MMAXQL
The MMAX2 Query Language
Reference Manual (draft)

© Christoph Müller
EML Research gGmbH
http://mmax.eml-research.de

3rd February 2005

Contents

1 Introduction

1.1 About MMAXQL

This document describes MMAXQL, the Multi-Level Query Language of the MMAX2 annotation tool. MMAX2 is a highly customizable annotation tool for creating, browsing, visualizing and querying linguistic annotations on multiple levels. The purpose of MMAXQL is to provide a means to perform queries on annotated MMAX2 documents. MMAXQL queries are mainly used for two purposes:

1. As input to the MMAX2 query console, for detecting, browsing and quantifying (sets of) markables with certain features, and

2. as part of the *pattern* specification in Markable Customizations, for creating markable sub-sets to which certain display styles are to be applied.

This document describes mainly the first of the above uses of MMAXQL. The central concepts, like e.g. how to specify attribute names and target values, are identical for both purposes anyway. Differences are detailed in Section 2.7.
The query examples described in this document use either the HTC (Heidelberg Text Corpus) or the Trainline[1] sample document supplied with the MMAX2 distribution. The MMAX2 query console can be opened by selecting the *Query Console* item in the *Tools* menu in the main application window.

1.2 Basics (updated for version 1.0 beta 4)

In MMAX2, annotation data is stored in so-called *markables*. A markable is an only formally defined entity which singles out and uniquely identifies one or more elements from the base data, i.e. from the words to which the annotation is added. Normally, each markable has at least one *feature* associated with it. In MMAX2, features can be either simple *attribute-value pairs* (e.g. `pos=det`), or *relations* that hold between one markable and one or more other markables. In the HTC data, e.g. a relation of type MARKABLE_SET is used to model coreference, and a relation of type MARKABLE_POINTER represents bridging relations. The Trainline data does not (yet) contain any markable relations.
The features (or feature combinations) that are allowed for a given markable are defined in an *annotation scheme*. The annotation scheme specifies the names of attributes that are applicable to a given markable, as well as the range of possible values that this attribute may have. It also specifies if and under which circumstances a markable can take part in some markable relation.
Markables are organized in *markable (or annotation) levels*. Each level contains markables of a particular type, i.e. all markables on the same level share the same annotation scheme. Each markable is associated with exactly one annotation level and thus exactly one annotation scheme.
The point of querying is to find (sets of) markables based on the features they have.
The result of a MMAXQL query is a list of markable *tuples*, i.e. a list of *n*-element ordered sets. In the most simple case, when only one markable level is concerned (cf. Section 2), each tuple contains only one element, i.e. the matching markable from the level under consideration. In more complex cases, where more than one markable level is concerned (cf. Section 4), each tuple contains two or more elements. The result tuples are displayed in the lower part of the MMAX2 query console in the form of a table with *n* columns. Each column has a header which contains the *name* of the markable level the markables in

[1] Taken from the SPAAC project, http://www.ling.lancs.ac.uk/groups/spaac/SPAAC.htm

this column come from, and the zero-based *column index* uniquely identifying each column. The header information is required when accessing particular columns within query results (cf. Section 4.2).
From version 1.0 beta 4 on, query results can now also be navigated: selecting a cell in the result table will highlight the corresponding markable in the MMAX2 display. For results of tuples with $n > 1$, this means that the table can be navigated both horizontally and vertically. This is particularly useful for results of queries related to markable sets (cf. Section 3): Moving up and down in the list will move from one set to the other, while moving left and right will step through the individual members of the set in the currently selected row.

2 Querying Markables on a Single Level

The most simple type of query is one that considers markables on a single level only. The syntax for this type of query runs as follows:

```
command [[attributes]] {level_name|set_query} [condition];
```

The content of each of these sub-expressions is detailed in the following.

