

# Emdros Database Designer's Guide

Ulrik Petersen

July 4, 2011

## Contents

### 1 Introduction

Welcome to the Emdros Database Designer's Guide. This guide will help you in designing your Emdros database. It will also help you in the task of populating your Emdros database.

The Guide has three parts:

1. **Foundations** explains some important concepts.
2. **Schema**: explains how to write a schema.
3. **Population**: explains how to populate your database.

To navigate, either use the menu on the left, or use the "prev(ious)", "up", and "next" links at the bottom. The "next" links will take you straight through the guide, whereas "up" will go up one level in the document hierarchy. "prev" will take you to the previous page.

### 2 Foundations

In this part, you will be taken through:

1. The EMdF model
2. A bird's eye view of the process of designing a database.

#### 2.1 EMdF model

We start by briefly recapitulating the most important parts of the EMdF model. Feel free to skip this if you know the EMdF model inside out already. If not, a brief refresher might be in order.

Table of contents:

1. Identifier
2. Monad
3. Object

4. Object type
5. Feature
6. Enumeration
7. Summary

### 2.1.1 Identifier

**Introduction** An important concept in Emdros is ”*identifier*”. An identifier is a sequence of characters, such as:

- Word
- Phrase
- Clause
- false
- true
- person\_1
- person\_2
- person\_3

**Motivation** Many concepts in this Guide are defined as being identifiers, e.g., object type names, feature names, enumeration names, enumeration constants, and others.

**Definition** For those who know the C language: Emdros identifiers are defined exactly as in C.

An identifier is a sequence of characters:

1. Which starts with a lower-case letter (a-z), upper-case letter (A-Z) or underscore (\_), and
2. Which, if it consists of more than one character, continues with lower-case letters (a-z), upper-case letters (A-Z), decimal digits (0-9), or underscores (\_), and nothing else.

### 2.1.2 Monad

**What is a monad?** This is one of the first things you have to grasp when designing an Emdros database:

**A monad is an integer –  
nothing more, nothing less!**

So, a monad is an integer. It is not a word, not a morpheme. It is an integer.

**Monads are consecutive** Because monads are integers, they follow one upon another in a natural succession. The most natural succession is "x plus 1", so, the sequence:

1,2,3,4,5,6,7,8,9,...

**What is the primary function of the monads?** Since monads form a sequence, and since text is sequential, the sequence of monads can be used to model the flow of the textual sequence.

So the primary function of the monads is to model the sequence of the text.

**What is the secondary function of the monads?** Objects (which we'll get to in a bit) consist of *sets of monads*. An object *is* a set of monads.

Thus the secondary function of the monads is to form the fabric out of which objects are made. An object *is* a set of monads.

**What are the minimum and maximum monads?** It depends on what you are asking.

If you are asking possible monads, then the minimum monad you can create at all is 1, while the maximum is 2,100,000,000 (2.1 billion).

If you are asking in a particular database, it depends on you. You need not start your database at monad 1: You can start it at whichever monad you like. And you can end it at whichever monad you like.

So the minimum monad in any given database is whatever you decided to use as the first monad for the first object, and similarly for the last monad of the last object.

## Summary

1. A monad is an integer.
2. Therefore, the monads form a sequence.
3. The most natural sequence is "1,2,3,4,5,6,...".
4. The primary function of the monads is to model the sequence of text.
5. The secondary function of the monads is to be the stuff out of which objects are made.
6. The minimum possible monad is 1.
7. The maximum possible monad is 2.1 billion.
8. You need not start the database at monad 1: You can choose any monad.
9. The same goes for where to end the database: It depends on the last object of the database.

### 2.1.3 Object

**What is an object?**

An object is a non-empty set of monads.

In addition:

An object **has**:  
- an *ID* (unique integer)  
and  
- zero or more other *features* (attributes).

Furthermore:

An object **belongs to** an *object type*.

It is the object type of an object which determines which features it has.

**What was an object again?** An object is a non-empty set of monads.

This means that the fabric out of which an object is made is the monads.

Since the set is non-empty, it means that you cannot have an object that is not tied to at least one monad.

**What about the set of monads?** First, it has to be non-empty. There must be at least one monad in it.

Second, it is quite arbitrary otherwise: You can have both singletons, contiguous objects, and discontinuous objects. What monads you assign to a given object depends on the meaning with respect to the textual flow and the other objects around it.

**What about the ID?** It's an integer that's unique in the database, identifying this particular object uniquely.

It's like a ROWID or a PRIMARY KEY ID in a relational database, except that it's globally unique, not just unique to the object type.

Well, the uniqueness is only guaranteed if you let Emdros assign the ID at object creation time. You can also specify it yourself when you issue the CREATE OBJECT or CREATE OBJECTS WITH OBJECT TYPE statement, but in that case, you are on your own with respect to uniqueness.

Technically, it's called an *id\_d*.

**Why is the ID called an *id\_d*?** For historical reasons, it's actually called an *id\_d* (ID in the **d**atabase). It's to distinguish it from an *id\_m*, which was something in the original MdF model on which the EMdF model is based. For full details of the MdF model, see the paper "The Standard MdF model" in the Emdros documentation.

