Appendix B: Some notes on Emu-Tcl

The purpose of this appendix is to give an overview of some of the commands and scripts in the Emu-Tcl library for building automatically trees and annotation structures. The material in this section is based on Cassidy (2000) who, as well as describing most of the available scripts in Emu-Tcl, also gives a brief overview of the necessary background to Tcl for implementing scripts from the Emu-Tcl library. For further details on Tcl/Tk see the Tcl Developer Xchange page at http://www.tcl.tk/. A very useful, readable and informative introduction to Tcl scripting is available on the web by Abelson, Greenspun and Sandon (2008).

There are two parts to this Appendix. The first (B.1) is concerned with an overview of some commands for building annotation structures from scratch and the second (B.2) with implementing existing scripts in the Emu-Tcl library for carrying out tasks such as parsing, syllabification, and the alignment of annotation strings.

B.1 Some basic Emu-Tcl commands

B.1.1 Testing evolving scripts in the Console

A good way to start with Emu-Tcl and Tcl in general is to see the effects of some commands. Start up Emu and then open a Tcl console window with `File -> Console` to enter commands to Tcl. Here are some very basic Tcl commands that will be used in the discussion of annotation structures below.

A Tcl command consists of a command name followed by one or more arguments separated by spaces. For example:

```
# Create the variable x and give it the values 10 20 30
set x {10 20 30}

# List the values of the variable x
set x
10 20 30

# List the first value
lindex $x 0
10

# List the number elements in x
llength $x
3

# Append some values to x
lappend x 50 60 40
set x
10 20 30 50 60 40
```

You will notice from the above that it is very important in Tcl to distinguish between the name of the variable, such as x, and the value(s) that is contains: to get at the values, the name must be preceded by a $ symbol. However, since the first thing you type into Tcl is always a command name or procedure, then $x on its own is uninterpretable - and that is why set x must be used to list the variable's contents.

Square brackets are used to substitute the values of a command into another command. For example:

```
# The same as set x {10 20 30}
set x [list 10 20 30]

# Create a variable p containing the 2nd element of x
set p [lindex $x 1]
set p
20
```

B.1.2 Emu-Tcl commands

The rest of this section is concerned with some commands for finding information from, and sometimes modifying, either a database, or a specific utterance of a database, or any annotation of a specific utterance.

B.1.2.1 Finding information about a database

The emutemplate command in the Emu-Tcl library (package require emu) can be used to list the available templates (i.e. databases) or to create a command for manipulating a database. All of the commands below are typed into the console window as before. (If you want to enter the following commands at the console, make sure you have downloaded the ae database and that it is accessible in Emu). For example:

```
# List the available databases
emutemplate

# Make a command, t, for manipulating the ae database
emutemplate t ae
```

The above command creates a command, t, which can be used to retrieve information from the ae database. Here are some examples:

```
# The tiers or levels of the ae  database
t getlevels
Utterance Intonational Intermediate Word Syllable Phoneme Phonetic Tone
Foot

# All ancestors (parent, grand-parent...) of Phoneme
t ancestors Phoneme
Utterance Intonational Intermediate Word Syllable Foot

# The child tier of Phoneme
 t children Phoneme
Phonetic

# All tiers that are descendents i.e. children of Text
t descendents Text
Syllable Phoneme Phonetic Tone

# The parent tier of Foot
t parents Foot
Intonational
```

```
# Does Foot dominate Syllable? (Returns 1 if so, otherwise 0).
t dominates Foot Syllable
1

# Get any tiers linearly linked to Word
t getlabels Word
Word Accent Text

# The utterances of ae
t utterances
msajc003 msajc010 msajc012 msajc015 msajc022 msajc023 msajc057

# Make a variable consisting of these utterances and list the 3rd one:
set u [t utterances]
lindex $u 2
msajc012

# Equivalently
lindex [t utterances] 2
msajc012
```

B.1.2.2 Finding segment numbers in an utterance

The preceding command, t, created with emutemplate, was used to obtain information about the `ae` database itself. In order to get information about a specific utterance, a command specific to that utterance needs to be created. This is done with the hierarchy sub-command as follows:

```
# Create a command, h, that can be used to access information from
# the utterance msajc003
# NB emutemplate t ae must have been entered first

t hierarchy h msajc003

# Equivalently
t hierarchy h [lindex [t utterances] 0]
```

h, just like t, is a command and like t, it can be followed by different sub-commands for finding information, in this case, about the utterance `msajc003`. One of these sub-commands, segments, lists the *segment numbers* at a particular tier.
For example:

```
# NB t hierarchy h msajc003 must have been carried out first
h segments Text
2 24 30 43 52 61 83
```

In order to make sense of the above output, open the hierarchy window for `msajc003` from the `ae` database in Emu and select `Display -> Toggle Segment Numbers` in the manner of Fig. B.1: this shows that the numbers returned by the preceding command are the segment numbers of the annotations at the Text tier.

