Appendix D

EQL EBNF

This chapter presents the Extended Backus-Naur Form (EBNF) that describes the EQL. As the original EBNF adapted from John (2012) was written in German, some of the abbreviation terms were translated into English abbreviations (e.g., `DOMA` is the abbreviation for the German term **Dom**inanz**a**bfrage and the newly translated `DOMQ` is the abbreviation for the English term **dom**ination **q**uery).

D.0.1 Terminal symbols of EQL2 (operators) and their meaning

The terminal symbols described below are listed in descending order by their binding priority.

Symbol	Meaning
`#`	Result modifier (projection)
`,`	Parameter list separator
`==`	Equality (new in version 2 of the EQL; added for cleaner syntax)
`=`	Equality (optional; for backwards compatibility)
`!=`	Inequality
`=~`	Regular expression matching
`!~`	Regular expression non-matching
>	Greater than
>=	Equal to or greater than
<	Less than
>=	Equal to or less than

Symbol	Meaning
|	Alternatives separator
&	Conjunction of equal rank
^	Dominance conjunction
->	Sequence operator

D.0.2 Terminal symbols of EQL2 (brackets) and their meanings.

Symbol	Meaning
'	Quotes literal string
(	Function parameter list opening bracket
)	Function parameter list closing bracket
[	Sequence or dominance-enclosing opening bracket
]	Sequence or dominance-enclosing closing bracket

D.0.3 Terminal symbols of EQL2 (functions) and their meanings.

Symbol	Meaning
Start	Start
Medial	Medial
End	Final
Num	Count

D.0.4 Formal description of EMU Query Language Version 2

EBNF term	Abbreviation	Conditions
EQL = CONJQ | SEQQ | DOMQ;	**E**MU **Q**uery **L**anguage	

EBNF term	Abbreviation	Conditions
`DOMQ = "[", ( CONJQ | DOMQ | SEQQ ), "^", ( CONJQ | DOMQ | SEQQ ), "]";`	**dom**inance **q**uery	levels must be hierarchically associated
`SEQQ = "[", ( CONJQ | SEQQ | DOMQ ), "->", ( CONJQ | SEQQ | DOMQ ), "]";`	**seq**uential **q**uery	levels must be linearly associated
`CONJQ = { "[" }, SQ, { "&", SQ }, { "]" };`	**conj**unction **q**uery	levels must be linearly associated
`SQ = LABELQ | FUNCQ;`	**s**imple **q**uery	
`LABELQ = [ "#" ], LEVEL, ( "=" | "==" | "!=" | "=~" | "!~" ), LABELALTERNATIVES;`	**label q**uery	
`FUNCQ = POSQ | NUMQ;`	**fun**ction **q**uery	
`POSQ = POSFCT, "(", LEVEL, ",", LEVEL, ")", "=", "0" | "1";`	**position q**uery	levels must be hierarchically associated; second level determines semantics
`NUMQ = "Num", "(", LEVEL, ",", LEVEL, ")", COP, INTPN;`	**number q**uery	levels must be hierarchically associated; first level determines semantics
`LABELALTERNATIVES = LABEL , { "|", LABEL };`	**label alternatives**	
`LABEL = LABELING | ( "'", LABELING, "'" );`	**label**	levels must be part of the database structure; `LABELING` is an arbitrary character string or a label group class configured in the emuDB; result modifier `#` may only occur once

EBNF term	Abbreviation	Conditions
`POSFCT = "Start" | "Medial" | "End";`	**pos**ition **f**un**ct**ion	
`COP = "=" | "==" | "!=" | ">" | "<" | "<=" | ">=";`	**c**omparison **o**perator	
`INTPN = "0" | INTP;`	**int**eger **p**ositive with **n**ull	
`INTP = DIGIT-"0", { DIGIT };`	**int**eger **p**ositive	
`DIGIT = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9";`	**digit**	

INFO: The `LABELING` term used in the `LABEL` EBNF term can represent any character string that is present in the annotation. As this can be any combination of Unicode characters, we chose not to explicitly list them as part of the EBNF.

