ToolboxSearch - an R package for working with Toolbox corpora
User Manual

Taras Zakharko

November 3, 2011

Contents

1 About ToolboxSearch

`ToolboxSearch` is a new package aimed at linguists who work with language corpora in Toolbox file format. The package contains utilities for loading and manipulating Shuebox/Toolbox corpora within R as well as a powerful corpus search toolkit which is based on a flexible declarative query language. Search patterns can be specified in a simple, readable and reusable way, e.g. the following R code:

```
corpus %%
"@record
{
  CONTAINS @word
  {
    CONTAINS
    [
      @morpheme{$mgl =~ 'DEM'}
      @morpheme{$mgl =~ 'LOC'}
    ]
  }
}"
```

will find all records in the corpus which contain at least one word that contains a `DEM`-glossed morpheme followed by a `LOC`-glossed morpheme.

Some key features of the package:

simple user interface

ToolboxSearch provides a number of new R commands for manipulating corpora. These

commands are designed to be intuitive and easy-to-learn, so that any user, independently of their R experience level, could start using the package right away.

great search facilities
: ToolboxSearch includes an intuitive and powerful corpus query language. Its simple and comprehensive syntax allows the user to craft complex search patterns quickly, without a steep learning curve. The results of the query can be stored to a Toolbox file for further processing or transformed into an R data frame for the purposes of statistic analysis.

"smart" import of Toolbox files
: Toolbox/Shuebox stores corpora as sequences of interlinear glosses using a text-based format. This format appears to be very simple, but in reality, Toolbox files often suffer from inconsistencies and coding errors. The import routines of TolboxSearch attempt to circumvent this by using a number of different algorithms and setting simultaneously. Because of this, ToolboxSearch is able to parse Toolbox files where other tools (e.g. ELAN) would produce erroneous results. The routines maintains a detailed error log for all records in a Toolbox file which could not be parsed successfully. These logs can be then used to detect and "repair" errors within the corpus.

export of Toolbox files
: ToolboxSearch is able to save the results of an R session back to a Toolbox-formated file. The written file is 100% correctly formatted Toolbox and can be imported by ELAN and other tools without the fear of data corruption.

performance
: The performance-critical parts of the package (i.e. much of the file import and search facility) is written in the C programming language. This makes ToolboxSearch very fast for most operations. For instance, it takes less than a second to perform a complex search query on a combined corpus with over 90000 records.

The last version of the package can be always found at `https://bitbucket.org/tzakharko/toolboxsearch`. Bug reports, suggestions and criticisms are always welcome!

2 This document

This document is a manual/tutorial which will guide you though all the important features provided by ToolboxSearch. First, we will begin with a short review of the Toolbox file format — this is important for understanding of some common problems you may encounter when importing your corpora with ToolboxSearch. After that, the manual will explain how corpora can be loaded into your R session and subsequently viewed, partitioned and manipulated. The next large section will introduce and explain the corpus query language used by ToolboxSearch. The final section of the

manual will describe some hypothetical usage scenarios which illustrate how the tools within the package can be used in a workflow of a linguist.

The manual assumes that the reader is already acquainted with the R shell and its basic concepts. The reader is encouraged to consult one of the general books/tutorials on R.

3 Notes on the anatomy of a Toolbox file

Toolbox is a popular software tool for transcription and interlinear glossing of language corpora. A Toolbox corpus is a sequence of *records*, which usually correspond to sentences or clauses. Within each record, Toolbox stores a number of parallel *annotation tiers*, such as morpheme glosses or speaker name. The records are stored in a text-based file format, as illustrated below (Example from the Chintang and Puma Documentation Project, 2011):

```
\ref CLLDCh2R06S02. 0001
\ELANBegin 00:00:00.824
\ELANEnd 00:00:06.198
\EUDICOp XYZ
\tx ne coha
\gw ne           coha
\mph ne           ci  -u     -hã
\mgl EXCLA.interj eat -3P.gm -PRSV.IMP
\lg C            C   -C     -C
\eng Take it and eat.
\dt 19/Mar/2010
```

The above example shows an excerpt from a Toolbox file which represents a single record. Each line stores a value associated with an annotation tier `\***`. The first tier
ref is the record marker, which signals the start of the new record.

Annotation tiers can be arranged into different *annotation levels*. In the above example, there are three such levels — record level, word level and the morpheme level. The record level includes annotations which concern the whole record, such as the timestamp (`\ELANBegin` and `\ELANEnd`), the speaker's name (`\EUDICOp`), the transcribed text and its translation (`\tx`, `\eng`) and the date of last edit (`\dt`). The word level includes the annotation of the running words — here it is only the word form `\gw`. Finally, the morpheme level includes the annotations of the individual morphemes: the transcription `\mph`, the gloss `\mgl` and the source language of the lexeme (e.g. for code switching studies) `\lg`.

