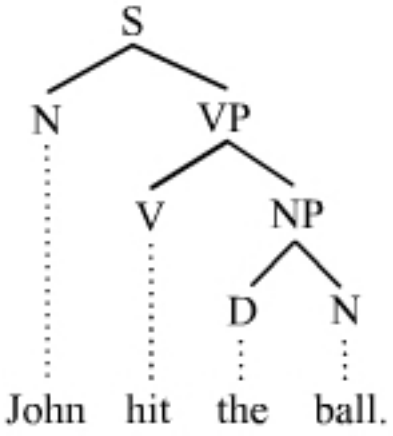
Annotation structures

This chapter introduces the reader to annotation structure modelling within emuR. Generally, there are two ways of thinking about annotations of speech data. Many linguists still think about the elements that build up an utterance in a purely `symbolic` way, i.e. a sequence of labels (e.g. phonemes). This approach is very useful in many sub-disciplines of linguistics, and certainly mainly known from introductory courses in syntax, in which even beginners start with learning that an utterance can be split up into its constituants, e.g. a nominal phrase and a verbal phrase, and that each of these can be split up further into smaller entities. A nominal phrase, for example, consists of a noun together with zero or more dependents of various types, like determiners, attributive adjectives, and many other possible candidates. A nominal phrase like "the green tree" would therefore consist of a determiner ("the"), an attributive adjective ("green"), and, of course, the obligatory noun ("tree"). The constituants of an entitiy are usually written down below of the symbolic representation of the entity itself, and linked with it, resulting in a tree-like structure. See e.g. a syntactic analysis of the sentence "John hit the ball":



The importancy of this analysis is expressed in the way how an entity can be shown to be build up by its constituants by symbolic **links** (here shown as lines) which show the **dominance** structure: entities dominate their constituants.

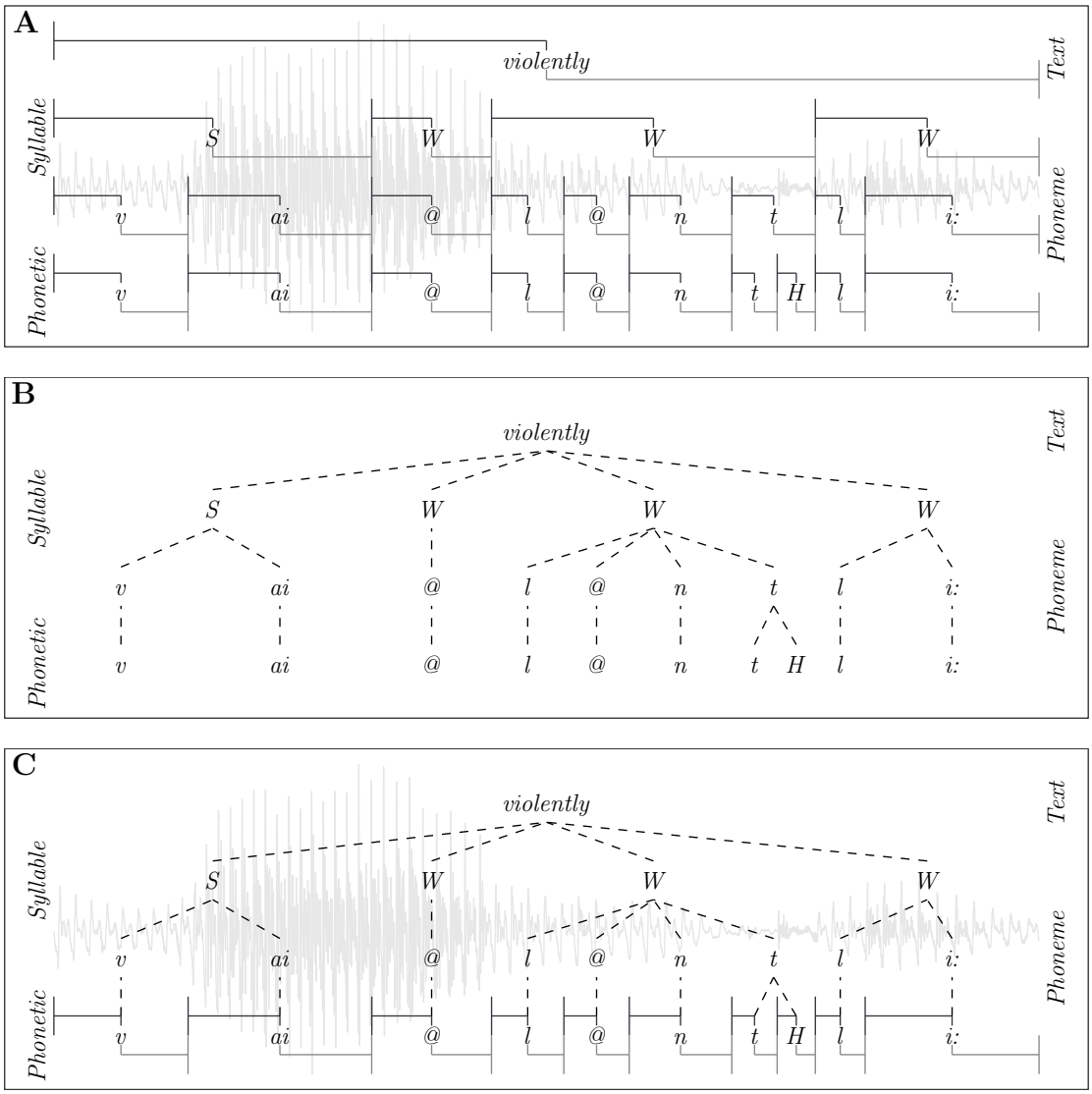
The sentence S in the top level consists of the N "John" and a verbal phrase called VP ("hit the ball"), and these constituants are shown in the next level; this VP on the second level can be split up into a verb V ("hit") and a nominal phrase NP ("the ball"), represented on a third level. A fourth level shows the analysis of the NP on level three, namely its constituants D (the determiner "the") and N (the noun "ball"). This analysis therefore reveals the internal syntactic structure of this sentence.