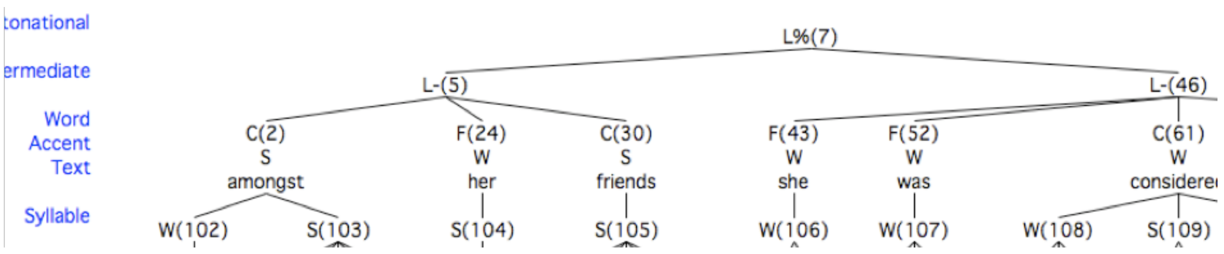


Fig. B.1. Part of the annotation structure of `msajc003` from the `ae` database. The numbers returned by h segments Text are those given at the linearly linked Word tier 2, 24, 30, ...

Thus analogously h segments Intonational returns `7`, because this is the segment number of the single annotation at the Intonational tier (see Fig. B.1).

B.1.2.3 Finding the annotations of segment numbers

The seginfo subcommand can be used to find out various kinds of information about any single segment number. Before examining seginfo in further detail, note from Fig. B.1, that:

- The first word `amongst` has a segment number 2.
- Its annotation (label) is obviously `amongst`.
- The annotations at the tiers Word and Accent linearly linked to `amongst` are `C` and `S` respectively.
- The segments at the child tier Syllable linked to `amongst` are 102 and 103.
- The segment at the grand-parent tier Intonational linked to `amongst` is `7`.

All these kinds of information are provided by seginfo together with the previously created h command together with the segment number of `amongst` itself (which, as already stated, is 2). For example:

```
# The first segment at the Text tier
lindex [h segments Text] 0
2

# The annotation of segment number 2 at the Text tier
h seginfo 2 label Text
amongst

# The annotation of segment number 2 at the Word tier
h seginfo 2 label Word
C

# The annotation of segment number 2 at the Accent tier
h seginfo 2 label Accent
S

# The segment numbers of the child annotations at the Syllable tier (see Fig. B.1)
h seginfo 2 children Syllable
102 103
```

```
# The segment number of the (grand)parent tier Intonational (see Fig. B.1)
h seginfo 2 parents Intonational
7
```

One of the very cumbersome aspects of seginfo is that you can only ever ask information about any single segment. In order to retrieve information from several segments, a for-loop would be needed. Here is an example that begins by retrieving the numbers of all the segments at the Text tier, as before. This time these segment numbers are additionally stored in the variable textnos:

```
set textnos [h segments Text]

# Get the annotations of each segment number
foreach j $textnos {
lappend  textlabs [h seginfo $j label Text]
}

# List the labels
set textlabs
amongst her friends she was considered beautiful
```

The above for-loop could be packed into a procedure (which you should copy into the console window) to get the labels from the segment number at any tier.

```
proc getlabels {x level} {
  # x contains segment numbers; level is the tier at which these occur
  foreach j $x {
    lappend  labs [h seginfo $j label $level]
  }
  return $labs
}
```

The annotations at the Text tier could now be retrieved with:

```
getlabels $textnos Text
amongst her friends she was considered beautiful
```

The above procedure together with the seginfo command could equally be used to find the annotations at the Phonetic tier dominated by `amongst`, thus:

```
set phonnum [h seginfo 2 children Phonetic]
getlabels $phonnum Phonetic
V m V N s t H

# Or equivalently in a single line
getlabels [h seginfo 2 children Phonetic] Phonetic
V m V N s t H
```

The query sub-command can be used to retrieve annotations at any tier in accordance with the Emu-QL search instructions described in Chapter 4.

#Get all annotations and their associated times at the Text tier
h query “Text != x”

```
{ae Text != x "segment"} {amongst 187.498000 674.237000 msajc003} {her
674.237000 739.994000 msajc003} {friends 739.994000 1289.494000 msajc003}
{she 1289.494000 1463.242000 msajc003} {was 1463.242000 1634.493000
msajc003} {considered 1634.493000 2150.242000 msajc003} {beautiful
2033.739000 2604.489000 msajc003}
```

B.1.2.4 Modifying annotations

The same subcommand seginfo can be used to modify the annotation of any individual segment number by supplying as a final argument the desired new annotation. For example, the following command changes the annotation of segment number 5 (currently L-, see Fig. B1) to H-

h seginfo 5 label Intermediate H-

# Verify that 5 now has the label H-
h seginfo 5 label Intermediate

```
H-
```

Changing the annotation of any linearly linked tier can be done in the same way. For example, to change the `c` annotation of segment 2 (`amongst`) to `F`:

h seginfo 2 label Word F

B.1.2.5 Modifying links

Links can be created with the same seginfo sub-command and deleted with the sub-command delete relation. For example, the following sub-commands can be used to re-link the associations between Syllable and Phoneme for the word `amongst` corresponding to a change from `a.mongst` in the current utterance to `am.ongst` as in Fig. B.2.

