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

Taras Zakharko
taras.zakharko@uzh.ch

July 10, 2012

The latest version of this package can be found at https://bitbucket.org/tzakharko/toolboxsearch.

This document uses examples from the Chintang Language Corpus for illustrative purposes. The corpus data is **not** distributed with this software or the document. Reference: Bickel, B., S. Stoll, M. Gaenszle, N. K. Rai, E. Lieven, G. Banjade, T. N. Bhatta, N. Paudyal, J. Pettigrew, I. P. Rai, M. Rai, 2012. Audiovisual corpus of the Chintang language, including a longitudinal corpus of language acquisition by six children, paradigm sets, grammar sketches, ethnographic descriptions, and photographs, http://www.spw.uzh.ch/clrp/. DOBES Archive, http://www.mpi.nl/DOBES.

Contents

1 About ToolboxSearch

ToolboxSearch is a new R package created for linguists who work with language corpora in Toolbox file format. The package contains utilities for loading and searching Shuebox/Toolbox corpora within R. Here are the key features of the package at one glance:

powerful search facility
The flexible corpus query language of ToolboxSearch makes it easy to extract parts of the corpora in accordance to a specific search pattern. The patterns are specified in a simple, readable and reusable way, e.g. the following R code, which will find all glossed utterances within the corpus that are uttered by adult speakers and contain at least one word whose gloss includes a demonstrative marker immediately followed by a locative marker:

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

The results of search queries can be converted into R data frames for subsequent statistical analysis. They can be also saved as a Toolbox file for editing or inspection.

"smart" import of Toolbox files

Toolbox/Shuebox uses plain text files to store corpora as sequences of interlinearly-glossed utterances. The vertical alignment of items (e.g. words and their glosses) is represented via string offsets, maintained by correct number of spaces. Unfortunately, the spacing within Toolbox files in real-word corpora is very often inconsistent due to unexpected behavior of Toolbox itself and usage of external editing tools which do not preserve the original spacing (such as a text editor). This makes it a difficult task to properly reconstruct the gloss structure. ToolboxSearch goes to great lengths to deal with this problem. Its import routines use adaptive parsing techniques, where each record in the file is repeatedly parsed using a number of different algorithms and settings, until it can be imported successfully. In addition, the package offers an alternative import algorithm which attempts to reconstruct the gloss structure based on morpheme hyphenation. Because of these advanced import features, ToolboxSearch is able to correctly parse Toolbox files where many other tools (e.g. ELAN at the moment of writing of this document) would produce erroneous results.

The import routines also 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 the 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.

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.

1.1 This document

This document is a user manual which will guide you through all the important features of ToolboxSearch. In the next section, we will review the Toolbox file format and point some common problems which arise when trying to import Toolbox to R. The remaining part of the manual explains how to load and search your corpora using ToolboxSearch.

The corpus examples from this manual are from the Chintang Language Corpus:

Bickel, B., S. Stoll, M. Gaenszle, N. K. Rai, E. Lieven, G. Banjade, T. N. Bhatta, N. Paudyal, J. Pettigrew, I. P. Rai, M. Rai, 2012. Audiovisual corpus of the Chintang language, including a longitudinal corpus of language acquisition by six children, paradigm sets, grammar sketches, ethnographic descriptions, and photographs, http://www.spw.uzh.ch/clrp/. DOBES Archive, http://www.mpi.nl/DOBES.

The manual assumes that the reader already has some basic familiarity with R and its command shell.

2 Notes on the anatomy of a Toolbox file

Toolbox is a popular software tool for interlinear glossing of language corpora and corresponding electronic dictionary creation. 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 transcribed words, morpheme glosses, speaker name, translation etc. The records are stored in a plain text file.

```
\ref CLLDCh2R06S02. 0001
\ELANBegin 00:00:00.824
\ELANEnd 00:00:06.198
\EUDICOp XYZ
\tx ne coha7
\gw ne           coha7
\mph ne           ci  -u     -hã7
\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 Chintang Corpus Toolbox file which represents a single record. Each line of text represents an annotation tier, the first item in the line (`\xxx`) is the symbolic name of the tier. The first tier (`\ref` is the record marker, which signals the start of a new record.

Annotation tiers can be arranged into different *annotation levels*. In the above example, there are three such levels — we will call them record level, word level and the morpheme level, respectively. The record level includes annotations which concern the whole record, such as the video timestamp (`\ELANBegin` and `\ELANEnd`), the speaker code (`\EUDICOp`), the transcribed text and its translation (`\tx`, `\eng`) and the date of last edit (`\dt`). The word level includes the grammatical words annotations — in this case it consists only of the word form `\gw`. Finally, the morpheme level consists of the morpheme annotations: the transcription `\mph`, the gloss `\mgl` and the source language of the lexeme (e.g. for code switching studies) `\lg`.

The Toolbox file format correctly stores the vertical alignment between the elements (i.e. the fact that morphemes `ci`, `-u`, `-hã7` comprise the word `coha7`). Consider the alignment of the tiers `\gw`, `\mph` and `\mgl` from the above example (with tier markers stripped and spaces visualized):

```
ne␣␣␣␣␣␣␣␣␣␣␣coha7
ne␣␣␣␣␣␣␣␣␣␣␣ci␣␣-u␣␣␣␣␣-hã7
EXCLA.interj␣eat␣-3P.gm␣-PRSV.IMP
```

Here, we can see that Toolbox automatically inserts spaces between tokens of different tiers such that corresponding elements occupy the same character starting position in their respective line (e.g. the word `coha?` and its first morpheme `ci`).

Unfortunately, this is not the complete story. Consider this example (taken from another record of the Chintang Corpus):

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

At first, it appears that the tokens are not aligned properly. For instance the morpheme `khat` is not properly aligned with its gloss `go` or the word `khai?ma` it belongs to. In fact, the alignment here is proper. The solution of the riddle lies in the (unfortunate) way Toolbox works with character encoding. Many corpora (including Chintang Corpus) use Unicode UTF-8 to encode the data. UTF-8 is a variable-byte encoding, which means that some characters (like `'n'`) are encoded as one byte of memory and some other characters (like `'ŋ'`, `'ʔ'`) as two or more. Unfortunately, for the purposes of alignment, Toolbox considers token length to be in bytes, and not in characters. Because of this, 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 `'ŋ'` twice results in the 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.

Unfortunately, the above rules do not appear to be absolute. Occasionally, Toolbox will count characters and not bytes, and/or accents as proper characters. Sometimes the behavior changes from one record to another (e.g. first record byte-aligned and the next one is character-aligned). To make the matter even worse, many existing Toolbox files are in even worse shape. Sometimes, the authors of the corpus will edit the toolbox file in a normal 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.

In conclusion, the format of the Toolbox file is very fragile and great care should be taken when trying to parse it.

2.1 ToolboxSearch import algorithm

ToolboxSearch goes to great lengths in order to ensure that Toolbox file import into R will be easy-to-setup, quick and error-prone. Currently, it implements two different import algorithms which

can be used in different scenarios.

The first and default algorithm is *position tracking*, which assumes that the vertical alignment of the tokens is correctly encoded via spaces. However, it remain flexible in regards to what 'correctly' actually means. The tokens might be aligned according to their byte length or their character length, with accents taken into consideration or ignored. The algorithm will try each of these possibilities for each record in the file separately. This way, the import will be successful even when the convention should change from one record to another. If the algorithm is still unable to parse the record, it will be ignored and a corresponding message will be logged. This algorithms works rather well for many corpora and requires no additional setup from the user.

If the spacing in the Toolbox file has been damaged beyond automatic repair (via manual edits or third-party tools), the second algorithm may be used. It is based on *sequence tracking*. The idea of the algorithm is to make an assumption that proper sequences in the interlinear gloss follow a specific pattern. The majority of corpora use hyphens along with morpheme tokens: *aaa*- to encode prefixes and -*aaa* to encode suffixes. Sometimes, = will be used to represent clitics. If a corpus uses some sort of morpheme hyphenation, it can be assumed that each proper word sequence has the form x-x-x-x-x etc., i.e. all morphemes which have hyphenation between them belong to the same word.

Following this idea, the algorithm will try to collect 'connected' morphemes into words. Thus, the algorithm does not rely on spacing at all, but it only works for hyphenated corpora. If the corpus has multiple morpheme tiers, it is enough if only one of them is hyphenated (although more then one may be), the rest of the tokens will be assigned to the structure based on one-to-one correspondence.

Both above algorithms can fail to parse a record which is too badly damaged. 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 edit 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.

3 Loading, viewing and partitioning the corpus

3.1 Installation and loading

Before the package can be used in R, it must be installed. Download the appropriate binary version for your operating system and install it using the R menu option Package Installer. Linux users can install the package from source by downloading the source code and executing

