# MQL Programmer's Guide

Ulrik Sandborg-Petersen

November 1, 2018

Please visit the Emdros website for the latest news and downloads:

`https://emdros.org`

**Abstract**

This is the MQL Programmer's Guide. It documents Emdros version 3.7.0 and upwards. If you just wish to use Emdros to query your data, then this might not be for you. Instead, you can consult the MQL Query Guide, which is available from the Emdros website, or with any recent distribution of Emdros.

In Chapter 1, we discuss some preliminaries, such as the history of Emdros, as well as giving an overview of the formalism used to define the MQL language, called Backus-Naur Form (or BNF).

In Chapter 2, we give an overview of the EMdF model, from a user's standpoint.

In Chapter 3, we describe the bulk of the MQL language.

In Chapter 4, we describe the query-subset of MQL. That is, the subset of MQL in which you can express queries that look for objects by object type, features, and structural relationships such as embedding and sequence.

# Contents

# Chapter 1

# Preliminaries

## 1.1 Introduction

Welcome to the MQL Programmer's Guide. MQL is the query language associated with the EMdF model. MQL is a "full access language," which simply means that MQL lets you create, update, delete, and query most of the data domains in the EMdF model – databases, objects, object types, features, etc. MQL is your front-end to the EMdF database engine. This guide helps you formulate MQL queries that do what you need done with your EMdF database.

This guide has four chapters. The first is this chapter on preliminaries. The second is a gentle introduction to the EMdF model, which underlies the MQL language. The third chapter deals with the bulk of the MQL query language, detailing all the different kinds of queries for creating, updating, deleting, and querying an EMdF database. The fourth chapter is a special chapter devoted to explaining those MQL queries that query for objects in the database. Since these queries are so rich, it was deemed necessary to devote a whole chapter to their treatment.

This chapter will proceed as follows. First, we present a short history of MdF, EMdF, and MQL. Second, we give a gentle introduction to Backus-Naur Form, or BNF, which will be used throughout chapters 3 and 4. Lastly, we explain the origin of the support for something called *regular expressions* in MQL. This is done so as to comply with the license for the library used.

But first, an explanation of where EMdF and MQL come from.

## 1.2 Origins of MdF, EMdF, and MQL

EMdF and MQL are not original works. They are merely derivative works based on someone else's hard labors. Most of the ideas underlying the EMdF model and the MQL query language are to be found in the PhD thesis of Crist-Jan Doedens, published in 1994 as [Doedens94]. This thesis described, among other things, the MdF database model and the QL query language. As one might guess, EMdF stems from MdF, and MQL stems from QL.

The EMdF model takes over the MdF model in its entirety, but adds a few concepts which are useful when implementing the MdF model in real life. Thus the 'E' in 'EMdF' stands for "Extended", yielding the "Extended MdF model". The EMdF model is the subject of chapter 2.

"MQL" stands for "Mini QL." Originally, I devised MQL as a subset of the QL query language developed in Dr. Doedens' PhD thesis, hence the "Mini" modifier. Since then, however, MQL has grown. QL was not a full access language, but specified only how to query an MdF database, i.e., how to ask questions of it. MQL, by contrast, is a full access language, allowing not only querying, but also creation, update, and deletion of the data domains of the EMdF model. The MQL query language is the subject of chapters 3 and 4.

Thus EMdF and MQL are derivatives of the MdF database model and the QL query language developed by Dr. Crist-Jan Doedens in his 1994 PhD thesis.

## 1.3 Introduction to Backus-Naur Form

### 1.3.1 Context-Free Grammars

BNF is a way of specifying the "syntactic rules" of a language. English also has "syntactic rules," and some of them can be specified using a "Context-Free Grammar." BNF is precisely a way of specifying a context-free grammar for a formal language. Thus it is beneficial first to see what a context-free grammar is, before looking at the details of BNF.

In English, the basic clause-pattern is "Subject - Verb - Object". For example, in the clause "I eat vegetables," the word "I" is the subject, the word "eat" is the verb, and the word "vegetables" is the object. A clause which exhibits exactly the same "Subject - Verb - Object" structure is "You drink coke." Here, "You" is the subject, "drink" is the verb, and "coke" is the object.

Consider the following context-free grammar:

$$
\begin{aligned}
&\texttt{Sentence} \rightarrow \texttt{NP}_{subj}\ \texttt{VP} \\
&\texttt{NP}_{subj} \rightarrow \texttt{"I"} \mid \texttt{"You"} \\
&\texttt{VP} \rightarrow \texttt{V}\ \texttt{NP}_{obj} \\
&\texttt{V} \rightarrow \texttt{"eat"} \mid \texttt{"drink"} \\
&\texttt{NP}_{obj} \rightarrow \texttt{"vegetables"} \mid \texttt{"coke"}
\end{aligned}
$$

This little context-free grammar is a toy example of a context-free grammar. However, despite its simplicity, it exemplifies all of the concepts involved in context-free grammars:

- Rule
- Non-terminal
- Terminal
- Choice
- Concatenation
- Start-symbol.

These will be described in turn below

#### 1.3.1.1 Rule

A "Rule" consists of three parts:

1. The left-hand side
2. The "production arrow" ("$\rightarrow$").
3. The right-hand side

An example of a rule in the above context-free grammar is:

$$\texttt{Sentence} \rightarrow \texttt{NP}_{subj}\ \texttt{VP}$$

It specifies that the left-hand side ("Sentence") can be *replaced with* the right-hand side, which in this case is two symbols, "$\texttt{NP}_{subj}$" followed by "VP". Sometimes, we also say that a left-hand side is *expanded to* the right-hand side.

#### 1.3.1.2 Non-terminal

There are only two kinds of symbols in a context-free grammar: Non-terminals and Terminals. They are a contrasting pair. In this section, we describe what a non-terminal is, and in the next section, what a terminal is.

A "Non-terminal" is a symbol in a rule which can be expanded to other symbols. Thus the symbols "Sentence", "NP$_{subj}$", "VP", "V", and "NP$_{obj}$" constitute all of the non-terminals of the above context-free grammar.

Only non-terminals can stand on the left-hand side of a rule. A non-terminal is defined as a symbol which can be expanded to or replaced with other symbols, and hence they can stand on the left-hand side of a rule. But as you will notice in the above context-free grammar, a non-terminal can also stand on the right-hand-side of a rule. For example, the non-terminal "V" is present both in the rule for how to expand the non-terminal "VP", and in the rule for how to expand itself. Thus, in order to expand "VP" fully, you must first expand to the right-hand-side "V NP$_{obj}$", and then expand both "V" and "NP$_{obj}$", using the rules for these two.

#### 1.3.1.3 Terminal

A "Terminal" is a symbol in a rule which cannot be expanded to other symbols. Hence, it is "terminal" in the sense that the expansion cannot proceed further from this symbol. In the above context-free grammar, the terminals are: "I", "You", "eat", "drink", "vegetables", and "coke". These are symbols which cannot be expanded further.

Terminals can only stand on the right-hand side of a rule. If they were to stand on the left-hand-side of the rule, that would mean that they could be expanded to or replaced with other symbols. But that would make them non-terminals.

#### 1.3.1.4 Choice

In the rule for "V" in the above grammar, we see an example of choice. The choice is indicated by the "|" symbol, which is read as "or". Thus, this example:

```
V → "eat" | "drink"
```

is read as 'V expands to "eat" *or* "drink"'.

#### 1.3.1.5 Concatenation

We have already seen an example of concatenation, namely in the rule for "Sentence":

Sentence → NP$_{subj}$ VP

Here, the symbols "NP$_{subj}$" and "VP" are *concatenated*, or placed in sequence. Thus "VP" comes immediately after "NP$_{subj}$".

"Concatenated" is simply a fanciful name for "being in sequence", but although it is a basic idea, we included it for completeness.

#### 1.3.1.6 Start-symbol

The start-symbol is merely the left-hand side non-terminal of the first rule in the grammar. Thus, in the above grammar, "Sentence" is the start-symbol.

### 1.3.2 Context-free grammars: Putting it all together

It is time to see how all of this theory works in practice. The above grammar can produce 8 sentences, some of which do not make sense:

1. I eat vegetables

2. I eat coke
3. I drink vegetables
4. I drink coke
5. You eat vegetables
6. You eat coke
7. You drink vegetables
8. You drink coke

Let us pick one of these sentences and see how it was produced from the above grammar. We will pick number 8, "You drink coke", and trace all the steps. We start with the start-symbol, "Sentence":

1. Sentence

   This is expanded using the rule for "Sentence":

2. NP$_{subj}$ VP

   The "NP$_{subj}$" non-terminal is then expanded to "You" using one of the choices in the rule for NP$_{subj}$:

3. "You" VP

   The "VP" is then expanded using the rule for "VP":

4. "You" V NP$_{obj}$

   The "V" non-terminal is then expanded to the terminal "drink":

5. "You" "drink" NP$_{obj}$

   The "NP$_{obj}$" non-terminal is then expanded to the terminal "coke":

6. "You" "drink" "coke"

   Which yields the final sentence, "You drink coke". This sentence has no non-terminals, only terminals, and therefore it cannot be expanded further. We are finished.

If you would like to, try to trace the production of one of the other sentences using pencil and paper, tracing each step as in the above example. When you have done so once or twice, you should understand all there is to understand about context-free grammars.

And BNF is simply a way of specifying a context-free grammar. So let us start looking at the details of BNF.

### 1.3.3 BNF

#### 1.3.3.1 Introduction

BNF comes in various variants, and almost everyone defines their usage of BNF a little differently from everyone else. In this document, we shall also deviate slightly from "standard BNF", but these deviations will only be very slight.

This treatment of BNF will be made from an example of a context-free grammar in "MQL BNF." This example covers every formalism used in "MQL BNF," and is a real-life example of the syntax of an actual MQL statement:

#### 1.3.3.2 Example

```
create_enumeration_statement : "CREATE"
            ("ENUMERATION" | "ENUM")
            enumeration_name "="
            "{" ec_declaration_list "}"
;
enumeration_name : T_IDENTIFIER
   /* The T_IDENTIFIER is a terminal
      denoting an identifier. */
;
ec_declaration_list : ec_declaration { "," ec_declaration }
;
ec_declaration : [ "DEFAULT" ]
              ec_name [ ec_initialization ]
;
ec_name : T_IDENTIFIER
;
ec_initialization : "=" T_INTEGER
;
```

#### 1.3.3.3 Elements of "MQL BNF"

All of the elements of "MQL BNF" can be listed as follows:

1. Rule

   This is just the same as in the context-free grammars above, except that the "→" production arrow is replaced by a ":" colon. Also, a rule ends with a ";" semicolon.

2. Non-terminal

   Non-terminals always start with a lower-case letter, e.g., "ec_declaration."

3. Terminal

   Terminals are either strings in "double quotes" or strings that start with "T_", e.g., "T_IDENTIFIER".

4. Choice

   This is the same as in the context-free grammars. The symbol "|" is used.

5. Concatenation

   This is the same as in the context-free grammars. A space between two symbols is used.

6. Start-symbol

   This is the same as in the context-free grammars. The left-hand side non-terminal of the first rule is the start-symbol.

7. Comment

   A comment is enclosed in /* slashes and stars */. The comments are not part of the grammar, but serve to explain a part of the grammar to the human reader.

8. Optional specification

   A symbol or sequence of symbols enclosed in [square brackets] is considered optional. For example, the `ec_initialization` non-terminal is optional in the `ec_declaration` rule. Both terminals and non-terminals can be optional. Here, the choice is between "ENUM" and "ENUMERATION", rather than, say, "CREATE ENUMERATION" and the rest of the rule.

9. Bracketing

   Sometimes, we do not go to the trouble of spelling out a choice with a rule by itself. Instead, (brackets) are used. For example, in the above grammar, there is a choice between "ENUMERATION" and "ENUM". Only one must be chosen when writing the CREATE ENUMERATION statement. The brackets serve to let you know the scope of the choice, i.e., between exactly which elements the choice stands.

10. Bracing (repetition)

    The { braces } are used to indicate that the enclosed part can be repeated zero or more times. For example, in the rule for `ec_declaration_list`, the first `ec_declaration` can be followed by zero or more occurrences of first a "," comma and then an `ec_declaration`. This effectively means that the `ec_declaration_list` is a comma-separated list of `ec_declaration`s, with one or more `ec_declaration`s.

    There is no comma after the last `ec_declaration`. To see why, notice that the part that is repeated starts with a comma and ends with an `ec_declaration`. Thus, no matter how many times you repeat this sequence, the `ec_declaration` will always be last, and the comma will never be last.

    That it can be repeated *zero* or more times simply means that it is optional. In the rule for `ec_declaration_list`, the first `ec_declaration` can stand alone.

    Notice also that, in the rule for `create_enumeration_statement`, there are braces as well. These braces, however, are enclosed in "double quotes" and are therefore terminals. Thus they do not have the meaning of repetition, but are to be written in the statement when writing a CREATE ENUMERATION statement. The braces without double quotes are repetition-braces and must not be written.

## 1.4 Acknowledgements

### 1.4.1 Thanks

Thanks to all the contributors mentioned in the AUTHORS file. Special thanks go to Dr. Crist-Jan Doedens, without whose PhD work, Emdros would not have existed. Also thanks to Prof. Dr. Eep Talstra, Prof. Dr. Nicolai Winther-Nielsen, and Prof. Dr. Peter Øhrstrøm for encouragement along the way. Thanks to Dr. Kirk E. Lowery for encouragement and other kinds of contributions. Thanks to Claus Tøndering for code contributions and other encouragements. Thanks to Constantijn Sikkel for encouragement and guidance, and for being an avid user of and bug-reporter for Emdros. Thanks to Prof. Dr. Oliver Glanz for being an avid user and requester of features.

### 1.4.2 Regular expression support

MQL has support for regular expressions in queries. Regular expression support is provided by the PCRE library package, which is open source software, written by Philip Hazel, and copyright by the University of Cambridge, England. The PCRE library can be downloaded from:

`ftp.csx.cam.ac.uk/pub/software/programming/pcre/`

The PCRE included with Emdros is a modified copy.
We'll get back to regular expressions in section 4.7.7.8.4 and in section A.3. See also the index.

# Chapter 2

# The EMdF model

## 2.1 Introduction

This chapter is a gentle introduction to the "EMdF model". The EMdF model is a "database model." As such, it provides a solid theoretical foundation for the EMdF database engine and the MQL query language. However, its importance goes beyond being a theoretical foundation. The EMdF model defines the very way we talk about text in an EMdF database, and as such, it provides the "vocabulary" of the language by which you, the user, communicate with the EMdF database engine. Put another way, the EMdF model defines the concepts which you use to talk about text when communicating with the EMdF database engine. Thus it is very important that you understand all of the concepts involved in the EMdF model. However, these concepts are neither many nor difficult to understand. This chapter is designed to help you understand them, whatever your background.

This chapter is built around the concepts in the EMdF model: monads, objects, object types, features, and a few other concepts. These four concepts: monads, objects, object types, and features, form the backbone of the EMdF model. Once you have understood these, the rest are mere extensions which follow fairly easily.

And so, with no further ado, let us jump into the first of the four major concepts, monads.

## 2.2 Monads

### 2.2.1 Monads and textual sequence

Language is linear in nature. We produce language in real-time, with sequences of sounds forming sequences of words. Text, being language, is also linear in nature. In the EMdF model, this linearity is captured by the monads.

A monad is simply an integer – no more, no less. The sequence of integers (1,2,3,4,...) forms the backbone to which the flow of the text is tied. Thus, because a monad is an integer, and because we can order the integers in an unambiguous way, we use the sequence of monads to keep track of the sequence of textual information.

The sequence of monads begins at 1 and extends upwards to some large number, depending on the size of the database.

### 2.2.2 Granularity

What unit of text does a monad correspond to? For example, does a monad correspond to a morpheme, a word, a sentence, or a paragraph?

The answer is that you, the database user, decide the granularity of the EMdF database. You do this before any text is put into the database. If you want each monad to correspond to a morpheme, you simply decide that this is so. A more common choice is for each monad to correspond to a word. However, there

is nothing implicit or explicit in the EMdF model that prevents the user from deciding that another unit of text should correspond to a monad. Be aware, however, that once the choice has been made, and the database has been populated with text, it is not easy to revoke the decision, and go from, say, a word-level granularity to a morpheme-granularity.

### 2.2.3 Text flow

What is the reading-order of an EMdF database? Is it left-to-right, right-to-left, top-to-bottom, or something else?

The answer is that the EMdF model is agnostic with respect to reading-order. In the EMdF model, what matters is the textual sequence, as embodied by the monads. How the text is displayed on the screen is not specified in the EMdF model.

The MQL query language provides for both 7-bit (ASCII) and 8-bit encodings of strings, which means that the database implementor can implement any character-set that can be encoded in an 8-bit encoding, including Unicode UTF-8.

### 2.2.4 Conclusion

A monad is an integer. The sequence of integers (1,2,3,4,...) dictates the textual sequence. The granularity of an EMdF database is decided by the database-implementor. The EMdF model is agnostic with respect to reading-order (right-to-left, left-to-right, etc.).

## 2.3 Objects

### 2.3.1 What is an object?

An object is a set of monads. Thus, for example, an object might consist of the monads {1,2,4}. This object could, for example, be a phrase-object consisting of three words (assuming the monad-granularity is "one monad, one word").

The EMdF model does not impose any restrictions on the set of monads making up an object. For example, objects can be discontiguous, as in the above example. In addition, objects can have exactly the same monads as other objects, and objects may share monads.

We need the last two concepts, object types and features, before we can understand exactly how an object can encode, say, a word or a phrase.

## 2.4 Object types

### 2.4.1 Objects are grouped in types

In any populated EMdF database, there will be at least one object type. Otherwise, no objects can be created, and thus the database cannot store textual information.

Objects are grouped in types, such as "word", "phrase", "clause", "sentence", but also "chapter", "part", "page", etc. Each of these are potential object types that the user can create. Once an object type has been created, objects of that type can also be created.

Any object is of exactly one object type. Object types are what gives objects their interpretation. For example, an object of type "phrase" is, by itself, just a set of monads, such as {1,2,4}. But seen in conjunction with its object type, it becomes possible to interpret those monads as a number of words that make up a phrase.

An object type is also what determines what *features* an object has. And thus we turn to the last major concept in the EMdF model, namely features.

## 2.5 Features

### 2.5.1 What is a feature?

A feature is a way of assigning data to an object. For example, say we have an object of type "word". Let us call this object "O", and let us say that it consists of the singleton monad set $\{1\}$. Assume, furthermore, that the object type "word" has a feature called "surface_text". Then this feature, taken on the object O, might be the string "In". This is denoted as "O.surface_text". If we have another object, $O_2$, which consists of the singleton monad set $\{2\}$, the value $O_2$.surface_text might be "the". Thus we know that the first text in this EMdF database starts with the words "In the".

### 2.5.2 Object types have features

An object type has a fixed number of features defined by the database implementor. For example, it might be necessary for a particular application to have these features on the object type "word":

- surface_text
- lexical_form
- part_of_speech
- preceding_punctuation
- trailing_punctuation
- ancestor

The "ancestor" feature would be a pointer to another object, allowing the user to create an immediate constituency hierarchy.

The object type "phrase" might have the following list of features:

- phrase_type
- ancestor

### 2.5.3 Features have types

An object type has a number of features. Each feature, in turn, has one type. In the current implementation of the EMdF model, a feature can have one of the following types:

- integer
- string (which is an 8-bit string)
- ascii (which is a 7-bit ASCII string)
- id_d (which is an object id – we will get to this later)
- enumeration (which we will also get to later in this chapter)
- list of integer
- list of id_d
- list of enumeration constants
- sets of monads

## 2.6 Example

We have now defined all of the four major concepts in the EMdF model: monad, object, object type, and feature. It is time to make them more concrete by giving an example of a very small EMdF database. Look at figure 2.1. At the top of the figure, you see the sequence of monads, 1 to 9. Just below this sequence, you

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| Word | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| surface | The | door, | which | opened | towards | the | East, | was | blue. |
| part_of_speech | def.art. | noun | rel.pron. | verb | prep. | def.art. | noun | verb | adject. |
| Phrase | 1 | | 2 | 3 | | 5 | | 6 | 7 |
| phrase_type | NP | | NP | VP | | NP | | VP | AP |
| Phrase | | | | | 4 | | | | |
| phrase_type | | | | | PP | | | | |
| Clause_atom | 1 | | 2 | | | | | 3 | |
| Clause | 1 | | 2 | | | | | 1 | |
| Sentence | 1 | | | | | | | | |

Figure 2.1: Exemplifying the four major concepts of the EMdF model: monad, object, object type, and feature.

see the object type "Word" with its object ordinals, 1 to 9. In this example, the granularity is "one monad, one word." Thus *Word* number 1 corresponds to *monad* number 1, but they are really separate entities. Word number 1 *consists of* the set of monads {1}.

This becomes clearer when you notice that Clause number 2 consists of the set of monads {3,4,5,6,7}. Thus there is a fundamental distinction between the number of an object (also called object ordinal), and the set of monads making up that object.

Some of the object types in the figure (Word and Phrase) have a number of *features*. The object type "Word" has the features "surface" and "part_of_speech". The "Phrase" object type has only one feature, namely "phrase_type".

Notice that objects can be discontiguous. The Clause object with object ordinal 1 consists of the monads {1,2,8,9}. Thus there can be gaps in an object.

Notice also that an object type need not have features. The object types Clause_atom, Clause, and Sentence have no features in the figure.

## 2.7 Other concepts

### 2.7.1 Introduction

Having learned the basic concepts of the EMdF model, we now turn to the additional concepts which we use to talk about EMdF databases. These concepts are:

1. The special object types pow_m, any_m, and all_m
2. object ids (id_d, id_m)

3. self
4. part_of
5. gaps
6. borders, first, and last
7. enumerations
8. min_m and max_m
9. arbitrary monad sets
10. databases

### 2.7.2 pow_m

In each EMdF database, we assume an abstract object type, pow_m. This object type has one object for every possible set of monads. Thus the pow_m object type has objects consisting of the sets {1},{2},{3}, ..., {1,2}, {1,3}, {1,4}, ..., {2,3},{2,4},{2,5},...,{1,2,3},{1,2,4}, ..., etc. Every possible set of monads is represented in the pow_m object type.

The pow_m object type is an *abstract* object type. That is, no objects of type pow_m actually exist in the EMdF database. However, it is useful to be able to talk about a particular pow_m object. In effect, a pow_m object is simply a set of monads, and sometimes, it is convenient to be able to talk about a particular pow_m object. This is especially true with gaps (see below).

The pow_m object type has no features.

### 2.7.3 any_m

The any_m object type is an abstract object type like pow_m. Each of its objects consist of a single monad. So the any_m objects are: {1}, {2}, {3}, ... etc. The any_m object type has no features.

### 2.7.4 all_m

The all_m object type has only one object, and it consists of all the monads in the database. That is, it consists of the monads from min_m to max_m (see sections 3.8.9 on page 57 and 3.8.10 on page 58), the smallest and the largest monads in use in the database at any given time. This one object is called all_m-1.

### 2.7.5 object ids (id_d, id_m)

Each object in the database (apart from pow_m objects) has an *object id_d*. An object id_d is a unique ID assigned to the object when the object is created. The id_d is used only for that particular object, and the id_d is never used again when the object is deleted.

A feature can have the type "id_d", meaning that the values of the feature are taken from the id_ds in the database.

Each object in the database (including pow_m, any_m, and all_m objects) also has an id_m. The id_m is simply the set of monads which makes up the object. This is not strictly an ID, since objects of the same object type may have exactly the same monads. However, for historical reasons, this is called an id_m. See [Doedens94] or [Standard-MdF] for details.

### 2.7.6 self

Each object type in the database (apart from the pow_m, any_m, and all_m object types) has a feature called "self". This is used to get the object id_d of the object in question.