Contrary to a popular belief, Toolbox format correctly stores the vertical alignment between the elements (i.e. the fact that morphemes `ci`, `-u`, `-hã` belong to the word `coha`). The vertical alignment is given by number of spaces which separate the annotations on different tier. To illustrate this fact, consider the alignment of the tiers `\gw`, `\mph` and `\mgl` from the above example (with tier markers stripped and spaces visualized):

`ne␣␣␣␣␣␣␣␣␣␣coha`

```
ne␣␣␣␣␣␣␣␣␣␣ci␣␣-u␣␣␣␣␣-hã
EXCLA.interj␣eat␣-3P.gm␣-PRSV.IMP
```

The gloss annotations contain the longest text elements and Toolbox automatically adds spaces to other tiers so that corresponding elements (e.g. the word `coha` and its first morpheme `ci`) occupy exactly the same character position in the corresponding line. Unfortunately, the reality is a bit more complicated, as illustrated by the example below:

```
baŋge␣␣␣␣wanda ␣␣␣␣␣␣khai ma␣␣␣␣␣␣din
baŋge␣␣␣␣wanda␣␣␣␣␣␣␣␣khat␣␣-ma␣␣␣␣␣din
a.place.n␣tomorrow.adv␣go.vi␣-INF.gm␣day.n
```

The elements appear to be shifted, e.g. the stem `khat`(go) is not properly aligned with its gloss or the word it appears in. In fact, the items are still aligned correctly. This seeming paradox is the consequence of how Toolbox works with character encoding. The UTF-8 encoding (internally used by Toolbox) represents some characters (like `'n'`) with one byte of information, but other characters (like `'ŋ'`, `' '`) are stored as two bytes. Because Toolbox counts bytes internally, it computes the length of the word `baŋge` as 6, even when it has 5 characters only! We can easily see that counting multibyte character twice results in a correct alignment:

```
baŊŊge␣␣␣␣wanda ␣␣␣␣␣␣khai ma␣␣␣␣␣␣din
baŊŊge␣␣␣␣wanda␣␣␣␣␣␣␣␣khat␣␣-ma␣␣␣␣␣din
a.place.n␣tomorrow.adv␣go.vi␣-INF.gm␣day.n
```

This is still not the end of the story, because Toolbox ignores some characters (like accents) completely when computing the length of the elements. This happens because these elements do not occupy horizontal space when displayed, but rather, are combined with the neighbor characters.

When parsing a Toolbox file, it is important to take all these details into account, which makes the import algorithm non trivial. To further complicate the matter, Toolbox files often diverge from the conventions described above. Sometimes Toolbox seems to count characters and not bytes, sometimes it also counts the accents (for reasons unknown to me). These rules may even differ from record to record within a single Toolbox file! Sometimes, the authors of the corpus will "correct" records by editing a toolbox file in a text editor (and destroying the carefully arranged spaces in the process); also, some intermediate tools used to process the Toolbox files may affect the spaces or even convert them to tabulator characters.

All in all, the Toolbox file format is very fragile and this fact does not help when one tries to parse it. ToolboxSearch currently uses two algorithms to deal with this problem. The first algorithm (*position tracking*) assumes that the spaces encoding the vertical alignment are largely correct and attempts to parse each record using different settings: e.g. on the first attempt it will count bytes, and if it does not result in a valid structure, it will try to count characters. Thus, the algorithm can correctly parse the records, no matter which settings were used by Toolbox to produce the gloss structure. This works rather well for the majority of corpora and is very straightforward for the user.

The second algorithm (*sequence tracking*) can be used if the proper alignment in the files was damaged beyond automatic repair (e.g. by manual edits). Nevertheless, it requires that morphemes within the file were coded for their position within the word (i.e. either as a root, prefix or a suffix). Most corpora do this by using the minus symbol `'-'` to code prefixes (e.g. `aaa-`) and suffixes (e.g. `-aaa`) and the equality symbol to code clitics (e.g. `=aaa`). Root morphemes usually don't have any special symbols attached to them. The second algorithm works by tracing the morphemes and trying to assign them to the words. The basic idea: the algorithm knows that a single word can consist of zero or more prefixes, followed by a root and followed by zero or more suffixes. Thus, it does not have to rely on the number of spaces to detect the vertical alignment.

Both above algorithms can fail to parse a damaged record. In this case, the record is skipped, and a detailed error message is generated. The rest of the file is still loaded. The user may choose to inspect the error log afterwards and redo the damaged records. This way, ToolboxSearch may be used as a validation tool for Toolbox corpora - which is important when you are using other tools (like ELAN) to work with your corpora.

4 Loading, viewing and partitioning the corpora

4.1 Initialization

Assuming you have already installed ToolboxSearch on your computer (either using the command line or the R GUI menu option), the package can be loaded as follows:

```
library(ToolboxSearch)
```

```
ToolboxSearch 0.5 loaded.
```

You must load the package before you can use its features!

4.2 Toolbox format descriptor

In order to be able to load a Toolbox file, the system must first learn some information about the annotation tiers and their structure within that file. For this purpose, ToolboxSearch uses a construct called the *Toolbox file descriptor*. The descriptor simply lists the names of the tiers we want to load from the file and their corresponding level on the vertical alignment hierarchy. Recall the example from the previous section:

```
\ref CLLDCh2R06S02. 0001
\ELANBegin 00:00:00.824
\ELANEnd 00:00:06.198
\EUDICOp XYZ
\tx ne coha
\gw ne           coha
\mph ne           ci  -u     -hã
```

```
\mgl EXCLA.interj eat -3P.gm -PRSV.IMP
\lg C            C   -C     -C
\eng Take it and eat.
\dt 19/Mar/2010
```

A suitable descriptor for this file can be declared as:

```
fmt ← toolboxFormat(
        record=c(ref, EUDICOp, age),
        word=gw,
        morpheme=c(mph, mgl, lg)
)
fmt
```

```
Toolbox format descriptor with 3 levels
  record marker \ref
  @record: \ref \EUDICOp \age
  @word: \gw
  @morpheme: \mph \mgl \lg
```

The construction of the format descriptor is very similar to construction of R lists (only that you have to ue toolboxFormat() instead of list ()). Each argument defines one structural level as a `name=values` pair, where `name` is the name of the level and `values` are the names of annotation tiers which belong to that level. In this example, the format descriptor defines three levels (`record`, `word` and `morpheme`). The names of the levels are arbitrary choices by the user. The levels constitute a hierarchical structure, such that one `record` can contain one or more `words` and one `word` can contain one or more `morphemes`. The `record` level includes the tiers `\ref`, `\EUDICOp` and `\age`; the `word` level includes only the tier `\gw` etc. The first tier of the top-most level (here — `\ref`) is automatically treated as the record marker.

