05 Querying Annotation Structures

First of all, let us once again create a temporary EMU-SDMS-database:

```
# load package
library(emuR)
# create demo data in directory
# provided by tempdir()
create_emuRdemoData(dir = tempdir())
# create path to demo database
path2ae = file.path(tempdir(), "emuR_demoData", "ae_emuDB")
# load database
ae = load_emuDB(path2ae, verbose = F)
summary(ae)
```

In the link definitions, we can see a very rich annotation structure. This will allow us to perform even more complicated queries. Let us have a look at the annotation structure:

```
serve(ae)
```

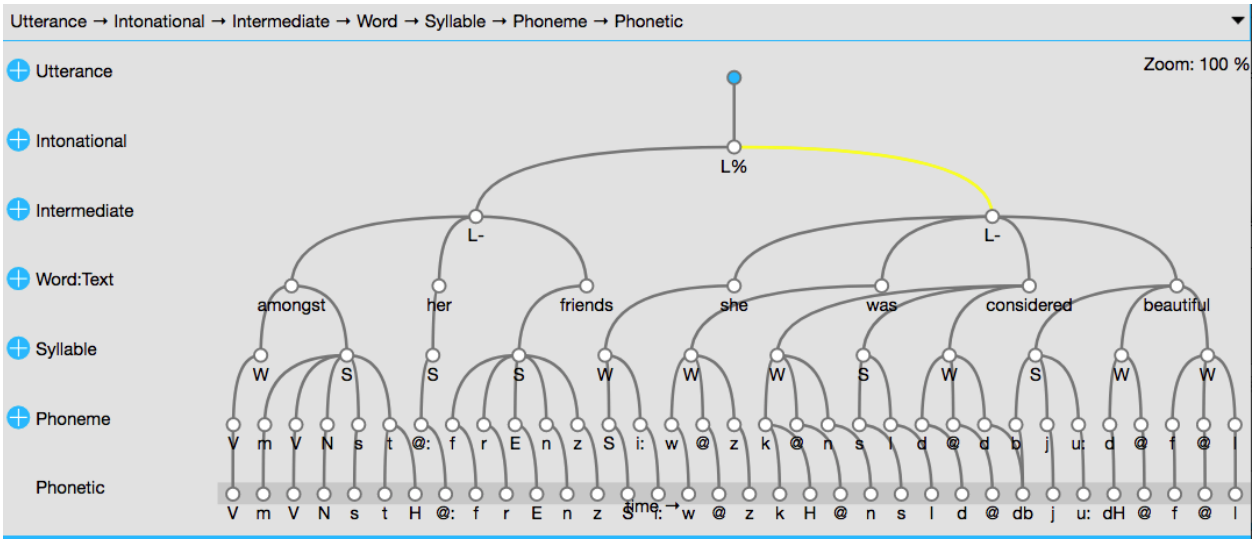

Figure 1: Figure 1: Hierarchy of the first utterance of the database *ae*

However, let us start with very basic queries. The command for conducting queries is simply `query`, called with at least two arguments, `emuDBhandle` and `query`:

```
query(emuDBhandle = ae,query = "Phonetic==V")
```

```
## segment  list from database:  ae
## query was:  Phonetic==V
##   labels    start      end session   bundle    level    type
## 1      V  187.425  256.925    0000 msajc003 Phonetic SEGMENT
## 2      V  340.175  426.675    0000 msajc003 Phonetic SEGMENT
## 3      V 1943.175 2037.425    0000 msajc057 Phonetic SEGMENT
```

We could have been less verbose instead to achieve the very same result, e.g. by typing:

```
query(ae,"Phonetic==V")
```

The expression "Phonetic==V" is a legal expression in the EMU Query Language (EQL) and could be translated into "which labels in the level Phonetic equal the label"V"". However, before we start to learn the EQL in detail, we will discuss other parameters of the query-command, and learn the structure of a result of the query-command.

```
help(query)
```

Arguments to query()

```
query(emuDBhandle, +
      query, +
      sessionPattern = ".*", +
      bundlePattern = ".*", +
      queryLang = "EQL2", +
      timeRefSegmentLevel = NULL, +
      resultType = NULL, +
      calcTimes = TRUE, verbose = FALSE)
```

Argument	Meaning
emuDBhandle	emuDB handle object
query	string (see vignette("EQL"))
sessionPattern	A regular expression pattern matching session names to be searched from the database
bundlePattern	A regular expression pattern matching bundle names to be searched from the database
queryLang	query language used for evaluating the query string
timeRefSegmentLevel	set time segment level from which to derive time information. It is only necessary to set this parameter if the queried parent level is of type ITEM and more than one child level contains time information.
resultType	type (class name) of result
calcTimes	calculate times for resulting segments (results in NA values for start and end times in emuseg/emuRsegs). As it can be very computationally expensive to calculate the times for large nested hierarchies it can be turned off via this boolian parameter.
verbose	be verbose. Set this to TRUE if you wish to choose which path to traverse on intersecting hierarchies. If set to FALSE (the default) all paths will be traversed (= legacy EMU behaviour).