The "self" feature is a read-only feature, which means that you cannot update an object's self feature, or write to it when creating the object. The value of the "self" feature is assigned automatically when the object is created.

The type of the "self" feature is "id_d".

### 2.7.7 part_of

If all of the monads of one object, $O_1$, are contained within the set of monads making up another object, $O_2$, we say that $O_1$ is part_of $O_2$.

For example, an object with the monads {1,2} would be part_of another object with the monads {1,2,4}.

In mathematical terms, $O_1$ is part_of $O_2$ if and only if $O_1 \subseteq O_2$.

### 2.7.8 gaps

Objects may have gaps. A gap in an object is a maximal stretch of monads which are not part of the object, but which are nevertheless within the boundaries of the endpoints of the object. For example, an object consisting of the monads {1,3,4,7,8,13} has three gaps: {2}, {5,6}, and {9,10,11,12}.

Note that gaps are always maximal, i.e., extend across the whole of the gap in the object. For example, {6} is not a gap in the above object: instead, {5,6} is.

### 2.7.9 borders, first, last

Each non-empty object, being a set of monads, has a left border and a right border. The left border is the lowest monad in the set, while the right border is the highest monad in the set. These are also called the first monad and the last monad in the object. If we have an object O, the notation for these monads is O.first and O.last.

For example, if we have an object O consisting of the monads {2,3,4,5}, then O.first = 2 and O.last = 5.

### 2.7.10 consecutive with respect to a set of monads

The basic idea is that two sets of monads are consecutive if they follow each other without any gaps in between. However, this idea is extended so that the "no gaps in between" is interpreted with respect to a reference set of monads called Su. For example, if Su = {1,2,5,6}, then the sets {2} and {5} are consecutive with respect to Su. However, the sets {2} and {6} are not consecutive with respect to Su, since there is a "gap" consisting of the monad 5 in between the two sets. Likewise, the sets {1} and {5} are not consecutive with respect to Su, because Su has a monad, 2, which is a "gap" between the two sets.

### 2.7.11 enumerations

#### 2.7.11.1 Definition

Each feature, it will be remembered, is of a certain type. These can be integers, strings, and id_ds, but they can also be enumerations. An enumeration is a set of pairs, where each pair consists of a constant-identifier and an integer value.

#### 2.7.11.2 Example

For example, the enumeration "phrase_type_t" might have the pairs of constants and values as in table 2.1 on the following page.

| constant | value |
|---|---|
| phrase_type_unknown | -1 |
| VP | 1 |
| NP | 2 |
| AP | 3 |
| PP | 4 |
| AdvP | 5 |
| ParticleP | 6 |

Table 2.1: phrase_type_t enumeration

#### 2.7.11.3 Default constant

Each enumeration has exactly one default constant which is used when the user does not give a value for a feature with that enumeration type. In this example, "phrase_type_unknown" might be the default.

#### 2.7.11.4 Terminology

The constants are called *enumeration constants*, while the type gathering the enumeration constants into one whole is called an *enumeration*.

#### 2.7.11.5 Names are identifiers

The names of both enumerations and enumeration constants must be *identifiers*. See section 3.1.3 on page 25 for information on what an identifier is.

#### 2.7.11.6 Each enumeration is a name-space

Each enumeration forms its own namespace. All name-spaces in the MQL language are orthogonal to each other. This means that two enumeration constants within the same enumeration cannot be called by the same constant-identifier, but two enumeration constants in two different enumerations may be the same. For more information, see section 3.1.4 on page 26 for more information.

#### 2.7.11.7 Enumeration constants must be unique

Enumeration constants must be unique within each enumeration, both in their values and in their names. For example, you cannot have two labels with the same name in the same enumeration. Nor can you have two labels with the same value in the same enumeration, even if the labels have different names.

This is different from C or C++ enumerations, where the same value can be assigned to different labels.

Thus an enumeration is effectively a one-to-one correspondence (also called a bijective function) between a set of label names and a set of values.

### 2.7.12 min_m, max_m

An EMdF database has a knowledge of which is the smallest monad in use (min_m) and which is the largest monad in use (max_m). Normally, you don't need to worry about these; the database maintains its knowledge of these monads without your intervention. You can, however, query the database for the minimum and maximum monads (see sections 3.8.9 on page 57 and 3.8.10 on page 58), and when you query the database for objects (section 3.8.1 on page 45), this is done within the confines of the minimum and maximum monads. Thus it is useful to know of their existence, but you needn't worry too much about them.

The associated statements are SELECT MIN_M (section 3.8.9 on page 57) and SELECT MAX_M (section 3.8.10).

### 2.7.13 Arbitrary monad sets

Each database has a central repository of monad sets which are not associated with any objects. That is, they are not objects, have no object type, and no features. They are just plain monad sets.

These monad sets can be used as the basis for searches. That is, when doing a SELECT ALL OBJECTS query (or SELECT FOCUS OBJECTS), one can specify within which arbitrary monad set the search should be conducted.

The associated statements are SELECT MONAD SETS (section 3.8.11 on page 58), GET MONAD SETS (section 3.8.12 on page 59), CREATE MONAD SET (section 3.11.2 on page 72), UPDATE MONAD SET (section 3.11.3 on page 72), and DROP MONAD SET (section 3.11.4 on page 73).

### 2.7.14 Computed features (monad_set_length, monad_count, first_monad, last_monad)

The following computed features can always be used on any object type, even though they are never declared:

- first_monad(<monad_set_name>)
- last_monad(<monad_set_name>)
- monad_count(<monad_set_name>)
- monad_set_length(<monad_set_name>)

The "<monad_set_name>" inside the parentheses must be a valid monad set name, such as the name of a feature of type "SET OF MONADS", or the privileged monad set feature, "monads".

If the <monad_set_name> to be used is the privileged "monads" monad set feature, the parenteses and the name "monads" can be omitted, as a short-cut.

The features mean the folllowing:

**first_monad(<monad_set_name>):** Retrieve the <monad_set_name> monad set, and yield its smallest (first) monad. For example, the monad set "{1-3}" has first_monad = 1.

**last_monad(<monad_set_name>):** Retrieve the <monad_set_name> monad set, and yield its largest (last) monad. For example, the monad set "{1-3}" has first_monad = 3.

**monad_count(<monad_set_name>):** Retrieve the <monad_set_name> monad set, and yields the number of monads in it, i.e., its cardinality. For example, the monad set "{1-3, 5}" has a monad_count of 4, since it has 4 monads in it ("1, 2, 3, 5"). Also, the monad set "{3}" has a monad_count of 1, while the monad set "{2-4}" has a monad_count of 3 ("2, 3, 4").

**monad_set_length(<monad_set_name>):** Retrieve the <monad_set_name> feature, and yield the number

last_monad(<monad_set_name>) - first_monad(<monad_set_name>) + 1

For example, the set "{1-3, 5}" has a monad_set_length of 5 - 1 + 1 = 5.
Note that if the monad set has no gaps, then this will be the same as monad_count(<monad_set_name>). If, on the other hand, the monad set has gaps, the monad_set_length(<monad_set_name>) will be strictly greater than monad_count(<monad_set_name>).

### 2.7.15 Databases

The EMdF model has a concept of "database." It is an organizational concept which generally corresponds to what the back-end database system calls a "database." Within a database, there is one string of monads starting at 1 and extending upwards to some very large number. Within this stretch of monads, the user is free to create objects.

You may need to issue the USE DATABASE statement (see section 3.3.3 on page 31) as the first thing you do before doing anything else, in order to tell Emdros which database you want to deal with. Ask the implementor of your Emdros application whether this is what you should do.

A database can be created with the CREATE DATABASE statement (see section 3.3.1 on page 29).

## 2.8 Encryption

Dr. D. Richard Hipp, the author of SQLite, makes an encryption-enabled version available for a fee. There is skeleton support for SQLite encryption in Emdros, meaning one should be able to use Dr. Hipp's encryption-enabled version of SQLite with Emdros and get encryption-support in Emdros on SQLite. This has not been tested, however; only the skeleton is there. See <http://www.hwaci.com>, the website of Dr. Hipp's consulting company, for more information about Dr. Hipp's encryption.

In this manual, when we speak of "encryption" on SQLite, please be aware that the actual encryption is not a part of Emdros, and you will achieve the exact same results and generate the exact same Emdros databases whether you use a key or not, unless you obtain an encryption-enabled SQLite from somewhere.

# Chapter 3

# MQL database manipulation

## 3.1 Preliminaries

### 3.1.1 Introduction

In this section on preliminaries, we will talk about four things. First, we describe what terminals are used in the grammar-fragments in this manual. Second, we describe the lexical conventions of MQL. Third, we describe the name-spaces available in MQL. And finally, we describe some top-level constraints in MQL syntax.

### 3.1.2 Terminals

The following terminals are used in this grammar:

- T_IDENTIFIER
- T_INTEGER
- T_STRING
- T_MARKS
- `"strings"`, e.g., `"OBJECT"`.

The fifth kind, e.g., `"OBJECT"` represent keywords in MQL. They are parsed as case-insensitive strings without the quotes.

### 3.1.3 Lexical conventions

The lexical conventions for MQL are as follows:

1. There are two kinds of comments:
   (a) Enclosed in "/*" (opening) and "*/" (closing). This kind of comment can span multiple lines. This is the same as C-style comments.
   (b) Starting with "//" and ending at the end of the line. This is the style used in C++.
2. All keywords (such as `"CREATE"`, `"SELECT"`, `"<="`, etc.) are case-*in*sensitive insofar as they are made up of letters. Keywords are enclosed in `"double quotes"` in the syntax-sections below.

| T_IDENTIFIER referent | Case-sensitivity |
|---|---|
| Database name | insensitive |
| Object type name | insensitive |
| Enumeration name | insensitive |
| Enumeration constant name | sensitive |

Table 3.1: Case-sensitivity of T_IDENTIFIERs

3. A T_IDENTIFIER begins with any letter (a-z,A-Z) or an underscore (_), followed by zero or more letters (a-z,A-Z), numbers (0-9), or underscores (_). For example, “Word”, “_delimiter”, “i18n”, and “phrase_type_t” are all identifiers. However, “8bf” is not an identifier because it does not start with a letter or an underscore. Neither is bf@foo.com an identifier, because it does not consist solely of letters, underscores, and numbers.

   Whether a T_IDENTIFIER is case-sensitive depends on what it stands for (i.e., what its “referent” is). See table 3.1 for a description.

4. A T_INTEGER is any sequence of one or more digits (0-9). For example, “0”, “42”, and “747” are all integers.

5. A T_STRING is one of two kinds:

   (a) A T_STRING can start with a single quote (’), followed by zero or more characters which are not single quotes, and ending with another single quote (’). Such a string can contain newlines.

   (b) A T_STRING can also start with a double quote ("), followed by zero or more characters, escape-sequences (see table 3.2), or newlines, and ending with a double quote (").

6. A T_MARKS is a sequence of one or more identifiers, each prefixed by a backping (‘). For example, the following are all T_MARKS: “‘yellow”, “‘red‘context”, “‘marks‘are‘useful”, “‘Flash_Gordon‘was‘a‘Hero”. More precisely, a T_MARKS conforms to the regular expression "(‘[a-zA-Z_][a-zA-Z_0-9]*)+".

7. White-space (spaces, newlines, and tabs) is ignored except in T_STRINGs.

| Escape sequences | Meaning |
|---|---|
| \n | newline (ASCII 10) |
| \t | horizontal tab (ASCII 9) |
| \v | vertical tab (ASCII 11) |
| \b | backspace (ASCII 8) |
| \a | bell (ASCII 7) |
| \r | carriage-return (ASCII 13) |
| \f | form-feed (ASCII 12) |
| \\ | slash (\) (ASCII 92) |
| \? | question-mark (?) (ASCII 63) |
| \’ | single quote (’) (ASCII 39) |
| \" | double quote (") (ASCII 34) |
| \ooo | Octal number (e.g., \377 is 255) |
| \xXX | Hexadecimal number (e.g., \xFF is 255) |

Table 3.2: Escape sequences in strings enclosed in double quotes.

### 3.1.4 Name-spaces

A name-space, in computer-terminology, is a language-domain with fixed borders within which names must be unique. *Within* a name-space, two different entities *cannot* be called by the same name without

causing a name-clash. In other words, within a name-space, names must be unique. However, if two name-spaces are *orthogonal* to each other, then a name from one name-space *can* be the same as a name from the other name-space *without* causing a name-clash.

In MQL, the following name-spaces exist. They are all orthogonal to each other:

- Each object type forms a name-space with respect to its features. That is, a single object type cannot have two features with the same name, but different object types can have features with the same name. The two features with the same name need not even have the same feature type. This is because all name-spaces are orthogonal to each other.
- Each enumeration forms a name-space with respect to its constants. That is, a single enumeration cannot have two enumeration constants with the same name, but different enumerations can have enumeration constants with the same name. Since all name-spaces are orthogonal to each other, the two enumeration constants with the same name need not have the same integer value.
- Each database forms a global name-space with respect to object type names. That is, object type names must be globally unique within a database. However, since all name-spaces are orthogonal to each other, you can have features or enumeration constants which have the same name as an object type.

### 3.1.5 Top-level constraints on MQL syntax

The MQL engine can receive any number (greater than or equal to 1) of MQL statements. The only requirement is that each statement must end with the keyword "`GO`". This keyword acts as a delimiter between each statement. The last statement may also be terminated with "`GO`", but need not be. Single statements on their own need not be terminated with "`GO`" either.

If you connect to the MQL engine in daemon-mode, you must append the meta-level statement "QUIT" after the "GO" of the last statement.

## 3.2 Return types

### 3.2.1 Introduction

MQL is made up of statements, each of which either returns something or doesn't. If it returns something, there are two possibilities for what the return-type can be. It can be:

1. A table, or
2. A sheaf

The sheaf is explained in detail in section 4.4 on page 83. In section 3.2.3 below, we will treat the tables. But first, a word about the two output-formats available (plus the third option for getting data back).

### 3.2.2 Output-formats

The MQL engine gives you three options for using the results of an MQL query:

1. You can specify that you want XML output.
2. You can specify that you want output for displaying on a console.
3. You can use the datatype provided if your program is in the same process as the mql library.

If you use the mql(1) program for output, please see the manual page for how to choose the output kind.

### 3.2.3 Tables

The tables will look differently, depending on whether you choose XML-output or console-output. In the descriptions below, we will give abstract schemas for the tables, such as the following:

| object_type_name : string | monad : monad_m | id_d : id_d |
|---|---|---|

This means that, in each row in the table, the first piece of data will be a string (called object_type_name), the second piece of data will be a monad_m (called monad), and the last piece of data will be an id_d (called id_d). And then the row stops. There will always be the same number of columns in each row.

A table of values may be empty, meaning it has no rows. In this case, there will still be a table heading with type-specifications.

Some MQL statements do not return a value. In this case, there will be no result, not even an empty table.

### 3.2.4 Atomic output-types in tables

The following types can get into a table and will be announced in the header of the table:

1. string
2. integer
3. boolean (true or false)
4. id_d

### 3.2.5 Other return values

A number of other values are also returned from each query:

1. A boolean indicating whether there were any compiler-errors.
2. A boolean indicating whether there were any database-errors.
3. An integer showing which stage of the compilation/interpretation we had come to when we exited the function (see table 3.3). In XML, this is a string as shown in the table, of the attribute "stage" attribute of the "error_stage" element.
4. A string carrying any error messages.

| Stage | Value | XML string |
|---|---|---|
| None | 0 | none |
| Parsing | 1 | parse |
| Weeding | 2 | weed |
| Symbol-checking | 3 | symbol |
| Type-checking | 4 | type |
| Monads-checking | 5 | monads |
| Execution | 6 | exec |

Table 3.3: Compiler stages.See include/mql_execution_environment.h for ready-made #define macros.

## 3.3 Database manipulation

### 3.3.1 CREATE DATABASE

#### 3.3.1.1 Syntax

```
create_database_statement : "CREATE"  "DATABASE"
    database_name opt_WITH_KEY opt_USING_ENCODING
;
database_name : T_IDENTIFIER
              | T_STRING
;
opt_WITH_KEY : /* Empty: No key is used. */
             | "WITH"  "KEY"  T_STRING
;
opt_USING_ENCODING : /* Empty: Default encoding is used. */
                   | "USING"  "ENCODING"  T_STRING
;
```

#### 3.3.1.2 Example

```
CREATE DATABASE book_test
GO
CREATE DATABASE book_test_utf8
USING ENCODING 'utf-8'
GO
CREATE DATABASE book_test_latin1
USING ENCODING 'iso-8859-1'
GO
```

#### 3.3.1.3 Explanation

The CREATE DATABASE statement creates and initializes a database. No text data is put into the database, and no object types are created, but the structures necessary for the EMdF engine to function are set in place. The user need not worry about these structures. Interested users are referred to [Relational-EMdF].

You must CREATE a database before you can USE it (see section 3.3.3). Alternatively, if you have a database that is already created but not initialized, you can use the INITIALIZE DATABASE statememt (see Section 3.3.2 on the next page).

If a transaction was in progress (see BEGIN TRANSACTION statement, section 3.4.1 on page 34), the transaction is automatically committed before the CREATE DATABASE statement is executed. Thus the user need not, cannot, and should not commit it or abort it.

The database name can be either a T_IDENTIFIER or a T_STRING. For MySQL and PostgreSQL, it must be a T_IDENTIFIER. For SQLite, it can be a T_STRING giving the filename (optionally including the full path) of the file in which the database is to be created. If no path is given, the file is created in the current working directory.

The optional "WITH KEY" syntax can be used on SQLite to send a key to SQLite's sqlite_open_encrypted API when opening the database. Note that this will not actually perform any encryption at all unless you obtain an encryption-enabled SQLite from somewhere, e.g., Dr. Hipp himself, the author of SQLite. See Section 2.8 on page 24 for more information.

The optional "WITH ENCODING" syntax can be used to specify the default encoding to be used for the database when creating it in the backend database. Currently, the following values are supported:

- "utf-8"
- "iso-8859-1"

If the WITH ENCODING clause is not supplied, then the default encoding is used. The default encoding for each database is given in the following list:

- PostgreSQL: iso-8859-1
- MySQL: iso-8859-1
- SQLite 2: iso-8859-1
- SQLite 3: utf-8

For SQLite 3, the only encoding available is "utf-8". To specify any other encoding would be an error.

Note that the encoding specified only has a bearing on how the database backend interprets the data, not on how Emdros interprets the data. In fact, Emdros most likely will not interpret the data at all, but rather will pass whatever is stored in the database on to the application using Emdros, which must the interpret the data according to domain-specific knowledge of which encoding has been used.

#### 3.3.1.4 Return type

There is no return value.

### 3.3.2 INITIALIZE DATABASE

#### 3.3.2.1 Syntax

```
initialize_database_statement : "INITIALIZE"  "DATABASE"
    database_name opt_WITH_KEY
;
database_name : T_IDENTIFIER
              | T_STRING
;
opt_WITH_KEY : /* Empty: No key is used. */
             | "WITH"  "KEY"  T_STRING
;
```

#### 3.3.2.2 Example

```
INITIALIZE DATABASE book_test
GO
```

#### 3.3.2.3 Explanation

The INITIALIZE DATABASE statement initializes a database without creating it first. That is, the database must exist before issuing this statement. It simple creates all the meta-data necessary for having an Emdros database. This is useful on MySQL and PostgreSQL if you don't have privileges to create databases, but you do have privileges to create tables in an already-provided database. On SQLite, it is also useful, if you want to add Emdros information to an already-existing SQLite database. Other than not creating the database, this statement accomplishes the same things as the CREATE DATABASE statement (see Section 3.3.1 on the preceding page).

For the optional "WITH KEY" syntax, please see the CREATE DATABASE statement.

There is no "WITH ENCODING" syntax for the INITIALIZE DATABASE statement. This is because the encoding is only used when CREATEing the database. However, the internal metadata of the database is set to the default given under the explanation for CREATE DATABASE (see Section 3.3.1 on the previous page).

#### 3.3.2.4 Return type

There is no return value.

### 3.3.3 USE DATABASE

#### 3.3.3.1 Syntax

```
use_database_statement : "USE"  [ "DATABASE" ]
    database_name  opt_WITH_KEY
;
database_name : T_IDENTIFIER
              | T_STRING
;
opt_WITH_KEY : /* Empty: No key is used. */
             | "WITH"  "KEY"  T_STRING
;
```

#### 3.3.3.2 Example

```
USE book_test
GO
```

This is equivalent to

```
USE DATABASE book_test
GO
```

On SQLite:

```
USE DATABASE "c:\\EmdrosDBs\\mydb.db" /* On SQLite you can do this. */
GO
```

With a key:

```
/* On SQLite you can get encryption ****IF**** you have an
   encryption-enabled SQLite. */
USE DATABASE "c:\\Emdros\\MySecretDB.db"
/* The format and length of the key depends on your SQLite
   encryption implementation. This is just an example. */
WITH KEY "\x45\x98\xbf\x12\xfa\xc6"
GO
```

#### 3.3.3.3 Explanation

Before you can start using a database you have CREATEd (see section 3.3.1 on page 29) or INITIALIZEd (see section 3.3.2 on the previous page), you must connect to it using the USE DATABASE statement. The keyword "DATABASE" is optional and can be left out.

If a transaction was in progress (see BEGIN TRANSACTION statement, section 3.4.1 on page 34), the transaction is automatically committed before the USE DATABASE statement is executed. Thus the user need not, cannot, and should not commit it or abort it.

The database name can be either a T_IDENTIFIER or a T_STRING. For MySQL and PostgreSQL, it must be a T_IDENTIFIER. For SQLite, it can be a T_STRING giving the filename (optionally including the full path) of the file holding the database to be used. If no path is given, the file must be in the current working directory.

#### 3.3.3.4 Return type

There is no return value.

### 3.3.4 DROP DATABASE

#### 3.3.4.1 Syntax

```
drop_database_statement : "DROP" "DATABASE"
    database_name
;
database_name : T_IDENTIFIER
              | T_STRING
;
```

#### 3.3.4.2 Example

```
DROP DATABASE book_test
GO
```

#### 3.3.4.3 Explanation

A previously CREATEd database (see section 3.3.1) can be completely removed from the system using this statement. All data in the database is irretrievably lost, including all objects, all object types, and all enumerations.

If a transaction was in progress (see BEGIN TRANSACTION statement, section 3.4.1 on page 34), the transaction is automatically committed before the DROP DATABASE statement is executed. Thus the user need not, cannot, and should not commit it or abort it.

The database name can be either a T_IDENTIFIER or a T_STRING. For MySQL and PostgreSQL, it must be a T_IDENTIFIER. For SQLite, it can be a T_STRING giving the filename (optionally including the full path) of the file holding the database to be dropped. If no path is given, the file must be in the current working directory.

#### 3.3.4.4 Return type

There is no return value.

### 3.3.5 VACUUM DATABASE

#### 3.3.5.1 Syntax

```
vacuum_database_statement : "VACUUM" [ "DATABASE" ]
                            [ "ANALYZE" ]
;
```

#### 3.3.5.2 Example

```
1. VACUUM DATABASE
   GO

2. VACUUM DATABASE ANALYZE
   GO
```

#### 3.3.5.3 Explanation

On PostgreSQL, this statement vacuums the database using the "VACUUM" SQL statement. If the optional keyword "ANALYZE" is given, the statement issues a "VACUUM ANALYZE" statement. See the PostgreSQL documentation for what this does.

On MySQL, this statement issues OPTIMIZE TABLE queries for all object types. If the ANALYZE keyword is given, ANALYZE TABLE queries are issued as well.

On SQLite, this statement first deletes all redundant sequence info (compacting the sequence tables), then issues a VACUUM statement to SQLite.

The significance of this statement to Emdros development is that, when populating a database, things will speed up dramatically if the database is VACUUM'ed after every 1000 objects created, or so.