D.0.5 Restrictions

A query may only contain a single result modifier `#` (hashtag).

Appendix E

EQL: further examples

Below are examples of query strings that have been adapted from Cassidy and Harrington (2001) and Harrington and Cassidy (2002) and which are displayed as questions and answers. All examples use the *ae* demo `emuDB`, which is provided by the `emuR` package, and were extracted from the `EQL` vignette of the `emuR` package. Descriptions (some of them duplicates of those in Chapter 5) of the various syntaxes and query components are also included for easier reading. R Example E.0.1 shows how to access the *ae* demo `emuDB`.

R Example E.0.1

```
# load the package
library(emuR)

# create demo data in directory provided by the tempdir() function
create_emuRdemoData(dir = tempdir())

# get the path to emuDB called 'ae' that is part of the demo data
path2directory = file.path(tempdir(), "emuR_demoData", "ae_emuDB")

# load emuDB into current R session
ae = load_emuDB(path2directory)
```

E.0.1 Simple equality, inequality, matching and non-matching queries (single-argument)

The syntax of a simple, equality, inequality, matching and non-matching query is `[L OPERATOR A]` where `L` specifies a level (or alternatively the name of a parallel attribute definition), `OPERATOR` is one of the following operators: `==` (equality); `!=` (inequality); `=~` (matching) or `!~` (non-matching), and `A` is an expression specifying the labels of the annotation items of `L`.

Example questions and answers:

- **Q**: *What is the query to retrieve all items containing the label "m" in the "Phonetic" level?*
- **A**:

```
query(emuDBhandle = ae,
      query = "[Phonetic == m]")
```

- **Q**: *What is the query to retrieve all items containing the label "m" or "n" in the "Phonetic" level?*
- **A**:

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

- **Q**: *What is the query to retrieve all items that do not contain the label "m" or "n"?*
- **A**:

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

- **Q**: *What is the query to retrieve all items in the "Syllable" level?*
- **A**:

```
query(emuDBhandle = ae,
      query = "[Syllable =~ .*]")
```

- **Q**: *What is the query to retrieve all items that begin with "a" in the "Text" level?*
- **A**:

```
query(ae, "[Text =~ a.*]")
```

- **Q**: *What is the query to retrieve all items that do not begin with "a" in the "Text" level?*
- **A**:

```
query(ae, "[Text !~ a.*]")
```

The above examples use three operators that are new to the EQL as of version 2. One is the `==` equal operator, which has the same meaning as the `=` operator of the EQL version 1 (which is also still available) while providing a cleaner, more precise syntax. The other two are `=~` and `!~`, which are the new matching and non-matching regular expression operators. Further, it is worth noting that the use of parentheses, blanks or characters that represent operands used by the EQL (see EBNF in Appendix D) as part of a label matching string (the string on the right hand side of one of the operands mentioned above), must be placed in additional single quotation marks to escape these characters. For example, searching for the items containing the labels `O_'` on the Phonetic level could not be written as `"[Phonetic == O_']"` but would have to be written as `"[Phonetic == 'O_'']"`. Reversing the order of single vs. double quotation marks is currently not supported, that is `'[Phonetic == ''O_''']'` will currently not work. Hence, to avoid this issue only double quotation marks for the outer wrapping of the query string should be used.

E.0.2 Sequence queries using the -> sequence operator

The syntax of a query string using the `->` sequence operator is `[L == A -> L == B]`, where item `A` on level `L` precedes item `B` on level `L`. For a sequential query to

work both arguments must be on the same level (alternatively, parallel attribute definitions of the same level may also be chosen).