We could e.g. restrict ourselves to only one sessionPattern (but there is only one in ae) or to only one bundlePattern; the latter case may be a not-too-unusual use case:

```
query(emuDBhandle = ae,query = "Phonetic==V",bundlePattern = "msajc003")
```

```
## segment  list from database:  ae
## query was:  Phonetic==V
##   labels   start     end session   bundle    level    type
## 1      V 187.425 256.925    0000 msajc003 Phonetic SEGMENT
## 2      V 340.175 426.675    0000 msajc003 Phonetic SEGMENT
```

Results of a query: `emuRsegs`

An emuR segment list is a list of segment descriptors. Each segment descriptor describes a sequence of annotation elements. The list is usually a result of an emuDB query using function `query`.

An `emuRsegs` object is an attributed data.frame, with one row per segment descriptor.

Data frame columns

- *labels*: sequenced labels of segment concatenated by '->'
- *start*: onset time in milliseconds
- *end*: offset time in milliseconds
- *session*: session name
- *bundle*: bundle name
- *level*: level name
- *type*: type of "segment" row: `ITEM`: symbolic item, `EVENT`: event item, `SEGMENT`: segment

Additional **hidden columns**

- *utts*: utterance name (for compatibility to emusegs class)
- *db_uuid*: UUID of emuDB
- *startItemID*: item ID of first element of sequence
- *endItemID*: item ID of last element of sequence
- *sampleStart*: start sample position
- *sampleEnd*: end sample position
- *sampleRate*: sample rate

Attributes

- *database*: name of emuDB
- *query*: Query string
- *type*: type (`SEGMENT` or `EVENT`, but not `ITEM`) (for compatibility to `emusegs` class)

Assign our query above to an object that we call `V_first_utt`:

```
(V_first_utt = query(emuDBhandle = ae,query = "Phonetic==V",bundlePattern = "msajc003"))
```

```
## segment  list from database:  ae
## query was:  Phonetic==V
##   labels   start     end session   bundle    level    type
## 1      V 187.425 256.925    0000 msajc003 Phonetic SEGMENT
## 2      V 340.175 426.675    0000 msajc003 Phonetic SEGMENT
```

```
class(V_first_utt)
```

```
## [1] "emuRsegs"   "emusegs"    "data.frame"
```

```
summary(V_first_utt)
```

```
## segment  list from database:  ae
## query was:  Phonetic==V
##  with 2 segments
##
```

```
## Segment distribution:
##
## V
## 2
```

Although every column can easily be called with the $-operator, there are some commands in emuR doing the very same job, e.g. the commands `label()`, `start()`, `end()`:

```
#Get labels:
#either
V_first_utt$labels
```

```
## [1] "V" "V"
```

```
#or
label(V_first_utt)
```

```
## [1] "V" "V"
```

Other examples:

```
#Get start times:
start(V_first_utt)
```

```
## [1] 187.425 340.175
```

```
#Get end times:
end(V_first_utt)
```

```
## [1] 256.925 426.675
```

Other more complex commands are simply a shorter way of doing rather simple things:

```
#Get durations:
dur(V_first_utt)
```

```
## [1] 69.5 86.5
```

```
#which is, of course, the same as:
end(V_first_utt) - start(V_first_utt)
```

```
## [1] 69.5 86.5
```

```
#We can also use the $-operator to access the columns of V_first_utt, because it is a data.frame
#see chapter 07
V_first_utt$end - V_first_utt$start
```

```
## [1] 69.5 86.5
```

Calculated times You might recall from chapter 04, that start and end times are stored internally as `sampleStart` and `sampleDur`. In the very same annot.json-file, there is also information about the sampling frequency:

```
{
    "name": "msajc003",
    "annotates": "msajc003.wav",
    "sampleRate": 20000,
...
        {
            "name": "Phonetic",
            "type": "SEGMENT",
            "items": [
```

```
        {
            "id": 147,
            "sampleStart": 3749,
            "sampleDur": 1389,
            "labels": [
                {
                    "name": "Phonetic",
                    "value": "V"
                }
            ]
        },
```

This means, that out first "V" start at the 3749th sample of 20000 samples per second. We can calculate:

```
3749/20000
```

```
## [1] 0.18745
```

