NAME

CQi tutorial -- how to run a CQP query

DESCRIPTION

This tutorial gives an introduction to the **Corpus Query Interface** (**CQi**). After a short description of the *data types* used by the CQi, a simple application is presented in detail. Code samples are given in a kind of pseudo-code that should be familiar to C, Java, and Perl programmers. For general information about the CQi architecture or for a full command reference, please refer to the published documentation.

CORPORA, SUBCORPORA, AND ATTRIBUTES

In the CQi, **Corpus names** are uppercase strings such as

```
BNC, HANSARD-EN, UP.
```

Subcorpus names begin with a capital letter followed by zero or more lowercase characters (including digits and hyphens). A full **subcorpus identifier** consists of both the name of the physical corpus and the subcorpus name, separated by a colon:

```
BNC:A, UP:Last, HANSARD-EN:Collocations-1 .
```

`Last` is a special subcorpus, which contains the results of the last CQP query. It will be automatically deleted whenever a new query is executed.

Attribute names are lowercase strings. Currently, the CQi protocol defines the following three types of attributes (typical attribute names are shown in parentheses):

- positional attributes (`word, pos, lemma`)
- structural attributes (`s, p, chapter`)
- alignment attributes (`hansard-fr`)

A full **attribute specifier** consists of corpus and attribute name, separated by a period:

```
BNC.word, UP.s, HANSARD-EN.hansard-fr
```

No distinction is made between the three types of attributes regarding naming conventions.

DATA TYPES

The following data types are defined in the CQi specification.

BYTE / BOOL
: A `BYTE` is an unsigned 8-bit integer. A `BOOL` is a `BYTE` that is guaranteed to be either `0` (*False*) or `1` (*True*).

WORD
: A `WORD` is an unsigned 16-bit integer. `WORD` quantities are used only in the client-server communication streams. As a command or response code, a `WORD` is interpreted as a sequence of two `BYTE`s.

INT
: An `INT` is a signed 32-bit integer. It is the standard numerical data type used by the CQi. When transmitted through a byte-stream connection, an `INT` must be sent in *network* (i.e. big-endian) byte order.

STRING
: A `STRING` is a sequence of `BYTE`s. Although CQi `STRING`s are *not* null-terminated, they *must not* contain `NULL` bytes. Internally, a `STRING` of length n is transmitted as

```
WORD    n
BYTE    <character_1>
...
BYTE    <character_n>
```

Since the characters in a `STRING` are unsigned bytes, valid character codes range from 1 to 255.

BYTE_LIST, BOOL_LIST, INT_LIST, STRING_LIST

These are the **array** data types available in the CQi. In this tutorial, the elements of an array variable

```
STRING_LIST a
```

are represented as

```
a[0], a[1], ... , a[n-1]
```

where

```
n = size(a)
```

is the size of the array. Internally, array data types are transmitted as the sequence

```
INT     n
STRING  a[0]
STRING  a[1]
...
STRING  a[n-1]
```

for string arrays, and correspondingly for other types of array.

CQi FUNCTIONS

The CQi functions are organised in **groups**. All function names in a group share a common prefix. In the following, all groups of functions are listed. The asterisk (`*`) is a placeholder for the ''individual'' part of a function name.

`CQI_CTRL_*`

General and administrative functions, mainly used for logging in to and out of the server.

`CQI_ASK_FEATURE_*`

The functions in this group are used to find out which parts and versions of the CQi protocol are supported by a server.

`CQI_CORPUS_*`

Corpus manager functions. Used to list available corpora and their attributes (e.g. `CQI_CORPUS_LIST_CORPORA()`), and to obtain additional information on the corpora (e.g. `CQI_CORPUS_FULL_NAME()`).

`CQI_CL_*`

The functions in this group provide low-level access to corpora (i.e. direct access to the *token sequence*, *lexicon*, and *index* of attributes). Most of these functions correspond to the **Corpus Library** functions and have similar names. A number of functions from this group (including `CQI_CL_CPOS2STR()` and `CQI_CL_REGEX2ID()`) will be used in the example program below.

`CQI_CQP_*`

This group contains all query processor functions. The most important function is `CQI_CQP_QUERY()`, which executes a CQP query. The group also includes all subcorpus operations (e.g. `CQI_CQP_DUMP_SUBCORPUS()`).