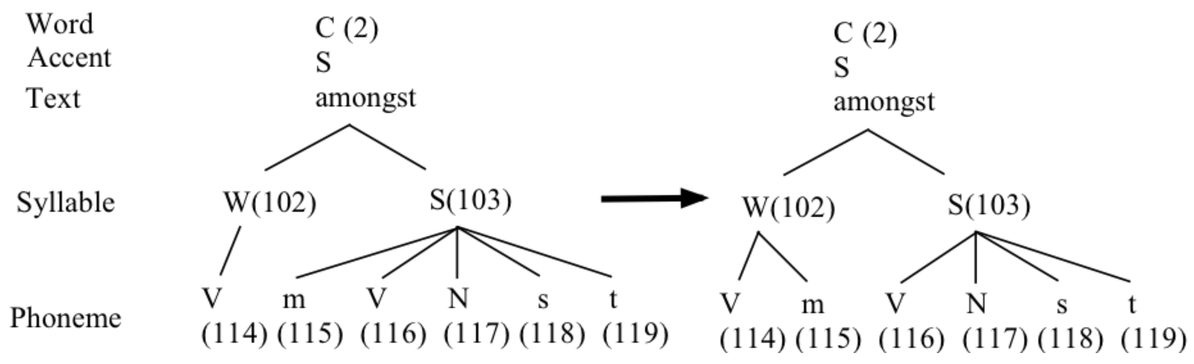


Fig. B.2. The relationship between segment numbers and annotations in the word `amongst` in utterance `msajc003` of the `ae` database (left) and the modifications that are to be made with Emu-Tcl commands (right).

# Delete the parent-child relation between the first syllable and /m/
h delete relation 103 115

# The second syllable should now consist of only four phonemes
h seginfo 103 children Phoneme

```
116 117 118 119
```

```
# Make /m/ a child of the first syllable
h seginfo 115 parents Syllable 102
```

```
# The first syllable should consist of two phonemes
h seginfo 102 children Phoneme
114 115
```

It is possible to use the parents and children sub-commands to link one segment with *multiple* segments. For example:

```
# Delete the parent-child relationships in the first syllable
h delete relation 102 114
h delete relation 102 115
```

```
# Make the first syllable a parent of "V" and "m"
set n {114 115}
h seginfo 102 children Phoneme $n
```

B.1.2.6 Adding and deleting segment numbers and their annotations

The subcommands append, delete, insert, prepend are for adding and deleting segments. In case of *timeless* tiers the use is straightforward. For example, an additional segment H% could be prepended at the Intonational tier as follows (the append sub-command works in the same way, except that new segments are inserted *after*, rather than *before*, any existing annotations).

```
h prepend Intonational H%
```

There should now be two segments at this tier and this is confirmed by the segments sub-command:

```
h segments Intonational
0 7
```

The delete subcommand can be used to delete the segment that has just been prepended:

```
h delete segments 0
```

```
# Verify that this segment has been deleted
h segments Intonational
7
```

The insert sub-command inserts a segment at a specific position. For example, to insert H- after the first L- tone (segment number 5, see Fig. B.1):

```
h insert Intermediate 5 H-
# Check that there are three segments at this tier
h segments Intermediate
5 0 46
```

All of the sub-commands could be used to insert several segments at once. For example:

```
# Append 4 segments L- H- L- L- at the Intermediate tier
set labs {L- H- L- L-}
h append Intermediate $labs

# get the segment numbers of all segments at the Intermediate tier
h segments Intermediate
5 0 46 1 3 4 6

# delete segment numbers 0, 1, 3, 4, 6
set n {0 1 3 4 6}
h delete segments $n

#Equivalently: Append 4 segments L- H- L- L- at the Intermediate tier and delete those
#segments again
set labs {L- H- L- L-}
set new [h append Intermediate $labs]
h delete segments $new
```

To append, delete, insert, prepend annotations at *event* tiers, it is necessary to set the time mark of the new segments.

```
#append an Event at the Tone tier at time mark 2660 ms
set ns [h append Tone L%]
h seginfo $ns times 2660
```

To do the same at *segment* tiers, it is necessary to specify the onset and offset times of the new segments as well as of the present segments if segments are to be inserted.

```
#get the offset time of the last segment at the Phonetic tier
set ls [lindex [h segments Phonetic] end]
set lsoffset [lindex [h seginfo $ls times] end]

#Append a segment at the Phonetic tier
set ns [h append Phonetic pause]
h seginfo $ns times $lsoffset [expr $lsoffset + 5]
```

B.1.2.7 Updating the annotation files

You can write out the results of the changes that you make to the annotation structure of any utterance with the write sub-command which will cause the utterance's hlb file to be overwritten (or one to be created, if none exists). When you open the utterance again in Emu, then any modifications that were made will be visible in the hierarchy window. The syntax for writing/updating the hlb file of the utterance `msajc003` is:

```
h write msajc003
```

If changes have been made to any time tier, then the sub-command writelabels causes the corresponding annotation files of time tiers to be over-writtten.

```
h writelabels
```

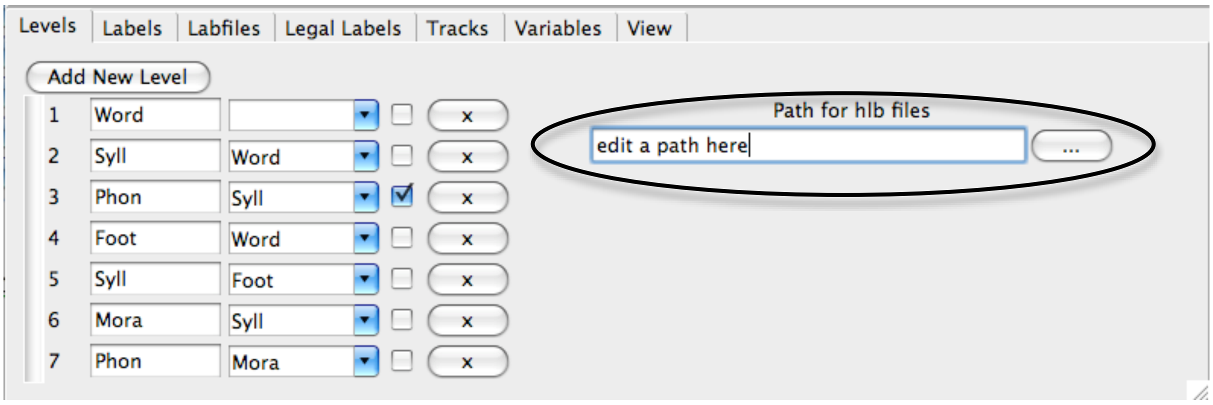
Alternatively (and perhaps preferably) if you do not want to overwrite the existing hlb file, then give the utterance a different basename first, thus

```
h basename m
h write m
```