... i.e., our first V starts a 187.45 milliseconds. Compare this to

```
start(V_first_utt)[1]
```

```
## [1] 187.425
```

In other words, `$start` and `$end` report times in milliseconds. As we have mentioned earlier, `$sampleStart` and `$sampleDur` are available in the `emuRsegs`-object, but are hidden. Nevertheless, we could call them with

```
V_first_utt$sampleStart
```

```
## NULL
```

```
V_first_utt$sampleDur
```

```
## NULL
```

As can be seen in the annot.json-file, our "V" on level "Phonetic" is of type `SEGMENT`, i.e. of a time-aligned type. This information can also be found in

```
V_first_utt
```

```
## segment  list from database:  ae
## query was:  Phonetic==V
##   labels   start     end session   bundle    level    type
## 1      V 187.425 256.925    0000 msajc003 Phonetic SEGMENT
## 2      V 340.175 426.675    0000 msajc003 Phonetic SEGMENT
```

What happens, if we were looking for a timeless `ITEM`?

```
(amongst=query(emuDBhandle = ae,query = "Text==amongst",bundlePattern = "msajc003"))
```

```
## segment  list from database:  ae
## query was:  Text==amongst
##    labels   start     end session   bundle level type
## 1 amongst 187.425 674.175    0000 msajc003  Text ITEM
```

We can see in `$type`, that the type of "amongst" is `ITEM`. However, we can see non-empty `$start` and `$end` columns, and

```
attributes(amongst)$type
```

```
## [1] "segment"
```

says, that its type was "segment" (for compatibility with the former emusegs format). So, although "amongst" is actually a timeless `ITEM` ...

```
        {
            "name": "Word",
            "type": "ITEM",
            "items": [
                {
                    "id": 2,
                    "labels": [
                        {
                            "name": "Word",
                            "value": "C"
                        },
                        {
                            "name": "Accent",
                            "value": "S"
                        },
                        {
                            "name": "Text",
                            "value": "amongst"
                        }
                    ]
                },
```

... times can be derived from this item's first and last time-aligned segments (our first "V" above is the first segment, and amongst end with a "t", that is aspirated (="H" on the Phonetic level, see Figure 1))

```
start(query(emuDBhandle = ae,query = "Text==amongst",bundlePattern = "msajc003"))
```

```
## [1] 187.425
```

```
start(query(emuDBhandle = ae,query = "Phonetic==V",bundlePattern = "msajc003")[1,])
```

```
## [1] 187.425
```

```
end(query(emuDBhandle = ae,query = "Text==amongst",bundlePattern = "msajc003"))
```

```
## [1] 674.175
```

```
end(query(emuDBhandle = ae,query = "Phonetic==H",bundlePattern = "msajc003")[1,])
```

```
## [1] 674.175
```

This, of course, will only work if Text (one attribute of Word) and Phonetic levels are linked (and they are, see also Figure 1):

```
list_linkDefinitions(ae)
```

```
##           type superlevelName sublevelName
## 1  ONE_TO_MANY      Utterance Intonational
## 2  ONE_TO_MANY   Intonational Intermediate
## 3  ONE_TO_MANY   Intermediate         Word
## 4  ONE_TO_MANY           Word     Syllable
## 5  ONE_TO_MANY       Syllable      Phoneme
## 6 MANY_TO_MANY        Phoneme     Phonetic
## 7  ONE_TO_MANY       Syllable         Tone
## 8  ONE_TO_MANY   Intonational         Foot
## 9  ONE_TO_MANY           Foot     Syllable
```

If the ITEM we are interested in was linked to several time-aligned segments, we would have to use query's parameter timeRefSegmentLevel to choose the segment level from which query derives time information.

Preview to requery_hier and requery_seq

We already know that the first "V" in our results belongs to (and is linked to) the word "amongst". By which words are the other "V"s dominated, then? We could find our by a hierarchical re-query:

```
#find all "V"-labels in `ae`
(V=query(emuDBhandle = ae,query = "Phonetic==V"))
```

```
## segment  list from database:  ae
## query was:  Phonetic==V
##   labels    start      end session   bundle    level    type
## 1      V  187.425  256.925    0000 msajc003 Phonetic SEGMENT
## 2      V  340.175  426.675    0000 msajc003 Phonetic SEGMENT
## 3      V 1943.175 2037.425    0000 msajc057 Phonetic SEGMENT
```

Now put this segment list into requery_hier() and look for the linked ITEM in Word:Text:

```
requery_hier(emuDBhandle = ae,seglist = V,level = "Text")
```

```
## segment  list from database:  ae
## query was:  FROM REQUERY
##      labels    start      end session   bundle level type
## 1   amongst  187.425  674.175    0000 msajc003  Text ITEM
## 2   amongst  187.425  674.175    0000 msajc003  Text ITEM
## 3 customers 1824.425 2367.775    0000 msajc057  Text ITEM
```

Your result will be the ITEM labels and calculated times (for the corresponding words).