2.1 `command`

The `command` part, i.e. the first part of each query, is an expression which tells the query interpreter what to do with the query result. Giving an explicit command is required for each query to be valid. At present, three types of commands are supported.
In the most simple case, the result can be displayed as a document-order list in the *Markable Tuples* tab in the lower part of the query console. For this, the obviously-named command `display` has to be used. The range of attributes to be displayed in the result can be controlled by the `attributes` parameter, cf. Section 2.2.
Using the command `statistics`, on the other hand, will not give the result list itself, but simply print to the *Statistics* tab some simple descriptive statistics about the result, including all attributes with their absolute and relative value frequencies as found in all markables in the result list.[2] By default, all attributes of the markables found in the result list will be considered. However, the range of attributes in the statistics report can be controlled by the `attributes` parameter, cf. Section 2.2. When the result of a set query is submitted to the `statistics` function, the statistic will consider features pertaining to markable sets, like average length etc.
As a third alternative, the query result can be assigned to a variable for later use. This can be done by using the `let` command, followed by a white space, the variable name, and the equals sign (=). Variable names must start with a $ sign. Variables need not be predefined or initialized, they are simply created when some value is assigned to them. A variable lives as long as the MMAX2 session lasts, or until it is explicitly overwritten. The MMAX2 query console has a special *debug* mode which can be used in connection with variables. If the debug mode is active, the results of variable assignments will also be displayed in the lower part of the MMAX2 query console. Use the command `debug` to activate and the command `nodebug` to deactivate this mode. The debug mode is particularly useful when incrementally building up complex queries from simpler ones (cf. Section 4.2).

2.2 `attributes` (updated for version 1.0 beta 4)

This is a new optional parameter which allows to specify a range of attributes. This can be used for mainly two purposes: Specifying which attribute values are to be displayed in the query result table along with the markable text, and specifying which attributes the statistics report should be restricted to. The command has to be given as a comma-separated list, and must be included in square brackets.

2.3 `level_name`

The `command` part of a query can be followed by an expression designating the markable level to which the query is to be applied. Markable levels are simply identified by their names, as they appear both in the

[2] In future versions, more refined statistics measures will be added.

markable level control window and on the tab panel elements in the MMAX2 attribute window. Markable level names are case-insensitive. If the supplied markable level name is not found in the currently opened MMAX2 document, a message will be displayed.
In addition to explicitly giving the name of a markable level, there is also the *default* markable level, which is initially set to the top-most markable level. The default markable level is referenced by using a single period (.) as the `level_name` value. The setting for the default markable level can be changed by using the *Default Level* sub-menu in the *Settings* menu in the query console window.

2.4 `set_query`

Alternatively, the `command` part of a query can also be followed by the name of a certain type of query for retrieving relations of type MARKABLE_SET. See Section 3.

2.5 `condition`

The `condition` part is the only optional (but cf. below) and at the same time most complex part of a query. If it is left out, all markables on the level `level_name` are submitted to the command specified in `command`. Thus, in the Trainline document,

```
display utterances;
```

is a completely well-formed (yet very simple) query, which displays all markables on the level *utterances*. Similarly,

```
let $all_turns = .;
```

is also well-formed and assigns all markables on the default level (supposing the *turns* level is still set to be default) to the variable $all_turns.
In most cases, however, a query will indeed contain a `condition` part. The function of the condition is to narrow down the set of markable tuples returned to the `command`. It does so by specifying matching conditions that markables must fulfill. The types of conditions that can be specified depend on the type of attribute they relate to.
When performing a query to retrieve relations of type MARKABLE_SET, the `condition` part is *not* optional. See Section 3.

