Emdros Query Guide

Ulrik Petersen

June 7, 2007

Abstract

This guide will show you how to use the Emdros Corpus Query System to query your data. It assumes that you have already imported your data into Emdros, and simply want to start querying. It is aimed at the non-technical person, though familiarity with corpus linguistics is assumed.

Contents

1 Introduction

This query guide will show you how to query your data with the Emdros Corpus Query System.[1] It is aimed at a non-technical (i.e., non-programmer) reader, but assumes familiarity with corpus linguistics.

2 The database model

The EMdF model underlying Emdros has four concepts:

1. Monads

[1] http://www.emdros.org/

2. Objects
3. Object types
4. Features

A monad is simply an integer, no more, no less.

An object is a set of monads, and belongs to an object type.

An object type groups a set of objects with similar characteristics. Examples would include “Word”, “Phrase”, “Clause”, “Page”, “Chapter”, “Line”, “Book”, etc.

The object type of an object determins what features it has. A feature is an attribute. Examples would include “Word.part_of_speech”, “Word.surface”, “Word.lemma”, “Phrase.phrase_type”, “Phrase.function”, “Chapter.chapter_number”, etc.

The set of monads of an object is quite arbitrary, in that it need not be contiguous, but may have one or more gaps. This is useful to model things like embedded relative clauses and postpositive conjunctions.

A feature “take on” exactly one type. This type is one of the following:

1. Integer (e.,g., 1, 3, 100, 133, etc.)
2. id_d (this is a unique integer identifying an object, e.g., 1,3,1003, etc.)
3. enumeration (see below)
4. list of any of the above
5. string of characters (e.g., ’This is my string.’)

An enumeration is a database-dependent set of labels. Thus the exact enumerations available to you depend on what enumerations are available in your database. Examples could be, if you have an enumeration called “part_of_speech”, it might contain labels like “noun”, “verb”, “adjective”, “adverb”, etc. Enumerations are also sometimes used for phrasal categories like “NP”, “PP”, etc. Again, the exact categories available to you are dependent on what is available in your database; these are just examples.

3 Getting started

At the beginning of every query, you must have this incantation:

```
SELECT ALL OBJECTS
WHERE
```

This tells Emdros that you wish to issue a linguistic query.[2] In this guide, we will mostly omit this incantation, since it is common to all queries.

NOTE: If you are using Emdros through an interface not provided by the author of Emdros, your interface designer may have chosen to let you omit this stanza.

[2]The MQL language caters to much more than just linguistic queries, but the rest is mainly concerned with database maintenance and display of data, and so are outside the scope of this query guide. See the MQL User’s Guide for more information on these other query types.

4 Comments

In this guide, we will often show comments in the queries. There are two kinds of comments, but we will only show examples of one kind, namely the one that begins with two slashes:

```
// This is a comment
```

This kind of comment starts with the two slashes, and extend to the end of the line. Such comments are ignored by Emdros.

The other kind is described in Appendix B on page 20.

5 Gentle introduction

5.1 Blocks

A "block" looks for something in the database. There are four kinds of blocks:

1. Object blocks – look for objects.
2. Power blocks – used to mean "arbitrary space within surrounding the context".
3. Gap blocks – look for "gaps" in the surrounding context.

5.1.1 Object blocks

A simple object block looks like this:

```
[word]
```

This looks for an object of type word.

5.1.2 Power block

A simple power block looks like this:

```
. .
```

It is simply two dots next to each other.

5.1.3 Gap blocks

A simple gap block looks like this:

```
[gap]
```

If you wish the gap to be optional, you can put a question mark after the "gap" keyword:

```
[gap?]
```

This is called an "optional gap block".

5.2 The overruling principle of MQL

The overarching principle of MQL is:

The structure of the query
mirrors
the structure of the objects found
with respect to sequence and embedding.

This means that:

1. If two blocks are next to each other in the query, the objects they find must be adjacent in the database:

```
[A]
[B]
```

2. If a block A is embedded inside another block B in the query, then the object that block A finds must be embedded inside the object that block B finds:

```
[B
   [A] // A object must be embedded in B
]
```

5.3 Strings of blocks

You can place blocks next to each other and thus look for a string of blocks. For example, the query:

```
[phrase]
[phrase]
```

looks for two phrases that are adjacent in the database.