#### 3.3.5.4 Return type

There is no return value.

### 3.3.6 DROP INDEXES

#### 3.3.6.1 Syntax

```
drop_indexes_statement : "DROP" "INDEXES"
                         "ON" "OBJECT" ("TYPE" | "TYPES")
                         "[" object_type_to_drop_indexes_on "]"
;
object_type_to_drop_indexes_on : object_type_name | "ALL"
;
object_type_name : T_IDENTIFIER
;
```

#### 3.3.6.2 Example

```
1. DROP INDEXES
   ON OBJECT TYPES
   [ALL]
   GO

2. DROP INDEXES
   ON OBJECT TYPE
   [Word]
   GO
```

#### 3.3.6.3 Explanation

Emdros creates indexes on the tables associated with object types when they are created. These indexes speed up retrieval, but slow down insertion. Therefore, if you are going to insert a large amount of objects, it is best to drop indexes on the object types you are going to modify (possible all object types), then create the indexes again after you have inserted all objects.

This statement drop indexes that have previously been create. It has no effect if the indexes have been dropped already. If "ALL" is specified as the object type, then all object types in the current database will have their indexes dropped (if not dropped already).

The manage_indices program that comes with the Emdros distribution can be used to achieve the same effect.

Note that the choice between "TYPE" and "TYPES" is just syntactic sugar. It doesn't matter which you use.

If a feature has been declared WITH INDEX, this index is dropped. However, the feature will have its index recreated upon a CREATE INDEXES statement affecting that object type.

#### 3.3.6.4 Return type

There is no return value.

### 3.3.7 CREATE INDEXES

#### 3.3.7.1 Syntax

```
create_indexes_statement : "CREATE" "INDEXES"
                           "ON" "OBJECT" ("TYPE" | "TYPES")
                           "[" object_type_to_create_indexes_on "]"
;
object_type_to_create_indexes_on : object_type_name | "ALL"
;
object_type_name : T_IDENTIFIER
;
```

#### 3.3.7.2 Example

```
1. CREATE INDEXES
   ON OBJECT TYPES
   [ALL]
   GO

2. CREATE INDEXES
   ON OBJECT TYPE
   [Word]
   GO
```

#### 3.3.7.3 Explanation

Emdros creates indexes on the tables associated with object types when they are created. These indexes speed up retrieval, but slow down insertion. Therefore, if you are going to insert a large amount of objects, it is best to drop indexes on the object types you are going to modify (possible all object types), then create the indexes again after you have inserted all objects.

This statement creates indexes that have previously been dropped. It has no effect if the indexes are there already. If "ALL" is specified as the object type, then all object types in the current database will have their indexes created (if not there already).

The manage_indices program that comes with the Emdros distribution can be used to achieve the same effect.

Note that the choice between "TYPE" and "TYPES" is just syntactic sugar. It doesn't matter which you use.

#### 3.3.7.4 Return type

There is no return value.

## 3.4 Transactions

### 3.4.1 BEGIN TRANSACTION

#### 3.4.1.1 Syntax

```
begin_transaction_statement : "BEGIN"  "TRANSACTION"
;
```

#### 3.4.1.2 Example

```
BEGIN TRANSACTION
GO
```

#### 3.4.1.3 Explanation

On PostgreSQL, this statement begins a transaction if no transaction is in progress already. The return value is a boolean saying whether the transaction was started (true) or not (false). If this value is false, the user should not subsequently issue a COMMIT TRANSACTION or ABORT TRANSACTION statement. If this value is true, the user should issue either a COMMIT TRANSACTION or an ABORT TRANSACTION later.

On MySQL, this has no effect, and always returns false.

On SQLite, the behavior is the same as on PostgreSQL.

The transaction, if started, is automatically committed if a CREATE DATABASE, USE DATABASE, DROP DATABASE or QUIT statement is issued before a COMMIT TRANSACTION or ABORT TRANSACTION statement has been issued.

Also, the transaction is automatically committed if the connection to the database is lost, e.g., if the mql(1) program reaches the end of the MQL stream (e.g., an MQL script) and thus has to close down. Transactions are not maintained across invocations of the mql(1) program. The transaction is also committed if the EMdFDB, CMQL_execution_environment, or CEmdrosEnv object is destroyed.

#### 3.4.1.4 Return type

A table with the following schema:

| transaction_started : boolean |
|---|

This table is empty if and only if there was a compiler error, i.e., if the syntax was not obeyed. The statement cannot fail with a database error. If no transaction was started, false is returned. If a transaction was started, true is returned.

### 3.4.2 COMMIT TRANSACTION

#### 3.4.2.1 Syntax

```
commit_transaction_statement : "COMMIT"  "TRANSACTION"
;
```

#### 3.4.2.2 Example

```
COMMIT TRANSACTION
GO
```

#### 3.4.2.3 Explanation

Commits the current transaction, if one is in progress. Has no effect if a transaction was not in progress. In such cases, false is returned.

If the commit failed, false is returned. If the commit succeeded, true is returned.

NOTE that this is slightly different from other statements which flag a DB error if unsuccessful. Here, no DB error is flagged, but false is returned in the table.

#### 3.4.2.4 Return type

A table with the following schema:

| transaction_committed : boolean |
|---|

This table is empty if and only if there was a compiler error, i.e., if the syntax was not obeyed. The statement cannot fail with a database error. If no transaction was started when the COMMIT TRANSACTION statement was invocated, false is returned. If a transaction was started, and it was committed successfully, true is returned. If a transaction was started, but it was not committed successfully, false is returned.

### 3.4.3 ABORT TRANSACTION

#### 3.4.3.1 Syntax

```
abort_transaction_statement : "ABORT"  "TRANSACTION"
;
```

#### 3.4.3.2 Example

```
ABORT TRANSACTION
GO
```

#### 3.4.3.3 Explanation

Aborts the current transaction, if one is in progress. Has no effect if a transaction was not in progress. In such cases, false is returned.

If the abort failed, false is returned. If the abort succeeded, true is returned.

NOTE that this is slightly different from other statements which flag a DB error if unsuccessful. Here, no DB error is flagged, but false is returned in the table.

#### 3.4.3.4 Return type

A table with the following schema:

| transaction_aborted : boolean |
|---|

This table is empty if and only if there was a compiler error, i.e., if the syntax was not obeyed. The statement cannot fail with a database error. If no transaction was started when the ABORT TRANSACTION statement was invocated, false is returned. If a transaction was started, and it was aborted successfully, true is returned. If a transaction was started, but it was not aborted successfully, false is returned.

## 3.5 Object type manipulation

### 3.5.1 CREATE OBJECT TYPE

#### 3.5.1.1 Syntax

```
create_object_type_statement : "CREATE"
    [ "OBJECT" ] "TYPE"
    opt_if_not_exists
    opt_range_type
    opt_monad_uniqueness_type
    "["  object_type_name
           [ feature_declaration_list ]
    "]"
;
opt_if_not_exists:
   /* empty: Throw an error if object type exists already */
  | "IF"  "NOT"  "EXISTS"
;
opt_range_type:
    /* empty: Same as WITH MULTIPLE RANGE OBJECTS */
  | "WITH"  "SINGLE"  "MONAD"  "OBJECTS"
  | "WITH"  "SINGLE"  "RANGE"  "OBJECTS"
  | "WITH"  "MULTIPLE" "RANGE"  "OBJECTS"
;
```

```
opt_monad_uniqueness_type :
    /* empty: same as WITHOUT UNIQUE MONADS */
  | "HAVING"  "UNIQUE"  "FIRST"  "MONADS"
  | "HAVING"  "UNIQUE"  "FIRST"  "AND"  "LAST"  "MONADS"
  | "WITHOUT"  "UNIQUE"   "MONADS"
;
object_type_name : T_IDENTIFIER
;
feature_declaration_list : feature_declaration
    { feature_declaration }
;
feature_declaration :  feature_name  ":"  feature_type
    [ default_specification ]   ";"
    | feature_name ":" list_feature_type ";"
;
feature_type :
       "INTEGER"  [with_index_specification]
    |  "ID_D"     [with_index_specification]
    | "STRING"    [from_set_specification]  [with_index_specification]
    | "ASCII"     [from_set_specification]  [with_index_specification]
    | set_of_monads_specification
    | T_IDENTIFIER /* For enumerations. */
;
list_feature_type :
      "LIST"  "OF"  "INTEGER"
    | "LIST"  "OF"  "ID_D"
    | "LIST"  "OF"  T_IDENTIFIER  /* For enumerations */
;
with_index_specification :
    | "WITH"  "INDEX"
    | "WITHOUT"  "INDEX"
;
from_set_specification : "FROM"  "SET"
;
set_of_monads_specification :
    | "SINGLE" "MONAD" "SET" "OF" "MONADS"
    | "SINGLE" "RANGE" "SET" "OF" "MONADS"
    | "MULTIPLE" "RANGE" "SET" "OF" "MONADS"
    | "SET" "OF" "MONADS" /* Same as MULTIPLE RANGE SET OF MONADS */
;
default_specification : "DEFAULT" expression
;
expression : signed_integer /* integer and id_d */
    | T_STRING
    | T_IDENTIFIER /* enumeration constant */
    | monad_set
;
signed_integer : T_INTEGER
    | "-" T_INTEGER
    | "NIL"
;
```

#### 3.5.1.2 Examples

```
CREATE OBJECT WITH
WITH SINGLE MONAD OBJECTS
[Word
  surface: STRING WITHOUT INDEX;
  lemma : STRING WITH INDEX;
  parsing_tag : STRING FROM SET WITH INDEX;
  psp : part_of_speech_t;
  parents : LIST OF id_d;
]
GO
CREATE OBJECT TYPE
IF NOT EXISTS
[Clause
   parent : id_d;
   clause_type : clause_type_t default NC;
   functions : LIST OF clause_function_t; // An enumeration
   descendants : LIST OF ID_D;
   parallel_monads : SET OF MONADS;
]
GO
```

The latter creates an object type called "Clause" with four features: parent (Immediate Constituent of), whose type is id_d, and clause_type, which has the enumeration-type clause_type_t and the default value NC (which must be an enumeration constant in the clause_type_t enumeration). In addition, the two features "functions" and "descendants" are created, both of which are lists. "functions" is a list of enumeration constants drawn from the enumeration clause_function_t, whereas the "descendants" feature is a list of id_ds, which should then point to the descendants in the tree. In addition, if the object type "Clause" exists already, no error is thrown, and the object type is left untouched.

#### 3.5.1.3 Explanation

This statement *creates an object type* in the meta-data repository of the current database. It starts out with the keywords "CREATE OBJECT TYPE", followed by an optional clause which states whether the objects will be single-range or multiple-range (see below). After that comes a specification of the object type name and its features enclosed in square brackets. The `feature_declaration_list` is optional, so it is possible for an object type to have no features.[1]

Each feature_declaration consists of a feature name, followed by a colon, followed by a feature type, followed by an optional specification of the default value.

An INTEGER, ID_D, STRING, or ASCII feature can be declared "WITH INDEX". This will put an index on the feature's column. The default is not to add an index. This index will be dropped if a DROP INDEXES statement is issued for the object type (see Section 3.3.6 on page 33), but it will be recreated if a CREATE INDEXES statement is issued for the object type (see Section 3.3.7 on page 34). If a feature is an enumeration, it is usually not a good idea to create an index on the feature. This is because enumeration constants are usually few in number, and it is generally not a good idea to index columns that draw their values from a small pool of values, since this can lead to speed decreases (O(NlogN) instead of O(N)). Therefore, the MQL language does not allow creating indexes on enumeration features. You can add them yourself, of course, if you like, with the backend's corresponding SQL interface.

A STRING or ASCII feature can be declared "FROM SET". The default is for it not to be from a set. This does *not* mean that the value of the feature *is* a set, but rather that the values are drawn FROM a set. Whenever an object is created or updated, and a feature is assigned which is declared "FROM SET", the

[1] Strictly, this is not true, since all object types (except pow_m, any_m, and all_m) have at least one feature, namely the one called "self". Please see section 2.7.6 on page 21 for more information.

string value is first looked up in a special table that maps strings to unique integers. Then this unique integer is used in lieu of the string in the feature column. If the string does not exist in the separate table, it is added, with a unique integer to go with it, and that integer is used. If the string is there, then the integer already associated with the string is used. This gives a space savings (on MySQL and PostgreSQL), and sometimes also a speed advantage, especially if used on the string features of a Word object type which do no have high cardinality (number of unique instances), and the words number many millions. SQLite and SQLite 3 may not see any speed or space savings advantage. Note that using FROM SET on a string may actually impede performance (especially database loading times), especially on MySQL, if the cardinality of the string data is high. This will likely be the case for strings like "surface" and "lemma", which generally should not be declared FROM SET. However, features like "part_of_speech", "case", "number", "gender", which all most likely have low cardinality, might be good candidates for using a STRING FROM SET. Thus STRING FROM SET is a performance-enhanced way of using general strings instead of enumerations, and should be just as fast as enumerations for most queries, provided it is not used with high-cardinality data.

The specification of the default value `(default_specification)` consists of the keyword "DEFAULT", followed by an expression. An expression is either a `signed_integer`, a string, or an identifier. The identifier must be an enumeration constant belonging to the enumeration which is also the type of the feature. The `signed_integer` is either a signed integer (positive or negative), or the keyword "NIL", meaning the id_d that points to no object.

The feature "self" is implicitly created. It is an error if it is declared. The "self" feature is a computed feature which holds the unique object id_d of the object. See also 2.7.6 on page 21.

The difference between "ASCII" and "STRING" is that the user promises only to store 7-bit data in an ASCII string, whereas a STRING string may contain 8-bit data.[2]

In previous versions, you could specify a string length in parentheses after the STRING or ASCII keyword. As of version 1.2.0, all strings can have arbitrary length, with no maximum.[3] The old syntax is still available, but is ignored.

If a feature is declared as a LIST OF *something*, that something has to be either INTEGER, ID_D, or an enumeration constant. Lists of strings are not supported. Also, you cannot declare a default value for a list – the default value is always the empty list.

This statement can be "hedged" with the "IF NOT EXISTS" clause. If this clause is included, the statement does not throw an error if the object type exists already. Instead, the object type is left as it is (no changes are made to what is in the database already), and the statement returns success. If the "IF NOT EXISTS" clause is omitted, the statement throws an error if the object type exists already.

An object type can be declared "WITH SINGLE MONAD OBJECTS", "WITH SINGLE RANGE OBJECTS" or "WITH MULTIPLE RANGE OBJECTS". The difference is that:

- An object type which has been declared "WITH SINGLE MONAD OBJECTS" can only hold objects which consist of a single monad (i.e., the first monad is the same as the last monad).
- An object type which has been declared "WITH SINGLE RANGE OBJECTS" can only hold objects which consist of a *single monad range*, i.e., there are no gaps in the monad set, but it consists of a single contiguous stretch of monads (possibly only 1 monad long).
- An object type which has been declared "WITH MULTIPLE RANGE OBJECTS" (the default) can hold objects which have arbitrary monad sets.

A single-monad object must consist of only 1 monad, e.g., {1}, {2}, {3}. A single-range object can consist of a single monad (e.g., { 1 }, { 2 }, { 3 }, etc.), or it can consist of a single interval (e.g., { 6-7 }, { 9-13 }, { 100-121 }, etc.). However, as soon as an object type needs to hold objects which can consist of more than one range (e.g., { 6-7, 9-13 }), then it must be declared WITH MULTIPLE RANGE OBJECTS. If neither is specified, then WITH MULTIPLE RANGE OBJECTS is assumed.

There is a speed advantage of using WITH SINGLE MONAD OBJECTS over WITH SINGLE RANGE OBJECTS, and again a speed advantage of using WITH SINGLE RANGE OBJECTS over WITH MULTIPLE RANGE OBJECTS. The latter is the slowest, but is also the most flexible in terms of monad sets.

[2] ASCII strings are stored exactly the same way as 8-bit STRINGs. This distinction is mostly obsolete.

[3] This is true for PostgreSQL (it is a TEXT). For MySQL, the maximum is 4294967295 (2^32 - 1) characters (it is a LONGTEXT). On SQLite, it is a TEXT, but it is unknown how much this can hold.

In addition, and orthogonally to the range type, an object type can be declared "HAVING UNIQUE FIRST MONADS", "HAVING UNIQUE FIRST AND LAST MONADS", or "WITHOUT UNIQUE MONADS". The difference is:

- An object type which has been declared "HAVING UNIQUE FIRST MONADS" can only hold objects which are unique in their first monad within the object type. That is, within this object type, no two objects may start at the same monad.
- An object type which has been declared "HAVING UNIQUE FIRST AND LAST MONADS" can only hold objects which are unique in their first monad and in their last monad (as a pair: You are allowed to have two objects with the same starting monad but different ending monads, or vice versa). That is, no two objects within this object type start at the same monad while also ending at the same monad. Note that for object types declared WITH SINGLE MONAD OBJECTS, a "HAVING UNIQUE FIRST AND LAST MONADS" restriction is upgraded to a "HAVING UNIQUE FIRST MONADS" restriction, since they are equivalent for this range type.
- An object type which has been declared "WITHOUT UNIQUE MONADS" (or which omits any of the "monad uniqueness constraints") has no restrictions on the monads, other than those implied by the range type.

#### 3.5.1.4 Return type

There is no return value.

### 3.5.2 UPDATE OBJECT TYPE

#### 3.5.2.1 Syntax

```
update_object_type_statement : "UPDATE"
    [ "OBJECT" ]   "TYPE"
    "[" object_type_name
          feature_update_list
    "]"
;
object_type_name : T_IDENTIFIER
;
feature_update_list : feature_update   { feature_update }
;
feature_update : [ "ADD" ]  feature_declaration
    | "REMOVE"   feature_name   ";"
;
feature_name : T_IDENTIFIER
;
```

#### 3.5.2.2 References

All the foreign non-terminals are defined in section 3.5.1.

#### 3.5.2.3 Example

```
UPDATE OBJECT TYPE
[Word
    ADD no_of_morphemes : integer;
    REMOVE surface_without_accents;
]
GO
```

This example ADDs the feature no_of_morphemes (being an integer), and REMOVEs the feature surface_without_accents.

#### 3.5.2.4 Explanation

This statement *updates an object type*. It can either add a feature or remove an already-existing feature. When adding a new feature, the ADD keyword is optional. Other than that, it has exactly the same notation as for feature declarations under the CREATE OBJECT TYPE statement.

Removing a feature requires the REMOVE keyword, the feature name, and a semicolon.

Both additions and removals must be terminated with semicolon, even if the `feature_update` is the only `feature_update` in the list of `feature_update`s.

Note that the statement does not allow for *changing* the type of an already existing feature, only for adding or removing features.

#### 3.5.2.5 Return type

There is no return value.

### 3.5.3 DROP OBJECT TYPE

#### 3.5.3.1 Syntax

```
drop_object_type_statement : "DROP"
    [ "OBJECT" ]   "TYPE"
    "["   object_type_name   "]"
;
object_type_name : T_IDENTIFIER
;
```

#### 3.5.3.2 Example

```
DROP OBJECT TYPE
[Sploinks]
GO
```

This example drops the object type "Sploinks" from the database.

#### 3.5.3.3 Explanation

This statement drops an object type entirely from the database. This deletes not only the object type, but also all the objects of that object type, as well as the object type's features. Enumerations which are used as a feature type are not dropped, however.

#### 3.5.3.4 Return type

There is no return value.

## 3.6 Enumeration manipulation

### 3.6.1 CREATE ENUMERATION

#### 3.6.1.1 Syntax

```
create_enumeration_statement : "CREATE"
    ("ENUM" | "ENUMERATION")
```

```
    enumeration_name   "="
    "{"  ec_declaration_list  "}"
;
enumeration_name : T_IDENTIFIER
;
ec_declaration_list : ec_declaration    { ","   ec_declaration }
;
ec_declaration : [ "DEFAULT" ]
                 ec_name
                 [ ec_initialization ]
;
ec_name : T_IDENTIFIER
;
ec_initialization : "=" signed_integer
;
```

#### 3.6.1.2 References

For a description of `signed_integer`, please see section 3.5.1 on page 36. Note, however, that NIL should not be used with enumerations.

#### 3.6.1.3 Example

```
CREATE ENUMERATION
phrase_type_t = { VP = 1, NP, AP,
   PP, default NotApplicable = -1 }
GO
```

This particular statement creates an enumeration called "phrase_type_t" with the following constants and values:

| Name | Value | Default |
|---|---|---|
| NotApplicable | -1 | Yes |
| VP | 1 | No |
| NP | 2 | No |
| AP | 3 | No |
| PP | 4 | No |

#### 3.6.1.4 Explanation

This statement creates a new enumeration and populates it with enumeration constants.

If there is no declaration that has the "default" keyword, then the first one in the list becomes the default.

If the first declaration does not have an initialization, its value becomes 1. This is different from C and C++ `enum`s, which get 0 as the first value by default.

If a declaration does not have an initialization, its values becomes that of the previous declaration, plus 1. This mimics C and C++ `enum`s.

Label names must be unique within the enumeration. That is, you cannot have two constants with the same name in the same enumeration.

Values must also be unique within the enumeration. That is, you cannot have two different labels with the same value in the same enumeration. This is different from C/C++ `enum`s, where two labels may have the same value.

#### 3.6.1.5 Return type

There is no return value.

### 3.6.2 UPDATE ENUMERATION

#### 3.6.2.1 Syntax

```
update_enumeration_statement : "UPDATE"
    ("ENUM" | "ENUMERATION")
    enumeration_name   "="
    "{"   ec_update_list   "}"
;
enumeration_name : T_IDENTIFIER
;
ec_update_list : ec_update   { ","   ec_update }
;
ec_update : [ "ADD" ] [ "DEFAULT" ]
    ec_name ec_initialization
  | "UPDATE"    [ "DEFAULT" ]   ec_name   ec_initialization
  | "REMOVE" ec_name
;
ec_name : T_IDENTIFIER
;
ec_initialization : "=" signed_integer
;
```

#### 3.6.2.2 References

For a description of `signed_integer`, please see section 3.5.1 on page 36. Note, however, that NIL should not be used with enumerations.

#### 3.6.2.3 Example

```
UPDATE ENUMERATION
phrase_type_t = {
    ADD default Unknown = -99,
    REMOVE NotApplicable,
    UPDATE PP = 5,
    AdvP = 4
}
GO
```

This alters the table made in the example in section 3.6.1 to be like this:

| Name | Value | Default |
|---|---|---|
| Unknown | -99 | Yes |
| VP | 1 | No |
| NP | 2 | No |
| AP | 3 | No |
| AdvP | 4 | No |
| PP | 5 | No |

#### 3.6.2.4 Explanation

This statement updates the enumeration constants of an already existing enumeration. The user can specify whether to add, remove, or update an enumeration constant.

It is an error (and none of the updates will be executed) if the user REMOVEs the default enumeration constant without specifying a new default.

Note that you are forced to specify values for all of the constants updated or added.

It is an error (and none of the updates will be executed) if the update would lave the enumeration in a state where two labels would have the same value. This is because an enumeration is effectively a one-to-one correspondence between a set of labels and a set of values.

It is the user's responsibility that the update leaves the database in a consistent state. For example, Emdros will not complain if you remove a constant with a given value without specifying a different constant with the same value, even if there are features that use this enumeration and have this value. This would mean that those feature-values could not be searched for, since there would be no label to look for. Neither would it be possible to get the feature values with GET FEATURES, since there would be no label to return.

#### 3.6.2.5 Return type

There is no return value.

### 3.6.3 DROP ENUMERATION

#### 3.6.3.1 Syntax

```
drop_enumeration_statement : "DROP"
    ("ENUM" | "ENUMERATION")
    enumeration_name
;
enumeration_name : T_IDENTIFIER
;
```

#### 3.6.3.2 Example

```
DROP ENUMERATION phrase_type_t
GO
```

#### 3.6.3.3 Explanation

This statement removes an enumeration altogether from the database, including all its enumeration constants.