The above two instructions will create a file `m.hlb` in the path that was given for storing hlb files in the Levels pane of the template file.

B.1.2.8 Building annotation structures: the mora database

There is now just about sufficient information to build some of the simpler annotation structures discussed in Chapter 4. The first example in this section involves building the structure for the `mora` database shown in Fig. 4.26 of Chapter 4. Before running the commands below, first edit the template file of the `moraanswer` database (which accesses the same data) and choose a new path for the hlb file as shown in Fig. B.3. Leave all other attributes of the template as they are.



1	Word		☐	x
2	Syll	Word	☐	x
3	Phon	Syll	✓	x
4	Foot	Word	☐	x
5	Syll	Foot	☐	x
6	Mora	Syll	☐	x
7	Phon	Mora	☐	x

Path for hlb files: edit a path here

Fig B.3. The modified template file of the database `moraanswer` in which a new path should be chosen for saving the hlb file.

The task will now be to build the annotation structure on the left of Fig. 4.26 in Chapter 4 linking Word, Foot, Syll, and Phon tiers.

```
# Load the template
emutemplate t moraanswer

# Load the utterance
t hierarchy h kitta
```

# Insert `O` at the Word tier, `F` at the Foot tier, and two `s` annotations at the Syll tier

```
h append Word O
h append Foot F
set syll {s s}
h append Syll $syll
```

```
# Make F a child of O
h seginfo [h segments Word] children Foot [h segments Foot]

# Make the first s a child of F
h seginfo [h segments Foot] children Syll [lindex [h segments Syll] 0]

# Make the first three phonetic segments children of the first s
# Either do this one phonetic segment at a time
h seginfo [lindex [h segments Syll] 0] children Phon [lindex [h segments Phon] 0]
h seginfo [lindex [h segments Syll] 0] children Phon [lindex [h segments Phon] 1]
h seginfo [lindex [h segments Syll] 0] children Phon [lindex [h segments Phon] 2]

# or use a for loop
for {set i 0} {$i <= 2} {incr i} {
  h seginfo [lindex [h segments Syll] 0] children Phon [lindex [h segments Phon] $i]
}

# Make the last two phonetic segments children of the 2nd syllable
h seginfo [lindex [h segments Syll] 1] children Phon [lindex [h segments Phon] end]
h seginfo [lindex [h segments Syll] 1] children Phon [lindex [h segments Phon] end-1]

# write out the results
h write kitta
```

The effect of the last instruction will be to write out the hlb file to whichever path you specified in the template file (Fig. B.3) so that when you load the utterance again, the links should be set as in the left panel of Fig. 4.26 of Chapter 4.

B.1.2.9 From console to AutoBuild scripts

The task is now to relate console commands to scripts run over an entire database as discussed in section 4.8 of Chapter 4. In question 4.3 of the section 4.11 Questions of Chapter 4, the Tcl script `ematcl.txt` was used to link the annotations as shown in Fig. B.4.

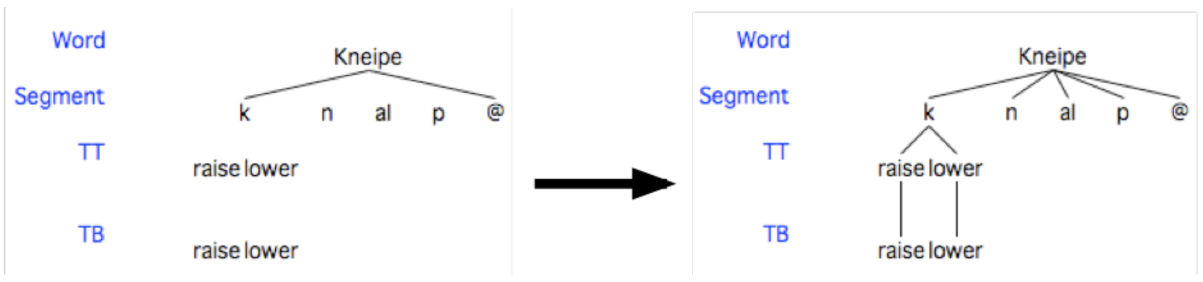


Fig. B.4. The annotation structure for the utterance `dfgspp_mo1_prosody_0020` before (left) and after (right) the application of a Tcl script ematcl.txt.