5.4 Embedding of blocks

You can embed (strings of) blocks in another block:

```
[Clause
   [Phrase]
   [Phrase]
   [Phrase]
]
```

This query would find clauses inside of which there are at least three phrases. The phrases must be adjacent.

If you use the "power block", you should always do so within the context of a surrounding block:

```
[Clause
   [Phrase]
   ..
   [Phrase]
]
```

This would find all clauses in which there were at least two phrases, but the phrases need not be adjacent.

The reason you should always use a surrounding context when using the power block is that otherwise, all combinations in the database of what appears before the power block and what appears after it will be retrieved, which will probably be more data than you will want to deal with. The language does not disallow using a power block at the outermost level, it might just return too much data for your liking.

6 Blocks in more detail

In this section, we explain blocks in more detail: First object blocks, then power blocks, and finally gap blocks.

6.1 Object blocks

As stated before, object blocks at their simplest look like this:

```
[Phrase]
```

This query will find all phrases in the database. The word right after the opening bracket ("[") is the object type you wish to search for. The exact categories of object type available to you depend on your database.

6.1.1 Feature-restrictions

You can search for feature-restrictions:

```
[Word surface='see']
```

This finds all words whose surface-feature is the string "see".

You can use arbitrary Boolean expressions with feature-restrictions with the operators AND, OR, NOT, and grouping (i.e., parentheses):

```
[Phrase phrase_type=NP
        AND (function = Subj OR function = Obj)
        AND NOT self = 13082
]
```

This will find all phrases whose type is NP, and whose function is either Subj(ect) or Obj(ect), and whose "self" feature is not 13082.

Op.	Meaning	Left-hand-side feature must be	Right-hand-side value must be
=	Equality	integer, string, id_d, enumeration, list	Same as left-hand-side
<>	Inequality	integer, string, id_d, enumeration	Same as left-hand-side
<	Less than	integer, string, id_d, enumeration	Same as left-hand-side
<=	Less than or equal to	integer, string, id_d, enumeration	Same as left-hand-side
>	Greater than	integer, string, id_d, enumeration	Same as left-hand-side
>=	Greater than or equal to	integer, string, id_d, enumeration	Same as left-hand-side
~	Regular expression	string	string
!~	Negated regular expr.	string	string
IN	List-membership	integer, id_d, enumeration	list
HAS	List-membership	list	integer, id_d, enumeration

Table 1: Comparison operators

6.1.2 Feature-comparison form

Each feature-comparison is of the form:

```
feature operator value
```

For example, in the feature-comparison

```
phrase_type = NP
```

"type" is the feature, "=" is the operator, and "NP" is the value.

The feature-comparisons must always appear in this order. Thus, for example, you cannot say:[3]

```
* NP = phrase_type // This won't work
```

6.1.3 Values

For details on values, such as integers and strings, please see Appendix A on page 20. Briefly:

- integers and id_ds are written as usual (e.g., 1, 100, 175, etc.).
- it is recommended that strings be written surrounded by 'single quotes', not "double quotes".[4]
- enumeration constants are written as they are declared in the database. Of course, this is database-dependent. Examples could be (this may differ from your database): NP, PP, AP, noun, verb.

6.1.4 Comparison operators

The operators available to you are listed in Table 1.

[3]The "*" in front is meant to signify that the example is erronerous, in accordance with the usual convention in linguistic writing.

[4]The reason is that double-quote-strings treat many characters specially, so you may need to "escape" certain characters. See Appendix B on page 20 for details.

6.1.5 The IN operator

The IN operator is used like this:

```
[Word psp IN (noun,adjective,conjunction,article)]
```

That is, the left-hand-side must be a feature that is either an integer, an id_d, or an enumeration, and the right-hand-side must be a comma-separated list of values in parentheses.

