer Roland Schäfer

August 28, 2013

1 R

1.1 Introduction

R is a free universal statistics (computing and plotting) environment, and programming language.[1] It works on all major platforms as a stand-alone (console) program or with diverse graphical interfaces. There is a lot of information and help out on the net regarding all aspects of R use and programming.[2] We use RStudio as a graphical environment for R, which is available for local installation and as a server version.[3] Point your browser to our RStudio server to log in and use your "k" account credentials:

```
https://hpsg.fu-berlin.de/rstudio/
```

By default, R (even under most GUIs) works as a console application. You type a command (terminated by `<RETURN>`), and R answers with a return value (if there is any) or an error message. Certain functions also lead to a plot being displayed. You will use primarily **operators** (like `+`) and **function calls** of the form `function(argument1, argument2, ...)`. Type

```
40 + 2
```

to see how R works as a calculator, and

```
R.Version()
```

which is a function without arguments, to see R's reaction to a function call. To get help for a function type `?function`, e. g., try

```
?log
```

In order to see (in the form of a context help) which arguments a function takes, type `function(`, then press the `<TAB>` key. Try

```
plot(<TAB>
```

In this course, we will also use **R scripts**. In scripts, you can define a series of commands to be executed when the script is loaded, and you can define your own functions to encapsulate any series of commands which you want to execute on diverse inputs in all kinds of contexts. Scripts are saved in files (simple text files, usually named `.r` or `.R`). We will use a script called `scqp.r`, for example, which contains a wrapper around some rcqp functions to get an easier start with rcqp. In RStudio, select *File – Open File…* and open `scqp.r`. After the file has loaded, press *Source* in the upper right corner of the script window. In the **Console** window, it should say *"You have loaded the simple rcqp wrapper for the HPSG pre-conf tutorial."*, and in the **Workspace** table, there should now be two entries under the header **Function**, namely `cqp(acorpus, aquery)` and `uk12(aquery)`. We will use these functions later.

The **Workspace** windows/table shows you the **objects** (in the broadest sense) which you have loaded/created in the current R session. Thus, you can see which data objects and functions you have available. Only your self-defined custom functions are shown. R has a lot of built-in functions, and **packages** also contribute functions, which are also not displayed in the **Workspace**.

The only **package** we need to load is rcqp, which was already pre-installed in your account. Go to the lower right panel of RStudio, select the **Packages** tab, search for rcqp in the list, and activate it by selecting the check box left to the rcqp entry.

The following list provides an overview of some of the things you can do with the R language. All examples in `typewriter font` you can try in the R **Console** and observe R's reaction.

1. Standard arithmetic using common operators:
   - `40 + 2`
   - `143 - 101`
   - `14 * sqrt(9)`
2. Assigning values to variables by the mostly (but not always) interchangeable operators `=` and `<-` (observe the **Workspace**):
   - `A = 42`
   - `B <- A/2`
3. Displaying data by just typing the name of the variable:
   - `B`
4. Removing data from **Workspace**:
   - `rm(A)`
5. Creating sequences of values (try to see the effect):
   - `1:10`
   - `rep(12, 7)`
   - `seq(12, 24, by=3)`
6. Accessing values in **vectors** (i. e., arrays, cf. more detail below) by index (execute these in sequence)
   - `MyVector = seq(12, 24, by=3)`
   - `MyVector`
   - `MyVector[2]`
   - `MyVector = seq(3, 300, by=3)`
   - `MyVector[2:12]`
7. Creating a vector of strings (data type **character**, enclosed in " or ') and selecting array elements by using comparison operators like `== or !=`:

[1] `http://cran.r-project.org/`

[2] `http://www.statmethods.net/` is probably one of the most popular R help website.

[3] `http://www.rstudio.com/`

- `Leaders = c("Merkel", "Obama", "Cameron", "the French guy")`
- `which(Leaders=="Cameron")`
- `Leaders[which(Leaders == "Cameron")]`
- `Leaders[which(Leaders != "Cameron")]`

8. Regular expressions to search in character vectors:
   - `grep(".*French.*", Leaders)`

1.2 The most common non-elementary data types

Above, we have used elementary data types (numbers, characters) and the `vector` data type. There are the following important non-elementary data types:

1. `vector`: an array (in the terminology of almost all other programming languages) of data objects of the same type.
   - Explicit creation:
     c(element1, element2, ...)
   - Implicit creation for example by:
     value1:value2
     seq()
     rep()
   - Member access by single subscripts in `[]` as in `MyVector[2]`
   - Note: What is called an `array` in R is actually an n-dimensional matrix. We do not use the `array` data type here.
2. `list`: a collection of data objects of arbitrary types.
   - Explicit creation:
     `list(4, "Merkel")`
   - Member access by single subscripts in `[]`
3. `matrix` a 2-dimensional matrix of values of identical data types, where rows and columns can be treated as vectors.
   - Explicit creation from a vector using `matrix()` as in this example for `Leaders` from above:
     `Statespersons <- matrix(Leaders, nrow=2, ncol=2)`
   - Member access by two-dimensional subscripts in `[row,column]`, where leaving out `row` or `column` selects all rows/columns:
     `Statespersons[2, 2]`
     `Statespersons[, 2]`
4. `data.frame`: like `matrix`, but columns can have different data types.
   - Explicit creation by `data.frame()`, typically by joining vectors as columns with `cbind()`:
     `Importance = c(2, 1, 3, 4)`
     `Politics = data.frame(cbind(Leaders, Importance))`
     Now click on `Politics` in the **Workspace** to see the data frame.