2.5.1 Nominal and Freetext Attributes

Nominal attributes are those for which the annotation scheme defines a closed set of possible values. In the MMAX2 attribute window, attributes of this type are displayed either as lists of radio buttons (nominal_button) or as drop-down list boxes (nominal_list). For freetext attributes, on the other hand, there is no predefined set of possible values. Instead, they can take any string value. Freetext attributes are displayed in the attribute window by means of a text field.
One way to query nominal and freetext attributes is by giving the attribute name and one or more values to match. In the Trainline document, e.g. the query

```
display utterances (type={q-wh, q-yn});
```

will display all markables on the level *utterances* whose *type* value is either *q-wh* or *q-yn*. The enumeration within the curly brackets is interpreted as OR-connected.
If an attribute of the supplied name is not defined on the specified markable level, a message is displayed. The same happens if an attribute is nominal and the specified value does not match any of the values defined for it. In both cases, no query will be performed.
A condition can be negated by prepending a ! sign to the attribute name. If what is negated is an enumeration, deMorgan's Law is applied. Thus, in the Trainline document,

```
display utterances (!type={q-wh, q-yn});
```

will display all markables on the level *utterances* whose *type* value is neither *q-wh* nor *q-yn*.
A second way to query nominal and freetext attributes is by means of a regular expression. In order to specify that a condition is to be interpreted as a regular expression match, an asterisk (*) has to be prepended to the attribute name. **Note:** If a regular expression match is negated, the negation sign ! must appear BEFORE the asterisk! Thus, in the Trainline document,

```
display utterances (*topic={.*date.*});
```

is a way to retrieve all markables from the level *utterances* which have in their (freetext) *topic* attribute a text that contains the substring *date*. In principle, regular expression matches can also be used for nominal attributes. However, in cases where the same result can be obtained by using the first query method (cf. above), using a regular expression is not really advisable. The power of regular expression queries is better used on either freetext attributes or the special `markable_text` attribute (cf. Section 2.5.2).

2.5.2 Special `markable_text` Attribute

`markable_text` is an attribute that is available on all markable levels. As the name suggests, it allows to formulate conditions based on the string that makes up the markable. The `markable_text` attribute can be used in queries just like nominal and freetext attributes (cf. Section 2.5.1). It is most useful, however, in combination with regular expression matches. E.g. on the Trainline document, the query

```
display utterances (*markable_text={.*please.*});
```

will return all markables from the level *utterances* that contain the string *please*.
Note: MMAX2 does not (yet) support full-text searches in the entire display. However, using the `markable_text` attribute and regular expression queries, this type of query can effectively be simulated. The only precondition is that there is at least one markable level which covers the entire set of base data elements. In the HTC and Trainline sample data, this is true for the *sentences* resp. *turns* level. Thus, in order to find all occurrences of e.g. numbers in the HTC document, the following query can be used:
Note also that discontinuous markables (supported from version 1.0 beta 4 on) contain the character sequence '] [' (incl. white space in between) at fragment boundaries.

```
display sentences (*markable_text={.*[0-9]+.*});
```

2.5.3 Markable Set Relations

Markable relations of type MARKABLE_SET allow the grouping of markables into markable sets. These sets are defined without any semantics, but simply state that an undirected and transitive relation holds

between all members of a set. On the individual markable, set membership is expressed by means of a particular attribute the value of which is shared by all set members. This value is only an identifier and cannot itself be queried directly (but cf. below). When querying markable relations of type MARKABLE_SET, the only admissible values are the keywords `empty`, `initial`, and `final`, or numbers.
The keyword `empty` means *empty set* and is matched by all markables that do not have a value in the corresponding attribute. Thus, on the HTC document,

```
let $singletons = coref (coref_class={empty});
```

will assign to the variable $singletons all markables from the level *coref* that are not coreferent with any other markable.
The keywords `initial` resp. `final` can be used to query markables depending on their position in a set (if any). Although the sets themselves are unordered, a relative ordering of the set members is established on the basis of the document order in which they appear. Thus, on the HTC document,

```
display coref (coref_class={initial});
```

will display all set-initial markables from the level *coref* (corresponding to the first mentions of entities referred to back later).
Numerical values are interpreted as set sizes. Thus, on the HTC document,

```
display coref (coref_class={2});
```