PSEUDO CODE

The pseudo code in this tutorial uses control structures similar to those found in C, Perl, and Java. All variables used in the code examples are declared as one of the CQi data types.

```
INT_LIST a
STRING full_name, corpus, subcorpus
```

Special functions or operators such as

```
size(a)
full_name = concatenate(corpus, ":", subcorpus)
```

should be self-explanatory. Comments begin with a double slash (`//`). Constant string values are enclosed in double quotes

```
STRING a = "simple"
```

whereas arrays are delimited by square brackets when given literally:

```
STRING_LIST b = [a, "simpler", "simplest"]
```

Each CQi function takes a certain number of arguments (shown in parentheses following the function name) and returns a single value (of the data type specified in the CQi documentation). Thus, a typical CQi function call is written as

```
STRING corpus_name
STRING_LIST subcorpora
subcorpora = CQI_CQP_LIST_SUBCORPORA(corpus_name)
```

in pseudo code. All CQi functions may return an error code instead of their return value. It is assumed here that some low-level component in the client library recognises error codes returned by the server and raises an exception, so that no explicit error checking is needed in the examples.

Some CQi functions do not have a return value. They merely return a status or error code. Such function calls appear in void context in the pseudo code examples:

```
CQI_CQP_QUERY(mother, subcorpus, query)
```

It is assumed that the program will abort if the query execution fails (e.g. if there is a syntax error in the query). Real-world applications should identifiy the status or error code returned by such functions, and distinguish between mere syntax or execution errors and other (non-recoverable) errors.

Finally, some functions return integer lists of fixed size (pairs or quadruples). Such return values are assigned to tuples of variables using the special syntax shown below.

```
// start & end positions of 3457th sentence in the BNC corpus
(from, to) = CQI_CL_STRUC2CPOS("BNC.s", 3456)
// boundaries of the 114th alignment block in the hansards
(f1, t1, f2, t2) = CQI_CL_ALG2CPOS("HANSARD-EN.hansard-fr", 113)
```

Most actual programming languages will require more complicated code to handle tuples as return values.

AN EXAMPLE PROPLEM

This section contains a step-by-step guide showing how to

- establish a CQi connection
- execute a CQP query
- access the resulting subcorpus
- display matches with part-of-speech annotations
- look up word forms in the corpus index
- get low-level access to the positional, structural, and alignment attributes of a corpus

Our pseudo-code example program will run the following CQP query on the `BNC` corpus and display the first ten matches with part-of-speech annotations.

```
(Q)  'interested' [pos='PRP'] [pos='AT0']? [pos='AJ0']* @[pos='NN.']
```

Then, a list of nouns appearing in the *target* field (marked `@`) will be produced. Finally, our program will look up words matching the regular expression

```
(R)  (over|under)estimate.*
```

in the `HANSARD-EN` corpus (the English part of the Canadian parliamentary debates), and display the sentences containing these words together with their French translations (from the aligned `HANSARD-FR` corpus).

Step 1: Establishing a CQi connection

A CQi session is initiated with the `CQI_CTRL_CONNECT()` command, which expects specifying *user name* and *password* as arguments.

```
// 'demo' user; password is _not_ encrypted
CQI_CTRL_CONNECT("demo", "secret")
```

A password is always required. Server administrators wishing to grant anonymous access should create an 'anonymous' user with the empty string as password.

When the connection has been established, the client should check whether the CQi server supports all required features.

```
BOOL ok_cqi, ok_cl, ok_cqp
// version numbers of the first cqpserver release:
// CQi specification v1.0
ok_cqi = CQI_ASK_FEATURE_CQI_1_0()
// corpus access functions CL v2.3
ok_cl = CQI_ASK_FEATURE_CL_2_3()
// CQP v2.3 corpus queries
ok_cqp = CQI_ASK_FEATURE_CQP_2_3()
```

Future releases of the CQi specification may define additional features and version numbers.