2 CQP and rcqp

2.1 Introduction and CWB's Internal Data Format

CQP is the query tool from the IMS Open Corpus Workbench (OCWB).[4] It can efficiently index and query corpora up to roughly 2 billion tokens. It is token-based, which means that trees and similar structures cannot be represented elegantly/ at all. However, regions of tokens can be easily annotated with attributes like chunks, sentences, etc.

- A CWB corpus is a sequences of positions of tokens. The index of the positions goes from 1 to the index which corresponds to the total number of tokens in the corpus.
- A **positional attribute** is an annotation of a position, for example `word` (the original token word form), in our case also `pos` (part of speech) and `lemma` (base form for `word`).
- A **structural attribute** is an annotation of a range of positions, for example (in UKCOW2012D) `nc` for noun chunks, `s` for sentences, `doc` for documents. Structural attributes have their own indices.
- Structural attributes can have **string values** associated with them, for example `doc_url` in COW (non-X(S)) corpora, providing the URL of the document or `s_freq` in COW XS corpora, specifying the frequency of the sentence in the original non-shuffled corpus.

For example, position `123243` in UKCOW2012D has the following attributes:

- Positional:
  - `word="theatrical"`
  - `pos="JJ"`
  - `lemma="theatrical"`
- Structural:
  - `nc=-1`
    (= It is not within any noun chunk.)
  - `doc=5589`
    (= It is within the document indexed as 5268.)

The `doc_url` indexed as 5589 is associated with the string:

`http://www.csv.org.uk/volunteering/environmental-conservation`

[4] `http://cwb.sourceforge.net/`

For the curious: This information can be obtained in R using rcqp as follows (including the expected R response after >):

1. `cqi_cpos2str("UKCOW2012D.word", 123243)`
   `> [1] "theatrical"`
2. `cqi_cpos2str("UKCOW2012D.pos", 123243)`
   `> [1] "JJ"`
3. `cqi_cpos2str("UKCOW2012D.lemma", 123243)`
   `> [1] "theatrical"`
4. `cqi_cpos2struc("UKCOW2012D.nc", 123243)`
   `> [1] -1`
5. `cqi_cpos2struc("UKCOW2012D.s", 123243)`
   `> [1] "5589"`
6. `cqi_struc2str("UKCOW2012D.doc_url", 5589)`
   `> [1] "http://..."`

2.2 The CQP Query Language

2.2.1 Basics

CQP can be used as a separate console application. rcqp, however, is a package for R which allows R users to get results and statistics from CQP directly in R.

If the rcqp library is loaded and the `scqp.r` script was "sourced" (as explained above), we can explore the CQP query language by querying UKCOW2012D using the wrapper function `uk12()`. The function takes a single character (= string) argument, the CQP query. It returns the internal name of a CQP subcorpus (something cryptic like `"Uzhtacpdye"`), which can be reused later, and it prints the concordance for the query at the console as a side effect.

Try:

```
uk12("'daylights'")
```

This is the simplest possible kind of CQP query: A single literal token in ' '. Notice that the outer " " are there for "telling" R that the argument your are passing to `uk12()` is actually a string, and the inner ' ' are part of the CQP query (R does not care about them, because they are inside " "). All CQP queries are a variation on this simple theme: **With CQP, you query for tokens or sequences of tokens.** A sequence of tokens requires **quoting every single token in ' '**. Internally, CQP simply constructs a list (a **subcorpus**) of the positions where the match (matching your query) begins, and where it ends. The start position is called **match** (or **match[0]**), the end position is called **matchend** (or **matchend[0]**).

Try:

```
uk12("'living' 'daylights'")
```

Later, we will look at a script that also gets statistical information about the construction.

2.2.2 Full token specification

The above token specification is an abbreviation for the following full variant, where a token specification is enclosed in [ ], and the positional attribute `word` is explicitly specified:

```
uk12("[word='living'][word='daylights']")
```

The advantage of this full mode of specification is that we can add more requirements for matching tokens with a logical *and* as `&`, for example `pos` and lemma conditions. Try this to check how well the tagger identified the lemma *house* used as a verb:[5]

```
[lemma='house' & pos='VB']
```

To search for a string in a **case insensitive mode**, append `%c` to the string after the closing ':

```
[word='obviOUSly'%c]
```

All conditions within `[ ]` must be fulfilled by a matching token. If you do not specify anything, each token matches:

```
[word='totally'][][word='guy']
```

2.2.3 Regular Expressions

The good news is that **CQP accepts regular expressions everywhere whithin ' '**:

```
[lemma='house' & pos='VB.*']
```

For those not familiar with regular expressions, here is a short list of things you can with them. Simply put, they allow you to underspecify strings of characters.