Such an analysis can get along **without any reference to physical time**, because the only - rather abstract - relation to time is the sequential order of the constituants of the sentence (e.g. word order). This is true not only for syntactic analyses, but also valid in other domains, e.g. in a purely phonemic analysis (cf. Figure 1B).

By contrast, physical **time**, and derived measures like durations etc., are a **very substantial part of phonetic analyses**. Many software packages like e.g. `praat` therefore represent the constituants of an utterance in a completely different way: they are represented on certain levels (e.g. the tiers in `praat`) and consist not only of a symbolic representation, but also of a relation to time and to the signal - whatever it may be - that was the result of any sort of measurement that was made during the utterance was spoken (e.g. in most cases an audio recording). The different levels, however, are completely independent of each other. Even if the researcher has a sort of link definition in his/her head, these links are *not* present in the actual annotation structure – even if it looks as if this would be the case (cf. Figure 1A).

The EMU legacy system combined the best of both worlds by mixing time-aligned and symbolic tree-like structures (cf. Figure 1C).This was achieved by providing software tools that allowed for these types of annotation structures to be generated, queried and evaluated. In practice, each annotation item had its own

unique identifier within the annotation. These unique IDs could then be used to reference each individual item and link them together using dominance relations to form the hierarchical annotation structure. On the one hand, this dominance relation implies the temporal inclusion of the linked sub-level items and was partially predicated on the no-crossing constraint, which does not permit the crossing of dominance relationships with respect to their sequential ordering. Since the dominance relations imply temporal inclusion, events can only be children in a parent-child relationship. To allow for timeless annotation items, a further **timeless level type** was used to **complement** the segment and event type levels used for **time-aligned annotations**. Each level of annotation items was stored as an ordered set to ensure the sequential integrity of both the time-aligned and timeless item levels.