```
R CMD install toolboxsearch
```

from the command line.

Please note that you will need R 2.14 or higher version to use ToolboxSearch.

After the package has been installed, loading it is as easy as any other R package. Simply type

```
library(ToolboxSearch)
```

in the R command line.

3.2 Toolbox format descriptor

To successfully load a Toolbox file, the parser needs some basic information about the file structure. In particular, you must specify the names of relevant (to-be-imported) annotation tiers and their relationship between each other. In ToolboxSearch, this information is stored within a *Toolbox format descriptor*. Consider an example Toolbox record from the Chintang Corpus:

```
\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
```

Let us assume that we are only interested in information about speaker (tiers `\EUDICOp`, `\age`), translation (`\eng`) and the interlinear gloss (tiers `\gw`, `\mph`, `\mgl`, `\lg`). The corresponding format descriptor is declared as:

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

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

The descriptor is set up in hierarchical levels (`record`, `word`, `morpheme`). Each level comprises of one or more annotation tiers. The names of the levels are arbitrary chosen by the user (we can also use

`clause`, `sentence` etc. instead `record` and `mor`, `m`, etc. instead `morpheme`). The first defined tier of the first level has a special meaning - it is treated as a record marker.

A step by step definition of a Toolbox format descriptor is as follows:

1. Decide which annotation tiers from the file you want to import
2. Divide these tiers into logical hierarchical levels and pick the names for these levels (the record marker must always belong to the outer-most level!). On practice, you will never need to set up more then three levels.
3. Define the R structure for the descriptor using the `toolboxFormat()` function. The levels are set up as arguments to this function as `level = content` pairs. Here, `level` is the name of the level and `content` is a vector (or a single value) of names of annotation tiers. You don't have to use quotation marks (although you can). The levels are declared in a hierarchical order, meaning that the first level will be the record-level one and the last level the morpheme one. The record marker should be the first declared tier of the uppermost level.

Note that you don't have to import the complete interlinear gloss. It is possible to import only some outer-level data, e.g.:

```
fmt ← toolboxFormat(
        record=c(ref, eng)
)
fmt
```

```
Toolbox format descriptor with 1 level
  record marker \ref
  @record: \ref \eng
```

or only word data:

```
fmt ← toolboxFormat(
        record=ref,
        word=gw
)
fmt
```

```
Toolbox format descriptor with 2 levels
  record marker \ref
  @record: \ref
  @word: \gw
```

By setting up the descriptor appropriately, you make sure that the data is imported into R in a shape you need. The following descriptor would load morpheme glosses as non-tokenized outer-level string (akin to translation):

```
fmt ← toolboxFormat(
        record=c(ref, mgl)
)
fmt
```

```
Toolbox format descriptor with 1 level
  record marker \ref
  @record: \ref \mgl
```

3.3 Importing Toolbox files

Importing Toolbox files with ToolboxSearch is very easy.

```
fmt ← toolboxFormat(
        record=c(ref, EUDICOp, age, eng),
        word=gw,
        morpheme=c(mph, mgl, lg)
)
crp ← readToolbox("dta/Budhohang_d.txt", fmt)
crp
```

```
Corpus with 91 entries (record) showing 1–3:

————————@1
\ref Budhohang_d.01
\gw he    parmeswora   sabai~caine ke      thippe              dhani
\mph he ś  paramevara sab~  caine ke      thippe              dhani
\mgl ADDR Lord          all    PTCL  FILLER deity(grandfather) owner
\lg N     N             N      N     C–RL   C–RL               N

————————@2
\ref Budhohang_d.02
\gw warimi  kumdami sirimi
\mph warimi  kumdami sirimi
\mgl a_samet a_samet a_samet
\lg C–RL     C–RL     C–RL

————————@3
\ref Budhohang_d.03
\gw ambira  legura ŋ ha  na    sabai kuro
\mph ambira  legura ŋ ha  na    sab    kura
\mgl a_place a_place king PTCL all    thing
\lg C–RL     C–RL     C–RL C     N      N
```

This code imports the Toolbox file `Budhohang_d.txt` from the folder `dta`. The corpus data from the file is stored in the variable `crp`. Typing the name of this variable in the R command line will show you the first few records from the corpus.

As already mentioned, ToolboxSearch uses a flexible import algorithm (see page ??), which can be tweaked by providing additional parameters to the `readToolbox()` function. In the default mode, the function will use position tracking mode, which assumes that vertical alignment of the tokens can be reconstructed from the spacing. If the spacing is broken, but the file is glossed using hyphens as morpheme connectors, you can tell `readToolbox()` to load the file using the sequence tracking mode:

```
crp ← readToolbox("dta/Budhohang_d.txt", fmt, morpheme='sequence')
crp
```

```
Corpus with 91 entries (record) showing 1-3:

----------@1
\ref Budhohang_d.01
\gw he    parmeswora   sabai~caine ke      thippe              dhani
\mph he ś  paramevara sab~  caine ke      thippe              dhani
\mgl ADDR Lord          all    PTCL  FILLER deity(grandfather) owner
\lg N     N             N      N     C-RL   C-RL               N

----------@2
\ref Budhohang_d.02
\gw warimi  kumdami sirimi
\mph warimi  kumdami sirimi
\mgl a_samet a_samet a_samet
\lg C-RL     C-RL     C-RL

----------@3
\ref Budhohang_d.03
\gw ambira  legura ŋ ha  na    sabai kuro
\mph ambira  legura ŋ ha  na    sab    kura
\mgl a_place a_place king PTCL all    thing
\lg C-RL     C-RL     C-RL C     N      N
```

This mode is activated by passing an additional parameter to the function. The parameter must have the same name as the level for which the sequence tracking should be activated — in our (and probably virtual any other) case — morpheme level. In the sequence tracking mode, ToolboxSearch assumes that - and = are morpheme connectors. If your corpus uses different connector symbols, you can specify them explicitly (e.g. if the connector is &):

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

Often, it is required to load more then one Toolbox file at once. ToolboxSearch is very convenient in regards to this. The first argument of readToolbox() function will accept a vector of file names. Each of these files will be imported and the results are collapsed to a single corpus object. The following code imports all files from the folder `dta`.