Example **Q** & **A**'s:

- **Q**: *What is the query to retrieve all sequences of items containing the label "@" followed by items containing the label "n" on the "Phonetic" level?*
- **A**:

```
# NOTE: all row entries in the resulting
# segment list have the start time of "@", the
# end time of "n", and their labels will be "@->n"
query(ae, "[Phonetic == @ -> Phonetic == n]")
```

- **Q**: *Same as the question above but this time we are only interested in the items containing the label "@" in the sequences.*
- **A**:

```
# NOTE: all row entries in the resulting
# segment list have the start time of "@", the
# end time of "@" and their labels will also be "@"
query(ae, "[#Phonetic == @ -> Phonetic == n]")
```

- **Q**: *Same as the first question but this time we are only interested in the items containing the label "n".*
- **A**:

```
# NOTE: all row entries in the resulting
# segment list have the start time of "n", the
# end time of "n" and their labels will also be "n"
query(ae, "[Phonetic == @ -> #Phonetic == n]")
```

Subsequent sequence queries using nesting of the -> sequence operator

The general strategy for constructing a query string that retrieves subsequent sequences of labels is to nest multiple sequences while paying close attention to the correct placement of the parentheses. An abstract version of such a query string for the subsequent sequence of arguments `A1`, `A2`, `A3` and `A4` would be: `[[[[A1 -> A2] -> A3] -> A4] -> A5]` where each argument (e.g. `A1`) represents an equality, inequality, matching or non-matching expression on the same level (alternatively, parallel attribute definitions of the same level may also be chosen).

Example questions and answers:

- **Q**: *What is the query to retrieve all sequences of items containing the labels "@", "n" and "s" on the "Phonetic" level?*
- **A**:

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

- **Q**: *What is the query to retrieve all sequences of items containing the labels "to", "offer" and "any" on the "Text" level?*
- **A**:

```
query(ae, "[[Text == to -> Text == offer] -> Text == any]")
```

- **Q**: *What is the query to retrieve all sequences of items containing labels "offer" followed by two arbitrary labels followed by "resistance"?*
- **A**:

```
# NOTE: usage of paste0() is optional
# as it is only used for formating purposes
query(ae, paste0("[[[Text == offer -> Text => .*] ",
                 "-> Text => .*] -> Text == resistance]"))
```

As the EQL1 did not have a regular expression operator, users often resorted to using queries such as `[Phonetic != XXX]` (where XXX is a label that was not part of the label set of the *Phonetic* level) to match every label on the `Phonetic` level. Although this is still possible in the EQL2, we strongly recommend using regular expressions as they provide a much clearer and more precise syntax and are less error-prone.

E.0.3 Conjunction operator &

The syntax of a query string using the conjunction operator can schematically be written as: `[L == A & L_a2 == B & L_a3 == C & L_a4 == D & ... & L_an == N]`, where items on level `L` have the label `A` (technically belonging to the first attribute of that level, i.e., `L_a1`, which per default has the same name as its level) also have the attributes `B`, `C`, `D`, ..., `N`. As with the sequence operator all expressions must be on the same level (i.e., parallel attribute definitions of the same level indicated by the `a2 - an` may to be chosen).

The conjunction operator is used to combine query conditions on the same level. This makes sense in two cases:

1. when to combining different attributes of the same level: `"[Phonetic == l & sonorant == T]"` when *Sonorant* is an additional attribute of level *Phonetic*;
2. when combining a basic query with a function (see sections Position and Count below): `"[phonetic == l & Start(word, phonetic) == 1]"`.