You could also wish to know what "V"s contexts are, e.g. the subsequent segments. We use the sequential structure of the database, and the command requery_seq():

```
requery_seq(emuDBhandle = ae,seglist = V,offset = 1)
```

```
## segment  list from database:  ae
## query was:  FROM REQUERY
##   labels    start      end session   bundle    level    type
## 1      m  256.925  340.175    0000 msajc003 Phonetic SEGMENT
## 2      N  426.675  483.425    0000 msajc003 Phonetic SEGMENT
## 3      s 2037.425 2085.175    0000 msajc057 Phonetic SEGMENT
```

We will discuss both commands more extensively later in the seminar, but wanted to show that it is possible to use the annotation structure and a given segment list to retrieve additional information afterwards. We could use both commands to express more complex queries: e.g. we could look for all "V" within the word "amongst" by querying "V", then requery all linked words, and then deletin all "V" that are not linked to "amongst". However, this would be rather cumbersome. A much easier way to conduct more complicated queries is the use of all possibilities of emuR's query language EQL within the command query.

The Emu Query Language EQL

To learn about the functionality of the EQL, you can always type

```
vignette("EQL")
```

As we have seen above, any query must be placed within " ". You minimally have to give a level, and some sort of representation for a label (this may be a regular expression), unless you do not use one of the `position` and `count` functions.

Single argument queries

Equality/inequality/matching/non-matching

In the examples above, we had looked for the equality of the labels to "V" on the level "Phonetic" (in the database `ae`):

```
query(emuDBhandle = ae, query = "Phonetic == V")
```

```
## segment  list from database:  ae
## query was:  Phonetic == V
##   labels    start      end session   bundle    level    type
## 1      V  187.425  256.925    0000 msajc003 Phonetic SEGMENT
## 2      V  340.175  426.675    0000 msajc003 Phonetic SEGMENT
## 3      V 1943.175 2037.425    0000 msajc057 Phonetic SEGMENT
```

So "==" is the equality operator. For backward compatibility, a single "=" is also allowed (but we ask you to prefer "==" instead):

```
query(emuDBhandle = ae, query = "Phonetic = V")
```

```
## segment  list from database:  ae
## query was:  Phonetic = V
##   labels    start      end session   bundle    level    type
## 1      V  187.425  256.925    0000 msajc003 Phonetic SEGMENT
## 2      V  340.175  426.675    0000 msajc003 Phonetic SEGMENT
## 3      V 1943.175 2037.425    0000 msajc057 Phonetic SEGMENT
```

We can also search everything *except* "V" by the use of `!=`

```
query(emuDBhandle = ae, query = "Phonetic != V")
```

(We do not show the resulting segment list, because it is very long.) So one way to get 'everything' would be to query something that is not in your database, like "xyz". However, there is a much better way: Using regular expressions. To use these, you have to type "=~":

```
Everything1 = query(emuDBhandle = ae, query = "Phonetic != xyz")
Everything2 = query(emuDBhandle = ae, query = "Phonetic =~ .*")
any(Everything1 != Everything2) # should result in FALSE if both are equal everywhere
```

```
## [1] FALSE
```

You can also negate the latter operator by "!~". A not very useful example would be:

```
(Nothing = query(emuDBhandle = ae, query = "Phonetic !~ .*"))
```

```
## segment  list from database:  ae
## query was:  Phonetic !~ .*
## [1] labels  start   end     session bundle  level   type
## <0 rows> (or 0-length row.names)
```

A more interesting example would be:

```
# What is the query to retrieve all ITEMs in the "Text" level that don't begin with 'a'?
query(emuDBhandle = ae, query = "Text !~ a.*")
```

So, there are four similar operators, two for equality matching, and two for inequalitiy:

Symbol	Meaning
==	equality
=~	regular expression matching
!=	inequality
!~	regular expression non-matching