6.1.6 The HAS operator

The HAS operator is the inverse: It looks for a single value in a list-feature:

```
[Word semantic_categories HAS royal]
```

6.2 Power blocks

Power blocks are used to mean "an arbitrary stretch of space":

```
[Clause
   [Phrase]
   ..
   [Phrase]
]
```

This will find all clauses which have at least two phrases, and inside such clauses, all combinations of two phrases. The two phrases need not be adjacent.

6.2.1 Limiting with < and <=

You can limit the scope of the power-block like this:

```
[Clause
   [Phrase]
   .. <= 5   // The space may only be up to 5 monads long
   [Phrase]
]
```

This also exists in a "strictly less than" version:

```
[Clause
   [Phrase]
   .. < 5    // The space may only be up to 4 monads long
   [Phrase]
]
```

Exactly how many linguistic units a monad constitutes in your database is dependent on how the database was designed. It may be "word", "morpheme", "phoneme", "sentence", or none of these. Ask the person who designed the database how they treated "monad granularity" if in doubt.

6.2.2 Limiting with BETWEEN X AND Y

The power block can also be used like this:

```
[Clause
   [Phrase]
   .. BETWEEN 3 AND 5 // The space must be at least 3
                      // and at most 5 monads long.
   [Phrase]
]
```

This is equivalent to "3 <= X <= 5", where X is the length of the stretch in monads.

6.3 Gap blocks

6.3.1 Introduction

Gap blocks are used to look for "gaps" in the surrounding context. For example, some linguists would hold that the sentence:

- The door, which opened towards the East, was blue.

in fact consists of two clauses, namely:

- The door . . . was blue.
- which opened towards the East

and that "which opened towards the East" is a *sibling*, not a child, of the clause "The door . . . was blue."

In such a scenario, there would be a "gap" in the clause "The door . . . was blue", corresponding to the embedded relative clause.

You can look for such cases with the gap block:

```
[Clause
   [gap
      [clause clause_type = relative]
   ]
]
```

6.3.2 Optional gap blocks

You can specify that a gap block may be optional, by placing a question mark after the "gap" keyword:

```
[Phrase
   [word psp=article]
   [gap?
      [word first and last psp = conjunction]
   ]
   [word psp=noun]
]
```

This would look for all phrases in which there is an article, followed optionally by a gap inside of which the sole word is a conjunction. After the optional gap, there must be a word which is a noun. This occurs, e.g., in classical Greek, where postpositive conjunctions abound. These are usually constituents at a higher level, but intervene in the phrase and/or clause in which they stand. Thus they would give rise to a "gap".

6.3.3 Automatic insertion of optional gap blocks

An optional gap block is inserted between other blocks by default. This is to safeguard against not finding cases such as the above with the postpositive conjunction. Thus the following:

```
[Phrase
   [word psp=article]
   [word psp=noun]
]
```

would also find the cases where a postpositive conjunction intervened between the article and the noun. Thus the above does not really mean that the article and the noun must be adjacent; it really means that they must be adjacent, *ignoring any gaps in between*.

If you want to turn this automatic insertion off, you can place an exclamation mark ("!") between the blocks:

```
[Phrase
   // The ! turns off insertion of optional gap block
   [word psp=article]!
   [word psp=noun]
]
```

This will ensure that the article and the noun really are adjacent, and that no gaps intervene.

7 Advanced topics

7.1 Introduction

This section explains some "advanced" topics. By "advanced" we do not mean that they are difficult to grasp; rather, we merely mean that they do not belong to the "basics" of writing an MQL query. In addition, taking a "spiral approach to learning" is a philosophy to which we subscribe.

7.2 Object blocks

7.2.1 Object references ("AS")

You can give an object a name, and refer back to it later in the query:

```
[Clause AS container // the AS keyword assigns the name
   [Phrase parent = container.self]
]
```

The AS keyword must appear right after the object type name ("Clause" in this example). After the AS keyword, you can write the name you want to give to the object.