Example questions and answers:

- **Q**: *What is the query to retrieve all items containing the label "always" in the "Text" attribute definition which also have the label "C" on a parallel attribute definition called "Word"?*
- **A**:

```
query(ae, "[Text == always & Word == C]")
```

- **Q**: *What is the query to retrieve all items of the attribute definition "Text" of the level "Word" that were also labeled as function words (labeled "F" in the "Word" level)?*
- **A**:

```
query(ae, "[Text =~ .* & Word == F]")
```

- **Q**: *What is the query to retrieve all items of the attribute definition "Text" of the level "Word" that were also labeled as content words (labeled "C" in the "Word" level) and as accented (labeled "S" in the attribute definition "Accent" of the same level)?*

- **A**:

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

E.0.4 Domination operator ^ (hierarchical queries)

A schematic representation of a simple domination query string that retrieves all items containing label `A` of level `L1` that are dominated by (i.e., are directly or indirectly linked to) items containing the label `B` in level `L2` is `[L1 == A ^ L2 == B]`. The domination operator is not directional, meaning that either items in `L1` dominate items in `L2` or items in `L2` dominate items in `L1`. Note that link definitions that specify the validity of the domination have to be present in the emuDB for this to work.

Simple domination

Example questions and answers:

- **Q**: *What is the query to retrieve all items containing the label "p" in the "Phoneme" level that occur in strong syllables (i.e., dominated by/linked to items of the level "Syllable" that contain the label "S")?*
- **A**:

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

- **Q**: *What is the query to retrieve all syllable items which contain a Phoneme item labeled "p"?*
- **A**:

```
query(ae, "[Syllable =~ .* ^ Phoneme == p]")
# or
query(ae, "[Phoneme == p ^ #Syllable =~ .*]")
```

- **Q**: *What is the query to retrieve all syllable items which do not contain a Phoneme item labeled "k" or "p" or "t"?*

- **A**:

```
query(ae, "[Syllable =~ .* ^ Phoneme != p | t| k]")
# or
query(ae, "[Phoneme != p | t | k ^ #Syllable =~ .*]")
```

Even though the domination operator is not directional, what you place to the left and right of the operator does have an impact on the result. If no result modifier (the hash tag `#`) is used, the query engine will automatically assume that the expression to the left of the operator specifies what is to be returned. This means that the schematic query string `[L1 == A ^ L2 == B]` is semantically equal to the query string `[#L1 == A ^ L2 == B]`. As it is more explicit to mark the desired result we recommend you always use the result modifier where possible.

Multiple domination

The general strategy when constructing a query string that specifies multiple domination relations of items is to nest multiple domination expressions while paying close attention to the correct placement of the parentheses. A dominance relationship sequence or the arguments `A1`, `A2`, `A3`, `A4`, can therefore be noted as: `"[[[[A1 ^ A2] ^ A3] ^ A4] ^ A5]"` where `A1` is dominated by `A2` and `A3` and so on.

Example questions and answers:

- **Q**: *What is the query to retrieve all items on the "Phonetic" level that are part of a strong syllable (labeled "S") and belong to the words "amongst" or "beautiful"?*
- **A**:

```
# NOTE: usage of R's paste0() function is optional
# as it is only used for formatting purposes
query(ae, paste0("[[#Phonetic =~ .* ^ Syllable == S] ",
                 "^ Text == amongst | beautiful]"))
```

- **Q**: *The same as the question above but this time we want the "Text" items.*
- **A**:

```
# NOTE: usage of R's paste0() function is optional
# as it is only used for formatting purposes
query(ae, paste0("[[Phonetic =~ .* ^ Syllable == S] ",
                 "^ #Text == amongst | beautiful]"))
```

E.0.5 Position

The EQL has three function terms to specify where in a domination relationship a child level item is allowed to occur. The three function terms are `Start()`, `End()` and `Medial()`.

Simple usage of `Start()`, `End()` and `Medial()`

A schematic representation of a query string representing a simple usage of the `Start()`, `End()` and `Medial()` function would be: `"POSFCT(L1, L2) == 1"` or `"POSFCT(L1, L2) == TRUE"`. In this representation `POSFCT` is a placeholder for one of the three functions where the level `L1` must dominate level `L2`. The `== 1` / `== TRUE` part of the query string indicates that if a match is found (match is `TRUE` or `== 1`), the according item of level `L2` is returned. If this expression is set to `== 0` / `== FALSE` (`FALSE`), all the items that do not match the condition of `L2` will be returned. A visualization of what is returned by the various options of the three functions is displayed in Figure 5.4.