will display all markables from the level *coref* that are members in two-member sets. Note that since the result is not a list of *sets*, but a list of *markables* (resp. markable *tuples*), one set with two markables will produce two result list entries.
Enumerating several values, as well as negation, works the same as with nominal or freetext attributes (cf. Section 2.5.1). **Note:** It is important to understand the effects of negating a query to a markable relation of type MARKABLE_SET: In the HTC document, a query like

```
display coref (!coref_class={initial});
```

does display *all* non-initial markables, regardless of whether they are in some set at all! What one wants, however, is probably a list of non-initial markables *in a set*. This has to be made explicit in the query by combining the above condition with another one. How this is done is described in detail in Section 2.6.
Regular expression matching is also supported, but should not really be needed. However, it is with regular expression matching that queries for set identifiers can be formulated.

2.5.4 Markable Pointer Relations

Markable relations of type MARKABLE_POINTER allow the creation of links between one *source* and one or more *target* markables. As with MARKABLE_SET relations, no semantics is associated with these relations either. On the individual markable, the markable(s) it points to are represented as a (list of) markable IDs. This value cannot itself be queried directly (but cf. below). When querying markable relations of type MARKABLE_POINTER, the only admissible values are the keywords `empty` and `target`, or numbers.
The keyword `empty` is matched by all markables that do not point to some other markable(s). Again, negation works here as well. Thus, in the HTC document, the query

```
display coref (!bridging_antecedent={empty});
```

will display all markables from the level *coref* that do have a bridging antecedent.
The keyword `target` works the other way round: It will match all markables on the specified level that have other markables pointing to them. In the HTC document, the query

```
let $all_bridging_antes = coref (bridging_antecedent={target});
```

will assign to the variable $all_bridging_antes all markables on the level *coref* that are pointed to by other markables.
Numerical values are interpreted as the number of targeta a given markable points to. Thus, in the HTC document, the query

```
display coref (bridging_antecedent={1,2});
```

will display all markables from the level *coref* that point to one or two target markables.

2.6 Queries with Complex Conditions

The queries described so far were simple in the sense that their condition parts only adressed one attribute at a time. In order to formulate more interesting queries, however, one needs to be able to combine several conditions to a complex query, or rather, to a query with a complex condition part. This can be done by using the logical connectors AND (&&) and OR (||), as well as negation (!) and explicit bracketing.
If only two simple conditions are to be combined, no explicit bracketing is necessary. E.g., in the Trainline document, a query like

```
display utterances (sp-act={inform} and polarity={negative});
```

will display all markables from the *utterances* level that have the value *inform* in their speech-act tag and the value *negative* in their *polarity* attribute. As another example, this time from the HTC document, the query

```
display coref (!coref_class={empty} and !coref_class={initial});
```

will display those markables from the *coref* level that are part of a markable set, but which are non-initial.[3]
Negation can be applied not only to simple conditions, but also to complex ones. Thus, the above query could be rewritten as

```
display coref (!(coref_class={empty} or coref_class={initial}));
```

In this case, explicit bracketing is required to identify the scope of the negation. Note that the negation sign MUST NOT be prepended to the outer pair of brackets: these are used to identify the condition part as a whole. Instead, an inner pair of brackets must be used to wjoich the negation can then be prepended.
More than two conditions can be combined in a similar way, still without need to explicitly bracket them, as long as the same type of logical connector is used. Thus, the above HTC example could be modified to contain another AND-connected condition (this time using the equivalent &&) notation, like

[3] In other words, this query returns all *anaphoric* markables from the *coref* level.

```
display coref (!coref_class={empty} &&
               !coref_class={initial} &&
               np_form={ne});
```