The script looks like this (and is stored in `path/ema/ematcl.txt` where `path` is the directory to which you downloaded the `ema` database):

```
package require emu::autobuild
proc AutoBuildInit {template} {}
proc AutoBuild {template h} {

  # get the segment numbers at level TB
  set tbsegs [$h segments TB]

  # get the first of these
  set tbfirst [lindex $tbsegs 0]

  # get the second of these
  set tbsecond [lindex $tbsegs 1]

  # get the segment numbers at tier TT
  set ttsegs [$h segments TT]

  # get the first of these
  set ttfirst [lindex $ttsegs 0]

  # get the second of these
  set ttsecond [lindex $ttsegs 1]

  # Link first segment at TB with the first segment at TT
  $h seginfo $ttfirst children TB $tbfirst

  # Link the 2nd segment at TB with the 2nd segment at TT
  $h seginfo $ttsecond children TB $tbsecond

  # Get the segment number at the Word tier
  set wordseg [$h segments Word]

  # Get the segments at the Segment tier that are children at the Word tier
  set phonsegs [$h seginfo $wordseg children Segment]

  # Get the first of these
  set phonfirst [lindex $phonsegs 0]

  # Link this first segment at the Segment tier to all segments at tier TT
  $h seginfo $phonfirst children TT $ttsegs

  # Make links for all segments between the first and last segments at
  # the Segment tier dominated by Word
  LinkSpans $h Word Segment
}
```

You can enter this script and apply it to the utterance `dfgspp_mo1_prosody_0020` in the console window with the following very few modifications:

# Load the Emu-Tcl library containing scripts like LinkSpans

```
package require emu::autobuild

# Make a command called template
emutemplate template ema

# Make a command, h, specific to one utterance
template h dfgspp_mo1_prosody_0020

# get the segment numbers at level TB
set tbsegs [h segments TB]

# get the first of these
set tbfirst [lindex $tbsegs 0]

# get the second of these
set tbsecond [lindex $tbsegs 1]

# get the segment numbers at tier TT
set ttsegs [h segments TT]

# get the first of these
set ttfirst [lindex $ttsegs 0]

# get the second of these
set ttsecond [lindex $ttsegs 1]

# Link first segment at TB with the first segment at TT
h seginfo $ttfirst children TB $tbfirst

# Link the 2nd segment at TB with the 2nd segment at TT
h seginfo $ttsecond children TB $tbsecond

# Get the segment number at the Word tier
set wordseg [h segments Word]

# Get the segments at the Segment tier that are children at the Word tier
set phonsegs [h seginfo $wordseg children Segment]

# Get the first of these
set phonfirst [lindex $phonsegs 0]

# Link this first segment at the Segment tier to all segments at tier TT
h seginfo $phonfirst children TT $ttsegs

# Make links for all segments between the first and last segments at
# the Segment tier dominated by Word
LinkSpans h Word Segment

#write the annotations to file
h write [h basename]
```

It seems then, that the differences between the structure of the AutoBuild script and the commands that you just typed into the console can be reduced to the following:

- The function proc AutoBuildInit {template} {} replaces the command line emutemplate template ema. The argument template has the same meaning in both cases and is used to load the database.
- All of the other command lines entered at the console are contained within a procedure AutoBuild that has two arguments: template (the same argument that was used in AutoBuildInit) and h, the command for manipulating a particular utterance's annotation structure. This is the same h that is used in the command that you typed into the console template h dfgspp_mo1_prosody_0020 - notice that this command from the console no longer appears in the AutoBuild procedure which replaces it.
- Instead of h typed into the console, there is $h, a variable in the script. Both h and $h have the same meaning: but in the script, it is a variable, $h, because the commands are to be applied not just to *one* utterance, as in the commands typed in at the console, but to *all* the utterances of the database.
- The AutoBuild procedure executes h write [h basename]
- Finally, the line package require emu::autobuild should always be included in the script (and is in fact also needed for the commands typed into the console in order to access the procedure LinkSpans).

The next section gives some further examples of AutoBuild using a number of existing procedures in the same Emu-Tcl library that contains LinkSpans.

B.2 Using EMU-Tcl: interface to a lexicon and some tree-building rules

In this section, an example is given, using existing Tcl scripts in the Emu-Tcl library, of how to interface an exhaustive segmentation into phonetic sized segments with a lexicon containing phonemic citation-form entries and to build an intervening syllable structure layer based on the maximum-onset-principle. As always, one of the main reasons for adding these layers of annotation structure is to enrich the range of queries that can be made – in order, for example, to be able to establish whether the phonetic properties of /t/ segments in word-initial position are different from those in word-medial and in syllable-initial position, and so on.

The various steps in building three layers of annotation semi-automatically on top of a segmentation into phonetics units are as follows. The user **enters the orthographic (text) representation of the utterance** that was produced by the speaker and for which a phonetic segmentation was made by a transcriber. This text is used to **access a lexicon containing entries of each word keyed to a citation-form phonemic pronunciation**, that is a broad transcription corresponding to the production of words in isolation. Next an Emu-Tcl script is applied to the citation-form phonemic string to **parse it into syllables**. The syllabification algorithm is based on the maximum onset principle (MOP) (Hoard, 1971; Selkirk, 1982) which groups as many consonants with a following vowel as possible, as long as the sequence is deemed to be phonotactically legal. Thus according to this principle, *athlete* would be syllabified as *ath.lete*, since /θl/ is not a possible onset cluster in English. It is emphasised here that the MOP is only one of many possible solutions to syllabification and it is quite possible for a user with a basic knowledge of Tcl programming to modify the existing Emu-Tcl script to produce other kinds of syllabification. The final step is to make use of an Emu-Tcl **matching algorithm** that finds the best possible alignment between two slightly different strings of segments. This matching algorithm is applied to align the citation-form or canonical phonemic representation with a phonetic segmentation. The difference between these two levels comes about because of the divergence between the citation-form pronunciations in the lexicon and those in continuous speech. For example, a task of the matching algorithm is to align a citation-form entry for a word like *actually*, which (for Southern British English) might be /aktʃʊəlɪ/ with possible spontaneous speech productions of [aktʃəli] or even [aʃli] (e.g., Laver, 1994). In general, this algorithm will be seen to do a reasonable job for the phonetic transcriptions taken from part of the Kiel Corpus of Read speech (Simpson et al, 1997), but also that some improvements can be made by invoking as a **phonetic-to-phoneme mapping rules** before this matching is done.