Later in the query, you can then refer to a feature on the named object by means of the "dot-notation". In the above example, the "parent" feature of the "Phrase" object type is compared with the "self" feature of the "Clause" object.[5]

This can be used with any operator, so long as the *left-hand-side* is a feature (e.g., "parent"), and the *right-hand-side* is the object reference usage (e.g., "container.self"). Thus you cannot say:

```
* [Clause AS container
      [Phrase container.self = parent] // This won't work
      // switch them around to make it work.
  ]
```

7.2.2 MARKS

You can specify "marks" on either an object block, a gap block, or an optional gap block. The marks look like this: "'red", "'yellow", "'context", "'red'context", "'Flash_Gordon". That is, they start with a backquote ('), followed by a sequence of letters, numbers or the underscore (_), where the first character must be either a letter or an underscore. This pattern can be repeated, as the marks "'red'context" shows.

```
[Clause`yellow
     [Phrase`red AS p1]
     ..
     [Phrase`blue phrase_type = p1.phrase_type]
]
```

The marks specification must come immediately after the object type, as shown by "Clause'yellow" above.

Emdros itself does nothing with the marks; it simply passes it on to the application lying on top of Emdros. Thus you need to consult any manual for your particular Emdros-application for whether it does anything with the marks. If not, there is no point in using them. In particular, Emdros does not assign any meaning to the sequences of charaters – for example, "'red" does not mean that Emdros will show anything in red, and "'context" does not mean that Emdros will recognize that such and such is context. The application lying on top of Emdros may do such things, but that is outside the scope of this manual.

7.2.3 FOCUS/RETRIEVE/NORETRIEVE

You can specify that an object must be in FOCUS:

```
[Clause FOCUS]
```

How this shows up in your results depends on the implementation of the display tool.

Alternatively, you can explicitly say that something must not be retrieved:

```
[Clause NORETRIEVE]
```

[5] The "self" feature gives the id_d of the object in question.

You can also explicitly say that it must be retrieved (this is unnecessary, as all objects are retrieved by default):

```
[Clause RETRIEVE]
```

If you have an object reference declaration on a block, then the FOCUS/RETRIEVE/NORETRIEVE keyword must come after the object reference declararition, and before any feature-restrictions:

```
[Clause
   AS C1                    // 1. Object reference declaration
   FOCUS                    // 2. Focus-specification
   clause_type = Wayyiqtol  // 3. Feature-restriction
]
```

7.2.4 Inner string of blocks

You can, as already shown, have an inner string of blocks inside an object block:

```
[Clause
    [Phrase]
    [Phrase]
    [Phrase]
]
```

This will find all clauses that have at least three phrases inside.

The inner string of blocks must come after any feature-restrictions:

```
[Clause
   AS C1                    // 1. Object reference declaration
   FOCUS                    // 2. Focus-specification
   clause_type = Wayyiqtol  // 3. Feature-restriction
     [Word]                 // 4. Inner string of blocks
     [Phrase]
     [Phrase]
]
```

7.2.5 FIRST/LAST/FIRST AND LAST

You can specify that an object block must be FIRST, LAST, or FIRST AND LAST in its surrounding context:

```
// Example 1:
[Clause
   [Phrase FIRST AND LAST] // must be the only phrase in its context
]
// Example 2:
[Clause
   [Phrase FIRST] // Must be first
   [Phrase LAST] // Must be last
]
```

The FIRST/LAST/FIRST AND LAST specification must come between any FOCUS/RETRIEVE/NORETRIVE specification and any feature-restrictions:

```
[Sentence
   [Clause
      AS C1                      // 1. Object reference declaration
      FOCUS                      // 2. Focus-specification
      FIRST AND LAST             // 3. FIRST/LAST/FIRST-AND-LAST spec.
      clause_type = Wayyiqtol    // 4. Feature-restriction
        [Word]                   // 5. Inner string of blocks
        [Phrase]
        [Phrase]
   ]
]
```

7.2.6 Regular expression operators

The "~" and "!~" operators work with Perl5[6]-compatible regular expressions[7] on the right-hand-side:

```
// finds both "see" and "See"
[Word surface ~ '[Ss]ee']
// finds everything that is neither "my" nor "your"
[Word surface !~ '(my)|(your)']
```

Note that if you use the "backslash" escape-operator with "double-quote-strings", you need to escape it twice:

```
// This will find a literal $ followed by a literal dot.
[Word surface ~ "\\$\\."]
```

Thus it is often easier to use 'single quote strings' with regular expressions:

```
// This will find a literal $ followed by a literal dot.
[Word surface ~ '\$\.']
```

For details, please see Appendix B on page 20.