As using `1` and `0` for `TRUE` and `FALSE` is not that intuitive to many R users, the EQL version 2 optionally allows for the values `TRUE`/`T` and `FALSE`/`F` to be used instead of `1` and `0`. This syntax should be more familiar to most R users.

Example questions and answers:

- **Q**: *What is the query to retrieve all word-initial syllables?*
- **A**:

```
query(ae, "[Start(Word, Syllable) == TRUE]")
```

- **Q**: *What is the query to retrieve all word-initial phonemes?*
- **A**:

```
query(ae, "[Start(Word, Phoneme) == TRUE]")
```

- **Q**: *What is the query to retrieve all non-word-initial syllables?*
- **A**:

```
query(ae, "[Start(Word, Syllable) == FALSE]")
```

- **Q**: *What is the query to retrieve all word-final syllables?*
- **A**:

```
query(ae, "[End(Word, Syllable) == TRUE]")
```

- **Q**: *What is the query to retrieve all word-medial syllables?*
- **A**:

```
query(ae, "[Medial(Word, Syllable) == TRUE]")
```

Position and boolean &

The syntax for combining a position function with the boolean operator is `[L == E & Start(L, L2) == TRUE]`, where item `E` on level `L` occurs at the beginning of item `L`. Once again, `L` has to dominate `L2` (optionally, parallel attribute definitions of the same level may also be chosen).

Example questions and answers:

- **Q**: *What is the query to retrieve all "n" Phoneme items at the beginning of a syllable?*
- **A**:

```
query(ae, "[Phoneme == n & Start(Syllable, Phoneme) == 1]")
```

- **Q**: *What is the query to retrieve all word-final "m" Phoneme items?*
- **A**:

```
query(ae, "[Phoneme == m & End(Word, Phoneme) == 1]")
```

- **Q**: *What is the query to retrieve all non-word-final "S" syllables?*
- **A**:

```
query(ae, "[Syllable == S & End(Word, Syllable) == 0]")
```

Position and boolean ^

The syntax for combining a position function with the boolean hierarchical operator is `[L == E ^ Start(L1, L2) == 1]`, where level `L` and level `L2` refer to different levels where either `L` dominates `L2`, or `L2` dominates `L`.

Example questions and answers:

- **Q**: *What is the query to retrieve all "p" Phoneme items which occur in the first syllable of the word?*
- **A**:

```
query(ae, "[Phoneme == p ^ Start(Word, Syllable) == 1]")
```

- **Q**: *What is the query to retrieve all phonemes which do not occur in the last syllable of the word?*
- **A**:

```
query(ae, "[Phoneme =~ .* ^ End(Word, Syllable) == 0]")
```

E.0.6 Count

A schematic representation of a query string using the count mechanism looks like `[Num(L1, L2) == N]`, where `L1` contains `N` items in `L2`. For this type of query to work, `L1` has to dominate `L2`. As the query matches a number (`N`), it is also possible to use the operators > (more than), < (less than) and `!=` (not equal). The resulting segment list contains items of `L1`.

Example questions and answers:

- **Q**: *What is the query to retrieve all words that contain four syllables?*
- **A**:

```
query(ae, "[Num(Word, Syllable) == 4]")
```

- **Q**: *What is the query to retrieve all syllables that contain more than six phonemes?*
- **A**:

```
query(ae, "[Num(Syllable, Phoneme) > 6]")
```

Count and boolean `&`

A schematic representation of a query string combining the count and the boolean operators looks like `[L == E & Num(L1, L2) == N]`, where items `E` on level `L` are dominated by `L1` and `L1` contains `N` `L2` items. Further, `L1` dominates `L2` on the condition that `L` and `L1` (not `L2`) refer to the same level (parallel attribute definitions of the same level may also be chosen).