```
crp ← readToolbox(dir('dta', full.names=T), fmt)
crp
```

```
Corpus with 1118 entries (record) showing 1-3:

----------@1
\ref Budhohang_d.01
\gw he    parmeswora   sabai~caine ke      thippe              dhani
\mph he ś  paramevara sab~  caine ke      thippe              dhani
\mgl ADDR Lord          all    PTCL  FILLER deity(grandfather) owner
\lg N     N             N      N     C-RL   C-RL               N

----------@2
\ref Budhohang_d.02
\gw warimi  kumdami sirimi
\mph warimi  kumdami sirimi
\mgl a_samet a_samet a_samet
\lg C-RL     C-RL     C-RL
```

```
————————@3
\ref Budhohang_d.03
\gw ambira  legura ŋ ha  na   sabai kuro
\mph ambira  legura ŋ ha  na   sab   kura
\mgl a_place a_place king PTCL all   thing
\lg C-RL    C-RL    C-RL C    N     N
```

This is equivalent to importing the files separately and then collapsing them using the concat.corpus () function:

```
crp.1 ← readToolbox('dta/file_1.txt', fmt)
crp.2 ← readToolbox('dta/file_2.txt', fmt)
...
crp.n ← readToolbox('dta/file_n.txt', fmt)
crp ← concat.corpus(crp.1, crp.2, ..., crp.n)
```

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

```
head(parse.log(crp))
```

```
                                Budhohang_d.01
"ok(settings: skip_invisibles=1, use_bytes=1)"
                                Budhohang_d.02
"ok(settings: skip_invisibles=0, use_bytes=0)"
                                Budhohang_d.03
"ok(settings: skip_invisibles=0, use_bytes=1)"
                                Budhohang_d.04
"ok(settings: skip_invisibles=1, use_bytes=0)"
                                Budhohang_d.05
"ok(settings: skip_invisibles=1, use_bytes=0)"
                                Budhohang_d.06
"ok(settings: skip_invisibles=1, use_bytes=0)"
```

The parse log can be used to detect glossing errors, for instance, cases when the number of morpheme glosses does not match the number of morphemes.

3.4 Viewing and partitioning the corpus

In the previous section we have loaded a Toolbox corpus consisting of multiple files and stored it in a variable named crp. In this section we will see how we can show and extract parts of the corpus.

Simply typing the variable into R command line will display the first few records of the corpus data (similarly as to how R displays values of other variables):

```
crp
```

```
Corpus with 795 entries (record) showing 1-3:

————————@1
\ref Budhohang_d.01
\gw he   parmeswora  sabai ~caine ke     thippe            dhani
\mph he ś  paramevara sab ~  caine ke     thippe             dhani
```

```
\mgl ADDR Lord          all    PTCL  FILLER deity(grandfather) owner
\lg N    N             N      N     C-RL   C-RL                N

------------@2
\ref Budhohang_d.02
\gw warimi  kumdami sirimi
\mph warimi  kumdami sirimi
\mgl a_samet a_samet a_samet
\lg C-RL    C-RL    C-RL

------------@3
\ref Budhohang_d.03
\gw ambira  legura ŋ ha  na   sabai kuro
\mph ambira  legura ŋ ha  na   sab   kura
\mgl a_place a_place king PTCL all   thing
\lg C-RL    C-RL    C-RL C    N     N
```

The function length.corpus () will show us information about number of distinct (non-empty) elements at a particular level in the corpus. Note that if a record is not glossed, its number of morphemes is 0!

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

```
[1] 795
```

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

```
[1] 3549
```

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

```
[1] 5174
```

We can also tell R to show us a particular set of records using the print () function, with the second argument being the number of the record we want to see:

```
print(crp, 2)
```

```
Corpus with 795 entries (record) showing 2:

------------@2
\ref Budhohang_d.02
\gw warimi  kumdami sirimi
\mph warimi  kumdami sirimi
\mgl a_samet a_samet a_samet
\lg C-RL    C-RL    C-RL
```

```
print(crp, 5)
```

```
Corpus with 795 entries (record) showing 5:

------------@5
\ref Budhohang_d.05
\gw jattigo~   caine ke     caurasko            dhani
\mph jattigo~   caine ke     cauras          -ko  dhani
\mgl as.much.as PTCL  FILLER every.direction -GEN owner
\lg C/N         N     C-RL   C-RL            -C   N
```

We can also specify a sequence (the notation `a:b` in R means a sequence of numbers from a to b) or an arbitrary vector of record indices:

```
print(crp, 5:8)
```

```
Corpus with 795 entries (record) showing 5–8:

————————@5
\ref Budhohang_d.05
\gw jattigo~   caine ke     caurasko             dhani
\mph jattigo~   caine ke     cauras          –ko  dhani
\mgl as.much.as PTCL  FILLER every.direction –GEN owner
\lg C/N         N     C–RL   C–RL            –C   N

————————@6
\ref Budhohang_d.06
\gw sabai~caine ke     bhanedekhilai   jattigo
\mph sab~  caine ke     bhanedekhi –lai jattigo
\mgl all   PTCL  FILLER FILLER     –DAT as.much.as
\lg N      N     C–RL   C–RL       –N   C/N

————————@7
\ref Budhohang_d.07
\gw~caine ke~    nau ŋ sie wa      garera~     caine ke     bhandekhinlai
\mph~caine ke~    nau ŋ si  wa      garera~     caine ke     bhandekhile
\mgl PTCL  FILLER nine horn chicken having.done PTCL  FILLER FILLER
\lg N      C–RL   N    C    C       N           N     C–RL   C–RL

————————@8
\ref Budhohang_d.08
\gw haniko~   caine ke     udhauli          sewa~   caine ke     bhandekhinlai
\mph hani –ko~ caine ke     udhauli          sewa~   caine ke     bhandekhile
\mgl 2p   –GEN PTCL  FILLER descending.time service PTCL  FILLER FILLER
\lg C     –C   N     C–RL   N               C/B     N     C–RL   C–RL
```

```
print(crp, c(2, 5, 8))
```

```
Corpus with 795 entries (record) showing 2, 5, 8:

————————@2
\ref Budhohang_d.02
\gw warimi  kumdami sirimi
\mph warimi  kumdami sirimi
\mgl a_samet a_samet a_samet
\lg C–RL     C–RL    C–RL

————————@5
\ref Budhohang_d.05
\gw jattigo~   caine ke     caurasko             dhani
\mph jattigo~   caine ke     cauras          –ko  dhani
\mgl as.much.as PTCL  FILLER every.direction –GEN owner
\lg C/N         N     C–RL   C–RL            –C   N

————————@8
\ref Budhohang_d.08
\gw haniko~   caine ke     udhauli          sewa~   caine ke     bhandekhinlai
\mph hani –ko~ caine ke     udhauli          sewa~   caine ke     bhandekhile
\mgl 2p   –GEN PTCL  FILLER descending.time service PTCL  FILLER FILLER
\lg C     –C   N     C–RL   N               C/B     N     C–RL   C–RL
```

Often we are only interested in a particular part of the corpus. We can use corpus *partitioning* (or *slicing*) to extract a subset of the corpus data. In ToolboxSearch, this works just like vector indexing:

```
crp.part ← crp[2]
crp.part
```

```
Corpus with 1 entries (record) showing 1:

──────────@1
\ref Budhohang_d.02
\gw warimi  kumdami sirimi
\mph warimi  kumdami sirimi
\mgl a_samet a_samet a_samet
\lg C-RL    C-RL    C-RL
```

The partition index works the same way as the second argument of `print()`. The main difference is that `print()` will only print the respective records, while corpus partitioning will copy the data from the original corpus and create a 'new' corpus object.

It is also possible to extract a different level of the corpus. For this, you need to specify the level explicitly. The following examples show how to extract first 5 word entries from the corpus:

```
crp[1:5, "word"]
```

```
Corpus with 5 entries (word) showing 1-3:

──────────@1
\gw he
\mph he
\mgl ADDR
\lg N

──────────@2
\gw parmeswora
\mph śparamevara
\mgl Lord
\lg N

──────────@3
\gw sabai
\mph sab
\mgl all
\lg N
```

or first few morphemes in even positions:

```
crp[c(2, 4, 6, 8), "word"]
```

```
Corpus with 4 entries (word) showing 1-3:

──────────@1
\gw parmeswora
\mph śparamevara
\mgl Lord
\lg N

──────────@2
```

```
\gw ~caine
\mph ~caine
\mgl PTCL
\lg N

————————@3
\gw thippe
\mph thippe
\mgl deity(grandfather)
\lg C–RL
```

You can also omit the partition index. Then, all elements will be extracted. This is a quick way to split the corpus into words or morphemes (e.g. if if you are interested in compiling the lists of morphemes):

```
crp[level="morpheme"]
```

```
Corpus with 5174 entries (morpheme) showing 1–3:

————————@1
\mph he
\mgl ADDR
\lg N

————————@2
\mph śparamevara
\mgl Lord
\lg N

————————@3
\mph sab
\mgl all
\lg N
```

This is equivalent to:

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

3.5 Index objects

Another way to do corpus partitioning is to use the special data objects provided by ToolboxSearch, the *corpus index objects*. The objects store the "coordinates" of a corpus partition, without doing the actual partitioning. Index objects will be an invaluable tool when we learn to use the ToolboxSearch corpus search functionality.

The function index.corpus() is used to create a index object. It takes the same arguments as the actual corpus partitioning:

```
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 selects the first three words of a corpus and index2 selects the second and the forth morphemes of a corpus. To perform the actual corpus partitioning and get the corresponding corpus subset, you can simply use the index object as a partition index:

```
crp[index1]
```

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

————————@1
\gw he
\mph he
\mgl ADDR
\lg N

————————@2
\gw parmeswora
\mph śparamevara
\mgl Lord
\lg N

————————@3
\gw sabai
\mph sab
\mgl all
\lg N
```

```
crp[index2]
```

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

————————@1
\mph śparamevara
\mgl Lord
\lg N

————————@2
\mph ~caine
\mgl PTCL
\lg N
```

Hence, a command like

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

is equivalent to

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

A very powerful feature of index objects is their ability to be combined using set operations. In combination with the corpus search functionality, this allows you to quickly combine different

search patterns (as explained in next section of the manual). Index objects support union, intersection or difference operations — they are also very easy to use, because they work just like the regular arithmetics 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)
```

Sometimes it is necessary to select all but the indexed elements. To create a complement of an index in respect to a particular corpus, you can simply subtracts the index from the corpus:

```
crp – index1
```

```
Corpus subset@word: 4–3549 (3546 elements)
```

```
crp[crp – index1]
```

```
Corpus with 3546 entries (word) showing 1–3:

————————@1
\gw ˜caine
\mph ˜caine
\mgl PTCL
\lg N

————————@2
\gw ke
\mph ke
\mgl FILLER
\lg C–RL

————————@3
\gw thippe
\mph thippe
\mgl deity(grandfather)
\lg C–RL
```

```
corpus ← crp
```

3.6 Doing statistics

The main goal of ToolboxSearch is to allow the user to quickly extract the interesting data for further processing. ToolboxSearch uses its own internal data format to store corpus data. However, a ToolboxSearch corpus can be quickly converted into an R data frame to do some statistics. Converting a corpus to the data frame is straightforward:

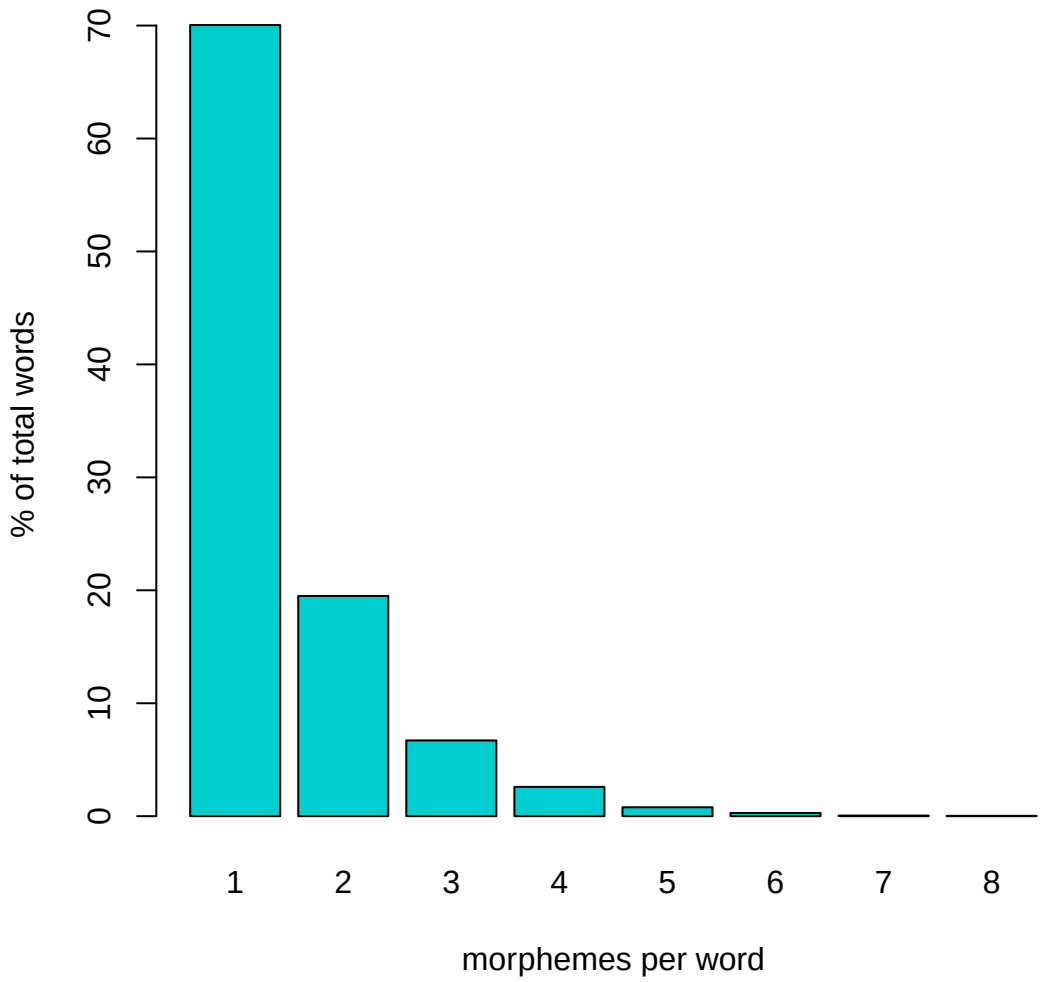
```
head(as.data.frame(crp))
```