This will display all coreferent, non-initial proper name markables.
The MMAXQL specification requires that all conditions within the same pair of brackets be connected with the same connector. Thus, explicit bracketing is also needed as soon as both AND and OR connectors are to be used in the same condition. If a query violates this requirement, a message is displayed, and no query is performed. As an example, in the HTC document, the query

```
display coref ((semantic_class={phys_obj} and agreement={3n}) or
                semantic_class={human});
```

will display all markables from the *coref* level that are tagged as either *physical_object* and *3rd ps neuter*, or which are tagged as *human*. If the brackets around the first AND-connected condition were missing, the query would not be performed, and a message reporting inconsistent connectors would be displayed.
Although conditions may have an arbitrary complexity and depth of embedding, it is advisable to try to keep them as simple as possible. One way to do this is to make use of the value enumeration feature for simple conditions. As was described in Section 2.5.1, value enumerations are interpreted as OR-connected. Thus, instead of writing

```
display coref ((agreement={3m} or agreement={3f}) and semantic_class={human});
```

one should use the (simpler yet equivalent)

```
display coref (agreement={3m,3f} and semantic_class={human});
```

notation, which will remove from the query an entire level of bracketing.

2.7 Using MMAXQL in Markable Customizations

Markable customizations are an instrument for creating permanent feature-dependent display styles for markables. These styles are permanent in the sense that they are applied to all matching markables simultaneously, and not just to the currently selected one. Markable customizations work by defining a set of customization rules, each of which defines a *feature pattern* and a *style* to be applied to all markables matching the pattern.
There is one set of customization rules for each markable level. They are stored in XML files that can be found in the `Customizations` subdirectory of the respective sample data set. All customization rules are applied sequentially, from top to bottom, and the styles defined by all matching rules will be *merged* to create the actual markable rendering style. Thus, customization rules work incrementally.
Patterns are specified by means of (somewhat simplified) MMAXQL expressions. The main difference is that in a customization pattern, neither the `command` nor the `level_name` part is necessary. The latter follows from the fact that each customization file is associated with exactly one markable level, so that no explicit markable level identification is required. It also follows from this that markable customizations on a given markable level *cannot access markables on another level*.
Apart from these differences, the syntax of MMAXQL expressions is the same. The customization rules actually used in the HTC sample document look as follows:

```
<rule pattern="{all}" style="foreground=blue"/>
<rule pattern="!coref_class={empty}" style="italics=true"/>
<rule pattern="type={bridging}" style="bold=true"/>
<rule pattern="type={bridging};bridging_antecedent={empty}" style="background=magenta"/>
<rule pattern="type={bridging};ante_sub_bridging={none}" style="background=magenta"/>
```

A few things are to be observed here:

- The pattern {all} can be used to match all markables on the respective level. In the above example, this is used to display all *coref* markables in blue.
- Several conditions in one pattern are separated by semicolon. Unless otherwise specified, they are interpreted as AND-connected. If OR-connection is required, the additional attribute `connector="or"` has to be added to the respective line. Thus, mixing AND and OR in a single rule is not possible (except by means of using the implicitly OR-connected value enumeration, cf. Section 2.6 above).
- Although the final style to be applied to a markable is built up incrementally (i.e. by merging all matched styles from top to bottom), the patterns are *NOT* interpreted in this fashion! In other words, there are no dependencies between rules such that a more specific rule automatically inherits sth. from a less specific one. This is apparent in the fact that in the above example, the condition `type={bridging}` is repeated in each rule in which this condition is required to hold.

For performance reasons, customizations should be kept as simple as possible. Although markable customizations (as well as the MMAX2 display proper) are reasonably fast, heavy use of customizations, particularly those using regular expressions, might slow down the display considerably.

3 Querying Markable Sets (updated for version 1.0 beta 4)

From version 1.0 beta 4 on, there is added support for formulating queries that produce not lists of markables, but lists of markable *sets*. The one query supported so far can be issued with the key word `all_markable_sets`. The query requires two string parameters: The name of the markable level on which to look for markable sets, and the name of the attribute, which must by of type MARKABLE_SET. Note that the two string parameters must be enclosed in single quotes. For example, the query

```
display all_markable_sets ('coref','coref_class');
```

when issued on the HTC sample data, will produce a query result with one line (for each coreference set). Each line will have as many columns as the markable set in it has members. The resulting result table can be navigated both horizontally (within the current set) as well as vertically (from set to set, if more than one set is available). The result of a set query cannot only be displayed. It can also, just like any other result, be asigned to a variable, or submitted to the `statistics` command.
When combining the above query with the optional [attributes] parameter, the result display can be made to show additional attribute values in the table cells, apart from just the markable text. Thus, the query

```
display [np_form] all_markable_sets ('coref','coref_class');
```

will produce the same result table as the query above, with the difference that each cell in addition contains the markable's value for the np_form attribute. More than one attribute can be accessed by supplying a comma-separated list of attribute names.