The OR operator

Use "|" to look for one label and another one(s), e.g.

```
query(emuDBhandle = ae, query = "Phonetic == m|n")
```

```
## segment  list from database:  ae
## query was:  Phonetic == m|n
##    labels    start      end session   bundle    level    type
## 1       m  256.925  340.175    0000 msajc003 Phonetic SEGMENT
## 2       n 1031.925 1195.925    0000 msajc003 Phonetic SEGMENT
## 3       n 1741.425 1791.425    0000 msajc003 Phonetic SEGMENT
## 4       n 1515.475 1554.475    0000 msajc010 Phonetic SEGMENT
## 5       n 2430.975 2528.475    0000 msajc010 Phonetic SEGMENT
## 6       n  894.975 1022.975    0000 msajc012 Phonetic SEGMENT
## 7       m 1490.425 1564.975    0000 msajc012 Phonetic SEGMENT
## 8       n 2402.275 2474.875    0000 msajc012 Phonetic SEGMENT
## 9       m  496.575  558.575    0000 msajc015 Phonetic SEGMENT
## 10      n 2226.575 2271.075    0000 msajc015 Phonetic SEGMENT
## 11      n 3046.125 3067.675    0000 msajc015 Phonetic SEGMENT
## 12      m 1587.175 1655.675    0000 msajc022 Phonetic SEGMENT
## 13      m  819.025  902.925    0000 msajc023 Phonetic SEGMENT
## 14      n 1434.775 1495.275    0000 msajc023 Phonetic SEGMENT
## 15      n 1774.925 1833.925    0000 msajc023 Phonetic SEGMENT
## 16      n  508.675  543.975    0000 msajc057 Phonetic SEGMENT
## 17      m 1629.675 1709.175    0000 msajc057 Phonetic SEGMENT
## 18      m 2173.425 2233.425    0000 msajc057 Phonetic SEGMENT
## 19      n 2447.675 2480.425    0000 msajc057 Phonetic SEGMENT
```

You can expand this as well:

```
mnN = query(emuDBhandle = ae, query = "Phonetic == m|n|N")
summary(mnN)
```

```
## segment  list from database:  ae
## query was:  Phonetic == m|n|N
##  with 23 segments
##
## Segment distribution:
##
##  m  n  N
##  7 12  4
```

Complex queries

Sequencial and dominance queries

Bracketing

In all hierarchical queries, bracketing with "[" and "]" is required to structure your query. In simple queries, however, brackets are optional.

```
mnN = query(emuDBhandle = ae, query = "[Phonetic == m|n|N]")
summary(mnN)
```

```
## segment  list from database:  ae
## query was:  [Phonetic == m|n|N]
##  with 23 segments
##
## Segment distribution:
##
##  m  n  N
##  7 12  4
```

However, this sequential query would fail, because of missing brackets:

```
query(ae, "Phonetic == V -> Phonetic == m")
```

Sequential queries

Use the "->" operator to find sequences of segments:

```
query(ae, "[Phonetic == V -> Phonetic == m]")
```

```
## segment  list from database:  ae
## query was:  [Phonetic == V -> Phonetic == m]
##   labels   start     end session   bundle    level    type
## 1   V->m 187.425 340.175    0000 msajc003 Phonetic SEGMENT
```

Note: all row entries in the resulting segment list have the start time of "V", the end time of "m" and their labels will be "V->m". Change this with the so-called `result modifier` hash tag "#":

```
query(ae, "[#Phonetic == V -> Phonetic == m]")
```

```
## segment  list from database:  ae
## query was:  [#Phonetic == V -> Phonetic == m]
##   labels   start     end session   bundle    level    type
## 1      V 187.425 256.925    0000 msajc003 Phonetic SEGMENT
```

```
query(ae, "[Phonetic == V -> #Phonetic == m]")
```

```
## segment  list from database:  ae
## query was:  [Phonetic == V -> #Phonetic == m]
##   labels   start     end session   bundle    level    type
## 1      m 256.925 340.175    0000 msajc003 Phonetic SEGMENT
```

Keep in mind that only one hash tag per query is allowed.

You can search sequences of sequences, however, you have to use bracketing; otherwise, you get an error

```
query(ae, "[Phonetic == @ -> Phonetic == n  -> Phonetic == s]")
```

The correct code would be as follows:

```
query(ae, "[[Phonetic == @ -> Phonetic == n ] -> Phonetic == s]")
```

```
## segment  list from database:  ae
## query was:  [[Phonetic == @ -> Phonetic == n ] -> Phonetic == s]
##    labels    start      end session   bundle    level    type
## 1 @->n->s 1715.425 1893.175    0000 msajc003 Phonetic SEGMENT
## 2 @->n->s 2382.475 2753.975    0000 msajc010 Phonetic SEGMENT
## 3 @->n->s 2200.875 2408.575    0000 msajc015 Phonetic SEGMENT
```

```
## What is the query to retrieve all sequences of ITEMs containing labels "offer" followed by two arbit
query(ae, "[[[Text == offer -> Text =~ .*] -> Text =~ .* ] -> Text == resistance]")
```

```
## segment  list from database:  ae
## query was:  [[[Text == offer -> Text =~ .*] -> Text =~ .* ] -> Text == resistance]
##                             labels    start      end session   bundle level
## 1 offer->any->further->resistance 1957.775 2753.975    0000 msajc010  Text
##   type
## 1 ITEM
```