Example questions and answers:

- **Q**: *What is the query to retrieve the "Text" of all words which consist of more than five phonemes?*
- **A**:

```
query(ae, "[Text =~ .* & Num(Text, Phoneme) > 5]")
# or
query(ae, "[Text =~ .* & Num(Word, Phoneme) > 5]")
```

- **Q**: *What is the query to retrieve all strong syllables that contain five phonemes?*
- **A**:

```
query(ae, "[Syllable == S & Num(Syllable, Phoneme) == 5]")
```

Count and ^

A schematic representation of a query string combining the count and the boolean operators is `[L == E ^ Num(L1, L2) == N]` where items `E` on level `L` are dominated by `L1` and `L1` contains `N` `L2` items. Further, `L1` dominates `L2` on the condition that `L` and `L1` do **not** refer to the same level.

Example questions and answers:

- **Q**: *What is the query to retrieve all "m" phonemes in three-syllable words?*
- **A**:

```
query(ae, "[Phoneme == m ^ Num(Word, Syllable) == 3]")
```

- **Q**: *What is the query to retrieve all "W" syllables in words of three syllables or less?*
- **A**:

```
query(ae, "[Syllable = W ^ Num(Word, Syllable) <= 3]")
```

- **Q**: *What is the query to retrieve all words containing syllables which contain four phonemes?*
- **A**:

```
query(ae, "[Text =~ .* ^ Num(Syllable, Phoneme) == 4]")
```

E.0.7 Combinations

^ and -> (domination and sequence)

A schematic representation of a query string combining the domination and the sequence operators is `[[A1 ^ A2] -> A3]`, where `A1` and `A3` refer to the same level (parallel attribute definitions of the same level may also be chosen).

Example questions and answers:

- **Q**: *What is the query to retrieve all "m" preceding "p" when "m" is part of an "S" syllable?*

- **A**:

```
query(ae, "[[Phoneme == m -> Phoneme =~ p] ^ Syllable == S]")
```

- **Q**: *What is the query to retrieve all "s" preceding "t" when "t" is part of a "W" syllable?*
- **A**:

```
query(ae, "[Phoneme == s -> [Phoneme == t ^ Syllable == W]]")
```

- **Q**: *What is the query to retrieve all "S" syllables, containing an "s" phoneme and preceding an "S" syllable?*
- **A**:

```
query(ae, "[[#Syllable == S ^ Phoneme == s] -> Syllable == S]")
```

- **Q**: *Same question as above but this time we want all "s" items where "s" is part of a "S" syllable and the "S" syllable precedes another "S" syllable.*
- **A**: `"[[Phoneme == s ^ Syllable == S] -> Syllable == S]"` would cause an error as `Phoneme == s` and `Syllable == S` are not on the same level. Therefore, the correct answer is:

```
query(ae, "[[Syllable == S ^ #Phoneme == s] -> Syllable == S]")
```

^ and -> and & (domination and sequence and boolean &)

Example questions and answers:

- **Q**: *What is the query to retrieve the "Text" of all words beginning with a "@" on the "Phoneme" level?*
- **A**:

```
# NOTE: usage of paste0() is optional
# as it is only used for formatting purposes
query(ae, paste0("[Text =~ .* ^ Phoneme == @ ",
                 "& Start(Text, Phoneme) == 1]"))
```

- **Q**: *What is the query to retrieve all word-initial "m" items in a "S" syllable preceding "o:"?*
- **A**:

```
# NOTE: usage of paste0() is optional
# as it is only used for formatting purposes
query(ae, paste0("[[Phoneme == m & Start(Word, Phoneme) == 1 ",
                 "-> Phoneme == o:] ^ Syllable == S]"))
```

- **Q**: *Same question as the question above, but this time we want the "Text" items.*
- **A**:

```
# NOTE: usage of paste0() is optional
# as it is only used for formatting purposes
query(ae, paste0("[[[Phoneme == m & Start(Word, Phoneme) == 1 ",
                 "-> Phoneme == o:] ^ Syllable == S] ",
                 "^ #Text =~ .*]"))
```