4 Multi-Level Queries

4.1 Introduction

The types of queries described in Section 2 were all confined to accessing markables from a single markable level at a time. Results produced by these queries always consist of tuples of width 1, i.e. 1-place markable tuples, which are equivalent to lists of matching markables.

In contrast to that, *multi-level* queries always access markables from *two* markable levels simultaneously. Consequently, a multi-level query always produces as a result a list of markable tuples of *at least* width 2. The actual width of a multi-level query result depends on the width of the tuples in the input markable tuple lists: If each of the input markable tuple lists is of width 1 (minimal case), the tuples in the result will be of width 2. In general, the width of the result will by default be the sum of the widths of the input tuple lists.

In each markable tuple list, one markable index is the *designated index*. The designated index is between 0 and the tuple's width-1, and is only important for markable tuples of width greater than 1. The main purpose of the designated index is to identify which markable in each tuple is to be used for determining the relation of the markable tuple. By default, the designated index is 0.

Formulating queries to markables from different levels based on markables' relations to each other requires a common reference system. In MMAXQL, the *temporal* relation is used for this purpose. The temporal relation between two markables is approximated on the basis of the relation of the base data elements that each markable spans. For this purpose, each markable is interpreted as an 'event' with a certain start and duration, which is expressed as a base data *interval*. Relations between two markables can thus be expressed in terms of *interval relationships*. James Allen[4] identifies and names seven basic interval relationships. Their application to the description of relations between markables is straightforward.

```
before(a, b)/after(b, a)         : level a:  [#######]
                                   level b:             [the   cat   sat   on   the   mat .]

meets(a, b)                      : level a:     [######.]
                                   level b:             [the   cat   sat   on   the   mat .]

overlaps(a, b)                   : level a:        [###. the]
                                   level b:             [the   cat   sat   on   the   mat .]

starts(a, b)/started_by(b, a)    : level a:             [the   cat]
                                   level b:             [the   cat   sat   on   the   mat .]

during(a, b)/contains(b, a)      : level a:                       [sat]
                                   level b:             [the   cat   sat   on   the   mat .]

finishes(a, b)/finished_by(b, a) : level a:                            [on   the   mat .]
                                   level b:             [the   cat   sat   on   the   mat .]

equals(a, b)                     : level a:             [the   cat   sat   on   the   mat .]
                                   level b:             [the   cat   sat   on   the   mat .]
```

[4] James Allen (1991): Time and Time Again, International Journal of Intelligent Systems 6(4), pp. 341-355.

4.2 Mapping Markables Based on their Interval Relations

The interval relations described in Section 4.1 can be used as query keywords. For the relations `starts`, `during` and `finishes` (and their respective inverse relations), two different versions are available. The standard version implements the matching conditions in a lax way, requiring only a *minimal* condition to hold:

- For `starts` and `started_by`, markables *a* and *b* must start at the same position.
- For `during` and `contains`, markable *a* must be completely contained in markable *b*.
- For `finishes` and `finished_by`, markables *a* and *b* must end at the same position.

This is the type of interpretation that one normally wants. The obvious effect is that more markables will be found. In contrast to this, by appending `_strict` to any of these six keywords (e.g. `started_by_strict`), a more strict interpretation will be used, preventing e.g. markables in a `starts` relation from being found when querying for `during`.
Multi-level queries are expressed declaratively by giving the relation name (as given above) and a pair of brackets with two comma-separated parameters. Each parameter is an expression which evaluates to a list of *n*-place markable tuples.