Step 2: Listing available corpora and attributes

A CQi client will usually want to present a list of available corpora and, for each corpus, a list of its attributes to the user. Our example program uses hard-coded corpus and attribute names. A more advanced version of this program should check whether the hard-coded corpus and attributes actually exist in order to avoid raising an exception.

```
STRING_LIST corpora
corpora = CQI_CORPUS_LIST_CORPORA()
// check here if BNC is in the list of available corpora
```

After selecting a corpus, a listing of its (positional and structural) attributes should be obtained.

```
STRING_LIST p_att, s_att
p_att = CQI_CORPUS_POSITIONAL_ATTRIBUTES("BNC")
s_att = CQI_CORPUS_STRUCTURAL_ATTRIBUTES("BNC")
// check here if word, lemma, pos, and s attributes are defined
```

We will be using the standard `word` (word form), `pos` (part-of-speech class), and `lemma` attributes, plus the structural attribute `s` (sentence boundaries) here. Every corpus *must* have a `word` attribute, but one or more of the other ''standard'' attributes may be absent, so client applications should always check the attribute lists before proceeding.

Finally, `CQI_CORPUS_FULL_NAME()`, `CQI_CORPUS_INFO()`, etc. provide some more information on a corpus (see CQi documentation for details). The *size* of a corpus, i.e. the number of tokens in the text, can be obtained using the `word` attribute:

```
INT size
size = CQI_CL_ATTRIBUTE_SIZE("BNC.word")
```

Step3: Executing a CQP query and accessing the result

The `CQI_CQP_QUERY()` command expects three arguments: a *corpus* or subcorpus on which the query is executed; a *subcorpus name* in which the query result will be stored; and a CQP *query string* (see the CQP manual for syntax). The following CQi pseudo code executes query (Q) on the BNC corpus.

```
STRING query
query = "'interested' [pos='PRP'] [pos='AT0']? [pos='AJ0']* @[pos='NN.']"
CQI_CQP_QUERY("BNC", "Results", query)
```

If the query was executed sucessfully, the results are now stored in the `BNC:Results` subcorpus. The command

```
STRING_LIST bnc_sub
bnc_sub = CQI_CQP_LIST_SUBCORPORA("BNC")
```

returns the subcorpus names `Last` and `Results`.

A CQi **subcorpus** can be interpreted as a table where each row corresponds to one query match. The total number of matches is

```
INT nr_matches = CQI_CQP_SUBCORPUS_SIZE("BNC:Results")
printf "%d matches.\n", nr_matches
```

In our example, *nr_matches* is 2324. The `BNC:Results` subcorpus contains 3 columns, which are called **fields**:

```
match field      position of first token in the match
matchend field   position of last token in the match
target field     position of token matching the marked (@) pattern
```

The *match* and *matchend* fields are always present. Applications that do not use hard-coded queries need to check whether any other fields are defined.

```
BOOL $match_ok, $matchend_ok, $target_ok
// these two are always True
$match_ok =
   CQI_CQP_SUBCORPUS_HAS_FIELD("BNC:Results", CQI_CONST_FIELD_MATCH)
$matchend_ok =
   CQI_CQP_SUBCORPUS_HAS_FIELD("BNC:Results", CQI_CONST_FIELD_MATCHEND)
// $target_ok is True if the the query contained an @-marked pattern
$target_ok =
   CQI_CQP_SUBCORPUS_HAS_FIELD("BNC:Results", CQI_CONST_FIELD_TARGET)
```

We will just display the first 10 matches in the subcorpus, i.e. we need to retrieve the first 10 entries from each field (column). Matches are numbered beginning with 0, so we request entries 0 through 9.

```
INT_LIST match, match_end, target
match = CQI_CQP_DUMP_SUBCORPUS(
           "BNC:Results",
           CQI_CONST_FIELD_MATCH,
           0, 9)
matchend = CQI_CQP_DUMP_SUBCORPUS(
           "BNC:Results",
           CQI_CONST_FIELD_MATCHEND,
           0, 9)
target = CQI_CQP_DUMP_SUBCORPUS(
           "BNC:Results",
           CQI_CONST_FIELD_TARGET,
           0, 9)
```

In our example, the table printed by the following code

```
INT i
printf "match\tm-end\ttarget\n"
printf "-----\t-----\t------\n"
for (i=0; i<5; i++) {
  printf "%d\t%d\t%d\n", match[i], matchend[i], target[i]
}
```

will look like this:

```
match   m-end   target
-----   -----   ------
45837   45839   45839
45885   45888   45888
54987   54989   54989
68633   68636   68636
73562   73565   73565
```

Each number in this table is a **corpus position**, i.e. the sequential number of the corresponding token in the BNC corpus. In the next step, we will use low-level corpus access functions to retrieve the query matches as plain text.