All these ideas of the legacy EMU system have been adopted to the EMU-SDMS. The elements needed to achieve the above mentioned properties are:

Level types

There are three types of levels in the EMU-SDMS. Two of them are time-aligned and one is purely symbolic:

- **SEGMENT**s (segments of the signal represented by a *start time* and a *end time*)

- **EVENT**s (single *points in time*)
- **ITEM**s (*timeless symbols*)

Relation types

There are two (self-explaining) types of relations in the EMU-SDMS:

- **dominance**
- **sequence**

Sequence is expressed by means of a numerical ID. E.g. the word label "amongst" is the 30st element in the first utterance (msajc003) of the `ae` database. This information is saved in the file msajc003_annot.json:


```
{
          "id": 30,
          "labels": [
            {
              "name": "Text",
              "value": "amongst"
            }
          ]
        },
...
{
          "id": 39,
          "labels": [
            {
              "name": "Syllable",
              "value": "W"
            }
          ]
        },
...
{
          "id": 53,
          "labels": [
            {
              "name": "Phoneme",
              "value": "V"
            }
          ]
        },
...
{
          "id": 87,
          "sampleStart": 3749,
          "sampleDur": 1389,
          "labels": [
            {
              "name": "Phonetic",
              "value": "V"
            }
          ]
        },
```

Dominance relationships are also saved in the annot.json files within each bundle, e.g.:

```
{
      "fromID": 30,
      "toID": 39
    },
...
{
      "fromID": 39,
      "toID": 53
    },
...
{
      "fromID": 53,
      "toID": 87
    },
```

i.e. (the timeless) "amongst" dominates the (timeless) weak syllable "W", which dominates (and consists of) the ("timeless") vowel phoneme "V", which dominates the phonetic (and therefore time-aligned) vowel "V".

The resulting annotation tree can be seen by browsing to http://ips-lmu.github.io/EMU-webApp/ and opening the first utterance of the demo database `ae`. In this demo database, you will find all three level types, i.e. timeless ITEMs, and time-aligned SEGMENTs and (tonal) EVENTs, respectively.

Reduced data redundancy

One obvious way to reduce data redundancy is the use of the level type ITEM, as it is a purely timeless symbol, i.e. only a symbol, but no time information has to be stored, alongside with information about the ITEM's position in a sequential order of ITEMs and the ITEM's dominance over SEGMENTs; the dominance relationship of an ITEM to one or more SEGMENTs allows, however, to deduce time information from the start time of the first dominated SEGMENT and the end time of the last dominated SEGMENT. So, if you are searching "amongst" in the `ae` database, you will not only get the information, that there is one "amongst" in the first utterance, but also the start time of its first phonetic segment and the end time of its last phonetic segment ("V", i.e. ID 87, and "t", i.e. ID 92, respectively).

Another way to reduce data redundancy was achieved by allowing parallel annotations (**multiple attributes**) to be defined in the form of linearly linked levels for any given level (e.g., a segment level bearing SAMPA annotations as well as IPA UTF-8 annotations). See examples of this in the `ae` demo database mentioned above.