4.3 Loading the Toolbox file

Now that we have defined the format descriptor, loading the actual corpus is a trivial task (the corpus file must be in your working directory):

```
corpus ← readToolbox("CLLDCh1R01S01.txt", fmt)
corpus
```

```
Corpus with 183 entries (record) showing 1-3:

----------@1
\ref CLLDCh1R01S01.002
\EUDICOp RM
\age 27;3.1
\gw chito      ottopida
\mph chito      ott              -u      -bid     -a
\mgl quick.adv break.for.sb.vt -3P.gm -BEN.v2 -IMP.gm
\lg N          C                -C      -C       -C
```

```
------------@2
\ref CLLDCh1R01S01.002a
\EUDICOp RM
\age 27;3.1
\gw chito      chito      cohakha                          na
\mph chito      chito      ca      -u      -a      -kha      na
\mgl quick.adv quick.adv eat.vt -3P.gm -IMP.gm -BGR.gm TOP.gm
\lg N          N          C       -C      -C      -C        C

------------@3
\ref CLLDCh1R01S01.005
\EUDICOp RM
\age 27;3.1
\gw hokhi luno              kancha
\mph hokhi lus      -no      kancha
\mgl how    feel.vi -NPST.gm youngest.one.male
\lg C      C        -C       N
```

As already mentioned, the import algorithms of ToolboxSearch can be tweaked to match the particular format and condition of the particular corpus. For this, readToolbox() offers a number of optional parameters.

Per default, the import function will attempt to parse the vertical alignment by checking the number of spaces which separate the annotations (as explained in the section about the Toolbox file format). Here, the algorithm will try different parsing options for each record in the file in the hope that one set of options will lead to success.

As an alternative, you may want to use the second algorithm, which tracks morpheme positions (i.e. words as sequence of prefixes, root and suffixes) to parse the gloss structure. To do this, you have to tell the function to parse the corresponding level in the *sequence mode*:

```
corpus ← readToolbox("CLLDCh1R01S01.txt", fmt, morpheme="sequence")
corpus
```

```
Corpus with 183 entries (record) showing 1-3:

------------@1
\ref CLLDCh1R01S01.002
\EUDICOp RM
\age 27;3.1
\gw chito      ottopida
\mph chito      ott             -u      -bid     -a
\mgl quick.adv break.for.sb.vt -3P.gm -BEN.v2 -IMP.gm
\lg N          C                -C      -C       -C

------------@2
\ref CLLDCh1R01S01.002a
\EUDICOp RM
\age 27;3.1
```

```
\gw chito      chito      cohakha                          na
\mph chito      chito      ca      -u      -a      -kha      na
\mgl quick.adv quick.adv eat.vt -3P.gm -IMP.gm -BGR.gm TOP.gm
\lg N          N          C       -C      -C      -C        C

----------@3
\ref CLLDCh1R01S01.005
\EUDICOp RM
\age 27;3.1
\gw hokhi luno              kancha
\mph hokhi lus      -no      kancha
\mgl how    feel.vi -NPST.gm youngest.one.male
\lg C      C        -C       N
```

When in the sequence mode, the algorithm assumes that morpheme position is coded via the symbols `'-'` and `'='` (e.g. prefixes like `'aaa-'` and suffixes like `'-aaa'`). If your corpus uses some sort of exotic coding, you can tell it to the import algorithm via the conn parameter, e.g.:

```
corpus ← readToolbox("myfile.txt", fmt,
           morpheme=list(mode="sequence", conn=c("&")))
```

The import algorithm generates a status report for each record it encounters within the file. If a record could not be parsed, an appropriate message will appear in the report. The report can be accessed via:

```
record.log(corpus)
```

The reader is encouraged to experiment with this functionality.

4.4 Viewing and partitioning the corpus

Now that we have loaded a Toolbox file, we have stored it in the variable with the name corpus. We can simply print the name of this variable into the R shell and it will display the first three records of the data (similarly as to how R displays values of other variables):

```
corpus
```

```
Corpus with 183 entries (record) showing 1-3:

----------@1
\ref CLLDCh1R01S01.002
\EUDICOp RM
\age 27;3.1
\gw chito      ottopida
\mph chito      ott              -u      -bid     -a
\mgl quick.adv break.for.sb.vt -3P.gm -BEN.v2 -IMP.gm
\lg N          C                -C      -C       -C

----------@2
```

```
\ref CLLDCh1R01S01.002a
\EUDICOp RM
\age 27;3.1
\gw chito      chito      cohakha                          na
\mph chito      chito      ca      -u      -a       -kha     na
\mgl quick.adv quick.adv eat.vt -3P.gm -IMP.gm -BGR.gm TOP.gm
\lg N          N          C       -C      -C       -C       C

----------@3
\ref CLLDCh1R01S01.005
\EUDICOp RM
\age 27;3.1
\gw hokhi luno              kancha
\mph hokhi lus      -no      kancha
\mgl how    feel.vi -NPST.gm youngest.one.male
\lg C      C        -C       N
```