```
  record.id word.id morpheme.id                 ref       gw     mph       mgl
1         1       1           1 appa_katha_talk.001 ajikali ajjoli thesedays
2         1       2           2 appa_katha_talk.001    appa      a-   1sPOSS-
3         1       2           3 appa_katha_talk.001    appa      pa    father
4         1       3           4 appa_katha_talk.001  manchi manchi       not
5         2       4           5 appa_katha_talk.007     tai     tai      even
6         2       5           6 appa_katha_talk.007 ŋ     ya ŋ   ya       ADD
```

Each annotation tier is stored as a column. Each row corresponds to the lowest level element (morpheme). The values of higher-level tiers are replicated accordingly. For instance, in this case the word *appa* consists of two morphemes, *a-* and *pa*. Thus, *appa* will be duplicated. The `.id` columns of the data frame indicate the element the current row (morpheme) belongs to. For instance, we can see that the first records spans morphemes $1-4$ and words $1-3$, while the second word spans morphemes 2 and 3.

As an illustration, the following code plots the frequency distribution of words in respect to their length in morphemes:

```
df ← as.data.frame(crp)
tab ← table(sapply(split(df$morpheme.id, df$word.id), length))
tab ← tab/sum(tab)*100
barplot(tab, ylab='% of total words', xlab='morphemes per word', col='cyan3')
```



3.7 Saving Toolbox files

An important function of ToolboxSearch is to export a corpus from R to Toolbox again. You can use it, for example, to save some interesting examples you have compiled from the corpus using the search facility. Saving Toolbox files is very simple:

```
# create a corpus partition
sub.crp ← crp[...]
writeToolbox("my_examples.txt", sub.crp)
```

This function produces correctly formatted Toolbox files which can be further edited in Toolbox or imported to a third party tool (e.g. ELAN).

4 Corpus search

The most powerful component of ToolboxSearch is its flexible search facility, which allows the linguist to extract elements from the corpus in according to a given pattern. The pattern is specified in a *query language*. This language has been specifically designed to be easy to learn, easy to write and easy to read. The distinguished feature of the query language is the ability to search for elements which contain particular sequences of subelements (such as words which contain a particular sequence of morphemes).

This section of the manual will introduce the query language and its elements.

4.1 An introduction to the query language

The query language allows the user to search for elements (e.g. records, words or morphemes) in the corpus which match a specific pattern. A search pattern combines a set of constrains, including constrains in regards to the annotation (e.g. 'find all morphemes with a particular gloss'), simple containment relations (e.g. 'find all words which contain a certain morpheme') or sequence containment relation (e.g. 'find all words which contain a particular morpheme sequence'). The following is a simple query which matches all records where the english translation (the annotation tier `\eng`) contains a substring 'beer':

```
@record{$eng ='beer'}
```

This example illustrates some basic principles of the query language. A declaration in form of $@L\{ \ldots \}$ is an *element pattern*. An element pattern matches a class of elements on a particular level L according to some conditions. The conditions are listed within the curvy brackets. In this example, we have one condition, which is `$eng =~ 'plum'`. This tells ToolboxSearch to constrain the list of resulting record-level elements to ones whose `\eng` annotation tier includes a substring 'beer' (which would find results like 'He had a beer' but also 'Beeri was the father of the prophet Hosea'). The operation '=~' here means 'match the regular expression'.

In ToolboxSearch, using the query language to search the corpus is very easy. The query is simply written as a string within R and the search is carried out using the `%%` operator:

```
ri ← crp %% "@record{$eng =~ 'beer'}"
ri
```

```
Corpus subset@record: 122, 149, 462, 470, 472 (5 elements)
```

The search returns a corpus index object as a result (see page ??). This index object contains the

indices of the elements which match the query. It can be used to extract these elements using corpus partitioning:

```
crp[ri]
```

```
Corpus with 5 entries (record) showing 1–3:

————————@1
\ref Burhahang_02.08
\eng (He is) offering the local beer and yeast.
\gw khamawa                maciya~soloiwa ŋʔ samami  garikana
\mph khamawa                maciya~soloiwa ŋʔ samami  garera
\mgl local.beer.and.liquor yeast  calabash materials having.done
\lg C–RL                   C–RL   C–RL     C–RL      N

————————@2
\ref Burhahang_02.42
\eng (He is) offering the local beer, yeast, calabash.
\gw ~khamauwa              maciwa~               soloiwa ŋʔ samami  garikana
\mph ~khamauwa              maciya~               soloiwa ŋʔ samami  garera
\mgl local.beer.and.liquor local.beer.and.liquor calabash materials having.done
\lg C–RL                   C–RL                  C–RL     C–RL      N

————————@3
\ref arkha_hengma.03
\eng Rice beer is needed
\gw khaca      caha lino
\mph khaca      caha lis       –no
\mgl grain.mash need be.needed –IND.NPST
\lg C          N    C/N       –C
```

We are not limited to searches at the record level. In fact, we can search at any level defined in the corpus. Here, we look for words which end in 'ka':

```
ri ← crp %% "@word{$gw =~ 'ka$'}"
crp[ri]
```

```
Corpus with 32 entries (word) showing 1–3:

————————@1
\gw ŋlaka
\mph ŋlaka
\mgl upside.down
\lg C–RL

————————@2
\gw ŋheka
\mph ŋheka
\mgl upside.down
\lg C–RL

————————@3
\gw ŋlaka
\mph ŋlaka
\mgl upside.down
\lg C–RL
```

As you can see, ToolboxSearch will automatically extract the correct element from the corpus: the resulting subcorpus becomes a list of word instead of a list of records.

A powerful feature of the query language is its compositionality. For instance, we can combine different search conditions. The following example will find all records which include substrings 'beer' and 'give' in the translation:

```
crp[crp %% "@record{$eng =~ 'beer' AND $eng =~ 'need'}"]
```

```
Corpus with 1 entries (record) showing 1:

----------@1
\ref arkha_hengma.03
\eng Rice beer is needed
\gw khaca       caha lino
\mph khaca       caha lis       -no
\mgl grain.mash need be.needed -IND.NPST
\lg C          N     C/N       -C
```

The conditions are combined with the help of logical operations `AND`, `OR` and `NOT`. You can also use brackets to enforce precedence. The usage of the logical operators closely follows the rules of the usual predicate logic. For example, to find records with 'beer' and either 'need' or 'offer' in the translation:

```
crp[crp %% "@record{$eng =~ 'beer' AND ($eng =~ 'need' OR $eng =~ 'offer')}"]
```

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

----------@1
\ref Burhahang_02.08
\eng (He is) offering the local beer and yeast.
\gw khamawa                maciya~soloiwa ŋʔ samami  garikana
\mph khamawa                maciya~soloiwa ŋʔ samami  garera
\mgl local.beer.and.liquor yeast  calabash materials having.done
\lg C-RL                   C-RL   C-RL     C-RL      N

----------@2
\ref Burhahang_02.42
\eng (He is) offering the local beer, yeast, calabash.
\gw ~khamauwa              maciwa~              soloiwa ŋʔ samami  garikana
\mph ~khamauwa              maciya~              soloiwa ŋʔ samami  garera
\mgl local.beer.and.liquor local.beer.and.liquor calabash materials having.done
\lg C-RL                   C-RL                 C-RL     C-RL      N

----------@3
\ref arkha_hengma.03
\eng Rice beer is needed
\gw khaca       caha lino
\mph khaca       caha lis       -no
\mgl grain.mash need be.needed -IND.NPST
\lg C          N     C/N       -C
```