**So is an *id\_d* a monad?** No!. An *id\_d* is separate from the monads. You could say that the monads are the stuff out of which an object is made, but the *id\_d* identifies the object uniquely.

The *id\_d* is akin to a person's "name" (say, "Fred Olington"), while the monads are his/her body.

**What about object type and feature?** We'll get to those in a moment.

**OK, OK, but I'd like an example first!** As you please:

```
// Create the sentence "Fred was a genius."
CREATE OBJECT
FROM MONADS = {1}
[Word
  surface := "Fred";
  psp := proper_noun;
]
GO

CREATE OBJECT
FROM MONADS = {2}
[Word
  surface := "was";
  psp := verb;
]
GO

CREATE OBJECT
FROM MONADS = {3}
[Word
  surface := "a";
  psp = indefinite_article;
]
GO

CREATE OBJECT
FROM MONADS = {4}
[Word
  surface := "genius.";
  psp = noun;
]
GO

// Querying. Finds "a genius" in the above.
SELECT ALL OBJECTS
WHERE
[Word psp=indefinite_article]
[Word psp=noun]
GO
```

#### 2.1.4 Object type

**What is an object type?** An object type is a named group of objects with similar characteristics.

Actually, the "similar characteristics" of the objects in the group boil down to having exactly the same *features*.

You see, what object type an object belongs to determines what features an object has.

**What about the name?** A, yes, an object type is a *named* group of objects with similar characteristics. The name of an object type identifies it in all MQL queries that deal with object types (which is most MQL queries).

The name must be an identifier.

#### 2.1.5 Feature

**What is a feature?** A feature is an attribute of an object. Thus it is a typed value which an object stores as part of itself.

The object type of an object determines which features an object has.

**You said the value was typed?** Yes. The type can be any of the following:

- STRING (8-bit string)
- ASCII (7-bit string)
- INTEGER (32-bit value)
- ID\_D (pointer to another object)
- an enumeration (set of named constants)
- LIST OF INTEGER
- LIST OF ID\_D
- LIST OF *enumeration name*

If it is a STRING, then the feature can hold arbitrarily long (even megabytes long) 8-bit strings.

If it is ASCII, then the feature can hold arbitrary long (even megabytes long) 7-bit strings. The bottom line difference between a STRING and an ASCII string is that ASCII strings are stored much more space-efficiently (i.e., raw, instead of encoded).

If it is an INTEGER, then the feature can hold signed 32-bit values.

If it is an ID\_D, then the feature can hold pointers to other objects. That is, each ID\_D feature value points to another object. The value can be "NIL", in which case it doesn't point to any object.

If it is an enumeration, then the feature can hold enumeration constants from that enumeration.

If it is a LIST OF something, then the feature can hold a list of values of the given "something" type.

**I've heard rumours of "self" being a feature?** Yes. Every object always has at least one feature, namely "self". Its type is ID\_D, and it gives the id\_d of the object in question.

This means that, in topographic queries, you can use "self" to compare with ID\_D features on other objects, using object references. For example:

```
SELECT ALL OBJECTS
WHERE
[Clause AS c1
 [Phrase
  // parent is here an id_d feature which
  // points to the parent in the syntax tree.
  parent = c1.self
 ]
]
```

**What's the minimum an object can be/have?**

- The minimum an object can *be* is a non-empty set of monads.
- The minimum an object can *have* is a unique id\_d, given by its feature "self". It can also have other features, but then it is no longer minimal.
- The minimum an object can *belong to* is one object type. But that is also the maximum it can belong to: Every object belongs to exactly one object type.

**So an id\_d is a "person's name", the monads are "his/her body", what's a feature, then?** The analogy was first introduced on this page. You might say that a feature is an attribute of the person, like height, weight, eye-color, shoe-size, preferred pet-animal, social security number, etc.

### 2.1.6 Enumeration

**What is an enumeration?** An enumeration is a named set of named integer constants.

**An enumeration is what?** Let's take an example:

```
CREATE ENUM gender_t = {
  NA = -1,
  feminine = 1,
  masculine = 2
}
```

This creates a *named set* (the name is "gender\_t") of *named integer constants* (for example, the integer constant "1" is given the name "feminine").

**OK, I get it. Any restrictions?** Yes. Both the enumeration name ("gender\_t" above) and the enumeration constant names (e.g., "NA", "masculine") must be identifiers.

Also, you can't have two constants with the same value in the same enumeration. It is a set, not only in the names, but also in the values.

### 2.1.7 Summary

- A monad is an integer.
- An object is a set of monads with an `id_d` and one or more features. It belongs to one object type.
- What object type an object belongs to determines what features it has.
- A feature is a typed value which is an attribute of an object.
- The feature types can be: `STRING`, `ASCII`, `INTEGER`, `ID_D`, an enumeration, `LIST OF INTEGER`, `LIST OF ID_D`, and `LIST OF enumeration`.
- An enumeration is a named set of named integer constants.

## 2.2 Bird's eye view of the process

### 2.2.1 What's this?

On this page, you will get a bird's eye view of the process of designing an Emdros database.