We can also ask R to show us a particular set of records by using the `print()` function (the notation `a:b` in R means a sequence of numbers from a to b):

```
print(corpus, 2)
```

```
Corpus with 183 entries (record) showing 2:

----------@2
\ref CLLDCh1R01S01.002a
\EUDICOp RM
\age 27;3.1
\gw chito      chito      cohakha                          na
\mph chito      chito      ca      -u      -a       -kha     na
\mgl quick.adv quick.adv eat.vt -3P.gm -IMP.gm -BGR.gm TOP.gm
\lg N          N          C       -C      -C       -C       C
```

```
print(corpus, 6:7)
```

```
Corpus with 183 entries (record) showing 6-7:

----------@6
\ref CLLDCh1R01S01.008
\EUDICOp RM
\age 27;3.1
\gw sa       napide
\mph sa       na-      pit      -e
\mgl who.pro 3>2.gm- give.vt -PST.gm
\lg C        C-       C        -C

----------@7
\ref CLLDCh1R01S01.009
\EUDICOp RM
\age 27;3.1
```

```
\gw bago ŋ                  saa                napide
\mph ba            -ko      sa ŋ      -a       na-      pit      -e
\mgl DEM.PROX.pro -GEN.gm who.pro -ERG.A.gm 3>2.gm- give.vt -PST.gm
\lg C             -C       C        -C        C-       C        -C
```

Thus, simply entering `corpus` is equivalent to `print(corpus, 1:3)`.

It is often useful to know how many records, words, morphemes etc. are described in a corpus. ToolboxSearch provides a special function for this purpose (second parameter is the level of the annotation to count):

```
length.corpus(corpus, "record")
```

```
[1] 183
```

```
length.corpus(corpus, "word")
```

```
[1] 434
```

```
length.corpus(corpus, "morpheme")
```

```
[1] 727
```

You can also extract parts of your corpus. By using the notation `cvar[index]`, where `cvar` is the variable which stores the corpus and `index` is the index of records (similar to what is used by `print()`), you tell R to copy the appropriate chunk of your corpus. This copy is independent of the original corpus data — you can modify it without the original data being affected. You can also store it in another variable for further processing. The example below copies the first 10 records from the original corpus and stores it another variable named `corpus.part`

```
corpus.part ← corpus[1:10]
corpus.part
```

```
Corpus with 10 entries (record) showing 1-3:

----------@1
\ref CLLDCh1R01S01.002
\EUDICOp RM
\age 27;3.1
\gw chito      ottopida
\mph chito      ott              -u      -bid     -a
\mgl quick.adv break.for.sb.vt -3P.gm -BEN.v2 -IMP.gm
\lg N          C                -C      -C       -C

----------@2
\ref CLLDCh1R01S01.002a
\EUDICOp RM
\age 27;3.1
\gw chito      chito      cohakha                        na
```

```
\mph chito      chito      ca      -u      -a      -kha      na
\mgl quick.adv quick.adv eat.vt -3P.gm -IMP.gm -BGR.gm TOP.gm
\lg N          N          C       -C      -C       -C       C

----------@3
\ref CLLDCh1R01S01.005
\EUDICOp RM
\age 27;3.1
\gw hokhi luno              kancha
\mph hokhi lus      -no      kancha
\mgl how    feel.vi -NPST.gm youngest.one.male
\lg C      C        -C       N
```

Operations like these are often called *slicing* in the programming jargon. Slicing is not restricted to the record level with ToolboxSearch, in fact, you can access any level:

```
corpus[1:3, "word"]
```

```
Corpus with 3 entries (word) showing 1-3:

----------@1
\gw chito
\mph chito
\mgl quick.adv
\lg N

----------@2
\gw ottopida
\mph ott              -u      -bid     -a
\mgl break.for.sb.vt -3P.gm -BEN.v2 -IMP.gm
\lg C                -C      -C       -C

----------@3
\gw chito
\mph chito
\mgl quick.adv
\lg N
```

```
corpus[1:3, "morpheme"]
```

```
Corpus with 3 entries (morpheme) showing 1-3:

----------@1
\mph chito
\mgl quick.adv
\lg N

----------@2
\mph ott
\mgl break.for.sb.vt
```

```
\lg C

————————@3
\mph –u
\mgl –3P.gm
\lg –C
```

The slicing functionality can be also used to split the corpus into words or morphemes (e.g. if if you are interested in compiling the lists of words):

```
corpus[level="morpheme"]
```

```
Corpus with 727 entries (morpheme) showing 1–3:

————————@1
\mph chito
\mgl quick.adv
\lg N

————————@2
\mph ott
\mgl break.for.sb.vt
\lg C

————————@3
\mph –u
\mgl –3P.gm
\lg –C
```

This is equivalent to:

```
corpus[1:length.corpus(corpus, "morpheme"), "morpheme"]
```

Another way to do slicing is to use the special data values provided by ToolboxSearch, the *corpus index objects*. These values represent the "coordinates" of a corpus slice, without doing the actual slicing. Consider:

```
index1 ← index.corpus(1:3, "word")
index1
```

```
Corpus subset@word: 1–3 (3 elements)
```

```
index2 ← index.corpus(c(2, 4), "morpheme")
index2
```

```
Corpus subset@morpheme: 2, 4 (2 elements)
```