The example to illustrate these various steps is of German sentences requiring a German lexicon, a statement of legal syllable-onsets in German, and a list of German vowel and diphthong nuclei. However, as will become apparent, there is nothing in what is to be presented that is inherently tied to German and the reader should be able to adapt these steps to any language, given an equivalent set of statements and, of course, an existing segmentation into phonetic segments. Such changes – in contrast to modifying the syllabification algorithm referred to earlier – require no prior knowledge of Tcl programming.

Download the `timetable` database which consists of five utterances, some sampled speech data, and an exhaustive phonetic segmentation of each utterance. As the template file for this database shows, there is a Phonetic tier of segments and then three tiers hierarchically stacked on top of it in the following path:

$$\begin{tikzcd}
\text{Text (-)} \arrow[d] \\
\text{Syllable (-)} \arrow[d] \\
\text{Phoneme (-)} \arrow[d] \\
\text{Phonetic (S)}
\end{tikzcd}$$

The first task is to get the text (orthography) for each utterance into the Emu hierarchy window. The text for each utterance is stored in the directory `x/timetable/orthography`, where `x` is the name of the directory to which you downloaded the `timetable` database. For utterance `HPTE002`, the text is in `HPTE002.txt` and consists of these words:

morgens zwischen acht und neun also nach acht

(*tomorrow between eight and nine that is after eight*). They could be entered by hand into the Emu hierarchy window of the `timetable` database in the manner explained in Chapter 4, but a quicker way is to make use of a procedure in the Emu-Tcl library AddLabelsFromFile which reads the annotation from a plain text file into an annotation tier. The command is:

```
AddLabelsFromFile $template $tree $wordpath Text
```

The first two variables $template and $tree have the same meanings as the commands t and h described in B.1 for manipulating respectively a database and the annotation structure of an utterance. The third variable $wordpath is the directory in which the files that contain the text (like `HPTE001.txt`) are located; and the final variable is Text because this is the annotation tier into which the orthography is to be incorporated.

AddLabelsFromFile must be called from inside the AutoBuildInit and AutoBuild procedures as described in earlier in B.1.2.9 and the skeleton of these procedures has already been included in `emutcl.txt` that can be found in `x/timetable`. The required modifications to the script are shown in bold following some further explanatory comments (after #).

```
package require emu::autobuild
proc AutoBuildInit {template} {}

proc AutoBuild {template tree} {
# x is the directory name to which you downloaded the timetable database
set wordpath x/timetable/orthography
AddLabelsFromFile $template $tree $wordpath Text
}
```

Then save the `emutcl.txt` and read it into the `timetable` database via the Variables pane of the template file as described in Chapter 4 (section 4.8, Fig. 4.13). If you save the template file with these modifications, then when you open an utterance from the `timetable` database, there will be a `Build Hierarchy` button which, if clicked, applies this Tcl script to the annotations, as shown in Fig. B.5.

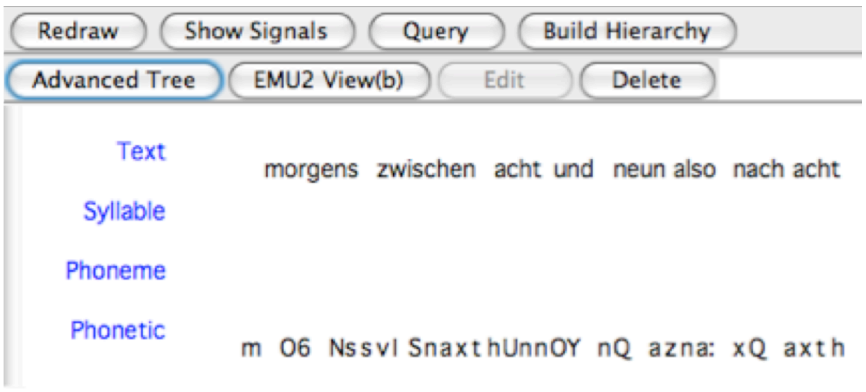


Fig. B5. The results of clicking on `Build Hierarchy` for the utterance `HPTE002.txt` in the `timetable` database after running a Tcl script to insert the orthography at the `Text` tier

It is best now **not to save the results of this operation**, since there are a few more procedures to be included in the script, and if you save it, then AddLabelsFromFile will add the same text annotations again, each time that the `Build Hierarchy` button is clicked (or, if you do save the annotation, delete all `hlb` files in `x/timetable/hlb`).