### 2.2.2 How do I design an Emdros database?

1. Create a schema. This will be in the form of a script with MQL statements.
2. Create the database from the schema. This involves either putting the MQL Schema script through the `mql(.exe)` program, or feeding it to an open Emdros database connection in a program. The latter can be done through an object of the `EmdrosEnv` class (1.2.0.preXX series), or the `CEmdrosEnv` class (1.1-series).
3. Populate the database with data.

It's that easy. And that complex, because steps (1) and (3) are not always easy. But this guide will tell you the nuts and bolts of how to do it.

## 3 Schema

### 3.1 What is an EMdF schema?

An **EMdF schema** is the collection of:

- **enumerations** and
- **object types**

that make up the types in an EMdF database.

## 3.2 What is an EMdF schema script?

An **EMdF schema script** is an MQL script that creates an EMdF schema.

## 3.3 Is this a data model?

Yes, an EMdF schema defines the data model of a database. It is similar to the DTD or XML schema of an XML file, although the syntax and semantics are, of course, very different.

## 3.4 Is this a database model?

As I understand and use the terms, a database model is the nuts and bolts that make up the building blocks out of which you can create your data model. The data model is how you apply the database model to modeling an application domain.

So the EMdF model is the database model that supports your creating your own data model which suits your specific application domain.

For more information on what a database model is, I suggest you take a look at E.F. Codd's (1980) classic paper, "Data Models in Database Management" from the International Conference on Management of Data, Proceedings of the 1980 workshop on Data Abstraction, databases, and conceptual models, Pingree Park, Colorado, USA, pp. 112-114. It can be downloaded for free from the ACM Portal. Don't be confused by the fact that Codd calls it a "data model": The terms have been used and abused and changed over the years.

## 3.5 Overview

1. HOWTO
2. Object types
3. Features
4. Enumerations
5. Summary

## 3.6 HOWTO

### 3.6.1 What do I need to decide?

In order to create an EMdF schema, you have to decide:

- What the name of the database should be.
- Which object types to create.
- Which features to create.
- Which enumerations to create.

### 3.6.2 How do I write the schema file?

First, you write the following commands:

```
ïPRE class="code"ï CREATE DATABASE your-database-name-here GO  
USE DATABASE same-database-name-here GO ï/PREï!- widthincm : 12
```

-ï

This sets up the database and connects to it.

Then, you write MQL statements to create:

1. the **enumerations** first,
2. then the **object types**.

This is because the enumerations need to be defined before they can be used for feature types in the object types.

## 3.7 Object types

### 3.7.1 What are good examples of object types?

Any of the following:

- Word
- Morpheme
- Phoneme
- Grapheme
- Phrase
- Clause
- Sentence
- Paragraph
- Episode
- Speaker\_turn
- Book
- Chapter
- Verse
- Page
- Line

In short, anything that arises as a set of textual units of some sort with similar characteristics is a good candidate for being grouped into an object type.

This is because the units, if they have sufficiently similar characteristics to be a group, will benefit from having specific characteristics that are shared by all the similar units (i.e., features).

### 3.7.2 Can object types form hierarchies?

Yes and no.

Yes because you, as the database designer, may be able to determine that objects of a certain object types will always be built from objects of a lower object type.

No because Emdros does not provide a way of making these "buildable\_from" relationships explicit. They must be implicit in the monads of the objects themselves.

But yes, you can, as the Emdros application designed, choose to always create objects in such a way that the object types form a hierarchy.

### 3.7.3 Can you have multiple hierarchies?

Yes, in the same sense as you can have single hierarchies: Implicitly in the monads of the objects. The multiple hierarchies can represent different perspectives on the same database.

For example, there may be one hierarchy, "Book  $\wr$  Chapter  $\wr$  Paragraph  $\wr$  Sentence  $\wr$  Word", and another hierarchy, "Book  $\wr$  Page  $\wr$  Line". It is easy to see that objects of the "Line" object type need not be embedded inside any "Sentence" objects. In fact, they will often overlap.

### 3.7.4 How do I decide which object types I should have?

That is a design question which cannot be answered in general, since it depends on the use to which you hope to put the database. But here are some ideas:

- Find out what the textual units are.
- Can you group the textual units into groups with similar characteristics? If so, make object types out of the groups.
- Try to identify unit hierarchies. They will often help you think clearly about the units and their groupings.
- Decide on a "lowest object type" that is at the bottom of most of the hierarchies. This is often "Word", but can also be "Grapheme", "Morpheme", "Phoneme", or other units.
- It is, however, not a *necessity* to have a single lowest unit, and in fact you can have any number of object types that make out the "lowest" levels of the hierarchies.

### 3.7.5 Help! I need meta-data!

Ah, a common problem. You need to be able to store data about the text, or data that is not textual, or data that is separate from the text. How do you do that?

There are two cases:

1. **Data about the text:** Create an object type, say "Text" that holds the data about the text. Give this object type the features you need to express the data about the text. Then create large objects that span the monads of each text. Then assign the data to the features of these objects.

2. **Non-textual data:** Well, if your data is truly non-textual, then just create object types to hold the kinds of data you need to store, then create objects with arbitrary monads that you make up.