E.0.8 A few more questions and answers (because practice makes perfect)

- **Q**: *What is the query to retrieve all "m" or "n" phonemes which occur in the word-medial position?*
- **A**:

```
query(ae, "[Phoneme == m | n & Medial(Word, Phoneme) == 1]")
```

- **Q**: *What is the query to retrieve all "H" phonetic segments followed by an arbitrary segment and then by either "I" or "U"?*
- **A**:

```
# NOTE: usage of paste0() is optional
# as it is only used for formatting purposes
query(ae, paste0("[[Phonetic == H -> Phonetic =~ .*] ",
                 "-> Phonetic == I | U]"))
```

- **Q**: *What is the query to retrieve all syllables which do not occur in word-medial positions?*
- **A**:

```
query(ae, "[Syllable =~ .* & Medial(Word, Syllable) == 0]")
```

- **Q**: *What is the query to retrieve the "Text" items of all words containing two syllables?*
- **A**:

```
query(ae, "[Text =~ .* & Num(Text, Syllable) == 2]")
```

- **Q**: *What is the query to retrieve the "Text" items of all accented words following "the"?*
- **A**:

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

- **Q**: *What is the query to retrieve all "S" (strong) syllables consisting of five phonemes?*
- **A**:

```
query(ae, "[Syllable = S ^ Num(Word, Phoneme) == 5]")
```

- **Q**: *What is the query to retrieve all "W" (weak) syllables containing a "@" phoneme?*
- **A**:

```
query(ae, "[Syllable == W ^ Phoneme == @]")
```

- **Q**: *What is the query to retrieve all Phonetic items belonging to a "W" (weak) syllable?*
- **A**:

```
query(ae,"[Phonetic =~ .* ^ #Syllable == W]")
```

- **Q**: *What is the query to retrieve "W" (weak) syllables in word-final position occurring in three-syllable words?*
- **A**:

```
# NOTE: usage of paste0() is optional
# as it is only used for formatting purposes
query(ae, paste0("[Syllable == W & End(Word, Syllable) == 1 ",
                 "^ Num(Word, Syllable) == 3]"))
```

- **Q**: *What is the query to retrieve all phonemes dominating "H" Phonetic items at the beginning of a syllable and occurring in accented ("S") words?*
- **A**:

```
# NOTE: usage of paste0() is optional
# as it is only used for formatting purposes
query(ae, paste0("[[[Phoneme =~ .* ^ Phonetic == H] ",
                 "^ Start(Word, Syllable) == 1] ^ Accent == S]"))
```

E.1 Differences to the legacy EMU query language

In this section summarizes the major changes concerning the query mechanics of `emuR` compared to the legacy R package `emu` Version 4.2. This section is mainly aimed at users transitioning to `emuR` from the legacy system.

E.1.1 Function call syntax

In `emuR` it is necessary to load an `emuDB` into the current R session before being able to use the `query()` function. This is achieved using the `load_emuDB()` function. This was not necessary using the legacy `emu.query()` function.

E.1.2 Empty result

The `query` function of `emuR` returns an empty segment list (row count is zero) if the query does not match any items. If the legacy EMU function `emu.query()` did not find any matches it, returned an error with the message:

```
## Can't find the query results in emu.query: there may have
## been a problem with the query command.
```

E.1.3 The result modifier hash tag

Compared to the legacy EMU system, which allowed multiple occurrences of the hash tag `#` to be present in a query string, the `query()` function only allows a single result modifier. This ensures that only consistent result sets are returned (i.e., all items belong to a single level). However, if multiple result sets in one segment list are desired, this can easily be achieved by concatenating the result sets of separate queries using the `rbind()` function.