It is an error (and impossible) to drop an enumeration which is in use by some object type.

#### 3.6.3.4 Return type

There is no return value.

## 3.7 Segment manipulation

### 3.7.1 Introduction

Segments were present in Emdros up to and including version 1.1.12. After that, support for segments was removed.

A segment used to be an arbitrary, contiguous stretch of monads which was given a name. Objects could not be created which crossed the boundaries of a segment. You could restrict your search to within a single segment with SELECT ALL OBJECTS.

However, segments were found to be ugly baggage, cumbersome and not useful. Therefore, they were removed.

The CREATE SEGMENT statement is retained for backward compatibility.

### 3.7.2 CREATE SEGMENT

#### 3.7.2.1 Syntax

```
create_segment_statement : "CREATE"   "SEGMENT"
    segment_name
    "RANGE"   "="   segment_range
;
segment_name : T_IDENTIFIER
;
segment_range : T_INTEGER "-" T_INTEGER
;
```

#### 3.7.2.2 Example

```
CREATE SEGMENT Old_Testament
RANGE = 1 - 500000
GO
```

This example used to create a segment named "Old_Testament" starting at monad 1 and ending at monad 500000.

Now it does nothing.

#### 3.7.2.3 Explanation

This statement currently does nothing. It will fail with a database error.

#### 3.7.2.4 Return type

There is no return value.

## 3.8 Querying the data

This section describes statements which can be used to query the *data* in an Emdros database, as opposed to querying the *schema*.

### 3.8.1 SELECT (FOCUS|ALL) OBJECTS

#### 3.8.1.1 Syntax

```
select_objects_statement : select_clause
    opt_in_clause
    opt_with_max_range_clause
    opt_returning_clause
    where_clause
;
/*
 * select-clause
 */
select_clause : "SELECT"   focus_specification   [ "OBJECTS" ]
;
focus_specification : "FOCUS" | "ALL"
;
/*
 * in-clause
```

```
 */
opt_in_clause : "IN"   in_specification
  | /* empty = all_m-1 */
;
in_specification : monad_set
  | "ALL" /* = all_m-1 */
  | monad_set
  | T_IDENTIFIER /* Named arbitrary monad set */
;
monad_set : "{"   monad_set_element_list   "}"
;
monad_set_element_list : monad_set_element
    { ","   monad_set_element }
;
monad_set_element : T_INTEGER
  | T_INTEGER   "-"   T_INTEGER
  | T_INTEGER   "-"   /* From T_INTEGER to "practical infinity"
                           (i.e., MAX_MONAD). */
;

/*
 *opt_with_max_range_clause
 */
opt_with_max_range_clause : "WITH" "MAX" "RANGE" "MAX_M" "MONADS"
   | /* empty; same as WITH MAX RANGE MAX_M MONADS. */
   | "WITH" "MAX" "RANGE" T_INTEGER "MONADS"
   | "WITH" "MAX" "RANGE" "FEATURE" "MONADS" "FROM" "[" object_type_name "]"
   | "WITH" "MAX" "RANGE" "FEATURE" feature_name "FROM" "[" object_type_name "]"
;
/*
 * returning-clause
 */
opt_returning_clause : /* Empty: Return full sheaf */
  | "RETURNING" "FULL" "SHEAF"
  | "RETURNING" "FLAT" "SHEAF"
  | "RETURNING" "FLAT" "SHEAF"  "ON"
    object_type_name_list
;
object_type_name_list :
  object_type_name { "," object_type_name }
;
/*
 * where-clause
 */
where_clause : "WHERE" mql_query
;
```

#### 3.8.1.2 References

For the `mql_query` non-terminal, please see section 4.3 on page 79.

#### 3.8.1.3 Example

```
SELECT ALL OBJECTS
```

```
IN { 1-4, 5, 7-9 }
WITH MAX RANGE 5 MONADS
RETURNING FULL SHEAF
WHERE
[Word lexeme = ">RY/"]
GO
```

#### 3.8.1.4 Explanation

This statement is a front-end to the MQL Query-subset (see chapter 4 starting on page 75).

The parameters to an MQL query are:

1. A universe U,
2. A substrate Su, and
3. A topograph.

The universe U is a contiguous stretch of monads. The search is restricted only to include objects which are wholly contained within this universe (i.e., which are part_of[4] the universe).

The substrate is used to further restrict the search. For there is the additional requirement that all objects found must be wholly contained within (i.e., part_of) the substrate as well. The substrate must be part_of the universe. Mathematically speaking, the substrate is the set intersection of whatever was in the IN clause and all_m-1 (i.e., the set of all monads in the database).

The topograph is what is specified as `mql_query` in the above grammar.

The IN-specification tells the query-engine what the substrate Su should be. There are three choices:

1. Specify an explicit monad set like "{ 1-3000, 7000-10000 }"
2. Specify a named arbitrary monad set (see section 2.7.13 on page 23).
3. Leave it blank. This means that the substrate is calculated as all_m-1 (i.e., all of the monads in the database; see [Standard-MdF] or [Doedens94] or page 20 in this Programmer's Guide.)

The universe U is then calculated as all the monads between the first and last monads of the substrate.

The "max range" specifies the maximum number of monads to take as a context for any power block at the outermost level. The significance of this is that it helps users not to get query results which are correct but useless because the query returns too many straws in the sheaf. This is done by putting an upper limit on the number of monads a power block may extend over. This limit can be "MAX_M" monads, meaning that there is in practice no limit to the stretch of monads which can be matched by a power block. It can also be empty, which is the same as "MAX_M" monads, i.e., no limit. It can also be an explicit number of monads, say, 5 or 100. The max range can also be taken as the length of the largest object of any object type. The set of monads to use can be either the privileged "monads" feature, or any feature whose type is SET OF MONADS. Note that the limit is NOT taken as the length of any actual object of the given object type which happens to be part_of the current stretch of monads under investigation. Rather, the cap is set before query execution time by inspecting the largest length of any monad set of the given monad set feature in the object type.

The difference between the RETURNING FULL SHEAF and the RETURNING FLAT SHEAF clause is that the latter applies the "flatten" operator to the sheaf before returning it, whereas the former does not. If the returning_clause clause is empty, it means the same thing as RETURNING FULL SHEAF. If the RETURNING FLAT SHEAF has an ON appendix with a list of object type names, then the two-argument flatten operator is applied using this list of object type names. See section 4.4.7 on page 85 for an explanation of flat sheaves and the "flatten" operator.

[4] For part_of, see section 2.7.7 on page 21.

#### 3.8.1.5 Monad set

The explicit monad set in the IN clause, if given, must consist of a comma-separated list of monad-set-elements enclosed in curly braces. A monad-set-element is either a single integer (referring to a single monad) or a range consisting of two integers (referring to a range of monads). The monad-set-elements need not occur in any specific order, and are allowed to overlap. The result is calculated by adding all the monads together into one big set. The ranges of monads must, however, be monotonic, i.e., the second integer must be greater than or equal to the first.

#### 3.8.1.6 Return type

A sheaf, either full or flat: All retrieved objects are included, but those objects that had the `focus` modifier in the query are flagged as such. Please see 4.4 on page 83 for an explanation of the sheaf. Appendix B on page 108 gives the grammar for the console-sheaf. Please see section 4.4.7 on page 85 for an explanation of the flat sheaf.

### 3.8.2 SELECT OBJECTS AT

#### 3.8.2.1 Syntax

```
select_objects_at_statement : "SELECT"   [ "OBJECTS" ]
    "AT"   single_monad_specification
    "["  object_type_to_find   "]"
;
single_monad_specification : "MONAD"   "="   T_INTEGER
;
object_type_to_find : object_type_name
;
object_type_name : T_IDENTIFIER
;
```

#### 3.8.2.2 Example

```
SELECT OBJECTS
AT MONAD = 3406
[Clause]
GO
```

This example selects all those objects of type Clause which start at monad 3406.

#### 3.8.2.3 Explanation

This statement returns a table containing the object id_ds of all the objects of the given type which start at the monad specified, i.e., whose first monad is the given monad.

The result is a table with one column, namely "id_d". Each row represents one object, where the id_d is its object id_d.

#### 3.8.2.4 Return type

A table with the following schema:

| id_d: id_d |
|---|

On failure, this table is empty. Note, however, that the table can also be empty because there were no objects of the given type having the given monad as their first monad. This is not an error.

### 3.8.3 SELECT OBJECTS HAVING MONADS IN

#### 3.8.3.1 Syntax

```
select_objects_having_monads_in_statement :
   "SELECT" "OBJECTS"
   "HAVING" "MONADS" "IN"
   monad_set
   "["  object_type_to_find   "]"
;
object_type_to_find : object_type_name | "ALL"
;
object_type_name : T_IDENTIFIER
;
```

#### 3.8.3.2 References

For the `monad_set` non-terminal, see section 3.8.1 on page 45.

#### 3.8.3.3 Example

```
SELECT OBJECTS
HAVING MONADS IN { 23-45, 68, 70, 87-93 }
[Clause]
GO

SELECT OBJECTS
HAVING MONADS IN { 1, 5-7, 103-109 }
[ALL]
GO

SELECT OBJECTS
HAVING MONADS IN { 23 }
[Word]
GO
```

#### 3.8.3.4 Explanation

This statement returns the object types and object id_ds of the objects that have at least one monad in the monad set specified. If "ALL" is specified as the object type, then this is done for all object types in the database. If a specific object type is specified, then that object type is used.

The returned table has one row for each object. Each object is represented only once. The monad in each row is guaranteed to be from the set of monads specified, and is guaranteed to be from the object in the row.

This statement is useful in much the same way that the SELECT OBJECTS AT statement is useful. It can be used, e.g., for getting id_ds of objects that must be displayed as part of the results of a query, but which are not in the query results. This statement can also be used like the SELECT OBJECTS AT statement by simply making the monad set a singleton set with only one monad. Note, however, that this statement does something a different from SELECT OBJECTS AT. Whereas SELECT OBJECTS AT will only retrieve an object if that object *starts on* the given monad, this present statement will retrieve the object if only the object *has at least one monad* from the monad set given. This statement also has the advantage that one can ask for all object types. This enables one to access objects which one knows might be there, but of which one does not know the object types. It also has the advantage of being much faster than a series of SELECT OBJECTS AT statements if one is looking for objects in more than one monad.

This statement was typically used in a series of SELECT OBJECTS HAVING MONADS IN, GET MONADS, and GET FEATURES statements, in order to obtain all information necessary for display of data. This sequence has been wrapped neatly into the GET OBJECTS HAVING MONADS IN statement, which is now the preferred method of doing this sequence.

Note to programmers: If you want to get objects not from all object types but from only a subset of all object types, the easiest thing is to issue the required number of copies of the statement with GO in between, varying only the object type. That way, if you are using the mql program as a proxy for the MQL engine, you don't incur the overhead of starting and stopping the mql program.

#### 3.8.3.5 Return type

| object_type_name : string | monad : monad_m | id_d : id_d |
|---|---|---|

On failure, this table is empty. Note, however, that the table can also be empty if the command were successful, if there were no objects that had at least one monad in the monad set specified.

### 3.8.4 GET OBJECTS HAVING MONADS IN

#### 3.8.4.1 Syntax

```
get_objects_having_monads_in_statement :
   "GET" "OBJECTS"
   "HAVING" "MONADS" "IN"
   gohmi_monad_set
   using_monad_set_feature
   "[" object_type_name
          [gohmi_feature_retrieval]
   "]"
;
gohmi_monad_set : "ALL" /* all_m-1 on the "monads" feature,
                           regardless of which monad set is actually used!
                        */
                | monad_set
;
using_monad_set_feature : /* empty: Use the object's monad set */
    | "USING" "MONAD" "FEATURE" feature_name
    | "USING" "MONAD" "FEATURE" "MONADS"
;
feature_name : T_IDENTIFIER
;
gohmi_feature_retrieval : "GET" feature_list
                        | "GET" "ALL"
;
feature_list : feature_name { "," feature_name }*
;
object_type_name : T_IDENTIFIER
;
```

#### 3.8.4.2 References

For the `monad_set` non-terminal, see section 3.8.1 on page 45.

#### 3.8.4.3 Example

```
GET OBJECTS
```

```
HAVING MONADS IN { 23-45, 68, 70, 87-93 }
[Clause]
GO

GET OBJECTS
HAVING MONADS IN { 1, 5-7, 103-109 }
USING MONAD FEATURE parallel_monads
[Phrase GET phrase_type, function]
GO

GET OBJECTS
HAVING MONADS IN { 23 }
[Word GET ALL]
GO
```

#### 3.8.4.4 Explanation

This statement returns the objects of the given object type that have at least one monad in the monad set specified. A flat sheaf is returned with one straw containing all the objects to be retrieved.

The monad set to use is the object's monad set by default. If the "using_monad_set_feature" variant is used, the monad sets used are the ones stored in this feature. The feature must be a set of monads.

This statement is useful in much the same way that the SELECT OBJECTS AT statement is useful. It can be used, e.g., for getting id_ds of objects that must be displayed as part of the results of a query, but which are not in the query results.

This is the preferred method for getting objects from the engine, rather than a sequence of SELECT OBJECTS HAVING MONADS IN, GET MONADS, and GET FEATURES. It is much faster than the combination of the three.

#### 3.8.4.5 Return type

A flat sheaf is returned which contains the objects in question. See Section 4.4.7 on page 85 for more information.

### 3.8.5 GET AGGREGATE FEATURES

#### 3.8.5.1 Syntax

```
get_aggregate_features_statement : "GET" "AGGREGATE" ("FEATURE" | "FEATURES")
    aggregate_feature_list
    "FROM" "OBJECTS" opt_in_clause
    "WHERE"
    "[" object_type_name
       feature_constraints
    "]"
;
aggregate_feature_list : aggregate_feature
                       | aggregate_feature_list "," aggregate_feature
;
aggregate_feature : aggregate_function "(" feature_name ")"
                  | "COUNT" "(" "*" ")"
                  | "COUNT" "(" feature_name "=" feature_value ")"
;
aggregate_function : "MIN" | "MAX" | "SUM"
;
```

```
feature_name : T_IDENTIFIER
;
object_type_name : T_IDENTIFIER
;
```

#### 3.8.5.2 References

For the `opt_in_clause` non-terminal, see section 3.8.1 on page 45. For the `feature_constraints` non-terminal, see section 4.3 on page 79.

#### 3.8.5.3 Examples

```
/*
 * Create the enumerations and object types of an
 * example database.
 *
 */
CREATE ENUMERATION boolean_t = {
    false = 0,
    true
}
GO
CREATE ENUMERATION part_of_speech_t = {
    Verb,
    Noun,
    ProperNoun,
    Pronoun,
    Adjective,
    Adverb,
    Preposition,
    Conjunction,
    Particle
}
GO
CREATE OBJECT TYPE
WITH SINGLE MONAD OBJECTS
[Token
    has_space_before : boolean_t;  // Any space before the surface?
    surface : STRING;              // The surface itself
    has_space_after : boolean_t;   // Any space after the surface?
    part_of_speech : part_of_speech_t;
    is_punctuation : boolean_t;     // true iff the surface is punctuation.
]
GO
CREATE OBJECT TYPE
WITH SINGLE RANGE OBJECTS
[Line
    actant_name : STRING FROM SET;
    gender : gender_t;
    words_spoken : INTEGER;
    line_number_in_act : INTEGER;
]
GO
/*
```

```
 * Example 1:
 *
 * Founds all Token objects in the entire database whose
 * is_punctuation feature equals false.
 *
 * Then retrieve one aggregate function, namely a count of
 * all Token objects whose part_of_speech is equal to Verb.
 */
GET AGGREGATE FEATURES
COUNT(part_of_speech=Verb)
FROM OBJECTS
WHERE
[Token is_punctuation=false]
GO
/*
 * Example 2:
 *
 * Finds all Line objects in the database (no monad restriction), and
 * does three aggregate functions:
 * - Find the sum of all words spoken by all actants.
 * - Find a count of all Line objects. (Note that this, together with the
 *   sum of all words spoken, can be used to calculate the average
 *   number of words spoken by any actant. This has to be done outside of
 *   Emdros, as Emdros does not yet support floating point return values.)
 * - Find a count of all Line objects whose gender feature is Man, i.e., the
 *   number of Line spoken by a Man.
 */
GET AGGREGATE FEATURES
SUM(words_spoken), COUNT(*), COUNT(gender=Man)
FROM OBJECTS
WHERE
[Line]
GO
/*
 * Example 3:
 *
 * Finds all Line objects where actant_name is equal to "Hamlet" in
 * the monad set { 1-12478 } (an arbitrarily chosen example).
 * Does three aggregate functions:
 * - Finds the maximum number of words spoken in any of "Hamlet"'s
 *   lines.
 * - Finds the minimum line number in any act (i.e., the first line
 *   in which "Hamlet" speaks.
 * - Finds the total count of Lines spoken by "Hamlet".
 */
GET AGGREGATE FEATURES
MAX(words_spoken), MIN(line_number_in_act), COUNT(*)
FROM OBJECTS
IN { 1-12478 }
WHERE
[Line actant_name="Hamlet"]
GO
/*
 * Example 4:
```

```
 *
 * Finds all Line objects in all monads,
 * where actant_name is equal to "Hamlet"
 * OR is equal to "Ophelia".
 * Does three aggregate functions:
 * - Finds the total sum of words spoken by either Hamlet or Ophelia.
 * - Finds the count of all Lines in which the actant_name is "Hamlet".
 * - Finds the count of all Lines in which the actant_name is "Ophelia".
 */
GET AGGREGATE FEATURES
SUM(words_spoken), COUNT(actant_name="Hamlet"), COUNT(actant_name="Ophelia")
FROM OBJECTS
IN ALL
WHERE
[Line actant_name="Hamlet" OR actant_name="Ophelia"]
GO
```

#### 3.8.5.4 Explanation

This statement returns a table giving the desired aggregate functions over the objects specified in the WHERE clause.

In essence, five aggregate functions are available:

**MIN(feature_name):** Retrieves the minimum value of the given feature. The feature must be of type integer. The result is also an integer.

**MAX(feature_name):** Retrieves the maximum value of the given feature. The feature must be of type integer. The result is also an integer.

**SUM(feature_name):** Retrieves the sum of all values of the given feature. The feature must be of type integer. The result is also an integer.

**COUNT(*):** Retrieves a count of all objects retrieved by the WHERE clause. The result is an integer.

**COUNT(feature_name=feature_value):** Retrieves a count of all objects retrieves, with the added restriction that the given feature name must have the given feature value. The result is an integer.

The standard SQL aggregate function "AVG" is missing. This is because Emdros does not (yet) support return values with type "floating point". Note that the same result as AVG can be achieved by retrieving two aggregate functions: SUM(feature_name) and COUNT(*), and then doing the appropriate division outside of Emdros.

The `opt_in_clause` can be used to limit the monad set within which to retrieve the objects. If omitted, it means that the entire database is searched, without monad restriction.

The object type name given after the "WHERE" and "[" tokens is also the object type on which any features in the aggregate feature list are found. Hence, the features mentioned in the aggregate feature list must exist on the object type.

It is possible to use an arbitrary Boolean expression after the object type name, just as in the topographic MQL queries explained in Chapter 75.

#### 3.8.5.5 Return type

Upon failure, an empty table.

Upon success, a table with one row, and as many columns as there are aggregate functions in the query. The column types are all "integer", and the column names are given as "Column1", "Column2", ..., "Column*N*", where the number given is the index (1-based) of the aggregate functions in the input query.

### 3.8.6 GET MONADS

#### 3.8.6.1 Syntax

```
get_monads_statement : "GET" "MONADS"
    "FROM"   ("OBJECT" | "OBJECTS")
    "WITH" id_ds_specification
    "[" object_type_name "]"
;
object_type_name : T_IDENTIFIER
;
```

#### 3.8.6.2 References

For a description of `id_ds_specification`, please see section 3.10.2 on page 64.

#### 3.8.6.3 Example

```
GET MONADS
FROM OBJECTS
WITH ID_DS = 10342, 10344, 10383
[Clause]
GO
```

#### 3.8.6.4 Explanation

This statement returns, for each object in the list of id_ds, a representation of its set of monads. The set is represented by maximal stretches of monads. For example, if an object consists of the monads { 1, 2, 4, 5, 6, 9, 11, 12 }, and its id_d is 10342, then the following will be in the results of the above example:

| object_id_d : id_d | mse_first : monad_m | mse_last : monad_m |
|---|---|---|
| 10342 | 1 | 2 |
| 10342 | 4 | 6 |
| 10342 | 9 | 9 |
| 10342 | 11 | 12 |

The "mse" in "mse_first" and "mse_last" stands for "Monad Set Element." A monad set element consists of a starting monad and an ending monad (always greater than or equal to the starting monad). It represents all of the monads between the two borders, including the borders. An mse's last monad is always greater than or equal to its first monad.

The mses in the list are always maximal. That is, there is a gap of at least one monad in between each of the MSEs.

In mathematical terms, suppose we have an MSE A. Then for all other MSEs B for the same object, it is the case that either A.last + 1 < B.first or B.last < A.first - 1

The MSEs will come in no particular order.

See [Monad Sets] for more information on monad sets and the way Emdros treats them.

It does not matter whether you write "OBJECT" or "OBJECTS": The choice is merely syntactic sugar.

There is no limit on how many id_ds can be specified. The algorithm will not balk at even many thousand id_ds, but it will, of course, take more time to get the monads of more objects.

This statement was typically used in a series of SELECT OBJECTS HAVING MONADS IN, GET MONADS, and GET FEATURES statements, in order to obtain all information necessary for display of data. This sequence has been wrapped neatly into the GET OBJECTS HAVING MONADS IN statement, which is now the preferred method of doing this sequence.

#### 3.8.6.5 Return type

A table with the following schema:

| object_id_d : id_d | mse_first : monad_m | mse_last : monad_m |
|---|---|---|

### 3.8.7 GET FEATURES

#### 3.8.7.1 Syntax

```
get_features_statement : "GET"
    ("FEATURE" | "FEATURES")
    feature_list
    "FROM"   ("OBJECT" | "OBJECTS")
    "WITH" id_ds_specification
    "["  object_type_name   "]"
;
/*
 * feature_list
 */
feature_list : feature_name   { ","  feature_name }
;
feature_name : T_IDENTIFIER
;
object_type_name : T_IDENTIFIER
;
```

#### 3.8.7.2 References

For a description of `id_ds_specification`, please see section 3.10.2 on page 64.

#### 3.8.7.3 Example

```
GET FEATURES surface, psp
FROM OBJECTS WITH ID_DS = 12513,12514
[Word]
GO
```

#### 3.8.7.4 Explanation

This statement returns a table containing feature-values of certain objects in the database.

Note how this is different from the "SELECT FEATURES" command. The "SELECT FEATURES" command queries an *object type* for a list of its *features*. The "GET FEATURES" command queries *objects* for the *values* of some of their features.

This statement was typically used in a series of SELECT OBJECTS HAVING MONADS IN, GET MONADS, and GET FEATURES statements, in order to obtain all information necessary for display of data. This sequence has been wrapped neatly into the GET OBJECTS HAVING MONADS IN statement, which is now the preferred method of doing this sequence.

#### 3.8.7.5 Return type

The return type is a table with a schema containing one string for each feature in the list of features. The order of the columns is that in the list of features. The first column in the table contains the object id_d involved in the row. Thus for $n$ features, the number of columns will be $n + 1$.

The return type of each feature is the same as the type of the feature. The exact representation depends on whether the output is console output or XML output. For XML, see the DTD. For console output, see

the examples below. Enumeration constants are shown as the enumeration constant label, not the integer value.