### 3.7.6 What are some examples of non-textual meta-data?

For example, say need to store a sequence of meta-data about hierarchy levels in your linguistic application. You can then create an object type called "HierarchyLevel" with the features you need for each hierarchy level, then create objects anywhere in the monad stream that describe the hierarchy levels in your application. You can put them at the monads 1, 2, 3, etc., or you can put them anywhere else in the monad stream. So long as you will never do a topographic query for both "HierarchyLevel" and real textual data at the same time, the monads can be shared.

Or say you need to store a set of labels. You create two object types: "LabelSet" and "Label", each with a string feature telling use what the name of the label set or label is.

```
// Create objects with any monadset, e.g., {1}
// The monad sets must, however, be unique and non-overlapping
// for each distinct label set.
CREATE OBJECT TYPE
[LabelSet
  set_name : STRING;
]
GO

// Objects of this type will have the same monad set as the
// corresponding LabelSet object.
CREATE OBJECT TYPE
[Label
  label_name : STRING;
]
GO
```

Then you create one LabelSet object at any monad (say, 1), and then create the Label objects at the same monad. In that way, you can get all label sets like this:

```
SELECT ALL OBJECTS
WHERE
[LabelSet
  [Label]
]
GO
```

## 3.8 Features

### 3.8.1 What is a feature again?

A feature is a typed value which is an attribute of an object. The object type of an object determines which features it has.

See here for more information.

### 3.8.2 How do I store the text?

Well, the text arises because objects exist in the monad stream which carry the textual information.

That may seem a bit cryptic, so let's try an example.

```
CREATE OBJECT TYPE
[Word
  surface : STRING;
]
GO
```

Here we have created an object type "Word" with a feature called "surface". The idea is that the objects of type Word will make up the text collectively, but filling the monad stream with word-sized chunks of the text.

Note that we need not have "Word" as the text-bearing unit: It can also be "Morpheme" or "Grapheme", or even "Sentence" if we don't care about individual words.

### 3.8.3 How do I store parts of speech?

The easiest is to create an enumeration with the parts of speech, then have this enumeration type as the type of a feature on the Word object type:

```
CREATE ENUMERATION pos\_t = {
  NA = -1,
  noun = 1,
  verb,
  adjective,
  adverb,
  preposition,
  conjunction,
  interjection,
  negative,
  article
}
GO
```

```
CREATE OBJECT TYPE
[Word
  surface : STRING;
```

```

    pos : pos_t;
    lemma : STRING;
]
GO

```

### 3.8.4 How do I store complex grammatical tags?

Well, there are two ways:

1. Either you use a single STRING or ASCII feature, or
2. You split up the grammatical tag into its constituent parts (for example, part of speech, person, number, gender, discourse type, etc. etc.) and store those as individual enumerations or strings.

The latter makes the grammatical "tag" searchable in its individual parts through the topographic search-features of MQL.

The former is more compact, is less prone to error, and is also searchable through regular expressions.

### 3.8.5 How do I store a phrase type?

Much in the same way as you store a part of speech (see above).

### 3.8.6 How do I point to other objects?

You use the ID\_D type and then let that feature point to the other object through that object's "self" feature.

That is, say you want object A to point to object B. Object A must have a feature of type ID\_D, say, "parent". Then, when creating object A, set object A.parent to B.self.

### 3.8.7 How do I create a syntax tree?

You can either express it top-down (with a LIST OF ID\_D pointing to the children) or bottom-up (with ID\_D pointers pointing to the parents).

### 3.8.8 Bottom-up with pointers to parents

If you want to do the tree bottom-up, and it really is a tree (not a Directed Acyclic Graph), then every node will have one or zero parents.

This can be elegantly modeled in EMdF with an id\_d feature called "parent" (or similar) on every node that may have a parent. Then that feature just has the id\_d of the parent object. Or, if there is no parent for this particular node, the feature can be the special value NIL.

```

CREATE OBJECT TYPE
[Word
    surface : STRING;
    parent : ID_D; // Points to phrase or clause

```

```

]
GO

CREATE OBJECT TYPE
[Phrase
    phrase_type : phrase_type_t; // A suitably defined enumeration
    parent : ID_D; // Points to phrase or clause
]
GO

CREATE OBJECT TYPE
[Clause
    parent : ID_D; // Points to clause or sentence (or phrase, in some languages)
]
GO

CREATE OBJECT TYPE
[Sentence
    parent : ID_D; // Points to paragraph...
]

CREATE OBJECT TYPE
[Paragraph] // The buck stops here.
GO

CREATE OBJECT TYPE
[Word
    surface : STRING;
    parent : ID_D; // Points to phrase or clause
]
GO

CREATE OBJECT TYPE
[Phrase
    phrase_type : phrase_type_t; // A suitably defined enumeration
    parent : ID_D; // Points to phrase or clause
]
GO

CREATE OBJECT TYPE
[Clause
    parent : ID_D; // Points to clause or sentence (or phrase, in some languages)
]
GO

CREATE OBJECT TYPE
[Sentence
    parent : ID_D; // Points to paragraph...
]

```

```

]

CREATE OBJECT TYPE
[Paragraph] // The buck stops here.
GO

```

Of course, if you have a Directed Acyclic Graph rather than a tree, then there may be zero *or more* parents. Then just substitute "LIST OF ID\_D" for "ID\_D" in the above, and call all the "parent" features "parents" instead.