4.2.1 Mapping Entire Levels

In the most simple case, the two parameters can be the names of markable levels, which evaluate to lists of 1-place markable tuples. As described above, the width of the result tuples will by default be the sum of the widths of each input tuple. Thus, in the Trainline document, the query

```
display started_by(turns, utterances);
```

will display a list of 2-place markable tuples where each tuple contains as its first element (index 0) one markable from the *turns* level and as its second element (index 1) one markable from the *utterances* level, such that the latter is the utterance that starts the turn. With no other modifications, using the inverse relation `starts` as in

```
display starts(utterances, turns);
```

will produce the equivalent result, with only the column positions being interchanged.
By default, `display` will always display the result list in its entire width (i.e. in as many columns as are elements in the result tuples). Sometimes, however, one might be interested in only one part of the result. E.g. one might be interested in a list of turn-initial utterances, but not in the turns themselves. While the *turns* level obviously has to appear in the query intself, it can be suppressed in the display by explicitly identifying the part of the result to be displayed. This can be done by appending to the query a period (.) and the (zero-based!) index of the column to be displayed. E.g. to have only the *utterances* column displayed, one has to write

```
display starts(utterances, turns).0;
```

since it is at index 0 in the result tuples that the markables from the *utterances* level appear. Another way of identifying a column in the result is by giving *the name of the markable level* the markables at this index come from. This name is shown in the header of the result table as displayed in the lower part of the MMAX2 query console. Thus, one could also write

```
display starts(utterances, turns).utterances;
```

Note, however, that obviously this will only work if the name is *unique* within each result. There are several reasons why this could not be the case. The most obvious one is when a relation query accesses the same markable level twice. In the Trainline document, e.g. the following query

```
display meets(utterances, utterances);
```

will retrieve a list of 2-place markable tuples corresponding to utterance *bigrams*. Trying to use the name *utterances* in the above query to identify a column in the result will not work, and a corresponding message will be displayed.
Instead of directly displaying the result, it can also be assigned to a variable. Thus, in the Trainline document, the query

```
let $turn-initial_utterances = starts(utterances, turns).utterances;
```

will assign the first column of the result list to the variable $turn-initial_utterances. Thus, $turn-initial_utterances will evaluate to a list of 1-place markable tuples, and can be used in any context where the original query could be used, e.g. in

```
display $turn-initial_utterances;
```

Note that since the width of the result list assigned to the variable $turn-initial_utterances is 1, no column identification is necessary here.
Of course, variables can also hold lists of markable tuples with *more* than one column. In the Trainline document, the query

```
let $utterance_bigrams = meets(utterances, utterances);
```

will assign the entire (2-place) query result to the variable $utterance_bigrams. Accessing result columns on variables works the same as with normal queries:

```
display $utterance_bigrams.0;
```

will display only the first column (index 0) of each bigram.

4.2.2 Mapping Markable Subsets

More interesting than the multi-level queries described so far are queries which do not map simply *all* markables from two levels to each other, but only subsets which were themselves produced by queries. This can be done by using simple MMAXQL queries (as described in Section 2 above) as parameters, which will also always evaluate to lists of 1-place markable tuples. E.g., to query from the Trainline document the initial utterances of turns spoken by speaker *a*, the following query can be used:

```
let $initial_a_utterances=
   started_by(turns (speaker={a}), utterances).utterances;
```