7.2.7 NOTEXIST

You can specify that an object block must "not exist" with the "NOTEXIST" keyword:

```
[Sentence
   NOTEXIST [Word surface = 'see']
]
```

This finds all sentences in which the word "see" does not occur.

Note how this is very different from saying:

[6]Perl is a programming language, and Perl5 is version 5 of the language.
[7]For details on regular expressions, please see Appendix C on page 21.

```
[Sentence
   [Word surface <> 'see']
]
```

This would find all sentences which has a word which is not “see”. That would include sentences which did have the word “see”, but which also had other words.

You are allowed to intermix NOTEXIST with other blocks in the same context. For example, this is allowed:

```
[Clause
   [Phrase]
   NOTEXIST [Word surface='food']
   [Word surface='glue']
]
```

What that means is that we want clauses inside of which there is a phrase, *right after which* there is a Word with surface=”glue”. From the end of the Phrase until the end of the Clause, there must not exist a Word with surface=”food”.

So: a) The NOTEXIST block is regarded as not being present when considering the surrounding blocks. That is why the “glue” word must be right after the Phrase in order for this query to match. Essentially, a NOTEXIST block has “zero width” with respect to consecutiveness. b) The NOTEXIST block is looked for starting at the end of the previous block (or the start of the context if the NOTEXIST block is the first) and running to the end of the context.

You are allowed to use NOTEXIST more than once in any given context. For example, this is allowed:

```
[Sentence
   NOTEXIST [Word surface = 'see']
   NOTEXIST [Word surface = 'the']
]
```

This would find all sentences inside of which neither a word with surface=”see” nor a word with surface=”the” exists. Because the NOTEXIST block with surface=”see” is the first in the context, the word “see” must not occur anywhere within the sentence. Because a NOTEXIST block has “zero width” with respect to consecutiveness, it means that the domain within which a word with surface=”the” must not occur is also anywhere within the sentence.

You cannot use an object reference that has been declared “inside” a NOTEXIST, except if you also use it “inside” the same NOTEXIST. Thus you cannot say:

```
* [Clause
     [Phrase
        NOTEXIST[Word as w1 surface='food']
     ]
     // OOPS! The NOTEXIST intervenes, so we can't "see" w1 here...
     [Word part_of_speech=w1.part_of_speech]
  ]
```

But you can say:

```
[Clause
   NOTEXIST [Phrase
                [Word as w1 part_of_speech=noun]
                ..
                // This is OK! NOTEXIST does not intervene,
                // but stands above both!
                [Word part_of_speech <> w1.part_of_speech]
            ]
]
```

7.3 Strings of blocks

7.3.1 OR between strings of blocks

A "string of blocks" is an unbroken sequence of object blocks, power blocks, and/or gap blocks. You can put an "OR" keyword in between two such strings. The result will be as though you had issued two separate queries, with one string of blocks taken away and the other left in (and the OR taken out as well), then vice-versa for the second query. This is useful, e.g. to search for different combinations of a given sequence of phrases with specific functions:

```
[Clause
    [Phrase function = Subj]
    [Phrase function = Pred]
    [Phrase function = Objc]     // Here the object comes before
    [Phrase function = Adjunct]  // the adjunct
    or
    [Phrase function = Subj]
    [Phrase function = Pred]
    [Phrase function = Adjunct]  // Here the adjunct comes before
    [Phrase function = Objc]     // the object
]
```

As mentioned, the OR construct works between strings of blocks. It doesn't matter what kind of block is involved (object block, power block, or gap block), so you could also say:

```
[Clause
    [Phrase function = Subj]
    ..
    [Phrase function = Objc]
    OR // The OR Works between on the one hand Subj..Objc
       // and on the other hand, Objc..Adjunct
    [Phrase function = Objc]
    ..
    [Phrase function = Adjunct]
]
```

Or even:

```
[Clause
```

```
    [gap [Clause clause_type = Appositional]]
    OR
    [Phrase function = Objc]
    [Phrase function = Adjunct]
]
```