Step 4: Displaying the matches

Each match of the query is a sequence of consecutive corpus positions. For instance, the first match in the table above consists of the tokens numbered 45837, 45838, and 45839. We will use the CQI_CL_CPOS2STR() function to obtain the tokens at the given corpus positions. Since this function takes a list of corpus position as its second argument, we can access all tokens in a match with a single function call. The entire code needed to print the first 10 matches is shown below.

```
INT i
STRING_LIST tokens, pos_tokens
for (i=0; i<10; i++) {
  printf "%2d. ", (i+1)
  // [x .. y] creates a list of integers ranging from x to y
  tokens = CQI_CL_CPOS2STR("BNC.word", [match[i] .. matchend[i]])
  // now get part-of-speech tags (BNC.pos attribute)
  pos_tokens = CQI_CL_CPOS2STR("BNC.pos", [match[i] .. matchend[i]])
  // print tokens with part-of-speech annotations
  for (j=0; j<size(tokens); j++) {
    printf "%s/%s ", tokens[j], pos_tokens[j]
  }
  printf "\n"
}
```

The output printed by this code example will look like this:

```
 1. interested/AJ0 in/PRP form/NN1
 2. interested/AJ0 in/PRP the/AT0 context/NN1
 3. interested/AJ0 in/PRP art/NN1
 4. interested/AJ0 in/PRP a/AT0 picture/NN1
 5. interested/AJ0 in/PRP the/AT0 market/NN1
 6. interested/VVN in/PRP copies/NN2
 7. interested/AJ0 in/PRP the/AT0 subject/NN1
 8. interested/AJ0 in/PRP action/NN1
 9. interested/AJ0 in/PRP film/NN1
10. interested/AJ0 in/PRP film/NN1
```

Finally, we want to get the full list of lemma annotations of the marked tokens (i.e. the entries in the *target* field), which can be used to compute the frequency distribution of nouns appearing in the target position of

query (Q).

```
INT i
INT_LIST target
STRING token
STRING_LIST temp

// use −1 .. −1 to dump entire subcorpus
target = CQI_CQP_DUMP_SUBCORPUS(
           "BNC:Results",
           CQI_CONST_FIELD_TARGET,
           −1, −1)
for (i=0; i<size(target); i++) {
  // −1 in target field means target was not set in this match
  // (although this cannot happen in our case)
  if (target[i] == −1) {
    token = "<undef>"
  }
  else {
    // CQI_CL_CPOS2STR() operates on lists, so we must pass a
    // a single corpus position as a one−element list. Note that
    // the return value is a list as well.
    temp = CQI_CL_CPOS2STR("BNC.lemma", [target[i]])
    token = temp[0]
  }
  // now insert <token> into frequency list or print
}
```

Step 5: Low-level corpus access

In the last part of our tutorial, we will show how to get direct access to the *index* and *lexicon* of positional attributes and how to compute sentence boundaries and alignment blocks.

All values of a positional attribute are stored in its *lexicon* with unique numeric *ID*s (but *not* in alphabetical order). We can look up the ID(s) of one or more words with the `CQI_CL_STR2ID()` function.

```
INT_LIST id
id = CQI_CL_STR2ID("HANSARD−EN.word", ["interesting", "fripping"])
```

In this example, `id[1]` is `−1`, which means the word form *fripping* was not found in the `HANSARD−EN` corpus. Lexicon entries matching a given regular expression can be obtained with the `CQI_CL_REGEX2ID()` function. Note that unlike in the previous example, the second argument is a single regular expression rather than a list, whereas the return value is still an integer list (because the regular expression will usually match several lexicon entries).

```
INT_LIST id
STR_LIST word_form
INT i

// <id> holds the IDs of word forms matching regular expression (R)
id = CQI_CL_REGEX2ID("HANSARD−EN.word", "(under|over)estimate.*")
// retrieve the corresponding word forms
word_form = CQI_CL_ID2STR("HANSARD−EN.word", id)

printf "%d word forms match /(under|over)estimate.*/:\n", size(word_form)
for (i=0; i<size(word_form); i++) {
  printf " −− %s\n", word_form[i]
}
```