### 3.8.9 Searching trees with parents

The idiom for searching a tree such as the above is as follows:

```

SELECT ALL OBJECTS
WHERE
[Clause AS c1
  [Phrase
    // This ensures that this Phrase is a direct descendant
    // of the clause
    parent = c1.self
  ]
]
GO

```

### 3.8.10 Top-down with pointers to children

You can also do it with children instead of parents:

```

CREATE OBJECT TYPE
[Word
  surface : STRING;
  // No parent or children feature: This is a leaf!
]
GO

CREATE OBJECT TYPE
[Phrase
  phrase_type : phrase_type_t; // A suitably defined enumeration
  children : LIST OF ID_D; // Points to word or phrase (or clause)
]
GO

CREATE OBJECT TYPE
[Clause
  children : LIST OF ID_D; // Points to clause or phrase
]

```

```

GO

CREATE OBJECT TYPE
[Sentence
  children : LIST OF ID_D; // Points to clause or sentence
]

CREATE OBJECT TYPE
[Paragraph
  children : LIST OF ID_D; // Points to sentence
]
GO

```

### 3.8.11 Searching trees with children

The idiom for searching trees with children is as follows:

```

SELECT ALL OBJECTS
WHERE
[Clause AS c1
  [Phrase
    // This ensures that this Phrase is a direct descendant
    // of the clause
    self IN c1.children
  ]
]
GO

```

## 3.9 Enumerations

### 3.9.1 What is an enumeration?

See here.

### 3.9.2 What are enumerations good for?

Enumerations are good for collecting named values in groups. If you have a set of values that keep recurring, consider making them into an enum.

You will have to deal with the restriction that constant names have to be identifiers, but if that is doable, then enumerations can be very useful.

### 3.9.3 How do I create an enumeration?

An example:

```

CREATE ENUMERATION boolean = {
  false = 0
  true,
}

```

```
}  
GO
```

#### 3.9.4 Can I assign specific values to enumeration constants?

Yes. As in the example above, where "false" gets the value "0".

#### 3.9.5 Can I let Emdros assign values?

Yes. Just don't assign a value. It will be given the value of the previous constant, plus 1. The first constant, if given no value, will be 0.

#### 3.9.6 How do I name enumerations?

You use identifiers.

If your enumeration has the same name as a feature, you can put a "\_t" suffix on the enumeration to distinguish it from the feature:

```
CREATE ENUMERATION phrase_type_t = {  
  NP = 1,  
  VP,  
  AP,  
  AdvP,  
  PP  
}  
GO
```

```
CREATE OBJECT TYPE  
[Phrase  
  phrase_type : phrase_type_t;  
]  
GO
```

The suffix can be anything, but "\_t" is a fairly common ending in programming languages, and can mean "\_type". "\_e" for "enumeration" would also do. Again: It doesn't matter that much.

### 3.10 Summary

- An EMdF schema is made up of enumerations and object types.
- You write the EMdF schema of a database in a schema script. This is a sequence of MQL statements that create the enumerations and object types.
- Start the script by issuing CREATE DATABASE and USE DATABASE statements.

- Decide which object types and enumerations should be present, and which features the object types should have. Then write MQL statements to create those in the EMdF schema script.
- Object types can form hierarchies, and there can be multiple, overlapping hierarchies. However, the hierarcies are only implicit in the monds of the objects, not explicit.
- Meta-data can be represented orthogonally to the rest of the data, by making object types that represent the meta-data and then creating objects anywhere in the monad stream that contain the data.
- The text is stored as features on objects. Thus the objects give rise to the text.
- Enumerations are good for grouping named values.

### 3.11 Example

#### 3.11.1 Can you give an example?

Sure.

```
// Create the database
CREATE DATABASE linguistic_database1 GO

// Connect to the database
USE DATABASE linguistic_database1 GO

CREATE ENUMERATION part_of_speech_t = {
    noun = 1,
    verb,
    adjective,
    adverb,
    preposition,
    conjunction,
    negative,
    personal_pronoun,
    demonstrative_pronoun,
    interrogative_pronoun,
    interrogative,
    interjection,
    article
}
GO

CREATE ENUMERATION person_t = {
    // we need a "Not Applicable" value for
    // those words that do not have person.
    pers_NA = -1,
```

```

    // we cannot use "first", since that is a reserved keyword
    pers_first = 1,
    pers_second = 2,
    pers_third = 3
}
GO

```

```

CREATE ENUMERATION number_t = {
    NA = -1,
    singular = 1,
    dual = 2,
    plural = 3
}
GO

```

```

CREATE ENUMERATION gender_t = {
    NA = -1,
    masculine = 1,
    feminine
}
GO

```

```

CREATE ENUMERATION phrase_type_t = {
    NP = 1,
    VP = 2,
    AP = 3,
    AdvP = 4,
    PP = 5
}
GO