Domination queries

Use the operator "^" for all queries, in which two linked levels are involved.

```
## What is the query to retrieve all ITEMs containing the label "p" in the "Phoneme" level that occur i
query(ae, "[Phoneme == p ^ Syllable == S]")
```

```
## segment  list from database:  ae
## query was:  [Phoneme == p ^ Syllable == S]
##   labels    start      end session   bundle   level type
## 1      p  558.575  639.575    0000 msajc015 Phoneme ITEM
## 2      p 1655.675 1698.675    0000 msajc022 Phoneme ITEM
## 3      p  863.675  970.425    0000 msajc057 Phoneme ITEM
```

However, the operator is *not* directional; although "Syllable" dominates "Phoneme", you could have asked

```
query(ae, "[Syllable == S ^ #Phoneme == p]")
```

```
## segment  list from database:  ae
## query was:  [Syllable == S ^ #Phoneme == p]
##   labels    start      end session   bundle   level type
## 1      p  558.575  639.575    0000 msajc015 Phoneme ITEM
## 2      p 1655.675 1698.675    0000 msajc022 Phoneme ITEM
## 3      p  863.675  970.425    0000 msajc057 Phoneme ITEM
```

So, "^" should not be translated with "is dominated by". However, you have to use the hash tag in order to get labels and times of the Phoneme level here. You can leave out the hash tag if the level you are interested in is the first one in your question.

You can query multiple dominations, however, like in the sequencing case, you have to use brackets:

```
## What is the query to retrieve all ITEMs on the "Phonetic" level that are part of a strong syllable (
query(ae, "[[#Phonetic =~ .* ^ Syllable == S] ^ Text == amongst | beautiful]")
```

```
## segment  list from database:  ae
## query was:  [[#Phonetic =~ .* ^ Syllable == S] ^ Text == amongst | beautiful]
##   labels    start      end session   bundle    level    type
## 1      m  256.925  340.175    0000 msajc003 Phonetic SEGMENT
## 2      V  340.175  426.675    0000 msajc003 Phonetic SEGMENT
```

```
## 3      N  426.675  483.425    0000 msajc003 Phonetic SEGMENT
## 4      s  483.425  566.925    0000 msajc003 Phonetic SEGMENT
## 5      t  566.925  596.675    0000 msajc003 Phonetic SEGMENT
## 6      H  596.675  674.175    0000 msajc003 Phonetic SEGMENT
## 7     db 2033.675 2150.175    0000 msajc003 Phonetic SEGMENT
## 8      j 2150.175 2211.175    0000 msajc003 Phonetic SEGMENT
## 9     u: 2211.175 2283.675    0000 msajc003 Phonetic SEGMENT
```

```
# same as
query(ae, "[[Phonetic =~ .* ^ Syllable == S] ^ Text == amongst | beautiful]")
```

```
## to get the "Text"-items instead, use
query(ae, "[[Phonetic =~ .* ^ Syllable == S] ^ #Text == amongst | beautiful]")
```

```
## segment  list from database:  ae
## query was:  [[Phonetic =~ .* ^ Syllable == S] ^ #Text == amongst | beautiful]
##      labels    start      end session   bundle level type
## 1   amongst  187.425  674.175    0000 msajc003  Text ITEM
## 2 beautiful 2033.675 2604.425    0000 msajc003  Text ITEM
```

Functions

The are three **position** and one **count** functions. As the latter function results in a number, queries involve a comparison with a number (by using one of "==", "!=", ">", ">=", "<", "<=", see below); The result of the position functions is logical; we therefore ask, whether a certain condition is `TRUE` or `FALSE`.