Database annotation structure definition

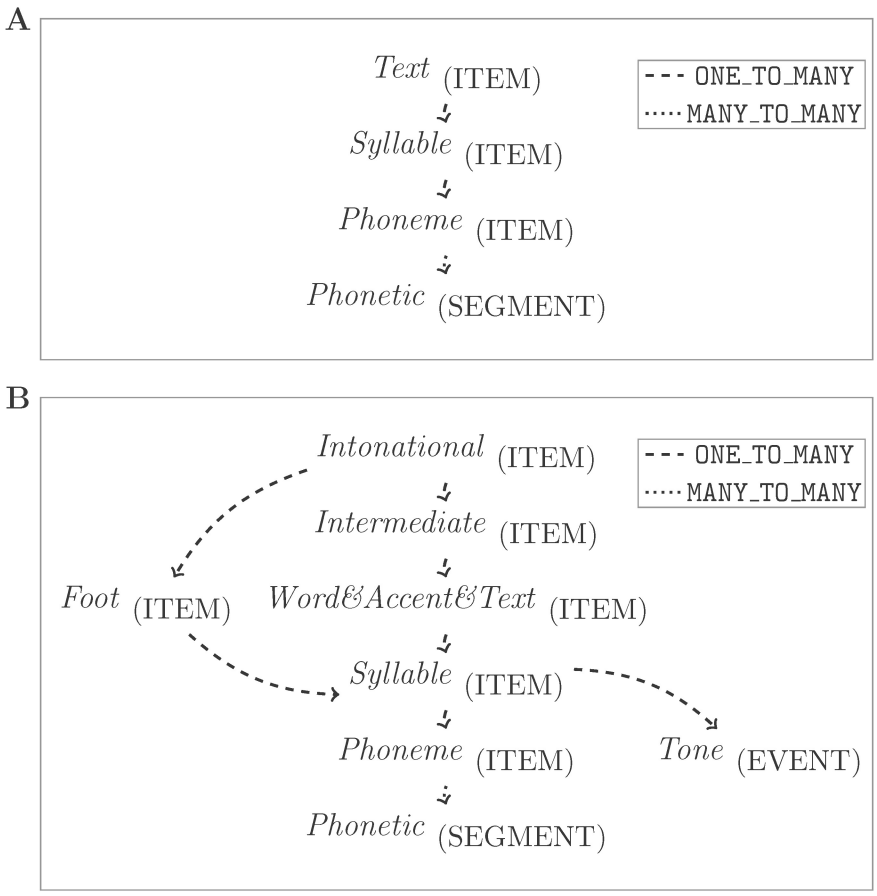
Unlike other systems, the EMU-SDMS requires the user to define the annotation structure formally for all annotations within a database. As mentioned above, the actual annotation files (. . . annot.json) of an emuDB contain the annotation items as well as their hierarchical linking information. To be able to check the validity of a connection between two items, the user specifies which links are permitted for the entire database just as for the level definitions. The permitted hierarchical relationships in an emuDB are expressed through link definitions between level definitions as part of the database configuration. There are three types of valid links:

ONE_TO_ONE

ONE_TO_MANY

MANY_TO_MANY

These links specify the permitted relationships between instances of annotation items of one level and those of another.



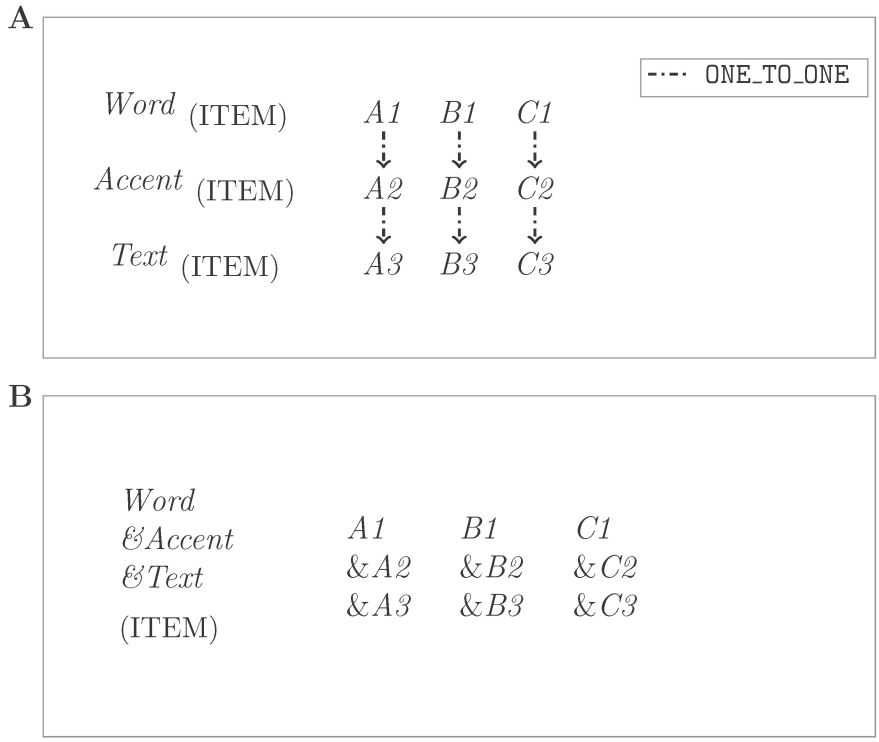
The structure in Figure 2A is a typical example of an EMU hierarchy where only the Phonetic level of type SEGMENT contains time information and the others are timeless as they are of the type ITEM. The top three levels, Text, Syllable and Phoneme, have a ONE_TO_MANY relationship specifying that a single item in the parent level may have a dominance relationship with multiple items in the child level. In this example, the relationship between Phoneme and Phonetic is MANY_TO_MANY: this type of relationship can be used to represent schwa elision and subsequent sonorant syllabi