For list features, the value is a space-surrounded, space-delimited list of values. Integers and ID_Ds are given as integers; enumeration constant values as their constant names (e.g., “first_person”).

| object_id_d: id_d | surface: string | psp: enum(psp_t) | number_in_corpus : integer | parent: id_d | parents: list_of_id_d |
|---|---|---|---|---|---|

The table contains the objects in no particular order.

## 3.8.8 GET SET FROM FEATURE

### 3.8.8.1 Syntax

```
get_set_from_feature_statement : "GET"   "SET"
    "FROM"   "FEATURE"
    feature_name
    "["  object_type_name   "]"
;
feature_name : T_IDENTIFIER
;
object_type_name : T_IDENTIFIER
;
```

### 3.8.8.2 Example

```
GET SET
FROM FEATURE lexeme
[Word]
GO
```

### 3.8.8.3 Explanation

This statement returns a table containing the set of existing feature-values for a feature declared FROM SET. See the CREATE OBJECT TYPE and UPDATE OBJECT TYPE statements on page 36 and 40 respectively for the syntax of the FROM SET declaration.

Note how this is different from GET FEATURES: The “GET FEATURES” command queries *objects* for the *values* of some of their features. The GET SET FROM FEATURE queries the *set* of existing values for a given feature, regardless of which objects have these values for this feature.

### 3.8.8.4 Return type

The return type is a table with a schema containing one string for each value in the set.

| value: string |
|---|

The order of the strings in the table is undefined.

## 3.8.9 SELECT MIN_M

### 3.8.9.1 Syntax

```
select_min_m_statement : "SELECT"   "MIN_M"
;
```

### 3.8.9.2 Example

```
SELECT MIN_M
GO
```

#### 3.8.9.3 Explanation

Returns the minimum monad in use in the database. The table returned has only one data row, namely the minimum monad. See section 2.7.12 on page 22 for more information.

#### 3.8.9.4 Return type

A table with the following schema:

| min_m : monad_m |
|---|

On failure, this table is empty.

### 3.8.10 SELECT MAX_M

#### 3.8.10.1 Syntax

```
select_max_m_statement : "SELECT"   "MAX_M"
;
```

#### 3.8.10.2 Example

```
SELECT MAX_M
GO
```

#### 3.8.10.3 Explanation

Returns the maximum monad in use in the database. The table returned has only one data row, namely the maximum monad. See section 2.7.12 on page 22 for more information.

#### 3.8.10.4 Return type

A table with the following schema:

| max_m : monad_m |
|---|

On failure, this table is empty.

### 3.8.11 SELECT MONAD SETS

#### 3.8.11.1 Syntax

```
select_monad_sets_statement : "SELECT" "MONAD" "SETS"
;
```

#### 3.8.11.2 Example

```
SELECT MONAD SETS GO
```

#### 3.8.11.3 Explanation

This statement returns a table listing the names of the monad sets stored in the database. These are the monad sets referred to in section 2.7.13 on page 23.

The monad set names come in no particular order.

#### 3.8.11.4 Return type

| monad_set_name : string |
|---|

### 3.8.12 GET MONAD SETS

#### 3.8.12.1 Syntax

```
get_monad_sets_statement : "GET" "MONAD" ("SET" | "SETS")
                           monad_sets_specification
;
monad_sets_specification : "ALL"
  | monad_set_list
;
monad_set_list : monad_set_name { "," monad_set_name }
;
monad_set_name : T_IDENTIFIER
;
```

#### 3.8.12.2 Example

```
GET MONAD SET My_research_collection
GO
GET MONAD SETS Historical_books, Former_prophets
GO
GET MONAD SETS ALL
GO
```

#### 3.8.12.3 Explanation

This statement returns a table listing the monads of each of the monad sets named in the query. These monad sets are the arbitrary monad sets described in section 2.7.13 on page 23.

It doesn't matter whether you say "SET" or "SETS". This is purely syntactic sugar.

If "ALL" is given as the monad_sets_specification, then all monad sets are listed, in no particular order.

In the output, each monad set is represented in the same way as described in section 3.8.6.4 on page 55. Each monad set is guaranteed to appear in the table in one contiguous stretch, that is, monad sets are not interleaved. Moreover, the monad set elements of each monad set is sorted on mse_first, in ascending order.

#### 3.8.12.4 Return type

| monad_set_name : string | mse_first : monad_m | mse_last : monad_m |
|---|---|---|

## 3.9 Schema reflection

This section describes those query statements which can be used to retrieve information about the schema.

### 3.9.1 SELECT OBJECT TYPES

#### 3.9.1.1 Syntax

```
select_object_types_statement : "SELECT"
    [ "OBJECT" ]    "TYPES"
;
```

#### 3.9.1.2 Example

```
SELECT OBJECT TYPES
GO
```

| Type name | Meaning |
|---|---|
| integer | integer |
| id_d | id_d |
| list of integer | list of integer |
| list of id_d | list of id_d |
| list of *something else* | list of enumeration constants from the enumeration given |
| string | 8-bit string of arbitrary length |
| ascii | 7-bit (ASCII) string of arbitrary length |
| set of monads | set of monads |
| *everything else* | enumeration by the name given |

Table 3.4: Possible type names in SELECT FEATURES

#### 3.9.1.3 Explanation

This statement returns a list of the names of all the object types available in the database.

#### 3.9.1.4 Return type

A table with the following schema:

| object_type_name: string |
|---|

On failure, this table is empty.

### 3.9.2 SELECT FEATURES

#### 3.9.2.1 Syntax

```
select_features_statement : "SELECT"   "FEATURES"
    "FROM"   [ [ "OBJECT" ]   "TYPE" ]
    "["   object_type_name   "]"
;
object_type_name : T_IDENTIFIER
;
```

#### 3.9.2.2 Example

```
SELECT FEATURES
FROM OBJECT TYPE
[Phrase]
GO
```

#### 3.9.2.3 Explanation

This statement returns a table with the features belonging to the given object type.

The type_name string in the result gives the type of the feature. It has the values as in table 3.4.

The default_value string in the result is a string representation of the default value. It must be interpreted according to the feature type.

The computed boolean in the result shows whether the feature is computed or not. Currently, the only computed features are: “self”, “first_monad”, “last_monad”, “monad_count”, and “monad_set_length”.

For lists, what is shown in the “default value” field is always “()” meaning “the empty list”.

For sets of monads, a string giving the canonical form of an empty set of monads is used: “ { } ”.

#### 3.9.2.4 Return type

A table with the following schema:

| feature_name: string | type_name: string | default_value: string | computed: boolean |
|---|---|---|---|

On failure, this table is empty. On success, the table cannot be empty, since every object type has the feature "self".

### 3.9.3 SELECT ENUMERATIONS

#### 3.9.3.1 Syntax

```
select_enumerations_statement : "SELECT"
    "ENUMERATIONS"
;
```

#### 3.9.3.2 Example

```
SELECT ENUMERATIONS
GO
```

#### 3.9.3.3 Explanation

This statement returns a table with the names of all the enumerations available in the database.

#### 3.9.3.4 Return type

A table with the following schema:

| enumeration_name: string |
|---|

On failure, this table is empty. Note, however, that it can also be empty because there are no enumerations in the database yet.

### 3.9.4 SELECT ENUMERATION CONSTANTS

#### 3.9.4.1 Syntax

```
select_enumeration_constants_statement : "SELECT"
    ("ENUM" | "ENUMERATION")   "CONSTANTS"
    "FROM"    [ ("ENUM" | "ENUMERATION" ) ]
    enumeration_name
;
enumeration_name : T_IDENTIFIER
;
```

#### 3.9.4.2 Example

```
SELECT ENUMERATION CONSTANTS
FROM ENUMERATION phrase_types
GO
```

#### 3.9.4.3 Explanation

This statement returns a table with the enumeration constants in a given enumeration.

Note that the syntax is made so that the query need not be as verbose as in the example just given. There is quite a lot of syntactic sugar[5] in this statement.

#### 3.9.4.4 Return type

A table with the following schema:

| enum_constant_name: string | value : integer | is_default : boolean |
|---|---|---|

On failure, this table is empty.

### 3.9.5 SELECT OBJECT TYPES USING ENUMERATION

#### 3.9.5.1 Syntax

```
select_object_types_which_use_enum_statement : "SELECT"
    [ "OBJECT" ]    "TYPES"
    "USING"
    ("ENUM" | "ENUMERATION") enumeration_name
;
enumeration_name : T_IDENTIFIER
;
```

#### 3.9.5.2 Example

```
SELECT OBJECT TYPES
USING ENUMERATION phrase_types_t
GO
```

#### 3.9.5.3 Explanation

This statement returns a table with the names of the object types which use a given enumeration. The rows of the table are not ordered. An object type uses an enumeration if at least one of its features is of the enumeration type.

#### 3.9.5.4 Return type

A table with the following schema:

| object_type_name: string |
|---|

On failure, this table is empty. Note, however, that it can also be empty because there are no object types using the enumeration.

## 3.10 Object manipulation

### 3.10.1 CREATE OBJECT FROM MONADS

#### 3.10.1.1 Syntax

```
create_object_from_monads_statement : "CREATE"   "OBJECT"
```

[5] "Syntactic sugar" is a term used by computer-scientists for niceties in the grammar of a language which help the user in some way, usually so that they do not have to type as much as would otherwise be required. Here, it simply means that some of the keywords are optional, or have shorthand forms.

```
    "FROM"   monad_specification
    [ with_id_d_specification ]
    object_creation_specification
;
/*
 * monad-specification
 */
monad_specification : "MONADS"   "="   monad_set
;
/*
 * with-id_d-specification
 */
with_id_d_specification : "WITH"   "ID_D"
    "="  id_d_const
;
id_d_const : T_INTEGER
  | "NIL"
;
/*
 * object-creation-specification
 */
object_creation_specification : "["
    object_type_name
    [ list_of_feature_assignments ]
    "]"
;
object_type_name : T_IDENTIFIER
;
list_of_feature_assignments : feature_assignment
                              { feature_assignment }
;
feature_assignment : feature_name  ":="  expression  ";"
    | feature_name  ":="  list_expression  ";"
;
feature_name : T_IDENTIFIER
;
expression : signed_integer /* integer and id_d */
  | T_STRING
  | T_IDENTIFIER /* enumeration constant */
  | monad_set
;
list_expression : "(" [list_values] ")"
;
list_values : list_value { "," list_value }
;
list_value : signed_integer
  | T_IDENTIFIER /* enumeration constant */
;
```

#### 3.10.1.2 References

For a description of `monad_set`, please see section 3.8.1 on page 45. For a description of `signed_integer`, please see section 3.5.1 on page 36. Note, however, that NIL should be used only with features whose type is id_d.

#### 3.10.1.3 Example

```
CREATE OBJECT FROM MONADS = { 1-2, 4-7 }
[Clause
   clause_type := NC;
   parent := 10033;
   descendants := (104546, 104547, 104549);
]
GO

CREATE OBJECTS FROM MONADS = { 35-37 }
WITH ID_D = 104546
[Phrase
   phrase_type := NP;
   parents := (104212, 104215, 104219);
]
GO
```

#### 3.10.1.4 Explanation

This statement creates a new object from a specified set of monads.

In creating an object, four items of information are necessary:

1. The new id_d,
2. The object type,
3. The set of monads,
4. Any features that need non-default values.

This statement creates an object of type "`object_type_name`" using the monads and features, and optional id_d, given. All features not specified will be given default values.

If you specify an id_d with the "WITH ID_D" specification, the system first checks whether that object id_d is already in use. If it is, the creation fails. If it is not, that id_d is used. If you do not specify an id_d, a unique id_d is auto-generated.

Note that when using WITH ID_D, it is not recommended to run several concurrent processes against the same database which issue CREATE OBJECT or CREATE OBJECTS statements. Doing so may cause the auto-generated object id_d sequence to become invalid. However, several concurrent processes may safely issue CREATE OBJECT(S) statements if none of them use WITH ID_D.

Note that objects of the special object types all_m, any_m, and pow_m cannot be created.

#### 3.10.1.5 Return type

A table with the following schema:

| object_id_d: id_d |
|---|

On success, there is always only one row in the table, namely the row containing the object id_d of the newly created object.

On failure, the table is empty.

### 3.10.2 CREATE OBJECT FROM ID_DS

#### 3.10.2.1 Syntax

```
create_object_from_id_ds_statement : "CREATE"   "OBJECT"
    "FROM"   id_ds_specification
```

```
    [ with_id_d_specification ]
    object_creation_specification
;
/*
 * id_ds-specification
 */
id_ds_specification : ("ID_D" | "ID_DS")
    "="   id_d_list
;
id_d_list : id_d   { "," id_d }
;
id_d : id_d_const
;
id_d_const : T_INTEGER
  | "NIL"
;
```

#### 3.10.2.2 References

For the non-terminals `with_id_d_specification` and `object_creation_specification`, please see section 3.10.1 on page 62.

#### 3.10.2.3 Example

```
CREATE OBJECT FROM ID_DS = 10028, 10029
[Clause
   clause_type := NC;
   parent := 10033;
]
GO
```

#### 3.10.2.4 Explanation

This statement creates a new object with the monads contained in the objects specified by their id_ds.

The id_ds specified are used only to calculate the set of monads to be used. This is calculated as the union of the set of monads of the objects with the id_ds specified. These id_ds can point to objects of any type, and it need not be the same type for all id_ds.

Note that there is a syntactic sugar-choice of whether to say "ID_DS" or "ID_D".

Note that objects of the special object types all_m, any_m, and pow_m cannot be created.

See the "CREATE OBJECT FROM MONADS" (section 3.10.1 on page 62) for further explanation and warnings. Especially about concurrent use of WITH ID_D.

#### 3.10.2.5 Return type

The return type is the same as for CREATE OBJECT FROM MONADS (section 3.10.1).

### 3.10.3 CREATE OBJECTS WITH OBJECT TYPE

#### 3.10.3.1 Syntax

```
create_objects_statement : "CREATE"  "OBJECTS"
                           "WITH"  "OBJECT" "TYPE"
                           "["  object_type_name  "]"
                           object_creation_list
;
```

```
object_creation_list : object_creation { object_creation }
;

object_creation : "CREATE"  "OBJECT"
                  "FROM"  monad_specification
                  [ with_id_d_specification ]
                  "["
                  [ list_of_feature_assignments ]
                  "]"
;
```

#### 3.10.3.2 References

For the non-terminals `monad_specification`, `with_id_d_specification`, and `list_of_feature_assignments` please see section 3.10.1 on page 62.

#### 3.10.3.3 Example

```
CREATE OBJECTS
WITH OBJECT TYPE [Phrase]
CREATE OBJECT
FROM MONADS = { 1-2 }
[
  phrase_type := NP;
  function := Subj;
]
CREATE OBJECT
FROM MONADS = { 3-7 }
[
  // Use default values for phrase_type and function
  // (probably VP/Pred in this fictive example)
]
CREATE OBJECT
FROM MONADS = { 4-7 }
WITH ID_D = 1000000 // Assign specific ID_D
[
  phrase_type := NP;
  function := Objc;
]
GO
```

#### 3.10.3.4 Explanation

This statement is for batch importing of objects. It is useful when populating databases, either from scratch or by adding large numbers of objects to an existing database. This statement is much faster than individual CREATE OBJECT statements.

The object type is specified only once, at the top. Note that no features can be assigned where the object type is specified: That comes later in the query, when each object is created.

Each object to be created must be given a monad set. The monad set follows the syntax specified in section 3.8.1 on page 45.

Optionally, an id_d can be specified. If an id_d is specified, it is the user's responsibility to ensure that the id_d assigned does not clash with another id_d in the database. This is mainly useful when dumping/restoring databases.

If no id_d is specified, a unique id_d is generated. This id_d is only guaranteed to be unique if no other objects are created with specific id_ds.

Note that when using WITH ID_D, it is not recommended to run several concurrent processes against the same database which issue CREATE OBJECT or CREATE OBJECTS statements. Doing so may cause the auto-generated object id_d sequence to become invalid. However, several concurrent processes may safely issue CREATE OBJECT(S) statements if none of them use WITH ID_D.

The feature-value assignments follow the same rules as for CREATE OBJECT FROM MONADS (see section 3.10.1 on page 62). If an object has a feature which is not assigned a value, the default value is used. The default value of a given feature can be specified when creating the object type, or when updating the object type (see section 3.5.1 on page 36 and section 3.5.2 on page 40).

A table is returned showing the number of objects created successfully. This number is valid even if the process failed half way through. In other words, if the process did not run to completion due to a DB error, the value in the return type will show how many objects, if any, were created successfully. This means that there is no way of knowing which object got which object id_d, a difference from the regular CREATE OBJECT statement.

#### 3.10.3.5 Return type

A table with the following schema:

| object_count: integer |
|---|

On both success and failure, the table contains one row showing the number of objects created successfully.

### 3.10.4 UPDATE OBJECTS BY MONADS

#### 3.10.4.1 Syntax

```
update_objects_by_monads_statement : "UPDATE"
    ("OBJECT" | "OBJECTS")
    "BY"   monad_specification
    object_update_specification
;
/*
 * object-update-specification
 */
object_update_specification : "["   object_type_name
    list_of_feature_assignments
    "]"
;
object_type_name : T_IDENTIFIER
;
```

#### 3.10.4.2 References

For the non-terminals `monad_specification` and `list_of_feature_assignments`, please see section 3.10.1 on page 62.

#### 3.10.4.3 Example

```
UPDATE OBJECTS BY MONADS = { 1-2, 4-7, 8-20 }
[Clause
    clause_type := VC;
]
GO
```

#### 3.10.4.4 Explanation

This statement finds all the objects of type `object_type_name` which are part_of the monads specified (i.e., they must be wholly contained within the monads specified), and updates their features according to the list of feature assignments.

Note that there is a syntactic sugar-choice of whether to say "OBJECTS" or "OBJECT". This is because the user may know that only one object is to be found within the monads, in which case having to write "OBJECTS" would be intellectually irritating.

Note that objects of the special object types all_m, any_m, and pow_m cannot be updated.

The feature "self" cannot be updated.

#### 3.10.4.5 Return type

A table with the following schema:

| object_id_d: id_d |
|---|

On success, the table contains one row for each updated object.

On failure, the table is empty.

### 3.10.5 UPDATE OBJECTS BY ID_DS

#### 3.10.5.1 Syntax

```
update_objects_by_id_ds_statement : "UPDATE"
    ("OBJECT" | "OBJECTS")
    "BY"   id_ds_specification
    object_update_specification
;
```

#### 3.10.5.2 References

For a description of `id_ds_specification`, see section 3.10.2 on page 64. For a description of `object_update_specification`, see section 3.10.4 on page 67.

#### 3.10.5.3 Example

```
UPDATE OBJECTS BY ID_DS = 10028, 10029
[Phrase
    parent := 10034;
]
GO
```

#### 3.10.5.4 Explanation

This statement updates all the objects of the given type with the given id_ds.

The id_ds should point to objects which are really of the given type. Otherwise, an error is issued.

Note that there is a syntactic sugar-choice between "OBJECTS" and "OBJECT."

Note that objects of the special object types all_m, any_m, and pow_m cannot be updated.

The feature "self" cannot be updated.

#### 3.10.5.5 Return type

The return type is the same as for UPDATE OBJECTS BY MONADS (section 3.10.4 on the preceding page).

### 3.10.6 DELETE OBJECTS BY MONADS

#### 3.10.6.1 Syntax

```
delete_objects_by_monads_statement : "DELETE"
    ("OBJECT" | "OBJECTS")
    "BY"   monad_specification
    object_deletion_specification
;
/*
 * object-deletion-specification
 */
object_deletion_specification : "["
    object_type_name_to_delete
    "]"
;
object_type_name_to_delete : object_type_name
  | "ALL"
;
object_type_name : T_IDENTIFIER
;
```

#### 3.10.6.2 References

For a description of `monad_specification`, see section 3.10.1 on page 62.

#### 3.10.6.3 Example

```
DELETE OBJECTS BY MONADS = { 1-20 }
[Clause]
GO
```

If "`object_name_to_delete`" is "ALL", then all objects of all types which are at these monads are deleted:

```
DELETE OBJECTS BY MONADS = { 28901-52650 }
[ALL]
GO
```

#### 3.10.6.4 Explanation

This command deletes all the objects of type `object_type_name` which are part_of the set of monads specified.

#### 3.10.6.5 Return type

A table with the following schema:

| object_id_d: id_d |
|---|

On success, the table contains one row for each deleted object.
On failure, the table is empty.

### 3.10.7 DELETE OBJECTS BY ID_DS

#### 3.10.7.1 Syntax

```
delete_objects_by_id_ds_statement : "DELETE"
    ("OBJECT" | "OBJECTS")
    "BY"   id_ds_specification
    object_deletion_specification
;
```

#### 3.10.7.2 References

For a description of `id_ds_specification`, please see section 3.10.2 on page 64. For a description of `object_deletion_specification`, please see section 3.10.6 on page 69.

#### 3.10.7.3 Example

```
DELETE OBJECTS BY ID_DS 10028, 10029
[Phrase]
GO
```

#### 3.10.7.4 Explanation

This statement deletes objects by their id_ds. Note that you cannot write "ALL" for `object_deletion_specification`. The id_ds given should point to objects of the type given.

#### 3.10.7.5 Return type

The return type is the same as for DELETE OBJECTS BY MONADS (section 3.10.6).

## 3.11 Monad manipulation

### 3.11.1 MONAD SET CALCULATION

#### 3.11.1.1 Syntax

```
monad_set_calculation_statement : "MONAD"  "SET"
    "CALCULATION"
    monad_set_chain
;
monad_set_chain : monad_set
                  { monad_set_operator  monad_set }
;
monad_set_operator : "UNION"
  | "DIFFERENCE"
  | "INTERSECT"
;
```

#### 3.11.1.2 References

For a description of `monad_set`, please see section 3.8.1 on page 45.

#### 3.11.1.3 Example

```
// Produces { 1-10 }
MONAD SET CALCULATION
{ 1-5, 7-8 }
UNION
{ 5-10 }
GO

// Produces { 2-5, 22-24 }
MONAD SET CALCULATION
{ 1-10, 20-30, 50-60 }
INTERSECT
{ 2-5, 22-24 }
GO

// Produces { 1-4, 8-10 }
MONAD SET CALCULATION
{ 1-10 }
DIFFERENCE
{ 5-7 }
GO

// Produces { 2-3, 5-6, 10-12 }
MONAD SET CALCULATION
{ 1-3, 5-9 }
INTERSECT
{ 2-6 }
UNION
{ 10-12 }
GO
```

#### 3.11.1.4 Explanation

This statement is for performing set-operations on sets of monads. The three standard set operations "union," "intersect," and "difference" are provided.

The return value is a representation of the resulting set of monads along the same lines as for the GET MONADS statement (see section 3.8.6).

The MSEs (see section 3.8.6) are listed in ascending order.

You can specify as many sets of monads as you want. The operations are done in succession from the first to the last set of monads. For example, in the last example above, the intersection is done first, and the union is done on the result of the intersection.

You can also specify only one set of monads, with no set operator. This is useful for creating a sorted, normalized set of monads from a number of different MSEs.

Note that this statement does not manipulate the stored arbitrary monad sets described in section 2.7.13 on page 23.

#### 3.11.1.5 Return type

A table with the following schema:

| mse_first : monad_m | mse_last : monad_m |
|---|---|

## 3.11.2 CREATE MONAD SET

### 3.11.2.1 Syntax

```
create_monad_set_statement : "CREATE" "MONAD" "SET"
            monad_set_name
            "WITH" "MONADS" "=" monad_set
;
monad_set_name : T_IDENTIFIER
;
```

### 3.11.2.2 References

For the monad_set non-terminal, please see section 3.8.1 on page 45.

### 3.11.2.3 Example

```
CREATE MONAD SET
My_research_collection
WITH MONADS = { 1-10394, 14524-29342, 309240-311925 }
GO
```

### 3.11.2.4 Explanation

This statement creates an arbitrary monad set in the database. These monad sets are the ones described in section 2.7.13 on page 23.