To find records with 'beer' but not 'need' in the translation:

```
crp[crp %% "@record{$eng =~ 'beer' AND NOT $eng =~ 'need'}"]
```

```
Corpus with 4 entries (record) showing 1-3:

----------@1
\ref Burhahang_02.08
\eng (He is) offering the local beer and yeast.
```

```
\gw khamawa                  maciya~soloiwa ŋʔ samami   garikana
\mph khamawa                  maciya~soloiwa ŋʔ samami   garera
\mgl local.beer.and.liquor yeast   calabash materials having.done
\lg C-RL                     C-RL    C-RL      C-RL       N

----------@2
\ref Burhahang_02.42
\eng (He is) offering the local beer, yeast, calabash.
\gw ~khamauwa                maciwa~                soloiwa ŋʔ samami   garikana
\mph ~khamauwa                maciya~                 soloiwa ŋʔ samami   garera
\mgl local.beer.and.liquor local.beer.and.liquor calabash materials having.done
\lg C-RL                     C-RL                    C-RL      C-RL       N

----------@3
\ref arkha_hengma.14
\eng Putting two third water of the rice beer
\gw khacakko         hicci bhaga       leki           cuwa  tima           kina
\mph khaca       -ko  hicce bhag -a     leki           cuwa  tis       -ma  kina
\mgl grain.mash -GEN two    part -NTVZ approximately water put.into -INF SEQ
\lg C           -C   C      N    -C     C              C     C         -C   C
```

And, find all words which either end in or start with 'ka'.

```
crp[crp %% "@word{$gw =~ 'ka$' OR $gw =~ '^ka'}"]
```

```
Corpus with 63 entries (word) showing 1-3:

----------@1
\gw ŋlaka
\mph ŋlaka
\mgl upside.down
\lg C-RL

----------@2
\gw ŋheka
\mph ŋheka
\mgl upside.down
\lg C-RL

----------@3
\gw ŋlaka
\mph ŋlaka
\mgl upside.down
\lg C-RL
```

The above examples feature a particular type of search condition: annotation condition. It has the form `$T` *op* `'val'`, where `T` is a name of an annotation tier, *op* is a comparison operator and `'val'` is a substring. The dollar sign tells ToolboxSearch that we want to match an annotation. The current version of ToolboxSearch can only do string-based match (so you can't do something like `$age > 5` yet).

Another type of search condition is the *containment condition*. It tells ToolboxSearch to find all elements which contain specific elements (which in turn, are matched using their own pattern). Here, for example, we will find all records which contain words ending in 'ka':

```
crp[crp %% "@record{CONTAINS @word{$gw =~ 'ka$'}}"]
```

```
Corpus with 29 entries (record) showing 1–3:

————————@1
\ref Burhahang_01.38
\eng Let it not be upside down. may it be well.
\gw ŋlaka ŋ       heka        lima ʔ     maha
\mph ŋlaka ŋ       heka        lis –ma ʔ maha
\mgl upside.down upside.down be  –INF no
\lg C–RL         C–RL         C   –C   C

————————@2
\ref Burhahang_02.66
\eng It is saying that let it not be upside down and fall down.
\gw ŋlaka ŋ       heka ʔʔ       yuimahaima         lima ʔ     maha bhanikana
\mph ŋlaka ŋ       heka ŋ       yu –ma  –hatt –ma   lis –ma ʔ maha bhonikana
\mgl upside.down upside.down be  –INF –TEL  –INF be  –INF no     having.said
\lg C–RL         C–RL         C   –C   –C    –C   C   –C   C      C–RL

————————@3
\ref chintang_sahid.023
\eng After that they let me free.
\gw uti        pachi     akka˜cai˜      uledehe
\mph utti       pachi     akka˜cai       u–  let       –e   –˜he
\mgl that.much later.on 1s   SPEC.TOP 3A– let.free –PST –ePST
\lg C          N         C    N         C–  C         –C   –C
```

Of course, containment conditions can be also combined with each other and other relations. Consider:

```
crp[crp %% "@record
{
  $eng =~ 'go' AND
  CONTAINS @word{$gw =~ 'ka$' OR $gw =~ '^ka'}
}"]
```

```
Corpus with 6 entries (record) showing 1–3:

————————@1
\ref chintang_sahid.183
\eng Goodness gracious, in the year 36 they caught him and...
\gw atterika ho chattis ʔ   salbe     na   ulabe               kina
\mph atterika ho chattis     sal  –ʔpe na   u–       lab   –e   kina
\mgl EXCLA     be thirty.six year –LOC PTCL 3nsS/A– catch –PST SEQ
\lg C         N  N           N    –C   C    C–       C     –C   C

————————@2
\ref chintang_sahid.238
\eng We also don't go.
\gw ŋkanaa ŋya ŋiŋkhacekean
\mph ŋkanaa ŋya khat –ce –kV ŋ  –a –iŋn
\mgl 1pe     ADD go    –ns –NPST –e  –NEG
\lg C       C   C     –C  –C     –C  –C

————————@3
\ref chintang_sahid.255
\eng When I go (there) these days...
\gw a       ajikali    na    akka ʔʔkhaiyaa            garda na
\mph ah      ajjoli     na    akka khat ŋ–a ʔ   –˜a      garda na
\mgl FILLER thesedays PTCL 1s    go    –1sS/P –1sNPST doing PTCL
\lg C       C/N        C     C     C     –C      –C       N      C
```

Or:

```
crp[crp %% "@record
{
  CONTAINS @word{$gw =~ 'ka$'}
  OR
  CONTAINS @word{$gw =~ '^ba'}
}"]
```

```
Corpus with 187 entries (record) showing 1–3:

————————@1
\ref Budhohang_d.04
\gw sirjana  gurjana~ caine ke     bhane~ baphaima punne
\mph sirjana  gurjani~ caine ke     bhane~ baphaima punne
\mgl creation creation PTCL  FILLER FILLER DEM      charity
\lg C/N–RL   C–RL     N     C–RL   C–RL   C–RL     N

————————@2
\ref Budhohang_d.11
\gw sabai~baphaima ?pinaani              kha
\mph sab~  baphaima pit  –na ? –~a     –i kha
\mgl all   DEM      give –1>2 –1sNPST –p FOC
\lg N     C–RL     C    –C   –C      –C C

————————@3
\ref Budhohang_d.14
\gw sabai jiudana      bardana  saranapicha na
\mph sab ā  jiudn ā      bardn   saranapicha na
\mgl all   gift.of.body blessing protection  PTCL
\lg N     N            N        C–RL        C
```

We can also nest containment relation to even further levels. The following will find all records which contain at least one word which contains a locative marker:

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

```
Corpus with 110 entries (record) showing 1–3:

————————@1
\ref Budhohang_d.20
\gw ba ?      thembeko      binti   na ?  nummaa
\mph ba       them –?pe –ko  binti   na   numd –ma ?  –~a
\mgl DEM.PROX what –LOC –GEN request PTCL do   –1sS/P –1sNPST
\lg C         C    –C   –C   N       C    C    –C     –C

————————@2
\ref Budhohang_d.22
\gw~ha ?    bagobe~  caine ke     haniko~   cai      ke     bhandekhilai
\mph~ha     bago –?pe~caine ke     hani –ko~ cai      ke     bhandekhile
\mgl FILLER DEM  –LOC PTCL  FILLER 2p   –GEN SPEC.TOP FILLER FILLER
\lg C/N     C    –C   N     C–RL   C    –C   N        C–RL   C–RL

————————@3
\ref Budhohang_d.23
```

```
\gw bago~ha ŋ     nawagi ʔ      sewabe~      cai
\mph bago~ha ā     nuwgi        sewa     –ʔpe~cai
\mgl DEM  FILLER first.fruits service –LOC SPEC.TOP
\lg C     C/N     N            C/B      –C    N
```