cation, as when the final syllable of "sudden" is /d@n/ at the Phoneme level but [dn] at the Phonetic level. Figure 2B displays an example of a more complex, intersecting hierarchical structure definition where Abercrombian feet (Abercombie, 1967) are incorporated into the Tones and Break Indices (ToBI) (Beckman and Ayers, 1997) prosodic hierarchy by allowing an intonational phrase to be made up of one or more feet (for further details see Harrington, 2010, page 98).

Parallel labels and multiple attributes

The legacy EMU system made a distinction between linearly and non-linearly linked inter-level links. Linearly linked levels were used to describe, enrich or supplement another level. For example, a level called Category might have been included as a separate level from Word for marking words' grammatical category memberships (thus each word might be marked as one of adjective, noun, verb, etc.), or information about whether or not a syllable is stressed might be included on a separate Stress tier (description taken from Harrington, 2010, page 77). Using ONE_TO_ONE link definitions to define a relationship between two levels, it is still possible to model linearly linked levels in the new EMU-SDMS. However, an additional, cleaner concept that reduces the extra level overhead has been implemented that allows every annotation item to carry multiple attributes (i.e., labels). The generic term **attribute** (vs. *label*) was chosen to have the exibility of adding attributes that are not of the type STRING (i.e., labels) to the annotation modeling

capabilities of the EMU-SDMS in future versions. Figure 3 shows the annotation structure modeling difference between linearly linked levels (see Figure 3A) and an annotation structure using multiple attributes (see Figure 3B). Figure 3A shows three separate levels (Word, Accent and Text) that have a ONE_TO_ONE relationship. Each of their annotation items is linked to exactly one annotation item in the child level (e.g., A1-A3). Figure 3B shows a single level that has three attribute definitions (Word, Accent and Text) and each annotation item contains three attributes (e.g., A1-A3). It is worth noting that every level definition must have an attribute definition which matches its level name. This primary attribute definition must also be present in every annotation item belonging to a level. As emuR's database interaction functions, such as `add_levelDefinition()`, and the EMU-webApp automatically perform the necessary actions this should only be of interest to (semi-)advanced users wishing to automatically generate the annot.json format.



Examples

Start once again with the conversion from a collection of TextGrids and correspondings audio files:

```
library(emuR)

# create demo data in directory provided by the tempdir() function
# (of course other directory paths may be chosen)
create_emuRdemoData(dir = tempdir())

# create path to demo data directory, which is
# called "emuR_demoData"
demoDataDir = file.path(tempdir(), "emuR_demoData")

# show demo data directories
list.dirs(demoDataDir, recursive = F, full.names = F)

# create path to TextGrid collection
tgColDir = file.path(demoDataDir, "TextGrid_collection")
```

```
# show content of TextGrid_collection directory
list.files(tgColDir)


# convert TextGrid collection to the emuDB format
convert_TextGridCollection(dir = tgColDir,
                           dbName = "myFirst",
                           targetDir = tempdir(),
                           tierNames = c("Text", "Syllable",
                                         "Phoneme", "Phonetic"))
# get path to emuDB called "myFirst"
# that was created by convert_TextGridCollection()
path2directory = file.path(tempdir(), "myFirst_emuDB")

# load emuDB into current R session
dbHandle = load_emuDB(path2directory, verbose = FALSE)


summary(dbHandle)

# list level definitions
# as this reveals the "-autobuildBackup" levels
# added by the autobuild_linkFromTimes() calls
list_levelDefinitions(dbHandle)
```