Here, index1 represents a slice of first to third words of a corpus and index2 represents the slice of the second and fourth morphemes of a corpus. You can perform the actual slice operation by using the index object as the actual slice index:

```
corpus[index1]
```

```
Corpus with 3 entries (word) showing 1-3:

----------@1
\gw chito
\mph chito
\mgl quick.adv
\lg N

----------@2
\gw ottopida
\mph ott              -u      -bid     -a
\mgl break.for.sb.vt -3P.gm -BEN.v2 -IMP.gm
\lg C                -C      -C       -C

----------@3
\gw chito
\mph chito
\mgl quick.adv
\lg N
```

```
corpus[index2]
```

```
Corpus with 2 entries (morpheme) showing 1-2:

----------@1
\mph ott
\mgl break.for.sb.vt
\lg C

----------@2
\mph -bid
\mgl -BEN.v2
\lg -C
```

Hence, a command like

```
corpus[1:3, "word"]
```

is equivalent to

```
index1 <- index.corpus(1:3, "word")
corpus[index]
```

The nice thing about index objects is that can be combined using set operations. This is a very powerful feature when combined with the query language (as explained in next sections of the manual). You can perform a union, intersection or difference operations with the help of the usual arithmetic operations:

```
index1 ← index.corpus(1:3, "word")
index2 ← index.corpus(2:4, "word")
# union
index1 + index2
```

```
Corpus subset@word: 1–2, 2–3, 3–4 (6 elements)
```

```
# intersection
index1 * index2
```

```
Corpus subset@word: 2–3 (2 elements)
```

```
# difference
index1 – index2
```

```
Corpus subset@word: 1 (1 elements)
```

You can perform "negative" slicing (i.e. exclude the parts of the corpus indexed by the index object) as follows:

```
index.corpus(1:3, "word")
```

```
Corpus subset@word: 1–3 (3 elements)
```

```
corpus – index1
```

```
Corpus subset@word: 4–434 (431 elements)
```

```
corpus[corpus – index1]
```

```
Corpus with 431 entries (word) showing 1–3:

————————@1
\gw chito
\mph chito
\mgl quick.adv
\lg N

————————@2
\gw cohakha
\mph ca      –u      –a       –kha
\mgl eat.vt –3P.gm –IMP.gm –BGR.gm
\lg C       –C      –C       –C

————————@3
\gw na
\mph na
\mgl TOP.gm
\lg C
```

Here, the variable containing the corpus data is interpreted as a corpus index selecting all data in the corpus.

4.5 Loading multiple Toolbox files

Usually, a corpus consists of more than one file, sometimes hundreds. With ToolboxSearch and R, you can easily load ad combine such huge corpora. The only requirement is that all files in a corpus can be described by the same format descriptor. The following code can be used to load all Toolbox files from the folder `data`:

```
files ← dir(path="data", pattern="txt$", full.names=T, recursive=T)
corpora ← lapply(files[1:5], function(fn) readToolbox(fn, fmt, morpheme="sequence"))
length(corpora)
```

```
[1] 5
```

First, the R function dir() is used to list all the files in the folder `data` and its subfolders. In my case, this folder contains a substantial part of the Chintang corpus (5 files). The list of these files is stored in the variable called lapply(). Then, we use the R function lapply(), which takes each file name from the files variable and passes it to the readToolbox() function. In the end, the variable corpora contains a list of 5 items, each of which represent the corpus data loaded from a Toolbox file. Please note that loading multiple Toolbox files may take some time, currently about half a minute for 300 files on my machine.

At this point, each corpus is still a separate data structure. You might with to "collapse" it into a single large "supercorpus", so you can work with all of your data at once. This is easily done with ToolboxSearch:

```
corpus ← concat.corpus(corpora)
corpus
```

```
Corpus with 5299 entries (record) showing 1–3:

————————@1
\ref CLLDCh2R02S05.001
\EUDICOp LDCh2
\gw ~hui u         u
\mph ~hui u         u
\mgl DEM DEM.pro DEM.pro
\lg C    N         N

————————@2
\ref CLLDCh2R02S05.002
\EUDICOp PMR
\gw ~ha
\mph ~ha
\mgl what.interj
\lg C/N

————————@3
\ref CLLDCh2R02S05.003
\EUDICOp PMR
```

```
\gw khoi
\mph khoi
\mgl where.pro
\lg C/N
```

4.6 Saving Toolbox files

ToolboxSearch also offers the ability to store slices of the corpora to regular Toolbox files. You can use it, for example, to save some interesting examples you have compiled from the corpus using the search facility (see next section). Writing Toolbox files is very simple:

```
slice ← corpus[...] # compute a corpus slice in some way
writeToolbox("my_examples.txt", slice)
```

Another useful application of this function is "cleaning up" the corpora. The writeToolbox() function produces carefully formatted Toolbox files which can be correctly parsed by ELAN and other tools. You can load your files to R using the ToolboxSearch sequence parsing algorithm (readToolbox(... , morpheme="sequence")) and immediately save them with writeToolbox(). This will ensure that the spaces (and thus the vertical alignment) is correctly formatted in the resulting files.