The last type of condition featured by ToolboxSearch is the *sequence pattern* condition. Sequences woks very similar to regular expressions: they match a particular sequence of elements. Let us suppose that we are interested in finding all records which contains a sequence of a word starting with 'ba' immediately followed by a word starting with 'ma'. This is easy with sequence conditions:

```
crp[crp %% "@record
{
  CONTAINS [@word{$gw=~'^ba'} @word{$gw=~'^ma'}]
}"]
```

```
Corpus with 1 entries (record) showing 1:

————————@1
\ref chintang_sahid.059
\eng ...the offsprings of my grandfather's youngest son do not live here.
\gw athippa         kanchako              sakha    santance ʔ     bai ʔ     manchi
\mph a– thippa       kancha             –ko   sakha    santan    –ce ʔbai      manchi
\mgl 2– grandfather youngest.one.male –GEN lineage offspring –ns DEM.PROX not
\lg C– C            N                  –C    C        N         –C  C         C
```

As you can see, sequence conditions look very similar to containment conditions. The only difference is that the sequence is written within square brackets. The sequence pattern consists of element patterns. Two consecutive element within a sequence mean that the respective elements must occur immediately after each other in order for the match to be successful. It is also possible to match an arbitrary element (similar to how . works for regular expressions) by using the special pattern ANY. For instance, if we require exactly one word to intervene in our last pattern:

```
crp[crp %% "@record
{
  CONTAINS [@word{$gw=~'^ba'} ANY @word{$gw=~'^ma'}]
}"]
```

```
Corpus with 2 entries (record) showing 1–2:

————————@1
\ref chintang_sahid.027
\eng That's it, there is not much (to tell).
\gw uti        ta    ni    baddhe na    manchi ni
\mph utti       ta    ni    baddhe na    manchi ni
\mgl that.much PTCL PTCL very    PTCL not     PTCL
\lg C/N        C     C     C       C     C       C

————————@2
\ref chintang_sahid.217
\eng No, not a lot.
\gw a    baddhe na ʔ  maha
\mph~a    baddhe na ʔ  maha
\mgl no   very    PTCL no
\lg C/N C       C     C
```

It is also possible to specify element repetition by putting $a : b$ before an element in a sequence pattern (a, b are numbers). A repetition will match a sequence of at least a and at most b elements which confirm to the element pattern.

```
crp[crp %% "@record
{
  CONTAINS [@word{$gw=~'^ba'} 1:3 ANY @word{$gw=~'^ma'}]
}"]
```

```
Corpus with 6 entries (record) showing 1-3:

----------@1
\ref chintang_sahid.027
\eng That's it, there is not much (to tell).
\gw uti       ta   ni   baddhe na   manchi ni
\mph utti      ta   ni   baddhe na   manchi ni
\mgl that.much PTCL PTCL very   PTCL not    PTCL
\lg C/N       C    C    C      C    C      C

----------@2
\ref chintang_sahid.124
\eng How old was your father when they killed him?
\gw ani kati     barsako~   huda buwalai     maro
\mph ani kati     barsa-ko~ huda buwa   -lai mar  -yo
\mgl and how.many year  -GEN be    father -DAT kill -PST
\lg N   C/N      C/N   -N   N    N      -N   N    -N

----------@3
\ref chintang_sahid.152
\eng Someone who's been born here...
\gw ba ? bai     ta   janma ŋ  lisago ?           mami
\mph ba ? bai     ta   janma   lis -a ŋ  -   -ko ?  mami
\mgl PRO DEM.PROX PTCL be.born be  -PST -1sA -NMLZ man
\lg C   C        C    N       C   -C   -C   -C    C
```

If $a = b$, we can omit the semicolon, e.g.:

```
crp[crp %% "@record
{
  CONTAINS [@word{$gw=~'^ba'} 3 ANY @word{$gw=~'^ma'}]
}"]
```

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

----------@1
\ref chintang_sahid.152
\eng Someone who's been born here...
\gw ba ? bai     ta   janma ŋ  lisago ?           mami
\mph ba ? bai     ta   janma   lis -a ŋ  -   -ko ?  mami
\mgl PRO DEM.PROX PTCL be.born be  -PST -1sA -NMLZ man
\lg C   C        C    N       C   -C   -C   -C    C

----------@2
\ref chintang_sahid.232
\eng Are there any (relatives of yours) near ŋBalakha?
\gw ani ŋbalakha tira cha ki ?manchi
\mph ani ŋbalakha tira cha ki manchi
\mgl and a_place  side be  or not
\lg N   N        N    N   N  C
```

```
————————@3
\ref budhohang_wal.048
\gw ?bagobe       banchuri    binchuri    numma          chakma ?       maha      kha
\mph bago –?pe    banchuri    binchuri    numd –ma       chakma ?       maha      kha
\mgl DEM –LOC.gm obstacle.n obstacle.n do.vt –INF.gm conscience.n no.interj PTCL.gm
\lg C     –C       C–RL        C–RL        C      –C       C–RL          C          C
```

Finally, a special repetition index $*$ means 'arbitrary number of times'. An a repetition $a : *$ will match at least a items (with no upper bounds). Using $*$ alone is equivalent to $0 : *$ (match zero or more items). The following query will find the records where a 'ba...' word precedes a 'ma...' word, with an arbitrary number of words between them:

```
crp[crp %% "@record
{
  CONTAINS [@word{$gw=~'^ba'} * ANY @word{$gw=~'^ma'}]
}"]
```

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

————————@1
\ref chintang_sahid.027
\eng That's it, there is not much (to tell).
\gw uti       ta    ni    baddhe na    manchi ni
\mph utti      ta    ni    baddhe na    manchi ni
\mgl that.much PTCL PTCL very    PTCL not     PTCL
\lg C/N       C     C     C       C     C       C

————————@2
\ref chintang_sahid.059
\eng ...the offsprings of my grandfather's youngest son do not live here.
\gw athippa         kanchako              sakha    santance ?     bai ?    manchi
\mph a– thippa       kancha             –ko  sakha    santan    –ce ?bai     manchi
\mgl 2– grandfather youngest.one.male –GEN lineage offspring –ns DEM.PROX not
\lg C– C            N                  –C   C        N         –C  C        C

————————@3
\ref chintang_sahid.124
\eng How old was your father when they killed him?
\gw ani kati      barsako~   huda buwalai      maro
\mph ani kati      barsa –ko~ huda buwa   –lai mar  –yo
\mgl and how.many year  –GEN be    father –DAT kill –PST
\lg N    C/N       C/N    –N    N     N       –N    N     –N
```

So far, the sequence patterns we examined are not anchored, which means that they will be matched independent of their position within the enclosing element. The symbol # allows us to anchor the sequence pattern on the boundary of the enclosing element. For instance, to find all words which end with a locative morpheme:

```
crp[crp %% "@word
{
  CONTAINS [@morpheme{$mgl=~'LOC'} #]
}"]
```

```
Corpus with 78 entries (word) showing 1–3:

————————@1
\gw ?bagobe
```

```
\mph bago –?pe
\mgl DEM –LOC
\lg C –C

———————@2
\gw ?sewabe
\mph sewa –?pe
\mgl service –LOC
\lg C/B –C

———————@3
\gw ?patibe
\mph pati –?pe
\mgl inn –LOC
\lg C/N –C
```

Here, #] means 'match the boundary'. It can be also used in the beginning of the sequence. The following example picks the words which start with a demonstrative marker:

```
crp[crp %% "@word
{
  CONTAINS [# @morpheme{$mgl=~'DEM'}]
}"]
```

```
Corpus with 160 entries (word) showing 1–3:

———————@1
\gw ˜baphaima
\mph ˜baphaima
\mgl DEM
\lg C–RL

———————@2
\gw ˜baphaima
\mph ˜baphaima
\mgl DEM
\lg C–RL

———————@3
\gw ba
\mph ba
\mgl DEM.PROX
\lg C
```

Of course, both anchors can be combined. Here, we find all words which start with a DEM marker and end with a LOC marker:

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

```
Corpus with 15 entries (word) showing 1–3:
```

```
———————@1
\gw ʔbagobe
\mph bago –ʔpe
\mgl DEM  –LOC
\lg C     –C

———————@2
\gw ŋʔhugoi
\mph hun –ko  –ʔi
\mgl DEM –GEN –LOC
\lg C    –C   –C

———————@3
\gw ŋʔhugoi
\mph hun –ko  –ʔi
\mgl DEM –GEN –LOC
\lg C    –C   –C
```

For a detailed reference of the query language, see Appendix ??.