Position functions

There are three position functions, `Start()`, `Medial()`, and `End()`. Example queries are:

```
## What is the query to retrieve all word-initial syllables?
query(ae, "[Start(Text, Syllable) == TRUE]", bundlePattern = "msajc003")
```

```
## segment  list from database:  ae
## query was:  [Start(Text, Syllable) == TRUE]
##   labels    start      end session   bundle    level type
## 1      W  187.425  256.925    0000 msajc003 Syllable ITEM
## 2      S  674.175  739.925    0000 msajc003 Syllable ITEM
## 3      S  739.925 1289.425    0000 msajc003 Syllable ITEM
## 4      W 1289.425 1463.175    0000 msajc003 Syllable ITEM
## 5      W 1463.175 1634.425    0000 msajc003 Syllable ITEM
## 6      W 1634.425 1791.425    0000 msajc003 Syllable ITEM
## 7      S 2033.675 2283.675    0000 msajc003 Syllable ITEM
```

```
## be careful with the direction of the arguments; the following gives a result, which makes no sense (
query(ae, "[Start(Syllable, Text) == TRUE]", bundlePattern = "msajc003")
```

```
## segment  list from database:  ae
## query was:  [Start(Syllable, Text) == TRUE]
##       labels    start      end session   bundle level type
## 1    amongst  187.425  674.175    0000 msajc003  Text ITEM
## 2    amongst  187.425  674.175    0000 msajc003  Text ITEM
## 3        her  674.175  739.925    0000 msajc003  Text ITEM
## 4    friends  739.925 1289.425    0000 msajc003  Text ITEM
## 5        she 1289.425 1463.175    0000 msajc003  Text ITEM
## 6        was 1463.175 1634.425    0000 msajc003  Text ITEM
```

```
## 7  considered 1634.425 2150.175    0000 msajc003  Text ITEM
## 8  considered 1634.425 2150.175    0000 msajc003  Text ITEM
## 9  considered 1634.425 2150.175    0000 msajc003  Text ITEM
## 10  beautiful 2033.675 2604.425    0000 msajc003  Text ITEM
## 11  beautiful 2033.675 2604.425    0000 msajc003  Text ITEM
## 12  beautiful 2033.675 2604.425    0000 msajc003  Text ITEM
```

Examples for Medial() and End() are:

```
## What is the query to retrieve all word-medial syllables?
query(ae, "[Medial(Word, Syllable) == TRUE]", bundlePattern = "msajc003")
```

```
## segment  list from database:  ae
## query was:  [Medial(Word, Syllable) == TRUE]
##   labels    start      end session   bundle    level type
## 1      S 1791.425 1945.425    0000 msajc003 Syllable ITEM
## 2      W 2283.675 2361.925    0000 msajc003 Syllable ITEM
```

```
## What is the query to retrieve all word-final syllables?
query(ae, "[End(Word, Syllable) == TRUE]", bundlePattern = "msajc003")
```

```
## segment  list from database:  ae
## query was:  [End(Word, Syllable) == TRUE]
##   labels    start      end session   bundle    level type
## 1      S  256.925  674.175    0000 msajc003 Syllable ITEM
## 2      S  674.175  739.925    0000 msajc003 Syllable ITEM
## 3      S  739.925 1289.425    0000 msajc003 Syllable ITEM
## 4      W 1289.425 1463.175    0000 msajc003 Syllable ITEM
## 5      W 1463.175 1634.425    0000 msajc003 Syllable ITEM
## 6      W 1945.425 2150.175    0000 msajc003 Syllable ITEM
## 7      W 2361.925 2604.425    0000 msajc003 Syllable ITEM
```

Everything not being first or last element is medial:

```
query(ae, "[Medial(Word, Phoneme) == TRUE]", bundlePattern = "msajc003")
```

```
## segment  list from database:  ae
## query was:  [Medial(Word, Phoneme) == TRUE]
##    labels    start      end session   bundle   level type
## 1       m  256.925  340.175    0000 msajc003 Phoneme ITEM
## 2       V  340.175  426.675    0000 msajc003 Phoneme ITEM
## 3       N  426.675  483.425    0000 msajc003 Phoneme ITEM
## 4       s  483.425  566.925    0000 msajc003 Phoneme ITEM
## 5       r  892.675  949.925    0000 msajc003 Phoneme ITEM
## 6       E  949.925 1031.925    0000 msajc003 Phoneme ITEM
## 7       n 1031.925 1195.925    0000 msajc003 Phoneme ITEM
## 8       @ 1506.175 1548.425    0000 msajc003 Phoneme ITEM
## 9       @ 1715.425 1741.425    0000 msajc003 Phoneme ITEM
## 10      n 1741.425 1791.425    0000 msajc003 Phoneme ITEM
## 11      s 1791.425 1893.175    0000 msajc003 Phoneme ITEM
## 12      I 1893.175 1945.425    0000 msajc003 Phoneme ITEM
## 13      d 1945.425 1966.675    0000 msajc003 Phoneme ITEM
## 14      @ 1966.675 2033.675    0000 msajc003 Phoneme ITEM
## 15      j 2150.175 2211.175    0000 msajc003 Phoneme ITEM
## 16     u: 2211.175 2283.675    0000 msajc003 Phoneme ITEM
## 17      d 2283.675 2302.925    0000 msajc003 Phoneme ITEM
## 18      @ 2302.925 2361.925    0000 msajc003 Phoneme ITEM
```

```
## 19      f 2361.925 2447.425    0000 msajc003 Phoneme ITEM
## 20      @ 2447.425 2506.275    0000 msajc003 Phoneme ITEM
```