4.7 Doing statistics with your corpora

In the preview release, the internal structure of the data is hidden away and is not directly accessible by R yet. However, it is possible to transform a corpus (or its part) to an R data frame structure, and use this structure for statistics. In such a data frame, each table row will correspond to a morpheme (or a lowest level defined by your format descriptor). This transformation is done simply using as.data.frame () function.

5 The corpus query language

5.1 The data model

The corpus query capabilities of TooolboxSearch operate with a fairly simple data model. The Toolbox corpus is divided into a number of *objects* which are associated with the levels defined in your format descriptor (e.g. records, words and morphemes). Each object can have one or more annotations (i.e. corresponding values of the annotation tiers). Also, an object may contain a sequence of other objects. This containment relation is strictly hierarchical (clauses contain words, words contain morphemes etc.), but it is also inherited between the objects of different levels. So, if a record contains a word and this word contains a morpheme, this morpheme is also contained in the record.

Consider the following excerpt from a Toolbox file:

```
\ref CLLDCh1R01S01.005
\EUDICOp AA
\gw hokhi luno              kancha
\mph hokhi lus     -no      kancha
\mgl how   feel.vi -NPST.gm youngest.one.male
\eng How does it taste, kancha?
```

This excerpt describes one object at the record level (with annotations `ref='CLLDCh1R01S01.005'`, `EUDICOp='AA'` and `eng='How does it taste, kancha?'`). This object contains three objects of the word level (associated with `gw` annotations `'hokhi'`, `'luno'` and `'kancha'`, respectively). Furthermore, the word objects contain (sequences of) morpheme objects. For instance, the second word object contains a sequence of two morphemes (`mph='lus'`, `mgl='feel.vi'` and `mph='-no'`, `mgl='-NPST.gm'`, respectively).

5.2 Query language - the basics

The query language allows the user to search for objects (i.e. records, words or morphemes) in the corpus which match a specific pattern. Such pattern can include conditions on the annotations (e.g. 'find all morphemes with a particular gloss') or containment relations in respect to other objects (e.g. 'find all words which contain a morpheme glossed as a case marker'). The following is a simple query which matches all records where the english translation (the annotation tier `\eng`) contains a substring 'plum':

```
@record{$eng =~ 'plum'}
```

This query illustrates some basic principles of the query language. An object pattern is declared in form `@level{ ... }`, where `level` is the level of the object (as declared in the format descriptor which was used to load the corpus) and `...` is a list of conditions which must apply in order for the object to match the pattern. The conditions are given within the scope of the object declaration, which is restricted by the curvy brackets.

`$eng =~ 'plum'` is one such condition, which matches the annotation `\eng` agains a substring 'plum'. The dollar sign is used to mark the annotation names and the operation '`=~`' means 'match a regular expression'. ToolboxSearch uses Perl-compatible regular expressions (see R help for ?regex).

Different conditions can be combined using logical operators `AND`, `OR` and `NOT`, e.g.:

```
@record{$eng =~ 'plum' AND $eng =~ 'house' AND NOT ($EUDICOp == 'HH')}
```

Here, we are looking for all records where the english translation contains substrings 'plum' and 'house' and the speaker is not `HH`. Another application would be to find all pronoun morphemes which begin with an 'a':

```
@morpheme{$mph =~ '^a' AND $mgl =~ 'PRO'}
```

In addition to conditions on annotations, ToolboxSearch can also do conditions on containment relation, e.g. find all records which contain a locative marker (a morpheme which is annotated as LOC):

```
@record{CONTAINS @morpheme{$mgl =~ 'LOC'}}
```

Furthermore, the containment relation can be extended into a sequence pattern, e.g. find all records which contain a sequence of pronoun + locative marker:

```
@record{CONTAINS [@morpheme{$mgl =~ 'PRO'} @morpheme{$mgl =~ 'LOC'}]}
```

5.3 Language in detail

This section describes the syntax and semantics of the query language in further detail. The syntax is described via a formal grammar (term expansion), the non-terminals are surrounded with the underscore.

A query in the query language is represented via a non-terminal _QUERY_. This is simply an object description:

```
_QUERY_ -> _OBJDESC_
```

An object description combines the object level (a literal which matches a level name declared by the format descriptor) and a set of conditions the object must comply to:

```
_OBJDESC_ -> level@{_CONDITIONS_}
```

Furthermore, a set of conditions is either a single condition or multiple conditions combined with the help of the logical operations. The precedence of logical operations is NOT>AND, OR (meaning that AND, OR have the same precedence); parentheses can be used to override precedence. E.g. `NOT A OR B` is equal to `(NOT A) OR B` and to put the scope of the negation onto the disjunction we have to write `NOT (A OR B)`. The full syntax is:

```
_CONDITIONS  -> (_CONDITIONS_)
_CONDITIONS_ -> _CONDITION_
_CONDITIONS_ -> NOT _CONDITIONS_
_CONDITIONS_ -> _CONDITIONS_ AND _CONDITIONS_
_CONDITIONS_ -> _CONDITIONS_ OR  _CONDITIONS_
```

A single condition is either an annotation condition, an object containment condition or a sequence containment condition:

```
_CONDITION_ -> _ANN_
_CONDITION_ -> _OBJCONT_
_CONDITION_ -> _SEQUENCECONT_
```

Below, we discuss the different types of conditions.