You can also have more than one OR between more than two strings of blocks:

```
// Finds all triples of object, adjunct, and complement
// where either the object or the complement is first.
// To find all six combinations (i.e., also adjunct first),
// simply add two more ORs with the right orders of phrases.
[Clause
    [Phrase function = Objc]
    [Phrase function = Adjunct]
    [Phrase functino = Complement]
    OR
    [Phrase function = Objc]
    [Phrase functino = Complement]
    [Phrase function = Adjunct]
    OR
    [Phrase functino = Complement]
    [Phrase function = Objc]
    [Phrase function = Adjunct]
    OR
    [Phrase functino = Complement]
    [Phrase function = Adjunct]
    [Phrase function = Objc]
]
```

7.3.2 Restrictions on OR (more on the AS keyword)

There is one restriction pertaining to OR: When you have a reference between two objects (using the AS keyword, see Section 7.2.1 on page 10), then both the object block on which you use the AS keyword, and the object block on which you use the reference, must be within the SAME string of blocks. The usage cannot cross an OR. Thus you cannot say:

```
* [Clause
     [Phrase AS p1]
     OR
     [Phrase function = p1.function] // OOPS! Illegal because it
                                     // crosses the OR construct!
 ]
```

Nor can you say:

```
* [Clause
     [Phrase
         [Phrase AS p2]
```

```
        ]
        OR
        [Phrase function = p2.function] // OOPS! Illegal because it
                                        // crosses the OR construct!
    ]
```

When we said that both the declaration (with the “AS” keyword) and the usage must be within the same string of blocks, we did not mean that they have to be at the same level, like this:

```
[Clause
    [Phrase AS p1]
    [Phrase function <> p1.function] // This is OK, since it does not
    OR                               // cross the OR.
    [gap]
]
```

These two, the declaration and the usage, are at the same level. But it is OK to have one of them be more deeply nested than the other:

```
[Clause
    [Phrase
        [Phrase AS p1] // This is more deeply nested than the usage
    ]
    [Phrase function <> p1.function] // This is OK, since it does not
    OR                               // cross the OR.
    [gap]
]
```

7.4 Grouping ([square brackets])

7.4.1 Introduction

“[Square brackets]” are used to group one or more strings of blocks, as if there were “parentheses” around them.

7.4.2 Examples

The following topograph illustrates the use of square brackets for grouping:

```
[Clause
    [Phrase function = Predicate]
    [
        [Phrase function = Objc]
        [Phrase function = Adjunct]
        OR
        [Phrase function = Indirect_object]
        [Phrase function = Complement]
    ]
]
```

This query finds all clauses in which there is a Phrase whose function is Predicate. Right after this phrase must come, either an Object followed by an Adjunct, or an Indirect object followed by a Complement.

Another example:

```
[Clause
    [
        [Phrase function = Subject]
        OR
        [Phrase function = Complement]
        [Phrase function = Adjunct]
    ]
    ..
    [Phrase function = Predicate]
]
```

This query finds all clauses in which there is, first either a Subject, or a Complement followed by an Adjunct. Then, after either of these, there can be arbitary space within the Clause (indicated by the ".." power block), and then a Predicate must appear.

7.5 Kleene star

7.5.1 Introduction

You can have a Kleene star construction on any object block or [group] in a query:

```
[Phrase
   // Note the * at the end
   [Word psp IN (article,noun,conjunction,adjective)]*
]
```

This query will find all phrases, inside of which there are zero or more adjacent words whose parts of speech are either article, noun, conjunction, or adjective. This would find many noun phrases.

You can also have a Kleene star on a group of blocks:

```
[Clause
   [
      [Word psp IN (article, noun, preposition)]
      [Word psp IN (noun, adjective)]
   ] *
]
```

This would find all clauses inside of which there are zero or more iterations of the pattern "a Word whose part of speech is either article, noun, or preposition", followed by a "Word whose part of speech is either noun or adjective."

NOTE: Because there are [square brackets] around the two words, and because the Kleene star applies to the group, it is the *group* that is repeated.