There are no link definitions:

```
list_linkDefinitions(dbHandle)
```

We want to add a few link definitions. We chose ONE_TO_MANY on order to link Text and Syllable (as one word may contain several syllables), and to link Syllable and Phoneme (as one Syllable usually consists of several Phonemes); however, we use MANY_TO_MANY when linking the Phoneme level with the Phonetic level (allowing insertions and deletions, respectively, on the Phonetic level)

```
# invoke autobuild function
# for "Text" and "Syllable" levels
autobuild_linkFromTimes(dbHandle,
                        superlevelName = "Text",
                        sublevelName = "Syllable",
                        convertSuperlevel = TRUE,
                        newLinkDefType = "ONE_TO_MANY")

# invoke autobuild function
# for "Syllable" and "Phoneme" levels
autobuild_linkFromTimes(dbHandle,
                        superlevelName = "Syllable",
                        sublevelName = "Phoneme",
                        convertSuperlevel = TRUE,
                        newLinkDefType = "ONE_TO_MANY")

# invoke autobuild function
# for "Phoneme" and "Phonetic" levels
autobuild_linkFromTimes(dbHandle,
                        superlevelName = "Phoneme",
                        sublevelName = "Phonetic",
```

```
                        convertSuperlevel = TRUE,
                        newLinkDefType = "MANY_TO_MANY")

# list level definitions
list_levelDefinitions(dbHandle)

list_linkDefinitions(dbHandle)

# remove the levels containing the "-autobuildBackup"
# suffix
remove_levelDefinition(dbHandle,
                       name = "Text-autobuildBackup",
                       force = TRUE,
                       verbose = FALSE)

remove_levelDefinition(dbHandle,
                       name = "Syllable-autobuildBackup",
                       force = TRUE,
                       verbose = FALSE)

remove_levelDefinition(dbHandle,
                       name = "Phoneme-autobuildBackup",
                       force = TRUE,
                       verbose = FALSE)

# list level definitions
list_levelDefinitions(dbHandle)

list_linkDefinitions(dbHandle)

serve(dbHandle)
```

Legal labels and label groups

We often might want to restrict ourselves to only a few "legal" labels; for examples, it might be useful to use only "S" and "W" for strong and weak syllables (therefore, "s" or "w" and any other label will be illegal; this will not delete labels other than "W" and "S" that are already in our database; it will, however, present us and out co-workers for introducing such labels in the future)

```
get_legalLabels(dbHandle,
                levelName = "Syllable",
                attributeDefinitionName = "Syllable")

set_legalLabels(dbHandle,
                levelName = "Syllable",
                attributeDefinitionName = "Syllable",
                legalLabels = c("S", "W"))
serve(dbHandle)
```

Another useful possibility is the grouping of certain labels, e.g.:

```
add_labelGroup(dbHandle,
               name = "Vowels",
               values = c("i:", "o:", "V"))
```

```
list_labelGroups(dbHandle)

#####Delete this again with ...
#delete_labelGroup(dbHandle,
#                 name = "Vowels")

query (emuDBhandle = dbHandle,
       query = "Phonetic == Vowels")
#instead of
query (emuDBhandle = dbHandle,
       query = "Phonetic == i: | o: | V")
```

Further information

Type

```
vignette()
```

to find all available vignettes (i.e. introductory help pages) of all installed packages.

For the emuR-package, you'll find 3 vignettes, "emuR_intro" (an introduction similar to chapters 01-03), "emuDB" (information about the database format of emuR), and "EQL" (an introduction to the emuR query language). To see the vignette concerned with the EMU-SDMS annotation structure, type

```
vignette("emuDB")
```