5.3.1 Annotation condition

An annotation condition states that the described object must be annotated in a specific way. The basic syntax of such condition is:

```
_ANN_ -> $annotation _OP_ 'pattern'
```

where `annotation` is the name of the annotation tier and `pattern` is a string or a regular expression which has to be matched against the content of the annotation tier. The `_OP_` is a comparison operator, which can be one of:

== exact match (the content of the annotation tier must be equal to pattern)

!= exact mismatch (the content of the annotation tier must not be equal pattern)

=~ regex match (pattern is a regular expression and the the content of the annotation tier must match this pattern)

!~ regex mismatch (pattern is a regular expression and the the content of the annotation tier must not match this pattern)

The regular expressions used by ToolboxSearch are the same as supported by the R grep(... , perl=TRUE) function (refer to R documentation for detailed list).

5.3.2 Object containment condition

This condition requires that the described object contains another object (e.g. word contained in a record, morpheme contained in a word). The syntax of the condition is

```
_OBJCONT_ -> CONTAINS _OBJDESC_
```

The contained object is again described via the `_OBJDESC_` rule (see above).

5.3.3 Sequence containment condition

This is similar to the object containment condition above, but in the sequence containment condition, we require that the object contains a set of other objects, which are positioned in a certain linear order. The sequence pattern is surrounded by edge brackets:

```
_SEQUENCECONT_ -> CONTAINS [_SEQPATTERN_]
```

The sequence patterns are very similar to (simplified) regular expression. Such pattern consists of one or more match conditions:

```
_SEQPATTERN_ -> _MATCHSEQ_
_MATCHSEQ_ -> _MATCH_
_MATCHSEQ_ -> _MATCH_ _MATCHSEQ_
```

A single match condition matches one or more elements (objects) in a sequence. A sequence of match conditions requires that objects matched by the individual match condition immediately follow after each other. The query language supports following match conditions:

Object desc match `_MATCH_ -> _REP_ _OBJDESC_`

Matches a sequence of one or more objects which can be described by the object descriptor. The optional prefix `_REP_` is the repetition statement. If it is absent, only one object in the pattern is matched. If present, it has the form `count` or `min : max`; in first case, exactly `count` objects with specified properties must be present in the sequence; in the later case, at least `min` objects with specified properties must be present in the sequence, in addition, up to `max` will be matched. `count` or `max` can be `*`, which means 'arbitrary number' (zero or more). Thus, `CONTAINS [@morpheme{`$mgl = {\sim}'N'$`} *@morpheme{`$mgl = {\sim}'LOC'$`} @morpheme{`$mgl = {\sim}'FOC'$`}]` will match morpheme sequences which contain a noun root morpheme followed by zero or more locative markers and a focus marker; these are sequences like *N FOC*, *N LOC FOC*, *N LOC LOC FOC* but not *N GEN FOC*.

Any match `_MATCH_ -> _REP_ ANY`

Matches a sequence of arbitrary elements. The `_REP_` is the same as above. This can be used if your pattern require an element to exist in a particular position, but you do not care what about the properties of this element. For example, the following pattern finds morpheme sequences which contain a noun root morpheme followed by a focus marker, with one or more additional morphemes in-between: `CONTAINS [@morpheme{`$mgl = {\sim}'N'$`} 1:* ANY @morpheme{`$mgl = {\sim}'FOC'$`}]`; these are sequences like *N GEN FOC N LOC FOC*, but not *N FOC*.

An important point is that sequences are matched independently to their initial position. So, the pattern like `CONTAINS [@morpheme{`$mgl = {\sim}'N'$`} @morpheme{`$mgl = {\sim}'LOC'$`}]` will match *N LOC* even if there are morphemes preceding or following this pattern, e.g. *DEF N LOC FOC*. Sometimes, though, we want to "anchor" the pattern on the container's boundary, for instance, if we want to find all words which start with a noun root morpheme and end with a locative marker. For this purpose, the query language provides the anchor symbol `#`. If a sequence patterns begins with `#`, e.g. `CONTAINS [# @morpheme{`$mgl = {\sim}'N'$`} @morpheme{`$mgl = {\sim}'LOC'$`}]`, then the pattern is only matched at the beginning of the container object (word, record, etc.), e.g. it will match *N LOC FOC* but not *DEF N LOC FOC*. Similarly, `CONTAINS [@morpheme{`$mgl = {\sim}'N'$`} @morpheme{`$mgl = {\sim}'LOC'$`} #]` will match *DEF N LOC* but not *N LOC FOC*; and `CONTAINS [# @morpheme{`$mgl = {\sim}'N'$`} @morpheme{`$mgl = {\sim}'LOC'$`} #]` will match only *N LOC*.

With the addition of the anchor symbol, the expansion rules for `_SEQUENCECONT_` must be extended as follows:

```
_SEQUENCECONT_ -> CONTAINS [_SEQPATTERN_]
_SEQUENCECONT_ -> CONTAINS [# _SEQPATTERN_]
_SEQUENCECONT_ -> CONTAINS [_SEQPATTERN_ #]
_SEQUENCECONT_ -> CONTAINS [# _SEQPATTERN_ #]
```

6 Query language - practical examples

The following examples show a few sample queries.