The Kleene star means "find me zero or more like this".

7.5.2 Specifying the number of iterations

You can also specify a set of integers that gives the number of times required:

```
// Example 1:
[Phrase
   [Word]*{0,1} // This makes the word optional (0 or 1 times)
]

// Example 2:
// This finds all clauses in which the first phrase is a subject,
// followed by exactly 3 non-subject, non-adjunct prases,
// followed by an adjunct phrase.
[Clause
   [Phrase FIRST function = Subject]
   // There must be exactly three phrases between the subject...
   [Phrase NOT function IN (Subject,Adjunct)]*{3}

   // ... and the adjunct
   [Phrase function = Adjunct]
]

// Example 3:
[Clause
   // Finds such phrases 1,2,3,5,6,7, or 9 and above times
   [Phrase
      function = Subj OR function = Obj
   ]*{1-3,5-7,9-}
   // But still only within the surrounding clause
]
```

7.5.3 Restrictions

The following restrictions apply:

- You cannot have an object reference declaration (using the AS keyword) on an object block on which you also have a Kleene Star. For example, this is NOT allowed:

```
* [Phrase as p1]* // OOPS! Not allowed to have both AS and Kleene Star!
  [Phrase function=p1.functino]
```

- You cannot use an object reference that has been declared inside an object block or group with a Kleene Star, if the usage is outside the object block or group with the Kleene Star. (If it is used inside, you can use it there, but not outside). Thus this is NOT allowed:

```
* [Phrase
     [Word as w1]
  ]* // OOPS! Kleene Star on Phrase...
  [Word surface=w1.surface] // so we can't "see" the reference here!
```

- Whereas this *is allowed*:

```
[Phrase
    [Word as w1]
    ..
    [Word lexeme=w1.lexeme]
]* // This * is OK; we don't "cross" it when we use the reference!
```

A Values

A.1 Atomic values

There are four kinds of atomic values:

1. integer: e.g. 0, 1, 42, 976, 1000, etc.
2. id_d: Like integers, but can also be NIL (no value).
3. enumeration: Whatever is defined in your database.
4. string: Enclosed in "double quotes" or 'single quotes'.

A.2 Lists

You can build lists out of integers, id_ds, and enumeration labels, but you cannot currently build lists of out strings.

Lists are enclosed in (parentheses), and are comma-separated. For example:

1. List of integer: (0,1000,23,76)
2. List of id_d: (NIL, 13200)
3. List of enum label: (NP,AP,PP)

A list can also have a single value inside, e.g., (NP).

B Lexical rules

1. Whitespace is ignored except to separate tokens, and in strings.
2. Everything except enumeration-labels and strings is case-INsensitive. Enumeration labels and strings ARE case-sensitive.
3. Reserved words (such as "object", "create", "type", etc.) may not be used except as reserved words. That is, you cannot, say, have a feature called "type" or an enumeration constant called "object".
4. Strings can be of two kinds: Either surrounded by "double quotes", or surrounded by 'single quotes'. Both may contain newlines. Backslashes inside "double-quote-strings" behave as escape-characters according to Table 2. Backslashes inside of 'single-quote-strings' behave as backslashes, and in fact you cannot escape anything inside of a 'single-quote-string'.

Escape	Meaning
\n	line feed (ASCII 0x0a)
\t	horizontal tab (ASCII 0x09)
\v	vertical tab (ASCII 0x0b)
\b	backspace (ASCII 0x08)
\a	alert/bell (ASCII 0x07)
\f	form feed (ASCII 0x0c)
\r	carriage-return (ASCII 0x0d)
\\	\
\?	?
\'	'
\"	"
\XYZ	The character with octal-based number XYZ (e.g., \040 meaning 32)
\xXY	The character with hexadecimal-based number XY (e.g. \x20 meaning 32)

Table 2: Backslash-escapes in "double-quote-strings"

5. Comments are ignored when parsing MQL. There are two kinds:

```
// This kind starts with two slashes.
// It extends to the end of the line.
// Thus if you want multiple lines commented out,
// you have to start each new line with the double slash.
/* This is the other kind of comment.
   It may extend over multiple lines. It begins with
   slash-star and ends with star-slash.
*/
```

6. An **identifier** starts with an underscore or the letters A-Z or the letters a-z. If it is longer than 1 character, it continues with underscores, letters from A-Z, letters from a-z, or digits in the range 0-9. Thus it conforms to the regular expression '[_A-Za-z][_A-Za-z0-9]*'.