```

```

CREATE ENUMERATION phrase_function_t = {
    Predicate,
    Subject,
    Object, // Cannot use "Object", since that is a reserved word.
    IndirectObject,
    Complement,
    Adjunct
}
GO

```

```

CREATE OBJECT TYPE
[Word
    surface : STRING; // 8-bit string
    lemma : STRING; // 8-bit string
    pos : part_of_speech_t;

```

```

        person : person_t;
        number : number_t;
        gender : gender_t;
        parent : id_d; // points to a phrase or a clause
    ]
GO

CREATE OBJECT TYPE
[Phrase
    phrase_type : phrase_type_t;
    function : phrase_function_t;
    parent : id_d; // points to a phrase or a clause
]
GO

CREATE OBJECT TYPE
[Clause
    parent : id_d; // points to a clause or a sentence
]
GO

CREATE OBJECT TYPE
[Sentence
    parent : id_d; // points to a paragraph
]
GO

CREATE OBJECT TYPE
[Paragraph]
GO

```

## 4 Population

This part tells you how to populate your database once you have created the schema.

### 4.1 Overview

1. Strategies
2. Creating objects
3. Monad assignment

### 4.2 Strategies

There are basically two strategies for populating an Emdros database:

1. Import it from existing data, or

2. Create it on-the-fly while analysis or processing takes place.

#### 4.2.1 Import

**What's this?** Say you have some linguistic data – for example, a corpus made by others, or some data you've collected. Let's call the representation you have already, the *external representation*.

You want to import it into an Emdros database for easy searching and/or update.

**How do I import linguistic data into Emdros?** Well, there are several strategies, but they all assume you've got a database schema figured out.

This is because you need a clear way of mapping the data to your EMdF data model. In other words, you need to know exactly how each part of your data maps into the EMdF database model.

Once you've figured that out, here are some variations over the theme of importing data. They all assume that the database has been created using the EMdF database schema script which you should have created.

1. Read your data into memory, processing on the fly, creating MQL statements which you store in a file for later import using the `mql(.exe)` program. A scripting language such as Python, Ruby, Perl, AWK, or a number of others are good for this purpose.
2. Same as (1), but with the MQL statements piped directly to the `mql(.exe)` program without storing in a file.
3. Read your data into memory, and use either the Emdros C++ libraries to issue the MQL statements directly to a live database connection, thus feeding the data into the database on-the-fly through the Emdros libraries.
4. Same as (3), but using one of the SWIG bindings (at the time of writing, Java, Perl, Python, and Ruby are supported).

**How do I read my data into memory?** Well, you probably need to create some kind of in-memory representation of the data, as an intermediate between the external representation (say, in XML) and the MQL statements which you will emit.

Object-oriented programming languages are easy to use here, because you can wrap each EMdF object in a language-specific object and give it methods to read from the external representation, to emit MQL, etc.

Often, however, you can probably make do with something like Python's dictionary, or Perl's or AWK's associate array.

#### 4.2.2 Create on-the-fly

**What's this?** Say you have some application that does analysis or annotation of text. It could be a parser, a morphological tagger, or similar. The program could do the analysis on its own, or via human input.

Now you want to store those analyses in an Emdros database for easy searching.

**How do I store my analyses?** Well, let us assume you have some in-memory representation of your analyses.

1. You first need to write a schema that fits your data model.
2. Then you need to map your data model to your schema.
3. Then you need to write code that issues MQL statements that create the objects in the database from your in-memory representation.

**How do I issue MQL statements?** You have several strategies at your option:

1. Given that this is an on-the-fly creation, the easiest is probably to have a live database connection into which you can pump the MQL on-the-fly. Use either the C++ libraries or one of the SWIG bindings.
2. You can also write the MQL to a file and then pump that file through the `mql(.exe)` program.
3. Finally, you can write the MQL to a pipe that is connected to `stdin` on a process running the `mql(.exe)` program.

**How do I create the objects?** See this section:

- Creating objects

### 4.3 Creating objects

You create objects by means of MQL statements.

#### 4.3.1 OK. Which MQL statements do I use?

There are basically three you can use:

- `CREATE OBJECTS WITH OBJECT TYPE`
- `CREATE OBJECT FROM MONADS`
- `CREATE OBJECT FROM ID_DS`

You should look them up in the MQL User's Guide for details.

#### 4.3.2 What are the pros and cons of each?

- Well, `CREATE OBJECTS WITH OBJECT TYPE` is for batch-loading, and is much faster if you create many objects than creating each object individually with a `CREATE OBJECT` statement. You should probably think about using this statement rather than one of the other two if at all possible. The downside is that you won't get the `ID_DS` from the newly created objects back. If you need them, for example, for referencing from other objects, you should probably use one of the `CREATE OBJECT` statements, which do return the `ID_D` of the newly created object.

- **CREATE OBJECT FROM MONADS** creates a single object, assigning the monads directly. Use this if you wish to create one object and know its monads. It is rather slow, but it returns the ID\_D of the newly created object.
- **CREATE OBJECT FROM ID\_DS** creates a single object, assigning the monads from the monads of already-existing objects with specific ID\_Ds that you supply. It is rather slow, but it returns the ID\_D of the newly created object.