### 3.11.2.5 Return type

There is no return value.

## 3.11.3 UPDATE MONAD SET

### 3.11.3.1 Syntax

```
update_monad_set_statement : "UPDATE" "MONAD" "SET"
     monad_set_name
     ("UNION" | "INTERSECT" | "DIFFERENCE" | "REPLACE")
     (monad_set | monad_set_name)
;
monad_set_name : T_IDENTIFIER
;
```

### 3.11.3.2 References

For the monad_set non-terminal, please see section 3.8.1 on page 45.

### 3.11.3.3 Examples

```
// Adds the specified monad set to "Historical_books"
UPDATE MONAD SET
Historical_books
UNION
{ 310320-329457 }
GO
// Remove the specified monad set from "Historical_books"
```

```
UPDATE MONAD SET
Historical_books
DIFFERENCE
{ 310320-329457 }
GO
// Intersects the monad set "My_research_collection"
// with the monad set "My_experimental_collection"
UPDATE MONAD SET
My_research_collection
INTERSECT
My_experimental_collection
GO
// Replaces the monad set "Lamentations" with
// the specified monad set
UPDATE MONAD SET
Lamentations
REPLACE
{ 380300-383840 }
GO
```

#### 3.11.3.4 Explanation

This statement is used to update an already-existing arbitrary monad set (see section 2.7.13 on page 23). Four operations are provided: set union, set intersection, set difference, and replacement. In all cases, the operation is done using two monad sets. The first set is the named set that is updated. The second set is either a set described in terms of monads, or the name of another arbitrary monad set.

The replacement operator effectively deletes the old set, replacing it with the new. Note, however, that this does not imply that the new is deleted – if you update one named monad set, replacing it with another named monad set, that other monad set is not deleted, but simply copied into the old monad set.

The other three operators are standard set-theoretic operators.

#### 3.11.3.5 Return type

There is no return value.

### 3.11.4 DROP MONAD SET

#### 3.11.4.1 Syntax

```
drop_monad_set_statement : "DROP" "MONAD" "SET"
                           monad_set_name
;
monad_set_name : T_IDENTIFIER
;
```

#### 3.11.4.2 Example

```
DROP MONAD SET Historical_books
GO
```

#### 3.11.4.3 Explanation

This statement drops an arbitrary monad set (i.e., deletes it) from the database. These are the arbitrary monad sets described in section 2.7.13 on page 23.

#### 3.11.4.4 Return type

There is no return value.

## 3.12 Meta-statements

### 3.12.1 QUIT

#### 3.12.1.1 Syntax

```
quit_statement : "QUIT"
;
```

#### 3.12.1.2 Example

```
QUIT
```

#### 3.12.1.3 Explanation

This causes the rest of the MQL stream not to be interpreted. It also causes the mql(1) program to quit after having executed this statement.

The QUIT statement can be used, e.g., if running the mql(1) program as a daemon through xinetd(8) or inetd(8), to end the connection.

The QUIT statement is special in that it does not need a "GO" keyword after it. You may supply the "GO" keyword if you wish, but it is not required.

If a transaction was in progress (see BEGIN TRANSACTION statement, section 3.4.1 on page 34), the transaction is automatically committed before the QUIT statement is executed.

#### 3.12.1.4 Return type

There is no return value.

# Chapter 4

# MQL Query subset

## 4.1 Introduction

This chapter is an introduction to the query-subset of MQL for programmers. That is, it introduces the important subset of MQL in which you can express queries that find objects and gaps in interesting environments, with specified interrelations, and with specified feature-values.

An easier-to-read MQL Query Guide is available from the Emdros website, or with the Emdros source-code in the doc/ directory (see [MQLQueryGuide]).

First, we give an informal introduction to MQL by means of some examples (4.2). Then we give a complete overview of the syntax of the MQL query-subset (4.3). Then we explain the sheaf, which is the data-structure that an MQL query-query returns (4.4). Then we explain what a Universe and a Substrate are, since they are important in understanding how a query works (4.5). After that, we explain two important properties of mql queries, namely consecutiveness and embedding (4.6). After that, we give detailed explanations of the blocks of the MQL query-subset, which are the "building blocks" out of which a query is made (4.7). Finally, we explain how strings of blocks are written, and what they mean (4.8).

## 4.2 Informal introduction to MQL by means of some examples

### 4.2.1 Introduction

This section informally introduces the query-part of MQL by way of a number of examples. The example database which we will use is the same as in Doedens' book, namely part of Melville's "Moby Dick":

> "CALL me Ishmael. Some years ago - never mind how long precisely - having little or no money in my purse, and nothing particular to interest me on shore, I thought I would sail about a little and see the watery part of the world. It is a way I have of driving off the spleen, and regulating the circulation. Whenever I find myself growing grim about the mouth; whenever it is damp, drizzly November in my soul; whenever I find myself involuntarily pausing before coffin warehouses, and bringing up the rear of every funeral I meet; and especially whenever my hypos get such an upper hand of me, that it requires a strong moral principle to prevent me from deliberately stepping into the street, and methodically knocking people's hats off - then, I account it high time to get to sea as soon as I can. [...]
>
> "[...] By reason of these things, then, the whaling voyage was welcome; the great flood-gates of the wonder-world swung open, and in the wild conceits that swayed me to my purpose, two and two there floated into my inmost soul, endless processions of the whale, and, mid most of them all, one grand hoofed phantom, like a snow hill in the air."

Suppose that we have in this EMdF database the domain-dependent object types "paragraph", "sentence", and "word", which correspond to paragraphs, sentences, and words of the text. And suppose that we add to the object type "sentence" the feature "mood," which draws its values from the enumeration type {

imperative, declarative }. And suppose that we add to the object type "word" the features "surface" (which gives the surface text of the word) and "part_of_speech" (which gives the part of speech of the word). The codomain of the feature "part_of_speech" on the object type "word" draws its values from the enumeration type { adjective, adverb, conjunction, determiner, noun, numeral, particle, preposition, pronoun, verb }. This hypothetical database will give the background for most of the examples in our informal introduction to MQL.

In the following, when we refer to an "MQL query", we will mean the query-subset of MQL. That is, we abstract away from the database-manipulation-part of MQL and concentrate on the query-queries. In addition, we will abstract away from the required "SELECT (FOCUS|ALL) OBJECTS" syntax that must precede an MQL query-query.

### 4.2.2 topograph

An MQL query is called a topograph. Consider the following topograph:

```
[sentence]
```

This topograph retrieves a list of all sentence objects in the database.

### 4.2.3 features

A query can specify which features an object must have for it to be retrieved. For example, consider the following topograph:

```
[word
   surface = "Ishmael" or part_of_speech = verb;
]
```

This topograph retrieves a list of all words which either have the surface "Ishmael", or whose part of speech is "verb."

### 4.2.4 object_block, object_block_first

There are several types of blocks. They are meant to come in a string of blocks, where each block in the string must match some part of the database in order for the whole string to match. Two such blocks are the `object_block` and the `object_block_first`.

Object blocks are the heart and soul of MQL queries. They are used to match objects and objects nested in other objects. An object block (be it an `object_block` or an `object_block_first`) consists of the following parts:

1. The opening square bracket, '`[`'.
2. An identifier indicating the object type of the objects which we wish to match (e.g., "phrase").
3. An optional T_MARKS (e.g., "'yellow" or "'red'context"). This will be put into the result set (i.e., sheaf) unchanged, and can be used to pass information back into the application from the user. The meaning of the T_MARKS is wholly application-dependent, since Emdros does nothing special with it — it just passes the T_MARKS on into the sheaf. See page 26 for the formal definition of T_MARKS.
4. An optional "object reference declaration." A reference to this object can be declared with the "`as`" keyword, like "`[word as w ...`". Subsequent blocks can then refer to features of this object as "`w`.*featurename*" (see section 4.7.8 on page 99).

5. An optional keyword which can be either of "noretrieve", "retrieve" or "focus". The default, when it is not specified, is "retrieve". The keyword "noretrieve" says as much as "I do not wish to retrieve this object, even if matched". It is useful for specifying the context of what we really wish to retrieve. The keyword "focus" specifies that this object is to be retrieved (it implies "retrieve"), and also that, when sifting the sheaf for focus objects, this object must go into the result (see section 4.7.5 on page 92).

6. An optional keyword, "first" or "last", which says as much as "this object must be first/last in the universe against which we are matching (see section 4.7.6 on page 94).

7. An optional Boolean expression giving what features need to hold true for this object for it to be retrieved (see section 4.7.7 on page 95). This boolean expression must be prefixed by one of the words "feature" or "features". It makes no difference which is used – it is merely syntactic sugar.

8. An optional inner blocks which matches objects inside the object (see section 4.8.3).

9. The closing square bracket, ']'.

Note that only the first object block in a string of blocks can have the "first" keyword, and only the last object_block in a string of blocks can have the "last" keyword.

Consider the following topograph:

```
[sentence'yellow
    mood = imperative;
    [word noretrieve first
         surface = "CALL";
    ]
    [word'red]
]
```

This topograph retrieves the set of sentences which are imperative, and whose first word is "CALL". Within each sentence in that set, we retrieve the second word, but not the first. The only sentence in our example database which qualifies is the first sentence.

### 4.2.5 power

The power construct is used to indicate that we allow some distance in between two blocks. A power construct must always stand between two other blocks, and can thus never be first or last in a query. It comes in three varieties:

- A "plain vanilla" power construct, syntactically denoted by two dots, "..", and
- A power construct with a single, upper limit. The limit specifies the maximum monads that can intervene between the two surrounding blocks. It is denoted as e.g., ".. < 5", or ".. <= 5".
- A power construct with a compound min/max limit. The limit specifies the minimum and maximum monads that can intervene. It is denoted as, e.g., ".. BETWEEN 1 AND 5".

Consider the following topograph:

```
[sentence
    [word
        part_of_speech = preposition]
    .. < 4
    [word
        part_of_speech = noun]
    ..
```

```
    [word last
        surface = "world"]
]
```

This topograph retrieves a list of sentences which have a word that has part of speech preposition, followed by a word which has part of speech noun, and which is within 4 monads of the preposition, followed by the last word of the sentence, which must be "world". Within that sentence, retrieve all the three words. The only sentence which qualifies is the second.

### 4.2.6 opt_gap_block

An `opt_gap_block` is used to match an optional gap in the text. It consists of:

1. The opening square bracket, '`[`'.
2. The keyword "`gap?`".
3. An optional T_MARKS (e.g., "'yellow" or "'red'context"). This will be put into the result set (i.e., sheaf) unchanged, and can be used to pass information back into the application from the user. The meaning of the T_MARKS is wholly application-dependent, since Emdros does nothing special with it — it just passes the T_MARKS on into the sheaf. See page 26 for the formal definition of T_MARKS.
4. An optional "`noretrieve`," "`retrieve`" or "`focus`." The default is "`noretrieve`". (See section 4.7.5 on page 92)
5. An optional `blocks` (see section 4.8.3 on page 103).
6. The closing square bracket, '`]`'.

The `opt_gap_block` matches gaps in the *substrate* against which we are matching. Thus if we look at the example in figure 2.1 on page 19, we can construct the following topograph:

```
[clause
    [clause_atom
        [word
             surface = "door,"
        ]
    ]
    [gap? noretrieve]
    [clause_atom noretrieve]
]
```

This retrieves all clauses which happen to have inside them a clause_atom which contains the word "door,", followed by a gap, followed by a clause_atom. The gap and the second clause_atom are not retrieved. This would retrieve clause-1. The gap need not be there.

The default is for the result of an `opt_gap_block` not to be retrieved. Thus one needs to explicitly write "`retrieve`" if one wishes to retrieve the gap.

### 4.2.7 gap_block

A `gap_block` is used to match a gap in the text. It consists of:

1. The opening square bracket, '`[`'.
2. The keyword "`gap`".

3. An optional T_MARKS (e.g., "'yellow" or "'red'context"). This will be put into the result set (i.e., sheaf) unchanged, and can be used to pass information back into the application from the user. The meaning of the T_MARKS is wholly application-dependent, since Emdros does nothing special with it — it just passes the T_MARKS on into the sheaf. See page 26 for the formal definition of T_MARKS.

4. An optional "`noretrieve`," "`retrieve`" or "`focus`." The default is "`noretrieve`". (See section 4.7.5 on page 92).

5. An optional `blocks`. (See section 4.8.3 on page 103).

6. The closing square bracket, '`]`'.

The `gap_block` is analogous to the `opt_gap_block` in all respects except that there *must* be a gap in order for the query to match.

### 4.2.8 object references

An object reference is a name given to a previously retrieved object with the "`as` *identifier*" declaration. An object reference can then be used in subsequent comparisons with features of other objects. This is done by selecting the desired feature from the object reference by using dot-notation, as in the example below:

```
[word as w
    part_of_speech = article;
]
[word'myhit
   (part_of_speech = noun
    or part_of_speech = adjective)
    and case = w.case
    and number = w.number
    and gender = w.gender;
]
```

Assuming that the `word` object type has features part_of_speech, case, number, and gender, this topograph retrieves all pairs of words which satisfy the following conditions:

- The first word has part of speech "article",
- The second word has part of speech "noun" or "adjective", and
- Both words have the same case, number, and gender.

This concludes our gentle, informal introduction to MQL.

## 4.3 Syntax of mql_query

### 4.3.1 Introduction

The `mql_query` non-terminal is the entry-point for the MQL query-subset. It is used in the WHERE clause of the SELECT (FOCUS|ALL) OBJECTS statement (section 3.8.1 on page 45). In this section, we give the full grammar of the MQL query-subset. It is important that you take some time to read through the grammar. Subsequent sections will build on the bird's-eye view given in this section.

### 4.3.2 Syntax

```
mql_query : topograph
;
topograph : blocks
;
blocks : block_string
;
block_string : block_string2
             | block_string2 "OR" block_string
;
block_string2 : block_string1
              | block_string1 block_string2
              | block_string1 "!" block_string2
;
block_string1 : block_string0
              | block_string0 "*" [monad_set]
;
block_string0 : block
              | "[" block_string "]"
;
block : opt_gap_block
      | gap_block
      | power_block
      | object_block
      | ("NOTEXIST" | "NOTEXISTS" ) object_block
;
opt_gap_block : "["   "GAP?"
                      [ marks_declaration ]
                      [ gap_retrieval ]
                      [ blocks ]
                "]"
;
marks_declaration : T_MARKS
;
gap_retrieval : "NORETRIEVE"
              | "RETRIEVE"
              | "FOCUS"
;
gap_block : "["   "GAP"
                  [ marks_declaration ]
                  [ gap_retrieval ]
                  [ blocks ]
            "]"
;
object_block :  "["   object_type_name
                     [ marks_declaration ]
                     [ object_reference_declaration ]
                     [ retrieval ]
                     [ firstlast ]
                     [ monad_set_relation_clause ]
                     [ feature_constraints ]
                     [ feature_retrieval ]
                     [ blocks ]
```

```
              "]"
;
object_reference_declaration : "AS" object_reference
;
object_reference : T_IDENTIFIER
;
retrieval : "NORETRIEVE"
          | "RETRIEVE"
          | "FOCUS"
;
firstlast : "FIRST"
          | "LAST"
          | "FIRST"   "AND"   "LAST"
;
feature_constraints  : ffeatures
;
ffeatures : fterm
          |  ffeatures   "OR"   fterm
;
fterm : ffactor
      | ffactor   "AND"   fterm
;
ffactor : "NOT"   ffactor
        | "("   ffeatures   ")"
        | feature_comparison
;
feature_comparison :
    comparison_feature_name  comparison_operator   value
  | comparison_feature_name  "="   ()
  | comparison_feature_name  "="   enum_const_set
  | comparison_feature_name  "="   "(" list_of_integer ")"
  | comparison_feature_name  "IN"   enum_const_set
  | comparison_feature_name  "IN"   "(" list_of_integer ")"
  | comparison_feature_name  "IN"   object_reference_usage
;
comparison_operator : "="
                    | "<"
                    | ">"
                    | "<>"  /* not equal */
                    | "<="  /* less than or equal */
                    | "=<"  /* less than or equal */
                    | ">="  /* greater than or equal */
                    | "=>"  /* greater than or equal */
                    | "~"   /* regular expression */
                    | "!~"  /* inverted regular expression */
                    | "HAS" /* lhs: list; rhs: atomic value.
                                 signifies list membership. */
;
list_of_integer : T_INTEGER { "," T_INTEGER }*
;
value : enum_const
      | signed_integer
      | T_STRING
      | object_reference_usage
```

```
;
enum_const : T_IDENTIFIER
;
object_reference_usage : object_reference
                         "."   feature_name
;
enum_const_set : "(" enum_const_list ")"
;
enum_const_list : enum_const { "," enum_const_list }
;
power : ".."   [ restrictor ]
;
restrictor : "<" limit
           | "<=" limit
           | "BETWEEN" limit "AND" limit
;
limit : T_INTEGER /* non-negative integer, may be 0. */
;
feature_retrieval : "GET" feature_list
  | /* empty: Don't retrieve any features */
;
monad_set_relation_clause : /* empty; which means: part_of(substrate) */
           | monad_set_relation_operation "(" universe_or_substrate ")"
           | monad_set_relation_operation "(" monad_set_or_monads "," universe_or_s
;
monad_set_relation_operation : "part_of" | "overlap"
;
universe_or_substrate : "universe" | "substrate"
;
monad_set_or_monads : "monads" | T_IDENTIFIER
;
comparison_feature_name : feature_name
           | computed_feature_name
;
computed_feature_name :
             "first_monad" computed_feature_monad_set_name
           | "last_monad" computed_feature_monad_set_name
           | "monad_count" computed_feature_monad_set_name
           | "monad_set_length" computed_feature_monad_set_name
;
computed_feature_monad_set_name : /* empty: Means "(monads)". */
           | "(" "monads" ")" /* Means: "(monads)"
           | "(" monad_set_name ")"
;
```

### 4.3.3 References

For the `signed_integer` non-terminal, please see section 3.5.1 on page 36. For `feature_name`, see 3.5.2 on page 40. For `feature_list`, see section 3.8.7 on page 56. For `monad_set`, see section 3.8.1 on page 45.

## 4.4 The sheaf

### 4.4.1 Introduction

The sheaf is the data structure that is returned from an MQL query-query. The structure of the sheaf closely reflects the structure of the query on which it is based. This section is meant as reading for implementors of Emdros-systems, not for end-users.

The sheaf has a specific structure, which we will look at next. After that, we will take a look at the meaning of the structures of the sheaf.

### 4.4.2 Structure of the sheaf

A sheaf consists of the following element types:

1. Sheaf
2. Straw
3. Matched_object

#### 4.4.2.1 What is a sheaf?

A sheaf *is* a list of straws.

#### 4.4.2.2 What is a straw?

A straw *is* a list of matched_objects.

#### 4.4.2.3 What is a matched_object?

A matched_object *is* one of the following:

1. (object id_d, focus boolean, marks, sheaf, object type, set of monads, list of feature-values)
2. (object id_m, focus boolean, marks, sheaf)

That is, a matched_object is an object id (either id_d or id_m), coupled with a boolean indicating whether the block that gave rise to the matched_object had the "FOCUS" modifier, coupled with a "marks" string, coupled with a sheaf. If the matched_object is of the first kind, then additionally, the object type and the object's set of monads are also available, and there is a (possibly empty) list of feature-values.

### 4.4.3 MQL is topographic

There is a correspondence between the way an MQL query is structured and the structure of the resulting sheaf. In fact, the two are isomorphic to some extent. Doedens, in [Doedens94], called this property "topographicity." Thus a `blocks` gives rise to a sheaf, a `block_str` gives rise to a straw, and a `block` gives rise to a matched_object. Inside a `block`, there is an optional inner `blocks`, which again gives rise to an inner sheaf. Hence a matched_object contains a sheaf. The origin of this sheaf is the optional inner `blocks` in the `block` which gave rise to the matched_object.

Note that this description applies to "full sheaves." Flat sheaves are a different matter. See section 4.4.7 on page 85 for a description of flat sheaves.

### 4.4.4 Meaning of matched_object

A matched_object is the result of one of the following matches:

1. An `object_block` against an object in the database.
2. An `opt_gap_block` against a gap.
3. A `gap_block` against a gap.

A matched_object's first component is either an id_d or an id_m. If the matched_object is the result of a match against an object_block or an object_block_first, then the id will be an id_d. If the matched_object is the result of a match against a `gap_block` or an `opt_gap_block`, the id is an id_m.

The second component is a boolean indicating whether the "FOCUS" keyword was present on the block.

The third component is a sheaf.

As we will see later, a sheaf is the result of matching against a `blocks`. It so happens that there is an optional `blocks` inside each of the four kinds of block (in the list above). The sheaf inside the matched_object is the result of a match against this `blocks`, if present. If the `blocks` is not present, then the sheaf is simply an empty sheaf.

For example, the following topograph:

```
[word FOCUS]
```

will contain one matched_object for each word-object within the substrate of the topograph. The sheaf of each of these matched_objects will be empty, and the FOCUS boolean will be "true" because we specified the FOCUS keyword.

### 4.4.5 Meaning of straw

A straw is the result of one complete match of a `block_str`. That is, a straw is a "string" of matched_objects corresponding to the blocks in the `block_str` which we should retrieve (which we can specify with the ("FOCUS"|"RETRIEVE"|"NORETRIEVE") keyword triad).

For example, consider the following topograph:

```
[word
   surface = "the";
]
[word
   part_of_speech = noun;
]
```

This will return a sheaf with as many straws as there are pairs of adjacent words where the first is the word "the" and the second is a noun. Each straw will contain two matched_objects, one for each word.

### 4.4.6 Meaning of the sheaf

A sheaf is the result of gathering all the matchings of a `blocks` non-terminal. There are four places in the MQL grammar where a `blocks` non-terminal shows up:

1. In the `topograph`,
2. In the `object_block`,
3. In the `opt_gap_block`, and
4. In the `gap_block`.

The first is the top-level non-terminal of the MQL query-query grammar. Thus the result of an MQL query-query is a sheaf.

Each of the last three is some kind of block. Inside each of these, there is an optional `blocks`. The result of matching this `blocks` is a sheaf.

But a sheaf is a list of straws. What does that mean?

It means that a sheaf contains as many matches of the strings of blocks (technically, block_string2) making up the `blocks` as are available within the substrate and universe that governed the matching of the `blocks`.

A straw constitutes one matching of the `block_string2`. A sheaf, on the other hand, constitutes all the matchings.

### 4.4.7 Flat sheaf

Most of the above description has applied to "full sheaves." We now describe flat sheaves.

A "flat sheaf," like a "full sheaf," consists of the datatypes "sheaf," "straw," and "matched_object." The difference is that a "matched_object" in a flat sheaf cannot have an embedded sheaf. This makes a flat sheaf a non-recursive datastructure.

A flat sheaf arises from a full sheaf by means of the "flatten" operator.

If "FullSheaf" is a full sheaf, then "flatten(FullSheaf)" returns a flat sheaf that corresponds to the full sheaf.

A flat sheaf contains the same matched_objects as its originating full sheaf. However, they are structured such that each straw in the flat sheaf contains only matched_objects of one object type. Each object type that is represented in the full sheaf results in one straw in the flat sheaf.

Thus a straw in a flat sheaf does not correspond to the matching of a block_string. Instead, it is a list of all the matched_objects of one particular object type in the originating full sheaf. All of the matched_objects in the full sheaf are represented in the flat sheaf, regardless of whether they represent the same object in the database.

The "flatten" operator is only applied to the output of an MQL query if the "RETURNING FLAT SHEAF" clause is given (see section 3.8.1 on page 45). The programmer of an Emdros application can also apply it programmatically.