4.2 Using corpus index objects to combine query results

As already mentioned, corpus query in ToolboxSearch return a corpus index object (see page ??). Because the index objects can be easily combined via set operations, we can use them to carry out complex searches by combining results of simple queries. Consider the following example:

```
# find all records with a demonstrative
i.dem ← crp %% "@record{CONTAINS @morpheme{$mgl =~ 'DEM'}}"
# find all records which contain at least one
i.v ← crp %% "@record{CONTAINS @morpheme{$mgl =~ '\\.(vi|vt|v2)$'}}"
# find all records which contain more then one verb stem
i.complex ← crp %% "@record{
  CONTAINS
  [
    @morpheme{$mgl =~ '\\.(vi|vt|v2)$'}
    * ANY
    @morpheme{$mgl =~ '\\.(vi|vt|v2)$'}
  ]}"
# pick only demonstratives within simple sentences
ri = (i.dem – i.complex)*i.v
crp[ri]
```

```
Corpus with 2 entries (record) showing 1–2:

———————@1
\ref budhohang_wal.048
\gw ʔbagobe       banchuri   binchuri   numma          chakma ʔ      maha     kha
\mph bago –ʔpe    banchuri   binchuri   numd  –ma      chakma ʔ      maha     kha
\mgl DEM  –LOC.gm obstacle.n obstacle.n do.vt –INF.gm conscience.n no.interj PTCL.gm
\lg C     –C      C–RL       C–RL       C     –C      C–RL          C        C

———————@2
\ref budhohang_wal.082
\gw ˜ha ʔ           bagobe      sabai   kuro    sima    lapma ʔ          maha
\mph ˜ha            bago –ʔpe   sab     kura    sima    lapt     –ma ʔ   maha
\mgl FILLER.interj DEM  –LOC.gm all.adv thing.n death.n catch.vt –INF.gm no.interj
\lg C/N            C    –C      N       N       C       C        –C      C
```

Our goal is to find all records with demonstrative, but only simple clauses (one verb stem per record). Doing this as one single query is complicated, so we can divide the query into a number of simpler ones. Here, i.dem is the result of the query which searches for a DEM marker. The i.v and i.complex are simple and complex records, respective. The `$mgl =~ '\\.(vi|vt|v2)$'` condition matches the end of the gloss (which is an inline part of speech tag) agains possible verb annotations. Finally, we combine the queries by omitting all complex records from the DEM-records and limiting the result to the records which are also simple.

Remember that we can also store the results of such queries in a Toolbox file for later processing:

```
writeToolbox(crp[ri], 'simple_dem.txt')
```

A Query language reference

This appendix is the reference to the ToolboxSearch query language. The query language is described in form of simple rewriting grammar rules. Language symbols delimited by an underscore denote non-terminals. A quotation mark before a symbol means that the occurrence of the symbol is optional in the rule.

A core element of the query language is the element pattern. A valid element pattern is also a valid query.

```
_ELEM_ ::= @level
_ELEM_ ::= @level{_CONDITIONS_}
```

Here, `level` is the level of the element and `_CONDITIONS_` is the list of conditions which the matched element must satisfy. The condition part can be omitted, in this case the element pattern will match any element of the respective level.

```
_CONDITIONS_ ::= _CONDITION_
_CONDITIONS_ ::= (_CONDITIONS_)
_CONDITIONS_ ::= NOT _CONDITIONS_
_CONDITIONS_ ::= _CONDITIONS_ AND _CONDITIONS_
_CONDITIONS_ ::= _CONDITIONS_ OR _CONDITIONS_
```

A condition can be one of: annotation condition, containment condition or sequence pattern condition.

```
_CONDITION_ ::= _ANN_COND_
_CONDITION_ ::= _CONTAINS_COND_
_CONDITION_ ::= _CONTAINS_SEQUENCE_COND_
```

Annotation condition match contents of an annotation tier.

```
_ANN_COND_ ::= $name _OP_ 'val'
_OP        ::= ==
_OP        ::= ==
_OP        ::= =~
_OP        ::= !~
```

Here, `name` is the name of an annotation tier and `val` is a string value which the contents of the annotation will be matched against. The match operator `_OP_` is one of:

== exact match

!= inequality

= match regular expression (case-insensitive)

!= do not match regular expression (case-insensitive)

For regular expression syntax, see R help on ?regex.

A containment condition matches a sub-element.

```
_ANN_COND_ ::= CONTAINS _ELEM_
```

Here, the nested `_ELEM_` describes the element which must be contained in the enclosed element.

Finally, a sequence pattern condition matches a sequence of sub-elements.

```
_CONTAINS_SEQUENCE_COND_ ::= CONTAINS [?#_SEQ_PATTERN_?#]
```

The anchor character # tells ToolboxSearch to match the boundary of the enclosing element (start, end, or both). Otherwise, the sequence is matched anywhere within the enclosing element.

```
_SEQ_PATTERN_ ::= _SEQ_ITEM_ ?_SEQ_PATTERN_
_SEQ_ITEM_    ::= ?_REP_ ANY
_SEQ_ITEM_    ::= ?_REP_ _ELEM_
```

A sequence pattern is a list of sequence items. Each sequence item can match a particular element pattern or any element (via special word `ANY`). Each sequence item is optionally prefixed by a repetition index.

```
_REP_ ::= num : num
_REP_ ::= num
_REP_ ::= *
_REP_ ::= num : *
```

Here, `num` is a non-negative integer number. A repetition index in form $a : b$ will match at least a and at most b items, a will match exactly a items, $a : *$ will match a or more items and $*$ will match zero or more items.

B Tips and tricks

This section contains small practical examples of how ToolboxSearch can be used.

Find all words with exactly one morpheme

```
crp[crp %% "@word{CONTAINS [# ANY #]}"]
```

```
Corpus with 2486 entries (word) showing 1-3:

----------@1
\gw he
\mph he
\mgl ADDR
\lg N

----------@2
\gw parmeswora
\mph śparamevara
\mgl Lord
\lg N

----------@3
\gw sabai
\mph sab
\mgl all
\lg N
```

To compile a list of such unique words we must do some R magic:

```
words <- as.data.frame(crp[crp %% "@word{CONTAINS [# ANY #]}"])
words <- unique(words$gw)
head(words)
```

```
[1] "he"         "parmeswora" "sabai"      "caine"      "ke"
[6] "thippe"
```

```
length(words)
```

```
[1] 715
```