### 4.3.3 Can you give me some examples?

Sure

```
// Create a clause object spanning the
// first four monads of the database.
CREATE OBJECT
FROM MONADS = {1-4}
[Clause]
GO
```

```
// Create the words of the clause "The door was blue."
CREATE OBJECTS
WITH OBJECT TYPE Word
CREATE OBJECT
FROM MONADS = {1}
[Word
  surface := "The";
]
CREATE OBJECT
FROM MONADS = {2}
[Word
  surface := "door";
]
CREATE OBJECT
FROM MONADS = {3}
[Word
  surface := "was";
]
CREATE OBJECT
FROM MONADS = {4}
[Word
  surface := "blue.";
]
GO
```

#### 4.3.4 How do I make a syntax tree?

See this page for how to do it in the schema. When you've read that, come back here.

Let's assume you have some in-memory representation of the tree which you can traverse both top-down and bottom-up.

Create the objects top-down (you will see later that you should probably assign monads bottom-up, but here we are concerned with how to create the tree objects). That is, create the highest nodes (for example, S nodes) first. Use CREATE OBJECT FROM MONADS so that you will get the object ID\_D of the object just created as a return value from MQL.

Then recurse down the tree, creating each lower unit pointing to its parent. You will know the ID\_D of the parent because you have just used CREATE OBJECT FROM MONADS to create the parent, which returns the ID\_D of the object created.

Continue until you hit the bottom of the tree (words or morphemes).

Voila! A syntax tree with parent pointers in each node.

### 4.4 Monad assignment

This chapter tells you how to assign monads to objects.

#### 4.4.1 Overview

1. You make them up!
2. Monad granularity
3. Let the objects be adjacent
4. Let the objects be embedded
5. Summary
6. Example

#### 4.4.2 You make them up!

**OK, so how do I assign monads?** The most **important principle** is:

You  
make them up  
yourself  
in a way  
that makes sense!

**That's not very helpful.** Whoa, there! The statement is deeper than you may think:

1. **You ... yourself:** YOU are the sole arbiter of how the monad stream is sliced up. Noone else can do it for you.

2. **make them up:** You literally have to invent a scheme (note: not schema) for how to assign the monads. This scheme should be generative or constructive in the sense that the assignment should follow from some textual principles that apply to your application. We will give examples of such principles later. For now, remember that you have to make the monads up.
3. **in a way that makes sense:** As we just said, the invention of the monads should follow some principles, and those principles should make sense in your application.

Don't worry. We'll get to some useful principles shortly.

#### 4.4.3 Monad granularity

**What on earth is monad granularity?!?** It's one of those useful principles for structuring the monad stream.

Monad granularity refers to:

The object type(s)  
that is lowest  
in all hierarchies  
of object types,  
and which therefore  
should get assigned  
the smallest (contiguous) monad sets.

**That's a long definition. Can you break it down?** Sure.

1. **The object type:** The monad granularity is defined in terms of one or more object types.
2. **that is lowest in all hierarchies of object types:** All object types will likely have a placement in a hierarchy of object types, where objects of each type are usually contained in each other in hierarchical fashion. This need not be strict hierarchies: They can be recursive. But usually, there will be some least textual element which is at the base of all hierarchies. Usually, this will be Word, Morpheme, or Grapheme (or letter). In other words, this is the stuff out of which the text is made at the lowest level. What the lowest level is depends on your application. If you are not interested in individual words, but in, say, speaker turns only, then speaker turns should be the lowest level.
3. **and which therefore should be assigned the smallest (contiguous) monad sets:** It stands to reason that if an object type is lowest in a containment hierarchy, then its units will also be the smallest textually. Therefore, they should have the smallest sets of monads.

However, notice what the definition does *not* say.

It does *not* say that these "smallest sets" should be made up of single monads (i.e., be singletons). They can be any size – 1,2,50,100 monads long.

They should, however, ideally be contiguous. This is because the topographic query language depends on the contiguity of the monad stream.

**Can you give another definition?** Here is a simpler one. Monad granularity refers to:

The object type(s)  
that make up the lowest element(s)  
out of which the text is made,  
and which therefore  
should be assigned  
the smallest (contiguous) monad sets.

**Size of least monad sets** We just said that the least monad sets (i.e., of the objects of the monad granularity) need not be singletons. So you may be wondering...

**What's this about non-singleton monad sets?** Well, objects of the monad granularity object type should have the smallest monad sets. But even though *the* smallest monad set is a singleton (e.g., 1, 2, 3), you need not use singletons for your monad granularity objects.

You could assign contiguous segments of monads, e.g., 1-100, 101-200, 201-300.

**How can this help?** An advantage of this is that you can then later **split the objects** without moving monads around. You would split them by first deleting the object, then recreating new objects in the same monad range as the original object, but split between the new objects.

For example, say you have an object A at monads 201-300. You want to split it into two objects, B and C, which occupy the same monads, but which share the monads of the original A.

You then drop A, and in its place create objects B and C, where B has monad set 201-250, and C has monad set 251-300. If you later need to split object C, you have ample room to do so.