There is a variant of the flatten operator which also takes a list of object type names, in addition to the full sheaf. Then only those object types which are in the list are put into the flat sheaf. If L is a list of object type names, and FullSheaf is a full sheaf, then flatten(FullSheaf, L) returns a flat sheaf with straws for only those object types which are in L. If L is empty, then this is interpreted as meaning that all object types in FullSheaf must go into the flat sheaf. In the this light, the single-argument flatten operator may be seen as being a special case of the two-argument flatten operator, with L being empty. That is, flatten(FullSheaf) is the same as flatten(FullSheaf, []).

## 4.5 Universe and substrate

### 4.5.1 Introduction

Two concepts which we shall need when explaining the blocks in MQL are "Universe" and "Substrate." In this section, we define and explain them.

### 4.5.2 Universe and substrate

A Universe is a contiguous set of monads. It always starts at a particular monad $a$ and ends at another monad $b$, where $a \leq b$. In more everyday language, a Universe is a stretch of monads that starts at one monad and ends at another monad later in the database. The ending monad may be the same as the starting monad.

A Substrate, on the other hand, is an arbitrary set of monads. It may have gaps (see section 2.7.8 on page 21). That is, while a Substrate always begins at a certain monad $a$ and always ends at another monad $b$, where $a \leq b$, it need not contain all of the monads in between.

A Universe always has an accompanying Substrate, and a Substrate always has an accompanying Universe. Their starting- and ending-monads are the same. That is, the first monad of the Universe is always the same as the first monad of the accompanying Substrate. And the last monad of the Universe is always the same as the last monad of the Substrate. So a Universe is a Substrate with all the gaps (if any) filled in.

See section 3.8.1.4 on page 47 for an explanation of how the initial substrate and universe are calculated for the query.

With that definition out of the way, let us proceed to describing, exemplifying, and explaining *blocks*.

## 4.6 Consecutiveness and embedding

Two important notions in the MQL query-subset are embedding and consecutiveness. If two blocks (be they object blocks or gap blocks) are consecutive in a query, it means that they will only match two objects or gaps which are consecutive with respect to the substrate. Likewise, a string of blocks (i.e., a `blocks`) which is embedded inside of a block of some sort will only match within the confines of the monads of the surrounding block.

For example, the following topograph:

```
[Word psp=article]
[Word psp=noun]
```

will match two adjacent (or consecutive) words where the first is an article and the second is a noun. The consecutiveness is calculated with respect to the current substrate (see section 2.7.10 on page 21).

Likewise, the following topograph:

```
[Clause
  [Phrase phrase_type = NP]
  [Phrase phrase_type = VP]
]
```

will match only if the two (adjacent) phrases are found *within the confines of* the monads of the surrounding Clause. In fact the monads of the surrounding clause serve as the substrate when matching the inner `blocks`.

You can specify the kind of containment you want: Either part_of or overlap. Part_of means that the inner object must be a subset (proper or not) of the outer object.

You can also specify whether the containment should be relative to the substrate or the universe. The universe is always the universe coming out the of the substrate that is the surrounding object or gap.

This is done as follows:

```
[Clause
  // Phrase is part_of the monads of the clause
  [Phrase part_of(substrate) phrase_type=NP]
  // Phrase is part_of the monads of the clause, including any gaps
  // in the clause
  [Phrase part_of(universe) phrase_type=VP]

  // Phrase has non-empty intersection with the monads of the clause
  [Phrase overlaps(substrate) phrase_type=AdvP]
  // Phrase is non-empty intersection with the monads of the clause,
  // including any gaps in the clause
  [Phrase overlaps(universe) phrase_type=PP]
]
```

The default is to use `part_of(substrate)`.

## 4.7 Blocks

### 4.7.1 Introduction

Blocks are the heart and soul of MQL query-queries. They specify which objects and which gaps in those objects should be matched and/or retrieved. With object blocks, you specify which objects should be matched. With gap blocks, you specify whether a gap should be looked for.

In this section, we treat the four kinds of blocks in MQL in some detail. First, we describe and explain the two kinds of object block (Object blocks, 4.7.2). Then we treat the two kinds of gap blocks (Gap blocks, 4.7.3). Then we describe how to specify whether to retrieve a block's contents (Retrieval, 4.7.5). After that we describe how to specify that an object block should be either first or last in its enclosing `blocks` (First and last, 4.7.6). Then we describe and explain how to specify constraints on features (Feature constraints, 4.7.7). Then we describe object references, which are a way of referring to other objects in a query (Object references, 4.7.8). Finally, we wrap up the syntactic non-terminals dealing with blocks by describing the `block` (Block, 4.7.9)

### 4.7.2 Object blocks

#### 4.7.2.1 Introduction

Object blocks specify which objects should be matched. Therefore, they are quite important in MQL. With object blocks, it is also possible to specify whether or not matched objects should be retrieved. You can also specify constraints on the features of the objects which should be matched; You can specify whether you want objects matched against a certain object block to be first or last in the string of blocks we are looking for at the moment; And finally, you can label objects matched in a query with object reference labels, so that those objects can be referred to later in the query (i.e., further down in the MQL query, and thus further on in the string of monads). In this subsection, we deal with the object blocks themselves, deferring the treatment of feature-constraints, first/last-specifications, and object references to later subsections.

First, we describe the syntax of object blocks, then we give some examples, and finally we give some explanatory information.

#### 4.7.2.2 Syntax

```
object_block :  "["   object_type_name
                      [ marks_declaration ]
                      [ object_reference_declaration ]
                      [ retrieval ]
                      [ firstlast ]
                      [ monad_set_relation_clause ]
                      [ feature_constraints ]
                      [ feature_retrieval ]
                      [ blocks ]
               "]"
;
object_type_name : T_IDENTIFIER
;
marks_declaration : T_MARKS
;
retrieval : "NORETRIEVE"
          | "RETRIEVE"
          | "FOCUS"
;
firstlast : "FIRST"
          |  "LAST"
          |  "FIRST"   "AND"   "LAST"
```

```
;
last : "LAST"
;
feature_retrieval : "GET" feature_list
  | /* empty: Don't retrieve any features */
;
monad_set_relation_clause : /* empty; which means: part_of(substrate) */
           | monad_set_relation_operation "(" universe_or_substrate ")"
           | monad_set_relation_operation "(" monad_set_or_monads "," universe_or_s
;
monad_set_relation_operation : "part_of" | "overlap"
;
universe_or_substrate : "universe" | "substrate"
;
monad_set_or_monads : "monads" | T_IDENTIFIER
;
```

#### 4.7.2.3 References

For `object_reference_declaration`, see section 4.7.8 on page 99. For `feature_constraints`, see section 4.7.7 on page 95. For `blocks`, see section 4.8.3 on page 103. For `feature_list`, please see section 3.8.7 on page 56.

#### 4.7.2.4 Examples

```
1. [Clause]

2. [Phrase noretrieve first
      phrase_type = NP
   ]

3. [Clause first and last]

4. [Word as w focus last
      psp = noun and number = pl
      GET surface, lexeme
   ]

5. [Clause'context
     [Phrase'red first
       phrase_type = NP and phrase_function = Subj
     ]
     [Phrase'green
       phrase_type = VP
       [Word
         psp = V
       ]
       [Phrase'blue
         phrase_type = NP and phrase_function = Obj
       ]
     ]
   ]

6. [Sentence
```

```
    NOTEXIST [Word surface = "saw"]
]
```

#### 4.7.2.5 Explanation

Firstly, it will be noticed that the first item after the opening bracket must always be an object type name. This is in keeping with all other parts of MQL where object type names are used.

Secondly, it will be noticed that all of the other syntactic non-terminals in the definition of the object blocks are optional.

The marks declaration comes after the object type name. The query-writer can use it to pass information back into the application that sits on top of Emdros. Emdros does nothing special with the T_MARKS, other than passing it on into the sheaf, that is, into the matched_object that arises because of the object_block. In particular, there is no semantics associated with the marks_declaration. See page 26 for the formal definition of T_MARKS.

The object reference declaration comes after the marks declaration, and will be dealt with below (4.7.8 on page 99).

The specification of the retrieval comes after the object reference declaration and will be dealt with in another section (4.7.5 on page 92).

The specification of the monad set relation clause has an impact on how the containment is calculated, and was dealt with above (4.6 on page 86).

The specification of first/last-constraints comes after the specification of retrieval, and will also be dealt with in another section (4.7.6 on page 94).

The specification of the monad set relation determines four things:

1. It determines which monad set will be used to match against the Substrate or Universe that accompanies the surrounding `blocks`. If the "`monad_set_or_monads`" specification is left out, the constituting monad set is used (i.e., the monad set which makes up the object). The same is true if the monad_set_or_monads specification is "`monads`". If the `monad_set_or_monads` specification is not left out, it must be a feature which must exist on the object type and be of the type "set of monads".

2. It determines which monad set to use as the Substrate of the inner `blocks`. The monad set used for the Substrate of the inner `blocks` is currently the same as the monad set used to match against the Universe or Substrate of the outer `blocks`. This may change in future releases of Emdros.

3. It determines whether to match against the Universe or Substrate of the outer b`locks`. This is done by the mention of "`universe`" or "`substrate`".

4. It determines which operation to use when matching against the Universe or Substrate of the surrounding `blocks`. This can be either "`part_of`" (the monad set of the object must be a subset of the Universe or Substrate) or "`overlap`" (non-empty set intersection). See section 2.7.7 on page 21 for details of the part_of relation.

It is possible to specify constraints on the features of objects. This is done in the `feature_constraints` non-terminal, which comes after the first/last-constraints. These constraints will be dealt with in a section below (4.7.7 on page 95).

A list of features can be given in the `feature_retrieval` clause. Their values for a given object are placed on the list of features in the matched_object in the sheaf.

The inner `blocks` syntactic non-terminal allows the writer of MQL queries the possibility of matching objects nested inside objects. Example 5 above shows several examples of this. Example 5 finds those clauses which have inside them first a phrase which is a Subject NP, then followed by a Phrase which is a VP, the first word inside of which is a verb, followed by an Object NP. Thus we have an object block ("Clause") with an inner `blocks` ("NP followed by VP"), where inside the VP we have another `blocks` (V followed by NP).

The inner blocks, if present, must match if the object block is to match. When entering the inner `blocks`, the Substrate for that `blocks` becomes the monads of the enclosing object. Let us call that

object O. The Universe for the inner `blocks` becomes the set of monads between and including the borders of the enclosing object (see section 2.7.9 on page 21), i.e., the stretch of monads between (and including) O.first and O.last. This is the same as the substrate, except with any gaps filled in.

If you want any objects or gaps inside the object block to be retrieved, then the retrieval of the enclosing object block must be either retrieve or focus. Since the default retrieval for object blocks is to retrieve them, this condition is satisfied if you write nothing for the retrieval.

An object, if it is to match against a given object block, must meet all of the following criteria:

1. The first/last constraints must be met.
2. The operation ("part_of" or "overlap") of the monad_set_relation_clause must be true on the given monad set and the Substrate or Universe.
3. The feature constraints must hold. See section 4.7.7 on page 95 for details.
4. The inner blocks must not return a failed sheaf.

You can optionally place the keyword "NOTEXIST" before the object block. This will result in matching those cases where the object block does not occur, and will result in a failed match where the object block does occur. This is most useful if you have some context, i.e., a surrounding context (e.g., a sentence which does not contain such and such a word, see example 6 above). You are allowed to have blocks before and after a NOTEXIST block. Let us say that there is a block before the NOTEXIST block. Then the Substrate within which the NOTEXIST block will be matched is the Substrate of the context, minus the monads from the beginning of the Substrate to the end of the MatchedObject matching the previous block. The Universe of the NOTEXIST block will be defined analogously on the Universe of the context.

The NOTEXIST block will have "zero width" with respect to consecutiveness: If it matches anything, the entire block_string fails. If it does not match, it is as though the NOTEXIST block had not been there, and any block after the NOTEXIST block will be attempted matched starting at the previous block's last monad plus 1.

The NOTEXIST keyword acts as an "upwards export barrier" of object reference declarations. That is, you cannot "see" an object reference declaration outside of the NOTEXIST, only inside of it.

### 4.7.3 Gap blocks

#### 4.7.3.1 Introduction

Gap blocks are used to match gaps in the substrate we are currently matching against. There are two kinds of blocks: plain gap blocks and optional gap blocks.

We start by defining the syntax related to gap blocks. We then give some examples of gap blocks. And finally, we provide some explanation.

#### 4.7.3.2 Syntax

```
gap_block : "["   "GAP"
                  [ marks_declaration ]
                  [ gap_retrieval ]
                  [ blocks ]
            "]"
;
opt_gap_block : "["   "GAP?"
                      [ marks_declaration ]
                      [ gap_retrieval ]
                      [ blocks ]
                "]"
;
marks_declaration : T_MARKS
```

```
;
gap_retrieval : "NORETRIEVE"
              | "RETRIEVE"
              | "FOCUS"
;
```

#### 4.7.3.3 Examples

```
1. [gap]

2. [gap?]

3. [gap noretrieve]

4. [gap¿yellow focus]

5. [gap`context`red retrieve
     [Word retrieve
       psp = particle
     ]
   ]

6. [gap`yellow
   ]
```

#### 4.7.3.4 Explanation

There are two differences between the two types of gap block: One is that the `gap_block` *must* match a gap in the substrate for the whole query to match, while the `opt_gap_block` *may* (but need not) match a gap in the substrate. The other is that the default retrieval of an opt_gap_block is NORETRIEVE, whereas the default retrieval of a gap_block is RETRIEVE. Otherwise, they are identical in semantics.

The retrieval will be dealt with more fully in the next section.

The inner `blocks`, if present, must match if the gap block is to match. When trying to match the inner `blocks`, both the Universe and the Substrate are set to the monads of the gap. So if the gap matches the monad-stretch $[a..b]$, then both the Universe and the Substrate for the inner `blocks` will be this stretch of monads.

The last point is important in example 5. Here the Word which we are looking for inside the gap will be looked for within the monads which made up the gap.

If you want any objects or gaps to be retrieved inside the gap (as in example 5 above, where we want to retrieve the Word), then the retrieval of the gap block must be either "retrieve" or "focus".

You can optionally specify a T_MARKS after the `gap` or `gap?` keyword. If you do, the MatchedObjects that arise because of this (opt_)gap_block will contain the same T_MARKS as you specified here. The query-writer can use it to pass information back into the application that sits on top of Emdros. Emdros does nothing special with the T_MARKS, other than passing it on into the sheaf, that is, into the matched_object that arises because of the (opt_)gap_block. In particular, there is no semantics associated with the marks_declaration. See page 26 for the formal definition of T_MARKS.

### 4.7.4 Power block

#### 4.7.4.1 Syntax

```
power : ".."   [ restrictor ]
;
restrictor : "<" limit
           | "<=" limit
```

```
            | "BETWEEN" limit "AND" limit
;
limit : T_INTEGER /* non-negative integer, may be 0. */
;
```

#### 4.7.4.2 Examples

```
1. [Word]

2. [Word psp=article]
   [Word psp=noun]
   .. <= 5
   [Word psp=verb]

3. [Phrase phrase_type = NP]
   ..
   [Phrase phrase_type = AdvP]
   .. BETWEEN 1 AND 5
   [Phrase phrase_type = VP]

4. [Chapter
     topic = "Noun classes in Bantu"
   ]
   [Chapter
     topic = "Causatives in Setswana"
   ]
   ..
   [Chapter
     topic = "Verb-forms in Sesotho"
   ]
```

#### 4.7.4.3 power

The power block means "before the start of the next block, there must come a stretch of monads of arbitrary length, which can also be no monads (0 length)". In its basic form, it is simply two dots, "`..`".

The stretch of monads is calculated from the monad after the last monad of the previous block. If the previous block ended at monad 7, then the power block starts counting monads from monad 8.

One can optionally place a `restrictor` after the two dots, thus making the power block look like this, e.g., "`.. < 5`", "`.. <= 5`", or "`.. BETWEEN 1 AND 5`".

The first two kinds of restrictor mean "although the stretch of monads is of arbitrary length, the length must be less than (or equal to) the number of monads given in the restrictor". Thus "`.. < 5`" means "from 0 to 4 monads after the end of the previous block", and "`.. <= 5`" means "from 0 to 5 monads after the end of the previous block". That is, if the previous block ended at monad 7, then "`.. < 5`" means "the next block must start within the monads 8 to 12", while "`.. <= 5`" means "the next block must start within the monads 8 to 13".

Similarly, the third kind, "`..` `BETWEEN` *`min`* `AND` *`max`*" means "there must be at least *`min`* monads in between, and at most *`max`* monads. This is construed as "`>=` *`min`* `AND <=` *`max`*".

### 4.7.5 Retrieval

#### 4.7.5.1 Introduction

Retrieval is used in four places in the MQL grammar. Once for each of the two object blocks and once for each of the two gap blocks. In this section we describe the three kinds of retrieval, specify the default behavior, and provide a bit of explanation.

#### 4.7.5.2 Syntax

```
retrieval : "NORETRIEVE"
          | "RETRIEVE"
          | "FOCUS"
;
gap_retrieval : "NORETRIEVE"
              | "RETRIEVE"
              | "FOCUS"
;
```

#### 4.7.5.3 Examples

```
1. [Word focus
     psp = verb
   ]

2. [gap? retrieve]

3. [Phrase noretrieve]

4. [gap focus]

5. [Phrase retrieve
     [Word focus
       psp = article
     ]
     [gap retrieve
       [Word focus
         psp=conjunction
       ]
     ]
     [Word focus
       psp = noun
     ]
   ]
```

#### 4.7.5.4 Explanation

Retrieval has to do with two domains pertaining to objects and gaps:

1. Whether to retrieve the objects or gaps, and
2. Whether those objects or gaps should be in *focus*.

Whether to retrieve is straightforward to understand. If we don't retrieve, then the object or gap doesn't get into the sheaf. The sheaf is the data-structure returned by an MQL query. The object or gap (if the gap is not optional) must still match for the overall match to be successful, but the object or gap won't get into the sheaf if we don't retrieve.

When an object is in focus, that means your application has the opportunity to filter this object out specifically from among all the objects retrieved. Exactly how this feature is used (or not used) will depend on your application. When is this useful?

Recall that, for objects in an inner `blocks` to be retrieved (in an object block or a gap block), the enclosing object or gap must also be retrieved. Thus you might end up with objects in the sheaf which you don't really care about. The focus-modifier is a way of signaling special interest in certain objects or gaps. Thus you can specify exactly which objects should be of special interest to the application. In example 5

above,[1] the outer Phrase must be retrieved, because we wish to retrieve the inner objects and gaps. The inner gap must also be retrieved because we wish to retrieve the inner Word. The three Words are what we are really interested in, however, so we mark their retrieval as "focus".

If we specify "focus" as the retrieval, then that implies "retrieve". Thus we can't not retrieve an object which is in "focus". This makes sense. If you have registered a special interest in an object, that means you want to retrieve it as well.

The default for object blocks of both kinds, when no retrieval is specified, is to assume "retrieve". The default for gap blocks of both kinds, on the other hand, is "noretrieve".

## 4.7.6 First and last

### 4.7.6.1 Introduction

The object blocks have the option of specifying whether they should be first and/or last in their enclosing `blocks`.

### 4.7.6.2 Syntax

```
firstlast : "FIRST"
          | "LAST"
          | "FIRST"   "AND"   "LAST"
;
```

### 4.7.6.3 Examples

```
1. [Clause first and last]

2. [Phrase first]

3. [Clause
     [Phrase first]
     [Word last
       psp = verb
     ]
   ]
```

### 4.7.6.4 Explanation

In example 1, the clause must be both first and last in its surrounding `blocks`. In the second example, the phrase must merely be the first. In the third example, the Phrase must be first in the clause, followed by a word, which must be a verb, and which must be last. This can be realized, e.g., in verb-final languages.

What does it mean to be "first" and "last" in the enclosing `blocks`?

Again we must appeal to the notion of Universe and Substrate. Each `blocks` carries with it a Universe and a Substrate. Let us say that an object block must be first, and let us say that we are trying to match an object O against this object block. Let us call the substrate of the enclosing blocks "Su". Then, for the object O to be first in the blocks means that O.first = Su.first. That is, the first monad of the object must be the same as the first monad of the Substrate.

Conversely, for an object O to be last in a blocks, means that O.last = Su.last. That is, the last monad of the object must be the same as the last monad of the Substrate.

[1] This construction actually does occur in at least one language, namely ancient Greek. It is due to post-positive particles and conjunctions such as "de", "gar", "men", and the like.

### 4.7.7 Feature constraints

#### 4.7.7.1 Introduction

Object blocks can optionally have feature constraints. The feature constraints are boolean (i.e., logical) expressions whose basic boolean building-blocks are “and”, “or”, and “not”. The things that are related logically are comparisons of features and values, i.e., a feature followed by a comparison-symbol (e.g., “=”), followed by a value. Parentheses are allowed to make groupings explicit.

In the following, we first define the syntax of feature constraints. We then make refer to other parts of this manual for details of certain non-terminals. We then give some examples, followed by explanations of those examples. We then give some explanation and elucidation on feature-constraints. We then describe the constraints on type-compatibility between the feature and the value. Finally we elaborate on comparison-operators.

#### 4.7.7.2 Syntax

```
feature_constraints  : ffeatures
;
ffeatures : fterm
          |  ffeatures   "OR"   fterm
;
fterm : ffactor
      | ffactor   "AND"   fterm
;
ffactor : "NOT"   ffactor
        | "("   ffeatures   ")"
        | feature_comparison
;
feature_comparison :
    feature_name  comparison_operator   value
  | feature_name  "IN"   enum_const_set
  | feature_name  "IN"   "("  list_of_integer  ")"
  | feature_name  "IN"   object_reference_usage
;
comparison_operator : "="
                    | "<"
                    | ">"
                    | "<>"  /* not equal */
                    | "<="  /* less than or equal */
                    | ">="  /* greater than or equal */
                    | "~"   /* regular expression */
                    | "!~"  /* inverted regular expression */
                    | "HAS" /* lhs: list; rhs: atomic value.
                                 signifies list membership. */
;
list_of_integer : T_INTEGER  { "," T_INTEGER }*
;
value : enum_const
      | signed_integer
      | T_STRING
      | object_reference_usage
;
enum_const : T_IDENTIFIER
;
object_reference_usage : object_reference
```

```
                          "."   feature_name
;
enum_const_set : "(" enum_const_list ")"
;
enum_const_list : enum_const { "," enum_const_list }
;
```

#### 4.7.7.3 References

For `signed_integer`, see section 3.5.1 on page 36. For object references, see the next section. For `feature_name`, see 3.5.2 on page 40.

#### 4.7.7.4 Examples

```
1. [Word psp = noun]

2. [Word gender = neut or gender = fem]

3. [Word psp = adjective and not case = nominative]

4. [Phrase (phrase_type = NP
            and phrase_determination = indetermined)
           or phrase_type = AP
   ]

5. [Word as w
     psp = article
   ]
   [Word
     psp = noun
     and case = w.case
     and gender = w.gender
     and number = w.number
   ]

6. [Word
     surface > "Aa" and surface ~ "[A-D]orkin"
   ]

7. [Word psp IN (verb, participle, infinitive)]

8. [Word psp = verb OR psp = participle OR psp = infinitive]
```

#### 4.7.7.5 Explanation of Examples

Example 1 above is the simple case where a feature ("psp") is being tested for equality with a value ("noun"). Example 2 is more of the same, except the gender can either be neuter or feminine, and the feature constraint would match in both cases. Example 3 finds those words which are adjectives *and* whose case is *not* nominative. Example 4 finds either adjectival phrases or NPs which are indetermined.

Example 5 is an example of usage of object references. The first Word is given the "label" (or "object reference") "w". Then the second Word's feature-constraints refer to the values of the features of the first Word, in this case making sure that case, number, and gender are the same.

Example 6 is an example of two different comparison-operators, "greater-than" and "regular expression-match".