Note how by specifying the *utterances* column, only the utterance markables are assigned to $initial_a_utterances. When the expressions yielding the parameter lists get more complex, it is advisable to use variables as parameters. This will avoid potential error sources and increase the readability of a query. Thus, in the Trainline document, the query sequence

```
let $a_turns=turns(speaker={a});
let $questions=utterances(type={q-yn,q-wh});
let $a_questions=during($questions, $a_turns).utterances;
display $a_questions;
```

will display a list of all questions of type *q-yn* or *q-wh* asked by speaker *a*. The above query has been typeset like this only to make it more readable. Actually, the entire sequence can be typed into the query console at once, *without* pressing enter in between. However, since variables are persistent and live as long as the MMAX2 session lasts, the query can also be entered and executed line by line. **Note:** In this case, using the `debug` command can be useful. After activating the debug mode (by typing the command `debug`), the result of each variable assignment will also directly be displayed. This way, immediate feedback is available if the query produced the type of result one had in mind, without having to display each variable explicitly. Use the `nodebug` command to deactivate the debug mode.
The first command in the above query retrieves a list of all turns of speaker *a*. The second command retrieves all utterances of *type* either *q-yn* or *q-wh*. The third command uses both these intermediate results to retrieve all elements from the first parameter list that occur during, i.e. are completely embedded in, elements from the second list, and extracts the column containing the elements from the first list (i.e. the utterances) only. The last command simply displays the final result.

4.2.3 Embedding Multi-Level Queries

Since the result of a multi-level query is itself a list of markable tuples, it can also be used as input parameter in another multi-level query. Although direct embedding of one relation query as the parameter of another one is possible, it is recommended to use an intermediate variable assigment instead. For simple queries, on the other hand, using direct embedding is sometimes more convenient.
When markable tuples of width greater than 1 are used as input to multi-level queries, setting the *designated index* becomes important. As mentioned earlier the designated index specifies from which column the markables should be taken when determining the relation between two given markable tuples. The default index is 0, which is not always correct. E.g., in order to retrieve a list of utterance *trigrams* from the Trainline document, one might be tempted to use the following query:

```
display meets(meets(utterances,utterances),utterances);
```

This query, however, will *not* produce tuples of three *subsequent* markables. Instead, the markables at index 1 and 2 will be identical! This is due to the fact that the outer `meets` query (which is evaluated after the inner one), considers at its first parameter only those markables in the result of the inner query that are at (the default) index 0. To make the above query work correctly, it has to be made explicit that the outer `meets` query should match markables that directly follow the *second* markable in the tuple. This is done by setting the designated index to 1:

```
display meets(meets(utterances,utterances).1,utterances);
```

Note that the syntax for setting the designated index within embedded multi-level queries and the syntax for identifying a result column for display or variable assignment is identical.on purpose.
The above query produced a result list of width 3 (= 2 from the embedded query + 1 from the outer query). This is obviously correct and desired if one is interested in retrieving trigrams. Thus, the above query is one example where explicitly setting the designated index is necessary. Frequently, however, only some

columns from a query accessing several markable levels will actually be relevant. In these cases, it is advisable to try to prevent unneeded temporal results from accumulating from query to query. This can best be achieved by

- generously using variables to store temporary results, and
- storing in these variables only relevant result columns.

Thus, in the Trainline document, the following query

```
let $a_requests=
   during(utterances (sp-act={reqinfo}), turns (speaker={a})).utterances;
let $b_utterances=
   during(utterances, turns (speaker={b})).utterances;
let $request_answer_pairs=
   meets($a_requests, $b_utterances);
```

will retrieve 2-place markable tuples where the first element (index 0) is an info request by speaker *a*, and the second element (index 1) is the utterance by speaker *b* directly following it.
The first command in the above query retrieves a list of all utterances with the speech act tag *reqInfo* spoken by speaker *a*. Note how by using the column specifier *.utterances*, only the relevant part of the result is stored. The turn information was only required to formulate the speaker condition, and is not needed beyond that. The second command retrieves all utterances by speaker *b*. The third command, finally, returns tuples of elements of both temporal results such that the utterance by speaker *b* directly follows the request by speaker *a*.
Now, if one is interested in descriptive statistics for either set of utterances, the command

```
statistics $request_answer_pairs.0;
```

and

```
statistics $request_answer_pairs.1;
```

can be used to to retrieve statistics for the speaker *a* and speaker *b* utterances, respectively.