A general rule of thumb is that if you ever need to split an object up into  $m$  distinct objects  $n$  times, then you need to make sure that the monad set is larger than  $m$  to the power of  $n$  ( $m^n$ ). Put another way,  $n$  should be less than the logarithm of the size of the monad set to base  $m$ .

**More than one least object type** We also said that there may be more than one "least" object type.

**Can you give an example?** Say it is important to you to distinguish between words and punctuation. You might want to have two different object types, one called "**Word**" and one called "**Punctuation**". The Word objects and the Punctuation objects would be adjacent, but they would be distinct object types.

In this way, you can search like this, e.g.:

```

SELECT ALL OBJECTS WHERE
[Word
  lexeme = "blue"
]
[Punctuation
  punct_type = period
]
GO

```

Alternatively, you can let the Word object and its associated Punctuation object occupy the same stretch of monads. E.g.:

```

SELECT ALL OBJECTS WHERE
[Word
  lexeme = "blue"
  [Punctuation
    punct_type = period and punct_location = after
  ]
]
GO

```

Incidentally, the "punct\_type" and "punct\_location" example features are here assumed to be defined as enumerations.

#### 4.4.4 Let the objects be adjacent

There is another very important principle:

Let objects  
that are textually adjacent  
be adjacent in  
their monad sets.

**What does that mean?** Well, two monad sets, A and B, are adjacent (in that order) if and only if  $A.last + 1 = B.first$ . In other words, if B begins one monad later than the last monad of A.

You should strive to make up the monad assignments in such a way that objects that are actually adjacent in the text are also adjacent in the monads.

**Examples?** Words are usually adjacent. Sentences are usually adjacent. Paragraphs are usually adjacent.

**Why should I make them adjacent?** Because the topographic part of MQL relies on it.

One of the great principles of the MQL query engine is that **objects that are adjacent in the database will be found by adjacent [object blocks]**

**in the query.** An "object block" is what is between [square brackets] in the queries. For example:

```
SELECT ALL OBJECTS
WHERE
[Word]
[Word]
GO
```

Since these two object blocks are adjacent in the query, this query will only work if the words to be found are actually adjacent in the database.

#### 4.4.5 Let the objects be embedded

The third great principle of monad assignment is:

Assign the monads  
to higher-level objects  
in such a way  
that they contain  
their constituents.

**What does that mean?** It simply means that if object D is a higher-level object (such as a phrase or a clause) that has a number of constituent objects A, B, and C (such as words or phrases), then object D's monad set should be the union of the monad sets of A, B, and C.

In other words, a higher-level object should completely subsume or contain the monad sets of its constituents.

**That sounds obvious!** I'm glad. But it needed to be stated explicitly, because it is an important principle.

**Why is it important?** Because the topographic query engine depends on it, that's why.

The second great principle of the MQL query engine is that **objects that are embedded in each other should be found by means of object blocks that are embedded in each other.**

For example:

```
SELECT ALL OBJECTS
[Clause
  [Phrase]
]
```

If this query is to work, then the phrases that are constituents of clauses must have their monad sets defined in such a way that the phrase-monad sets are subsets of their parent clauses.

#### 4.4.6 Summary

1. YOU make up the monads. This should be done in a way that makes sense in your application.
2. **First principle:** There should be a **monad granularity**.
  - The monad granularity is the least object(s) that make up the stuff of the text.
  - This can be Graphemes, Morphemes, Words, Sentences, or whatever is the smallest unit in your text, seen from the viewpoint of your application.
  - These smallest units should have the smallest monad sets.
  - The smallest monad sets need not be singletons, they can be arbitrarily large segments. This is helpful if you ever wish to split an object without moving monads around.
  - There can be more than one "least" object type.
3. **Second principle:** Let textually **adjacent** objects be adjacent in their monads.
  - In other words, objects that are adjacent in the text should be neighbors in their monads.
  - If an object B starts at monad B.first, then object A must end at monad B.first-1 if A and B are to be adjacent (in that order).
4. **Third principle:** Let textually **embedded** objects be embedded in their monads.
  - This means that higher objects should be the union set of their constituent objects.

#### 4.4.7 Example

Suppose you are constructing a linguistic database with a syntax tree of Words, Phrases, and Clauses, as in this example.

How do you assign monads?

You probably have some in-memory representation of the syntax tree. This should be traversable both up and down the tree. Let us also assume that you have a linked list of the words.

1. Start by assigning monads to the words. We will assume that a word occupies a single monad. Go through the linked list of words, starting at the first word of the text (assigning the monad set 1), then going further to the next word, assigning monad set 2, etc. until you reach the end of the text.
2. You now need to assign monads to the syntax-level units. Start at the top-level nodes (paragraphs). Go depth-first into the tree until you reach the word-level.

3. At the level just above words (phrase-level), assign monads by using the Emdros set of monads class and using its union operator to union the sets of all the constituent words together.
4. Go upwards in the tree, at each level using the union operator on the Emdros set of monads to union together the sets of the constituents. Then assign that monad set to that particular object, then move one step further up and repeat the process.
5. Once you hit the top, you will have assigned all objects their monads.