The output printed by the example above should look like this:

```
6 word forms match /(under|over)estimate.*/:
 -- underestimate
 -- underestimated
 -- underestimates
 -- overestimated
 -- overestimate
 -- overestimates
```

Using the list of IDs, we can now look up these word forms in the index of the `HANSARD-EN.word` attribute.

```
INT_LIST cpos
STRING_LIST temp

// <cpos> is a sorted list of corpus positions -- for a single word
// form the CQI_CL_ID2CPOS() function is faster
cpos = CQI_CL_IDLIST2CPOS("HANSARD-EN.word", id);

// print first ten tokens in HANSARD-EN matching (R)
for (i=0; i<10; i++) {
  // temp[0] is the token at corpus position cpos[i]
  temp = CQI_CL_CPOS2STR("HANSARD-EN.word", cpos[i])
  printf "%7d:  %s\n", cpos[i], temp[0]
}
```

This code produces the following (or similar) output:

```
  27627:  underestimate
 608702:  underestimated
1589347:  underestimated
1606570:  underestimates
1787174:  overestimated
2013493:  underestimated
2420116:  underestimate
2495773:  underestimated
3295899:  overestimated
3459443:  underestimate
3795709:  underestimate
```

The next piece of sample code uses structural attributes to display the first ten sentences containing a token that matches the regular expression (R). A **structural attribute** is a sequence of non-overlapping, but not necessarily adjacent regions in a corpus (similar to SGML regions). In our example, we assume that sentences in the `HANSARD-EN` corpus are encoded in the structural attribute `HANSARD-EN.s`. The regions of a structural attribute are numbered beginning with 0. The number of sentences in the `HANSARD-EN` corpus is

```
INT n
n = CQI_CL_ATTRIBUTE_SIZE("HANSARD-EN.s")
```

For each token in the `cpos` array, the code below first computes the number of the sentence containing that token, and then it obtains the start and end position of that sentence in the token sequence.

```
INT start, end, i, j
INT_LIST sentence
STRING_LIST tokens

sentence = CQI_CL_CPOS2STRUC("HANSARD-EN.s", cpos);
for (i=0; i<size(sentence); i++) {
```

```
  // -1 means that the token at position cpos[i] is not contained
  // in an <s>..</s> region -> skip this match
  if (sentence[i] == -1) {
    printf "%7d:  <no sentence found>\n", cpos[i]
  }
  else {
    // <start> is the number of the first token in the sentence
    // <end> is the number of the last token in the sentence
    (start, end) = CQI_CL_STRUC2CPOS("HANSARD-EN.s", sentence[i])
    // note that the 2nd argument is _not_ a list!

    // get tokens from <start> to <end> and print the sentence
    tokens = CQI_CL_CPOS2STR("HANSARD-EN.word", [start .. end])
    printf "%7d:  <s> "
    for (j=0; j<size(tokens); j++) {
      printf "%s ", tokens[j]
    }
    printf "</s>\n"
  }
}
```

Finally, we will determine regions in the HANSARD-FR corpus that are aligned to these sentences. The name of the **alignment attribute** corresponds to the name of the aligned corpus, hence we have to access the HANSARD-EN.hansard-fr attribute. Like structural regions, alignment blocks are numbered beginning with 0. All output code is omitted from the following example.

```
INT s1, s2, t1, t2, i
INT_LIST alignment

alignment = CQI_CL_CPOS2ALG("HANSARD-EN.hansard-fr", cpos);
for (i=0; i<size(alignment); i++) {
  if (alignment[i] == -1) {
    // -1 means that no alignment block was found for this token
  }
  else {
    (s1, s2, t1, t2) = CQI_CL_ALG2CPOS("HANSARD-EN.hansard-fr", alignment[i])
    // meaning that the region [s1 .. s2] in HANSARD-EN is aligned
    // to the region [t1 .. t2] in HANSARD-FR; [s1 .. s2] contains
    // the token at corpus position cpos[i], i.e.  s1 <= cpos[i] <= s2
  }
}
```

COPYRIGHT