The next task is to read into the `Phoneme` tier, the phonemic forms of these words that are stored in a dictionary. To do this, you have to make a dictionary consisting of orthographic entries keyed to phonemic forms. For the present database, there is a dictionary available that is a modification of the one available with the Kiel Corpus of Read Speech (Simpson et al, 1997) and it can be found in `x/timetable`. The dictionary name is `dictgerman.txt` and its entries look like this:

```
…
mir m i:6
mischt m I S t
mit m I t
mittags m I t a: k s
```

```
morgen m O6 g @ n
morgens m O6 g @ n s
muss m U s
musst m U s t
musste m U s t @
maechtig m E C t I C
….
```

That is, there is the orthography, a space, and then a space between every phoneme of the phonemic entry. In Emu, such a dictionary is case-sensitive and when you prepare one for your own utterances, be sure that there is one (and only one) entry for each word of your database. (It so happens that `dictgerman.txt` has a large number of entries; but in practice the dictionary need only contain the words of your database's utterances).

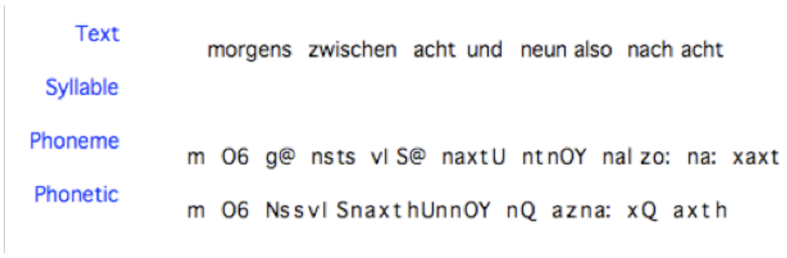
The phonemic forms of the dictionary entries are read into the database with the commands InitialiseDict which tells Emu that there is a dictionary available and LevelFromDict which does the job of reading in the phonemic forms. The required modifications to `emutcl.txt` are shown in bold, again with comments before the new lines:

```
package require emu::autobuild
proc AutoBuildInit {template} {}

proc AutoBuild {template tree} {
set wordpath x/timetable/orthography
AddLabelsFromFile $template $tree $wordpath Text

# Give the path where the dictionary is located as the 2nd argument
InitialiseDict lex x/timetable/dictgerman.txt
# Insert dictionary phonemes. $tree is the 2nd argument  of Autobuild;
# Text is the tier name for the orthography.  Phoneme is the tier name where you want
# to put the dictionary phonemes; lex is a variable name as defined above
LevelFromDict $tree Text Phoneme lex
}
```

When you now reload the utterance and click `Build Hierarchy` in the manner before, the result is to include the dictionary phonemes for each word, as in Fig. B.6.



```
Text        morgens  zwischen  acht und  neun also  nach acht
Syllable
Phoneme    m  O6  g@  nsts  vl S@  naxtU  ntnOY  nal zo:  na:  xaxt
Phonetic   m  O6  NssvlSnaxthUnnOY  nQ  azna:  xQ  axth
```

Fig. B.6. The results of including the procedures to read in dictionary phonemes into the Phoneme tier.

In order to syllabify according to the maximum onset principle, you have to declare what you consider to be the legal string of initial consonant phonemes. The legal strings that have been declared for German are in the file `clusters.txt` in `x/timetable`. It is a file containing statements of legal single-consonant, two-consonant, and three-consonant strings as follows:

```
set cons(triples) {Spr Spl Str skr skl}
set cons(pairs) {pr br tr dr kr gr fr Sr pl bl kl gl fl Sl Sm kn gn Sn kv
Sv Sp St sk tsv pfl pfr}
set cons(singles) {p b t d k g f v s z S Z C x m n N l r j h Q ts pf}
```

Notice that the legal single consonants include affricates `ts` and `pf` as in German *zu* (/tsu/, *to*) and *Pfeffer* (/pfɛfɐ/, *pepper*) because this is the way that that the dictionary has been coded (i.e., the entry for *zu* is `ts u` with no space between `t` and `s`). For this reason, the cluster `tsv` (as in `tsv aI`, /tsvaI/, *zwei* = *two*) is declared as a two-phoneme sequence in the line `set cons (pairs)`. Again, this is all dependent on the way that the phonemic entries are coded in the dictionary. If you want to apply this methodology to data for whichever language you are working with, then you will need to declare the legal consonant sequences in a similar way, using whatever machine readable phonetic notation you have used in your dictionary.

The required modifications to the Tcl script (shown in bold) involve the procedures source to read in `clusters.txt` containing the statements of legal consonant clusters and Syllabify to syllabify word-internally according to the MOP.

```
package require emu::autobuild
proc AutoBuildInit {template} {}

proc AutoBuild {template tree} {
set wordpath x/timetable/orthography
AddLabelsFromFile $template $tree $wordpath Text
InitialiseDict lex x/timetable/dictgerman.txt
LevelFromDict $tree Text Phoneme lex

# Read in the file clusters.txt
# containing the legal sequences of consonant clusters
source x/timetable/clusters.txt
# The 3rd and 4th arguments to Syllabify are the tier names to be linked; cons
# is the variable that is declared in the file clusters.txt
Syllabify $template $tree Phoneme Syllable cons
}
```

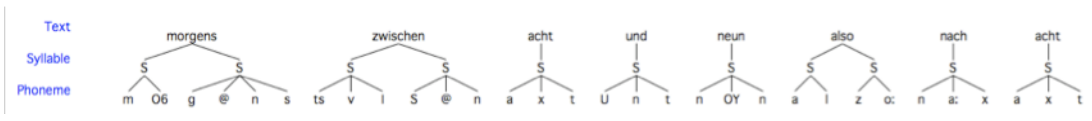


Fig. B.7. The result of applying Syllabify

Before you can run this script, Emu must also be told which phonemes can constitute possible syllable nuclei. This is done by naming a feature of the Phoneme tier that must be called `vowel` in the Legal labels pane of the template file. This has already been done for the `timetable` database, as is apparent in inspecting the Legal Labels pane of the template file (the file `vowelsgerman.txt` in `x/timetable` also contains a list of these vowel phonemes). If you are using these Tcl commands on a database from another language, then this feature will need to be modified, depending on which segments you wish to declare to be the possible syllable nuclei.