E.1.4 Interpretation of the hash tag # in conjunction operator queries

legacy EMU

```
emu.query(template = "andosl",
          pattern = "*",
          query = "[Text=spring & #Accent=S]")}
```

yielded:

```
## moving data from Tcl to R
## Read 1 records
## segment  list from database:  andosl
## query was:  [Text=spring & #Accent=S]
##   labels    start      end     utts
## 1 spring 2288.959 2704.466 msajc094
```

and

```
emu.query(template = "andosl",
          pattern = "*",
          query = "[#Text=spring & #Accent=S]")
```

yielded the identical:

```
## moving data from Tcl to R
## Read 1 records
## segment  list from database:  andosl
## query was:  [#Text=spring & #Accent=S]
##   labels    start      end     utts
## 1 spring 2288.959 2704.466 msajc094
```

Hence, the hash tag **#** had no effect.

emuR

```
query(emuDBhandle = andosl,
      query = "[Text == spring & #Accent == S]",
      resultType = "emusegs")
```

```
## segment  list from database:  andosl
## query was:  [Text=spring & #Accent=S]
##   labels    start      end          utts
## 1      S 2288.975 2704.475 0000:msajc094
```

Returns the same item but with the label of the hashed attribute definition name. The second legacy example is not a valid `emuR` query (two hash tags) and will return an error message.

```
query(dbName = "andosl",
      query = "[#Text == spring & #Accent == S]")
```

```
## Error in query.database.eql.KONJA(dbConfig, qTrim) :
## Only one hashtag allowed in linear query term: #Text=spring & #Accent=S
```

E.1.5 Bugs in legacy EMU function `emu.query()`

Alternative labels in inequality queries

Example:

legacy EMU

It appears that the OR operator | was mistakenly ignored when used in conjunction with the inequality operator `!=`:

```
emu.query(template = "ae",
          pattern = "*",
          query = "[Text != beautiful | futile ^ Phoneme = u:]")
```

yielded:

```
## moving data from Tcl to R
## Read 4 records
## segment  list from database:  ae
## query was:  [Text!=beautiful|futile ^ Phoneme=u:]
```

```
##      labels    start      end     utts
## 1       new  475.802  666.743 msajc057
## 2    futile  571.999 1091.000 msajc010
## 3        to 1091.000 1222.389 msajc010
## 4 beautiful 2033.739 2604.489 msajc003
```

emuR

The query engine of the `emuR` package respects the presence of the OR operator in such queries:

```
query(emuDBhandle = ae,
      query = "[Text != beautiful | futile ^ Phoneme == u:]",
      resultType = "emusegs")
```

```
## segment  list from database:  ae
## query was:  [Text!=beautiful|futile ^ Phoneme=u:]
##   labels    start      end          utts
## 1     to 1091.025 1222.375 0000:msajc010
## 2    new  475.825  666.725 0000:msajc057
```

Errors caused by missing or superfluous blanks or parentheses

Some queries in the legacy EMU system required blanks around certain operators to be present or absent as well as parentheses to be present or absent. If this was not the case the legacy query engine sometimes returned cryptic errors, sometimes crashing the current R session. The query engine of the `emuR` package is much more robust against missing or superfluous blanks or parentheses.

Order of result segment list

To our knowledge, the order of a segment list in the legacy EMU system was never predictable or explicitly defined. In the new system, if the result type of the `query()` function is set to `"emuRsegs"` the resulting list is ordered by UUID, session, bundle and sample start position. If the parameter `calcTimes` is set to `FALSE` it is ordered by UUID, session, bundle, level, seq_idx. If it is set to `"emusegs"` the resulting list is ordered by the fields utts and start.

Additional features

- The query mechanics of `emuR` accepts the double equal character string `==` (recommended) as well as the single `=` equal character string as an equal operator.
- The EQL2 is capable of querying labels by matching regular expressions using the `=~` (matching) and `!~` (non-matching) operators.
    - For example: `query("andosl", "Text =~ .*tz.*")`