7. Database names,[8] object type names, feature names, enumeration names, enumeration labels, and monad set names must be identifiers.

C Regular expressions

This is a crash course in regular expressions. Regular expressions (or RegExes) are a way of specifying a set of strings, which in Emdros can be used to compare a string-feature against many values at once. For example, if you wish to search for both "See" and "see" at once, you can use the regular expression comparison:

```
surface ~ '[Ss]ee'
```

The effect is as if you had said:

```
surface = 'See' OR surface = 'see'
```

[8]Except on SQLite, where a database name may be non-identifiers. In that case, however, it must be expressed as a "string" or a 'string'.

C.1 Character classes

You can specify character-classes with the [square brackets]. A character-class is a set of characters that are looked for at once. A simple example would be:

```
[AaBbCc]
```

This would look for the letters A, B, C, a, b, and c all at once. If just one of them was present, the whole character class would match.

The above could also be rewritten as:

```
[A-Ca-c]
```

This is because, inside a character class, the dash (also known as minus) means "from the previous character to the next character, both inclusive". Thus if you wish to search for the characters A-Z, you can say [A-Z]. If you wish to search for a minus, and include the minus in the character class, you can put it last in the character class:

```
[A-Z-]
```

This would search for the letters A-Z, but would also search for the minus.

If you wish to negate the character class, you can put the "hat" ("^") at the beginning of the class:

```
[^A-Z]
```

This would search for all characters *except* the letters A-Z (thus it would also search for the letters a-z, since regular expressions are case-sensitive).

C.2 Grouping

You can group sequences of characters or character classes with parentheses:

```
(se[ea])
```

The utility of grouping will be apparent shortly.

C.3 Kleene Star (*)

You can specify that something must occur zero or more times:

```
[A-Za-z0-9]*
```

This would search for the characters A-Z, a-z, and 0-9, and they may occur 0 or more times after each other. Thus both "", "a", "aA, "aAZ", and "abc0" would match.

The Kleene Star applies only to the previous character, character class, or group. Thus if you wish a whole string of characters to be repeated, you must use grouping:

```
(elar)*
```

This would match "", "elar", "elarelar", "elarelarelar", etc.

If you say:

```
elar*
```

then "ela", "elar", "elarr", "elarrr", etc. will be matched.

C.4 Kleene Plus (+)

The Kleene Plus (specified with "+") is similar to the Kleene Star, except that it matches one or more times, not zero or more times:

```
utterance ~ 'My precious+'
```

would match any of "My precious", "My preciouss", "My preciousss", etc.

Again, the Kleene Plus applies only to the previous character, character class, or group. If you wish to repeat a whole string, then it must be grouped with parentheses.

C.5 OR (|)

You can specify that either of two characters, character classes, or groups should match, with the "or" construct (which in the regular expressions is a "|"):

```
(sea)|(lake)
```

This would match either "sea" or "lake".

As with the Kleene Star, the | applies only to the surrounding two characters, character classes, or groups. Thus if you do wish to match either of two strings, you must put parentheses around both strings, as in the "sea or lake" example above. If you say:

```
sea|lake
```

then the two strings "selake" or "seaake" will be matched.

C.6 Escapes

If you wish to match one of the characters that have a special meaning, e.g., "[", "]", "*", "+", "|", etc., then you must put a backslash ("\") in front: "\[", "\]", "*", etc.

Of course, a backslash also has special meaning, so if you wish to match a backslash, you must escape it, too: "\\".

C.7 Any character

If you wish to match "any character", there is a shorthand for the character class that matches "all characters": It is simply a dot (also known as period):

```
^We the people.*
```

This would match any string which started with the letters "We the people" and which then continued with zero or more characters of any kind.

Note that if you wish to match a period, you need to escape the period: "\.".