The effect of including the above commands in `emutcl.txt` and of clicking `Build Hierarchy` is to build the tree between `Word` and `Phoneme` levels shown in Fig. B.7.

The command InsertWordBoundaries accomplishes the final step of linking the Phoneme and Phonetic tiers (the command is a misnomer, because rather than inserting anything, it finds the best match between two strings of symbols). The discrepancies between the annotations at these two tiers come about because of reduction processes that are typical of spontaneous speech, of which some examples from the `timetable` database are shown in Table I.

Table I: The relationship between dictionary entries and the phonetic forms actually produced for some of the sentences of the Kiel Corpus of read speech.

Utterance	**Lexical entry**	**Lexical phonemic form**	**Phonetic form produced**	**Gloss**
HPTE001	morgen	mO6g@n = /mɔɐɡən/	mo6gN = [mɔɐɡŋ]	tomorrow
HPTE002	morgens	mO6g@ns = /mɔɐɡəns/	mO6Ns = [mɔɐŋs]	in the morning
HPTE004	gegen	ge:g@n = /ɡe:ɡən/	ghe:N = [ɡ^{x}e:ŋ]	about
HPTE004	jedenfalls	je:d@nfals = /je:dənfals/	je:nfals = [je:nfals]	in any case
HPTE005	möglichst	m2:klICst = /møklɪçst/	m2:klIC = [møklɪç]	as possible
HPTE005	Abend	a:b@nt = /a:bənt/	a:bn = [a:bn̩]	evening

So the matching algorithm has to be able to align, for example, the citation-form dictionary entry `ge:g@n` (row 3) with the form that was actually produced, `ghe:N` which includes an `h` to mark the frication stage of the /ɡ/-release, and in which citation-form `g@n` surfaces without the body of the syllable and with a progressively assimilated velar nasal `N`.

The matching algorithm (Harrington et al., 1993), which is based on a form of dynamic programming, requires as a minimum a two-columned table consisting of **all** unique occurrences of the phonemes in the lexicon and the most probable phonetic forms that they map onto. This is included in `x/timetable/phonemesgerman.txt`. For the most part, there is likely to be a one-to-one correspondence between the phoneme in the lexicon (left column) and the most likely corresponding phonetic form, for example:

```
Y6 Y6
6 6
@ @
p p
b b
t t
d d
k k
g g
pf p f
ts t s
h h
```

The same phoneme could be repeated if it is realized as more than one phonetic form in the database. For example, it would be possible to have:

```
E     E
E     EC
```

to denote that the phoneme `E` in the lexicon could be phonetically either `E` (modal voice) or `EC` (creaky voice). A phoneme can also map onto a sequence of two phonetic segments, as in the above example in which the affricate phoneme `ts` is realised phonetically as `t` followed by `s`. However, this type of implicit rewrite rule needs to be used with great care and generally only if it is **always** the case that such a correspondence can be made. So to have phonemic `p` map onto a sequence of phonetic `p h` to mark aspiration is inadvisable, given that stops are not aspirated in all contexts. In general, the safest option is the minimal one, in which the unique phonemes in the left column are duplicated in the right column. As described in further detail below, it is then possible to supplement these simple rewrite statements with some additional **phonetic-to-phoneme mapping rules** to model a certain amount of allophonic variation.

The required modification to the Tcl script to include this matching algorithm is as follows:

```
package require emu::autobuild
proc AutoBuildInit {template} {}

proc AutoBuild {template tree} {
set wordpath x/timetable/orthography
AddLabelsFromFile $template $tree $wordpath Text
InitialiseDict lex x/timetable/dictgerman.txt
LevelFromDict $tree Text Phoneme lex
source x/timetable/clusters.txt
Syllabify $template $tree Phoneme Syllable cons

# Read in the table of phoneme correspondences
InitialiseDict phonemes x/timetable/phonemesgerman.txt

# The third argument in the line below is the variable name phonemes
# defined above; Phoneme and Phonetic  are the two tiers to be linked
InsertWordBoundaries $template $tree phonemes Phoneme Phonetic
}
```

Fig. B.8 shows that the matching algorithm has done a reasonable job of parsing the evident mismatches between the Phonetic and Phoneme tiers for some words in the fourth utterance: it has coped with the deletions in *morgens*, *gegen*, and *jedenfalls*, and it has associated the velar nasal `N` at the Phonetic tier with `n` at the Phoneme tier (it has also parsed the Phonetic sequence `t s` as the Phoneme affricate `ts` but this is to be expected because it was explicitly listed in the phoneme table of correspondences discussed earlier). The *, denoting silence, has been associated with the final `n` of *zehn* (*ten*): in fact the matching algorithm cannot get the right answer here, firstly because the parsing is exhaustive (every segment on the Phonetic tier is parsed) and also because * is not listed in the lexicon as a separate entry.