Example 7 shows the comparison IN. It takes a comma-separated list of enumeration constant names in parentheses as its right-hand-side. The effect is the same as an OR-separated list of "=" feature-comparisons. So 7. and 8. are equilalent.

#### 4.7.7.6 Explanation

While the syntax may look daunting to the uninitiated, the system is quite straightforward. At the bottom, we have feature comparisons. These consist of a feature, followed by a comparison-operator (such as "="), followed by a value. These feature-comparisons can be joined by the three standard boolean operators "and", "or", and "not".

The precedence of the operators follows standard practice, i.e., "not" has highest precedence, followed by "and", followed by "or". Parentheses are allowed to make groupings explicit. That is, "and" "binds" more closely than "or" so that the interpretation of this expression:

f_1 = val_1 "and" f_2 = val_2 "or" f_3 = val_3

is the following:

(f_1 = val_1 "and" f_2 = val_2) "or" f_3 = val_3

Note that if you use "not" on a feature comparison, and if you have another feature comparison before it, then you must explicitly state whether the relationship between the two is "and" or "or". Thus the following is illegal:

f_1 = val_1 "not" f_2 = val_2

The following, however, would be legal:

f_1 = val_1 "and" "not" f_2 = val_2

The "in" comparison-operator can only be used with a comma-separated list of enumeration constant names on the right-hand-side. The effect is the same as if all of the enumeration constants had been compared "=" to the feature, with "OR" between them.

#### 4.7.7.7 Type-compatibility

The feature and the value with which we compare both have a *type*. The type is, one of "integer", "7-bit (ASCII) string", "8-bit string", "enumeration constant", "id_d". Thus a type tells us how to interpret a value.

The types of the two values being compared must be *compatible*. Table 4.1 summarizes the type-compatibility-constraints.

| If value's type is... | Then feature's type must be... |
|---|---|
| enumeration constant | The same enumeration as the value |
| (enumeration constant-list) | The same enumeration as all the values |
| signed_integer | integer or id_d |
| 7-bit or 8-bit string | 7-bit or 8-bit string |
| object reference usage | The same type as the feature in the object reference usage, or a list of the same type |

Table 4.1: Type-compatibility-constraints

The 8-bit strings need not be of the same encoding.

#### 4.7.7.8 Comparison-operators

Table 4.2 summarizes the comparison-operators.

| op. | meaning |
|---|---|
| = | Equality |
| < | Less-than |
| > | Greater-than |
| <> | Inequality (different from) |
| <= | Less-than-or-equal-to |
| >= | Greater-than-or-equal-to |
| ~ | Regular expression-match |
| !~ | Negated regular-expression-match |
| IN | Member of a list of enum constants |
| HAS | List on left-hand-side, atomic value on right-hand-side. Signifies list membership. |

Table 4.2: Comparison-operators

**4.7.7.8.1 Inequality** The inequality-operator "<>" is logically equivalent to "not ... = ...". The negated regular-expression-match "!~" is logically equivalent to "not ... ~ ...".

**4.7.7.8.2 Equality** Equality is defined as follows: If the type is id_d, then both must be the same id_d. If the type is integer, then both must be the same number. If the type is string, then both must be byte-for-byte identical, and of the same length. If the type is enumeration, then both must have the same numerical value. That is, the enumeration constants must be the same, since an enumeration is a one-to-one correspondence between a set of labels and a set of values. If the type is a list, the two lists must be identical, i.e., consist of the same sequence of values.

**4.7.7.8.3 Less-than/greater-than** The four less-than/greater-than-operators use 8-bit scalar values for the comparison of strings. That is, it is the numerical value of the bytes in the strings that determine the comparison. In particular, the locale is not taken into consideration. For comparison of id_ds, the id_ds are treated as ordinary numbers, with nil being lower than everything else. For comparison of integers, the usual rules apply. For comparison of enumeration constants, it is the values of the enumeration constants that are compared, as integers.

**4.7.7.8.4 Regular expressions** There are two regular expression-comparison-operators ("~" and "!~"). They operate on 8-bit strings. That is, both the feature-type and the value against which the match is made must be 8-bit strings. The negated match matches everything that does not match the regular expression provided as the value.

The value that they are matched against must be a string.

The regular expressions are the same as in Perl 5. See section 1.4.2 on page 15 for details of where regular expression-support comes from. See `http://www.perl.com/` for details of Perl regular expressions.

Before version 1.2.0.pre46, regular expressions were anchored, meaning that they always started matching at the start of the string. As of 1.2.0.pre46, regular expressions are not anchored, meaning that they can start their match anywhere in the string.

**4.7.7.8.5 IN** The IN comparison operator must have:

1. either an enumeration feature on the left hand side and a comma-separated list of enumeration constants in parentheses on the right-hand-side (or an object refence usage resolving to a list-of-enum-constants of the same type),

2. or an INTEGER feature on the left hand side, and a list of integers on the right-hand-side (or an object reference usage resolving to this),

3. or an ID_D feature on the left hand side, and a list of integers on the right-hand-side (or an object reference usage resolving to a list of ID_Ds).

For the first case, all of the enumeration constants must belong to the enumeration of the feature. The meaning is "feature must be in this list", and is equivalent to a string of "=" comparisons with "OR" in between, and with parentheses around the string of OR-separated comparisons.

For the second and third cases, the meaning is the same, but applied to integers and id_ds respectively.

**4.7.7.8.6 HAS** The HAS comparison operator must have:

1. Either a list-of-enumeration constant on the left hand side and an enumeration constant belonging to the same enumeration on the left hand side (or an object reference usage resolving to this),

2. or a list-of-INTEGER feature on the left hand side, and an atomic integer value the right-hand-side (or an object reference usage resolving to this),

3. or a list-of-ID_D feature on the left hand side, and an atomic id_d value on the right-hand-side (or an object reference usage resolving to this).

This signifies list-membership of the right-hand-side in the list on the left-hand-side.

## 4.7.8 Object references

### 4.7.8.1 Introduction

Object references are a way of referring to objects in a query outside of the object block which they matched. This provides the possibility of matching objects on the basis of the features of other objects earlier in the query.

In this subsection, we first give the syntax of object references, their declaration and their usage. We then provide some examples, followed by an explanation of those examples. We then give some explanation of object references. Finally, we document some constraints that exist on object references.

### 4.7.8.2 Syntax

```
object_reference_declaration : "AS" object_reference
;
object_reference : T_IDENTIFIER
;
object_reference_usage : object_reference
                         "."   feature_name
;
feature_name : T_IDENTIFIER
;
```

### 4.7.8.3 Examples

```
1. [Clause
     [Phrase as p
       phrase = NP
     ]
     ..
     [Phrase
       phrase = AP
       and case = p.case
```

```
        and number = p.number
        and gender = p.gender
      ]
    ]

2. [Clause as C
      [Phrase
        phrase_type = NP
        parent = C.self
      ]
    ]

3. [Sentence as S]
    ..
    [Sentence
      head = S.self
    ]
```

#### 4.7.8.4 Explanation of examples

Example 1 finds, within a clause, first an NP, followed by an arbitrary stretch of text, followed by an AP. The AP's case, number, and gender-features must be the same as the NP's case, number, and gender-features respectively.

Example 2 finds a clause, and within the clause an NP which is a direct constituent of the clause. That is, its parent feature is an id_d which points to its parent in the tree. This id_d must be the same as the clause's "self" feature. See section 2.7.6 on page 21 for more information about the "self" feature.

Example 3 finds a sentence and calls it S. Then follows an arbitrary stretch of text. Then follows another sentence whose head feature is an id_d which points to the first sentence. That is, the second sentence is dependent upon the first sentence.

#### 4.7.8.5 Explanation

The `object_reference_declaration` non-terminal is invoked from the object blocks, right after the object type name. That is, the `object_reference_declaration` must be the first item after the object type name, if it is to be there at all, as in all of the examples above. The object reference declaration says that the object that matched this object block must be called whatever the `object_reference` is (e.g., "p", "C", and "S" in the examples above). Then object blocks later in the query can refer to this object's features on the right-hand-side of feature comparisons. See section 4.7.7 on page 95 for details of feature-comparisons.

The `object_reference_usage` non-terminal shows up on the right-hand-side of feature comparisons, as in the examples above. It consists of an object reference followed by a dot followed by a feature name.

#### 4.7.8.6 Constraints on object references

The following are the constraints on object references:

- Object references must be declared before they can be used. That is, they must appear in an `object_reference_declaration` earlier in the query (i.e., further towards the top).
- The feature name on an object reference usage must be a feature of the object type of the object that had the corresponding object reference declaration.
- The feature type of the object reference usage must be the same as the feature type of the feature with which it is compared (not just compatible with).

- An object reference must only be declared once in a query. That is, no two object references must have the same name.
- A "Kleene Star" construct (see Section 4.8.4.5 on page 104) acts as an "export barrier" upwards in the tree for object reference declarations. Thus any object reference usages which are separated from the object reference declaration by a Kleene Star cannot be "seen". For example, this is not allowed:

```
[Clause
   [Phrase
      [Word as w1]
   ]* // Kleene Star acts as an export barrier!
   [Word surface=w1.surface] // So we can't see the declaration here...
]
```

- You also cannot have an object reference declaration on an object block that itself bears the Kleene Star. Thus this is not allowed:

```
[Clause
   [Phrase as p1]* // This is NOT allowed!
   [Phrase function=p1.function]
]
```

- It is not allowed to have an object reference declaration that is used "above" an OR. That is, all object reference declarations and usages should be within the same block_string2 (see Section 4.8.4 on page 103 and Section **??**). "OR" acts as an "export barrier" on object reference declarations, not allowing them to be "seen" beyond the "OR". Thus this is *not* allowed:

```
[Phrase as p1]
OR
[Phrase function=p1.function] // Oops! Can't see declaration from here!
```

Whereas this *is* allowed:

```
[Phrase as p1
   [Phrase function=p1.function] // This is OK
   OR
   [Phrase function<>p1.function] // This is also OK
]
```

The reason the second is allowed but the first is not is that it is the object reference *declaration* which is under embargo above (not below) an OR, whereas the object reference *usage* is free to see an object reference *declaration* that has been declared above an OR.

## 4.7.9 Block

### 4.7.9.1 Introduction

The non-terminal `block` is a choice between three kinds of block: `opt_gap_block`s, `gap_block`s, and `object_block`s. It is used in the grammar of MQL queries in the definition of `block_string`s, that is, in when defining strings of blocks. See section **??** on page **??** for more information on `block_string`s.

### 4.7.9.2 Syntax

```
block : opt_gap_block
      | gap_block
      | power
      | object_block
      | ("NOTEXIST" | "NOTEXISTS") object_block
;
```

## 4.8 Strings of blocks

### 4.8.1 Introduction

Having now described all the syntax and semantics of individual blocks, we now go on to giving the bigger picture of MQL queries. This section describes strings of blocks, as well as the higher-level non-terminals in the MQL query-query subset.

We first describe the `topograph`, the top-level entry-point into the MQL query-query grammar (4.8.2). We then describe the `blocks` non-terminal, which shows up inside each of the three kinds of blocks as an inner `blocks` (4.8.3). We then describe the `block_str` non-terminal, which provides for strings of blocks optionally connected by power blocks (the "`..`" blocks which have been exemplified previously, and which mean "an arbitrary stretch of space") (**??**).

### 4.8.2 topograph

#### 4.8.2.1 Introduction

The `topograph` non-terminal is the entry-point for the MQL query-query subset.[2] It simply consists of a `blocks` non-terminal. The topograph passes on a Universe and a Substrate to the `blocks` non-terminal, and these will be described below.

#### 4.8.2.2 Syntax

```
topograph : blocks
;
```

#### 4.8.2.3 Examples

```
1. [Word]

2. [Word psp=article]
   [Word psp=noun]

3. [Clause
     [Phrase phrase_type = NP]
     ..
     [Phrase phrase_type = VP]
   ]

4. [Book
     title = "Moby Dick"
     [Chapter chapter_no = 3
       [Paragraph
         [Word surface = "Ishmael"]
       ]
       ..
       [Paragraph
         [Word surface = "whaling"]
       ]
     ]
   ]
```

[2] Even though `mql_query` is really the proper entry-point for an MQL query-query, we may consider the topograph to be the top-level syntactic non-terminal in the MQL query-subset. The topograph has historical primacy, since it was defined first in Doedens' QL (see [Doedens94]). The `mql_query` non-terminal simply acts as a proxy, passing control to the topograph immediately.

#### 4.8.2.4 Explanation of examples

Example 1 simply finds all words within the topograph's Universe and Substrate.

Example 2 finds all pairs of adjacent words in which the first word is an article and the second word is a noun, within the topograph's Universe and Substrate.

Example 3 finds all clauses within which there are pairs of first an NP, followed by an arbitrary stretch of monads, then a VP. Within the topograph's Universe and Substrate, of course.

Example 4 finds a book whose title is "Moby Dick", and within the book it finds chapter 3, and within this chapter it finds a Paragraph within which there is a word whose surface is "Ishmael". Then, still within the chapter, after an arbitrary stretch of monads, it finds a Paragraph within which there is a word whose surface is "whaling".

#### 4.8.2.5 Universe and Substrate

In order to understand how the Universe and Substrate are calculated, it is necessary to refer back to the definition of the SELECT OBJECTS query. Please consult section 3.8.1.4 on page 47 for details.

### 4.8.3 blocks

#### 4.8.3.1 Introduction

The `blocks` non-terminal is central in the MQL query-query subset. It shows up in five places:

- In the `topograph`,
- Inside the `object_block` as the inner `blocks`,
- Inside the `gap_block` and the `opt_gap_block` as the inner `blocks`.

#### 4.8.3.2 Syntax

```
blocks : block_string
;
```

### 4.8.4 block_string

#### 4.8.4.1 Introduction

A "block_string" is basically either a "block_string2" or it is a block_string2 followed by the keyword "OR" followed by another block_string. Or, put another way, a block_string is a string (possibly 1 long) or block_string2's, separated by "OR".

#### 4.8.4.2 Syntax

```
block_string : block_string2
             | block_string2 "OR" block_string
;
block_string2 : block_string1
              | block_string1 block_string2
              | block_string1 "!" block_string2
;
block_string1 : block_string0
              | block_string0 "*" [monad_set]
;
block_string0 : block
              | "[" block_string "]"
;
```

#### 4.8.4.3 Examples

```
1. [Clause
      [Phrase function = Predicate]  // This...
      [Phrase function = Objc]       // ... is a block_string2
      OR
      [Phrase function = Predicate]  // And this...
      [Phrase function = Complement] // is another block_string2
   ]
2. [Sentence
      [gap [Clause function = relative] // This is a block_string2
      OR
      [Clause AS c1 function = Subject] // And this...
      ..                                // ... is also ...
      [Clause daughter = c1.self]       // ... a block_string2
   ]
```

#### 4.8.4.4 Explanation

Block_strings are recursive in that the lowest level (Block_string0) can be either a Block, or a full Block-String in [square brackets].

Notice that Kleene Star (*) binds more tightly than concatenation. Thus if you wish to use Kleene Star with more than one block, you must wrap those blocks in a [square bracket group].

Notice also that OR binds less tightly than concatenation. Thus OR works between strings of blocks.

The first example finds all clauses in which it is either the case that there exist two phrases inside the clause where the first is a predicate and right next to it is an object, or the first is a predicate and right next to it is a complement (or both might be true, in which case you'll get two straws inside the inner sheaf of the clause).

The second example finds all clauses in which it is the case that there either is a gap with a relative clause inside it, or there are two clauses (possibly separated) where the first clause is a subject in the second. (This assumes a data model where mother clauses do not include the monads of their daughter clauses).

See Section 4.7.8 on page 99 for some restrictions on object references regarding OR.

#### 4.8.4.5 The "*" construct

The "*" construct is a so-called "Kleene Star". It allows searching for object blocks or groups that are repeated. It has two forms: One with and one without a trailing set of integers (with the same syntax as a set of monads). For example:

```
SELECT ALL OBJECTS
WHERE
[Sentence
  [Phrase FIRST phrase_type = NP]
  [Phrase phrase_type IN (VP, NP, AP)]*
  [Phrase function = adjunct]* {1-3}
]
GO
```

This finds sentences whose first phrase is an NP, followed by arbitrarily many phrases which can either be VPs, NPs, or APs (or any combination of those), followed by between 1 and 3 phrases whose function is adjunct.

A less contrived example:

```
SELECT ALL OBJECTS
```

```
WHERE
[Sentence
  [Word psp=verb]
  [Word psp=article or psp=noun
        or psp=adjective or psp=conjunction]*{1-5}
]
GO
```

This finds sentences where there exists a word whose part of speech is verb, followed by 1 to 5 words whose parts of speech may be article, noun, adjective, or conjunction (or any combination of those). Presumably this would (in English) be a VP with (parts of) the object NP after the verb.

The Kleene-Star without a trailing set of integers means "from 0 to MAX_MONADS". Note, however, that there is no performance penalty involved in such a large end: The algorithm stops looking when getting getting one more fails.

If 0 is in the set of integers, then this means that the object need not be there. This means that the following:

```
SELECT ALL OBJECTS
WHERE
[Sentence
  [Word psp=verb]
  [Word psp=article]*{0-1}
  [Word psp=noun]
]
GO
```

would find sentences where there exists a verb followed by 0 or 1 articles followed by a noun. Thus the case of "verb immediately followed by noun" would also be found by this query. Thus the "*{0-1}" is equivalent to "?" in regular expressions.

The set of integers has the same syntax as monad sets. Therefore, to obtain a “no upper bound” star-construct, use {<lower-bound>-}, e.g., {10-} to mean “from 10 times to (practically) infinitely many times.”

The following restrictions apply:

- You cannot have a Kleene Star on an object block which also has the `NOTEXIST` keyword in front.
- You cannot have a Kleene Star on an object block which has the "`noretrieve`” keyword.

#### 4.8.4.6 The bang (“!”)

You can optionally place a bang (“`!`”) between any of the `blocks` in a `block_string2`. The bang indicates that there must be no gaps between the two blocks. It is an implicit rule of MQL that there is a hidden `opt_gap_block` between each pair of `blocks` in a `block_string` which are not mediated by a bang.

The reason for having the `opt_gap_block` is the following: It protects you from what you do not know. In some languages, there can be gaps in clauses because of post-positive conjunctions “sticking out” at a higher level. One would not wish to have to specify all the time that one wanted to look for gaps, for one would invariably forget it sometimes, thus not getting all the results available. Thus MQL inserts an `opt_gap_block` between each pair of blocks that are not mediated by a bang. The bang is a way of specifying that one does not wish the hidden `opt_gap_block` to be inserted.

The `opt_gap_block` that is inserted is not retrieved.

# Appendix A

# Copying

## A.1 Introduction

Emdros is covered by two different licenses, both of which allow you freely to copy, use, and modify the sourcecode. The parts which were written by Ulrik Sandborg-Petersen are covered by the MIT License. The pcre library, which provides regular expressions-capabilities, is covered by a different license. Some parts were contributed by Kirk E. Lowery, Martin Korshøj Petersen, or Claus Tøndering; they are Copyright Sandborg-Petersen Holding ApS and are also under the MIT.

SQLite is in the Public Domain. See `www.sqlite.org` for details.

All Emdros documentation (including this document) is covered under the Creative Commons Attribution-Sharealike license version 4.0. Please see the Creative Commons website for the details.

## A.2 MIT License

MIT License

```
Copyright (C) 1999-2018 Ulrik Sandborg-Petersen
Copyright (C) 2018-present Sandborg-Petersen Holding ApS
Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
"Software"), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:
The above copyright notice and this permission notice shall be
included in all copies or substantial portions of the Software.
THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE
LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION
OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION
WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
```

## A.3 PCRE license

PCRE is a library of functions to support regular expressions whose syntax and semantics are as close as possible to those of the Perl 5 language.

Release 6 of PCRE is distributed under the terms of the "BSD" licence, as specified below. The documentation for PCRE, supplied in the "doc" directory, is distributed under the same terms as the software itself.

The basic library functions are written in C and are freestanding. Also included in the distribution is a set of C++ wrapper functions.

### A.3.1 THE BASIC LIBRARY FUNCTIONS

Written by: Philip Hazel

Email local part: ph10

Email domain: cam.ac.uk

University of Cambridge Computing Service,

Cambridge, England. Phone: +44 1223 334714.



### A.3.2 THE C++ WRAPPER FUNCTIONS

Contributed by: Google Inc.



### A.3.3 The “BSD” license

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- Neither the name of the University of Cambridge nor the name of Google Inc. nor the names of their contributors may be used to endorse or promote products derived from this software without specific prior written permission.

# Appendix B

# Console sheaf grammar

## B.1 Introduction

The sheaf's contents were explained in section 4.4 on page 83. In this appendix, we give the grammar for the sheaf as it is output with console output (as opposed to XML output).

## B.2 Sheaf grammar

```
/* Sheaf */
sheaf : failed_sheaf | successful_sheaf
;
failed_sheaf : "//"  /* A failed sheaf means
                        that the query failed
                        in some way. */
;
successful_sheaf : "//" straws /* A successful sheaf
                                  means that the query
                                  did not fail. It may
                                  however, be empty,
                                  in which case the
                                  list_of_straws will
                                  not be there, and the
                                  sheaf will look like
                                  this: "// < >".
                               */
;
straws : "<" list_of_straws ">"
;
list_of_straws : { straw }
;

/* Straw */
straw : "<" list_of_matched_objects ">"
;
list_of_matched_objects : { matched_object }
;
/* Matched object */
matched_object : mo_id_d | mo_id_m
;
```

```
/* Matched object with id_d */
mo_id_d : "[" object_type_name
          id_d_of_object
          monad_set
          is_focus
          [marks]
          inner_sheaf
          "]"
;
object_type_name : T_IDENTIFIER
;
id_d_of_object : T_INTEGER
;
is_focus : "true" | "false" /* Was the block against
                               which this matched_object
                               was matched a "focus"
                               block or not? I.e., was
                               they keyword "focus"
                               present in the block?
                            */
;

marks : T_MARKS
;
inner_sheaf : sheaf
;

/* Matched object with id_m (see [Standard-MdF]) */
mo_id_d : "[" "pow_m"
          monad_set
          is_focus
          [marks]
          inner_sheaf
          "]"
;
```

## B.3 References

For the `monad_set` non-terminal, please see section 3.8.1 on page 45.

# Bibliography


[Doedens94] Doedens, Crist-Jan. 1994: *Text Databases, One Database Model and Several Retrieval Languages.* 'Language and Computers,' Volume 14. Editions Rodopi Amsterdam, Atlanta, GA. ISBN 90-5183-729-1.

[Ulrik PhD] Sandborg-Petersen, Ulrik. 2008. *Annotated text databases in the context of the Kaj Munk archive: One database model, one query language, and several applications*. PhD dissertation, Department of Communication and Psychology, Aalborg University, Denmark. Obtainable from URL: `http://ulrikp.org/`

[Standard-MdF] Petersen, Ulrik. 2002: *The Standard MdF Model*. Unpublished article. Obtainable from URL: `http://emdros.org/`

[Monad Sets] Petersen, Ulrik. 2002: *Monad Sets – Implementation and Mathematical Foundations*. Unpublished article. Obtainable from URL: `http://emdros.org/`

[COLING 2004] Petersen, Ulrik. 2004. *Emdros -- a text database engine for analyzed or annotated text*. In ICCL, Proceedings of COLING 2004, held August 23-27 in Geneva. International Commitee on Computational Linguistics, pp. 1190--1193. `http://emdros.org/`

[Relational-EMdF] Petersen, Ulrik. 2007: *Relational Implementation of EMdF and MQL.* Unpublished working-paper. Obtainable from URL: `http://emdros.org/`

[MQLQueryGuide] Petersen, Ulrik. 2007: *MQL Query Guide*. Obtainable from URL: `http://emdros.org/`

# Index