Count function

The count function's name is `Num()`. `Num(x,y)` counts, how many y are in x. You can therefore ask things like the following:

```
## What is the query to retrieve all words that contain two syllables?
query(ae, "[Num(Text, Syllable) == 2]", bundlePattern = "msajc003")
```

```
## segment  list from database:  ae
## query was:  [Num(Text, Syllable) == 2]
##    labels   start     end session   bundle level type
## 1 amongst 187.425 674.175    0000 msajc003  Text ITEM
```

```
## What is the query to retrieve all syllables that contain more than four phonemes?
query(ae, "[Num(Syllable, Phoneme) > 4]", bundlePattern = "msajc003")
```

```
## segment  list from database:  ae
## query was:  [Num(Syllable, Phoneme) > 4]
##   labels   start      end session   bundle    level type
## 1      S 256.925  674.175    0000 msajc003 Syllable ITEM
## 2      S 739.925 1289.425    0000 msajc003 Syllable ITEM
```

Conjunction

You can use "&" to search within several attribute definitions on the same level. For example, the level Word in `ae` has several attribute definitions

```
list_attributeDefinitions(ae,level="Word")
```

```
##     name   type hasLabelGroups hasLegalLabels
## 1   Word STRING          FALSE          FALSE
## 2 Accent STRING          FALSE          FALSE
## 3   Text STRING          FALSE          FALSE
```

We could, therefore, look for all accented ("S") words by ...

```
query(ae, "[Text =~.* & Accent == S]")
```

```
## segment  list from database:  ae
## query was:  [Text =~.* & Accent == S]
##        labels    start      end session   bundle level type
## 1     amongst  187.425  674.175    0000 msajc003  Text ITEM
## 2     friends  739.925 1289.425    0000 msajc003  Text ITEM
## 3   beautiful 2033.675 2604.425    0000 msajc003  Text ITEM
## 4      futile  571.925 1090.975    0000 msajc010  Text ITEM
## 5     further 1628.475 1957.775    0000 msajc010  Text ITEM
## 6  resistance 1957.775 2753.975    0000 msajc010  Text ITEM
## 7       chill  379.525  744.525    0000 msajc012  Text ITEM
## 8        wind  744.525 1082.975    0000 msajc012  Text ITEM
## 9      caused 1082.975 1456.475    0000 msajc012  Text ITEM
## 10     shiver 1650.975 1994.975    0000 msajc012  Text ITEM
## 11  violently 1994.975 2692.325    0000 msajc012  Text ITEM
## 12 emphasized  425.375 1129.075    0000 msajc015  Text ITEM
## 13  strengths 1213.075 1797.425    0000 msajc015  Text ITEM
```

```
## 14 concealing 2104.075 2693.675    0000 msajc015  Text ITEM
## 15 weaknesses 2780.725 3456.825    0000 msajc015  Text ITEM
## 16     itches  299.975  662.425    0000 msajc022  Text ITEM
## 17     always  775.475 1280.175    0000 msajc022  Text ITEM
## 18         so 1113.675 1400.675    0000 msajc022  Text ITEM
## 19   tempting 1400.675 1806.275    0000 msajc022  Text ITEM
## 20    scratch 1890.275 2469.525    0000 msajc022  Text ITEM
## 21         no 1774.925 1964.425    0000 msajc023  Text ITEM
## 22      risks 1964.425 2554.175    0000 msajc023  Text ITEM
## 23    display  666.675 1211.175    0000 msajc057  Text ITEM
## 24       more 1578.675 1824.425    0000 msajc057  Text ITEM
## 25       ever 2480.425 2794.925    0000 msajc057  Text ITEM
```

Another usage of "&" is to combine a basic query with a function, e.g.

```
## What is the query to retrieve all non-word-final "S" syllables?
query(ae, "[Syllable == S & End(Word, Syllable) == FALSE]", bundlePattern = "msajc003")
```

```
## segment  list from database:  ae
## query was:  [Syllable == S & End(Word, Syllable) == FALSE]
##   labels    start      end session   bundle    level type
## 1      S 1791.425 1945.425    0000 msajc003 Syllable ITEM
## 2      S 2033.675 2283.675    0000 msajc003 Syllable ITEM
```