Stand-ins for characters:

- Any character: `.`
  For example: `ba.h` matches *bath* and *bash*, but also *ba3h* or *ba#h*.
- One out of a class of characters listed in `[ ]`
  For example: `ba[ts]h` matches only *bath* and *bash*.
- One character from a range of characters specified in `[ ]` with `-`
  For example: `[a-z]` matches *a* to *z*, but not *A* or *5*, etc.
- An example of the combination of the last two: `[a-z378]` matches lowercase letters from *a* to *z* and the numbers *3*, *7*, and *8*.
- Alternatives with `|` as *or* and groups in `( )`
  For example: `h(ouse|erring)s` matches *houses* and *herrings*.

Repetitions of characters and groups:

- Zero or arbitrarily many repetitions: `*`
  For example: `BAM*` matches *BA*, *BAM*, *BAMM*, etc.
- One or arbitrarily many repetitions: `+`
  For example: `BAM+` matches *BAM*, *BAMM*, etc.
- From `n` to `m` repetitions: `{n,m}`
  For example: `BAM{2,3}` matches exactly *BAMM* and *BAMMM*.

Because some characters acquire a special meaning, they have to be **quoted** with a preceding `\` when used literally:

- literal . is denoted by `\.`
- literal + is denoted by `\+`
- literal * is denoted by `\*`
- literal *{}()[]* are denoted by `\{\}\(\)\[\]`

[5] From now on, the `uk12("")` wrapper is omitted for space reasons.

2.2.4 Quantification

Token specifications in the CQP language can be quantified using the same quantifiers as used for regular expressions. Therefore, you can (by way of example):

- Search for sequences of one or more adjectives:
  `[pos='JJ']+`
- Search for sequences of two or more auxiliaries:
  `[pos='MD']{2,}`
- Search for sequences of one determiner, zero or many adjectives, and a single noun (long query time!):
  `[pos='DT'][pos='JJ']*[pos='NN']`

2.2.5 Integrating Structural Attributes

Beginnings and ends of structural attributes can be queried in combination with token specifications. For example:

- Search all NP chunks beginning with *few*:
  `<nc> [word='few'] []* </nc>`
  Expected query time is around 4 minutes!
- Search an arbitrary word in documents from a specific source:
  `[word='Merkel'] :: match.doc_url = '.*\\.guardian\\.co\\.uk.*'`

2.3 The `cqi_` Functions in rcqp

rcqp provides two ways of accessing CQP corpora:

1. the low-level `cqi_` functions (preferred in the COW barn)
2. a high-level wrapper around `cqi_`, implemented in the `corpus` and `subcorpus` data types

We demonstrate the use of the `cqi_` functions in the script `daylights_simple.r`, which can be loaded like `scqp.r`.[6] However, a few elementary functions are first explained here before you should look at the scripts. All commands are given by example, and the examples should work in our R installation.

1. Show installed corpora:
   `cqi_list_corpora()`
2. Show information for an installed corpus:
   `cqi_corpus_info("UKCOW2012D")`
3. Show subcorpora (= queries) for a corpus:
   `cqi_list_subcorpora("UKCOW2012D")`

Notice that the following functions provide a very quick way of getting frequency information from CQP subcorpora (=queries).

- `cqi_fdist1()`
  Get a one-dimensional frequency distribution over a single positional attribute.
- `cqi_fdist2()`
  Get a two-dimensional frequency distribution over two positional attributes.

This shows you how to combine the `uk12()` wrapper with these functions. Remember that the wrapper returns the CQP subcorpus for a query, and that we can assign this to a variable and reuse it.

1. Perform the query and catch the subcorpus name:
   `Admire = uk12("[lemma='admire'] [pos='DT'] [pos='NN']")`
2. Get the frequency table for the lemmas at matchend:
   `AdmireNounFreq = cqi_fdist1(Admire, "matchend", "lemma")`
3. Lookup the strings for the lemma indices (lexicon lookup):
   `AdmireLemmas = cqi_id2str( "UKCOW2012D.lemma", AdmireNounFreq[,1])`
4. Put lemmas and frequencies in a data frame:
   `AdmireFinal = data.frame( cbind(AdmireLemmas, as.data.frame(AdmireNounFreq[,2])))`
5. Give the columns in the data frame nice names:
   `colnames(AdmireFinal) = c("Noun", "Frequency")`
6. Now click on `AdmireFinal` in the **Workspace** to inspect the data frame.

[6] The script `scqp.r` uses the high-level functions, so you can look at it to see how you can use them and mix them with `cqi_` functions.