Find all records which contain a word with a locative marking:

```
p ← corpus %%
"@record
{
  CONTAINS @word
  {
    CONTAINS @morpheme{$mgl =~ 'LOC'}
  }
}"
```

```
Query done in 0.013 seconds
```

```
corpus[p]
```

```
Corpus with 463 entries (record) showing 1-3:

----------@1
\ref CLLDCh2R02S05.040
\EUDICOp Parbati
\gw them ŋŋ    yusandose                                    lo      ni      yogoi
\mph them ŋ    yus    -u     -dhend  -u ŋ    -s     -e      lo      ni      yo
-ko     -i
\mgl what.pro keep.vt -3P.gm -TEL.v2 -3P.gm -PRF.v2 -PST.gm PTCL.gm PTCL.gm DEM.ACROSS.gm
\lg C        C       -C     -C      -C     -C      -C      C       C/N     C
-C      -C

----------@2
\ref CLLDCh2R02S05.062
\EUDICOp PMR
\gw yoba ŋ                  yusa
\mph yo             -ʔpe ŋ   yus    -a
\mgl DEM.ACROSS.gm -LOC.gm keep.vt -IMP.gm
\lg C              -C      C       -C

----------@3
\ref CLLDCh2R02S05.063
\EUDICOp PMR
\gw hou ŋ       hugoi              ta
```

```
\mph ho ŋ          hu -ko      -i        ta
\mgl yes.interj DEM -GEN.gm -LOC.gm FOC.gm
\lg C/N          C    -C        -C        C
```

The same, but the word must end with the locative marker:

```
p ← corpus %%
"@record
{
  CONTAINS @word
  {
    CONTAINS [@morpheme{$mgl =~ 'LOC'} #]
  }
}"
```

```
Query done in 0.019 seconds
```

```
corpus[p]
```

```
Corpus with 367 entries (record) showing 1-3:

----------@1
\ref CLLDCh2R02S05.040
\EUDICOp Parbati
\gw them ŋŋ     yusandose                                          lo        ni        yogoi
\mph them ŋ     yus     -u      -dhend  -u ŋ     -s      -e        lo        ni        yo
-ko      -i
\mgl what.pro keep.vt -3P.gm -TEL.v2 -3P.gm -PRF.v2 -PST.gm PTCL.gm PTCL.gm DEM.ACROSS.gm
\lg C         C         -C      -C       -C      -C       -C       C         C/N      C
-C       -C

----------@2
\ref CLLDCh2R02S05.062
\EUDICOp PMR
\gw yoba ŋ                    yusa
\mph yo              -ʔpe ŋ    yus     -a
\mgl DEM.ACROSS.gm -LOC.gm keep.vt -IMP.gm
\lg C              -C       C        -C

----------@3
\ref CLLDCh2R02S05.063
\EUDICOp PMR
\gw hou ŋ        hugoi                ta
\mph ho ŋ         hu -ko      -i        ta
\mgl yes.interj DEM -GEN.gm -LOC.gm FOC.gm
\lg C/N          C    -C        -C        C
```

Find the list of words which contain a demonstrative morpheme and a locative marker (in that particular order):

```
p ← corpus %%
"@word
{
  CONTAINS [@morpheme{$mgl =~ 'DEM'} * ANY @morpheme{$mgl =~ 'LOC'}]
}
"
```

```
Query done in 0.028 seconds
```

```
corpus[p]
```

```
Corpus with 312 entries (word) showing 1-3:

-----------@1
\gw yogoi
\mph yo -ko -i
\mgl DEM.ACROSS.gm -GEN.gm -LOC.gm
\lg C -C -C

-----------@2
\gw yoba
\mph yo -?pe
\mgl DEM.ACROSS.gm -LOC.gm
\lg C -C

-----------@3
\gw ŋhugoi
\mph ŋhu -ko -i
\mgl DEM -GEN.gm -LOC.gm
\lg C -C -C
```

If you want to compile the list of unique words like that, its a bit more complicated. We will need to transform the result into a data frame and perform some R code which picks up the unique entries:

```
p ← corpus %%
"@word
{
  CONTAINS [@morpheme{$mgl =~ 'DEM'} * ANY @morpheme{$mgl =~ 'LOC'}]
}"
```

```
Query done in 0.028 seconds
```

```
# transform it into a data frame
words ← as.data.frame(corpus[p])
# and leave only the annotations we are interested in
words.pure ← words[, c("gw", "mph", "mgl")]
head(words.pure)
```

```
       gw   mph              mgl
1   yogoi    yo DEM.ACROSS.gm
2   yogoi  -ko         -GEN.gm
3   yogoi   -i         -LOC.gm
4    yoba    yo DEM.ACROSS.gm
5    yoba -?pe         -LOC.gm
6 ŋhugoi ŋ hu              DEM
```

```
# split the structure into individual words
words.pure ← split(words.pure, words$word.id)
words.list ← unique(words.pure)
head(words.list)
```

```
[[1]]
     gw mph              mgl
1 yogoi  yo DEM.ACROSS.gm
2 yogoi -ko         -GEN.gm
3 yogoi  -i         -LOC.gm

[[2]]
    gw  mph              mgl
4 yoba    yo DEM.ACROSS.gm
5 yoba -?pe         -LOC.gm

[[3]]
      gw mph       mgl
6 ŋhugoi ŋhu       DEM
7 ŋhugoi -ko -GEN.gm
8 ŋhugoi  -i -LOC.gm

[[4]]
       gw mph       mgl
9 ŋ hugoi ŋhu       DEM
10 ŋhugoi -ko -GEN.gm
11 ŋhugoi  -i -LOC.gm

[[5]]
     gw mph             mgl
12 ?hai  ba DEM.PROX.pro
13 ?hai  -i        -LOC.gm

[[6]]
     gw mph        mgl
14 ?bai  ba DEM.PROX
15 ?bai -?i       -LOC
```