DMLex: a data model for lexicography (draft)

OASIS LEXIDMA

1 Introduction

DMLex is a data model for modelling dictionaries (here called lexicographic resources) in computer applications such as dictionary writing systems.

DMLex is a data model, not an encoding format. DMLex is abstract, independent of any markup language or formalism. At the same time, DMLex has been designed to be easily and straightforwardly implementable in XML, JSON, as a relational database, and as a Semantic Web triplestore.

1.1 Modular structure of DMLex

The DMLex specification is divided into a core with several optional modules.

- **DMLex Core** allows you to model the basic entries-and-sense structure of a monolingual lexicographic resource.

- **DMLex Bilingual Module** and **DMLex Multilingual Module** extend DMLex Core to model either bilingual or multilingual lexicographic resources. These two modules are mutually exclusive: you can always only implement one of them.

- **DMLex Linking Module** extends DMLex Core and allows you to model various kinds of relations between entries, senses and other objects, including semantic relations such as synonymy and antonymy and presentational relations such as subentries and subsenses, both within a single lexicographic resource and across multiple lexicographic resources.

- **DMLex Inline Markup Module** extends DMLex Core to allow the modelling of inline markup on various objects such as example sentences, including the modelling of collocations and corpus patterns.
1.2 Schema formalism

DMLex models a lexicographic resource as a **hierarchical list of objects**. Each object has a name, a value and an optional list of child objects, each of which can in turn also have a **name**, a **value** and an optional list of child objects.

The data model is defined in this standard through the means of a formalism which defines, for each object: (1) what its name is, (2) what its value is supposed to be (from a list of predefined primitive types) and (3) which child objects it may contain, with what arities.

The arities of child objects are indicated with the following codes:

- \((0..1)\) zero or one
- \((0..n)\) zero or one or more
- \((1..1)\) exactly one
- \((1..n)\) one or more
- \((2..n)\) two or more

The primitive types of the values of objects are given with the following codes:

- `<string>` a non-empty string
- `<stringOrEmpty>` a string which may be empty
- `<number>` a positive integer number
- `<id>` an alphanumeric identifier
- `<idref>` a reference to something through its alphanumeric identifier
- `<uri>` a URI
- `<langCode>` an IETF language code
- `<empty>` nothing: the object serves only as a container for child objects
- `<symbol>` one of a specified finite number of values

When the primitive type of a child object is absent, this means that the schema for objects of that name is defined elsewhere in the code.

1.3 Implementing DMLex

DMLex is an abstract data model which can be implemented in many different programming environments and serialization languages. In this document, we give recommended implementations in XML, JSON and SQL. Examples of what such implementations look like with real-world data are given in the Appendix.
1.3.1 Implementing DMLex in XML

The XML implementation of DMLex shown in this document follows these principles:

- The top-level `lexicographicResource` object is implemented as an XML element.
- All other objects are implemented as XML attributes of their parents, unless:
  - the object has an arity other than (0..1) and (1..1)
  - or the object can have child objects
  - or the object’s value is human-readable text, such as a headword or a definition.

In such cases the object is implemented as a child XML element of its parent.

1.3.2 Implementing DMLex in JSON

The XML implementation of DMLex shown in this document follows these principles:

- The top-level `lexicographicResource` object is implemented as a JSON object: `{...}`.
- All other objects are implemented as JSON name-value pairs inside their parent JSON object: `{"name": ...}`.
- The values of objects are implemented:
  - If the object has an arity of (0..1) or (1..1):
    * If the object cannot have any child objects: as a string or number.
    * If the object can have child objects: as a JSON object.
  - If the object has any other arity:
    * If the object cannot have any child objects: as an array of strings or numbers.
    * If the object can have child objects: as an array of JSON objects.

1.3.3 Implementing DMLex as a relational database

The SQL implementation of DMLex shown in this document follows these principles:

- The `lexicographicResource` object is implemented as table. (Alternatively, it can be left unimplemented if the database is going to contain only one lexicographic resource.)
- Other objects with an arity other than (0..1) and (1..1) are implemented as tables.
The values of objects, and objects with an arity of (0..1) or (1..1) are implemented as columns in those tables.

The parent-child relation is implemented as a one-to-many relation between tables.

2 DMLex Core

The DMLex Core provides data types for modelling monolingual dictionaries (called lexicographic resources in DMLex) where headwords, definitions and examples are all in one and the same language. DMLex Core gives you the tools you need to model simple dictionary entries which consist of headwords, part-of-speech labels, senses, definitions and so on.

2.1 lexicographicResource

Represents a dictionary. A lexicographic resource is a dataset which can be used, viewed and read by humans as a dictionary and – simultaneously – ingested, processed and understood by software agents as a machine-readable database. Note that the correct name of this data type in DMLex is lexicographic, not lexical, resource.

```xml
<lexicographicResource id="..." uri="..." language="...">
  <title>...</title>
  <entry.../>
  <tag.../>
</lexicographicResource>

```JSON

```json
{
  "id": "...",
  "title": "...",
  "language": "...",
  "entries": [...],
  "tags": [...]
}
```
SQL

create table lexicographicResources (  
id int primary key,  
title varchar(255),  
language varchar(10)  
)

Comments

- language identifies the language of headwords, definitions and examples in this dictionary. DMLex is based on the assumption that all headwords in a lexicographic resource are in the same language, and that definitions and examples, if any occur in the lexicographic resource, are in that language too. The language child object of lexicographicResource informs potential users of the lexicographic resource which language that is.

- The main role of a lexicographic resource is to contain entries (entry objects). The other two object types that can optionally occur as children of a lexicographicResource, especially tag, are for lists of look-up values such as part-of-speech labels.

2.2 entry

Represents a dictionary entry. An entry contains information about one headword.

entry: <id>  
  headword: (1..1) <string>  
  homographNumber: (0..1) <number>  
  partOfSpeech: (0..n)  
  label: (0..n)  
  pronunciation: (0..n)  
  inflectedForm: (0..n)  
  sense: (0..n)

XML

<entry id="..." homographNumber="...">  
  <headword>...</headword>  
  <partOfSpeech.../>  
  <label.../>  
  <pronunciation.../>  
  <inflectedForm.../>  
  <sense.../>  
</entry>
JSON

```json
{
   "id": "...",
   "headword": "...",
   "labels": [...],
   "pronunciations": [...],
   "inflectedForms": [...],
   "senses": [...] 
}
```

SQL

```sql
create table entries ( 
   lexicographicResourceID int foreign key references lexicographicResource(id),
   id int primary key,
   headword varchar(255),
   homographNumber int
)
```

Comments

- **headword** contains entry’s headword. The headword can be a single word, a multi-word expression, or any expression in the source language which is being described by the entry.
- Entries in DMLex do not have an explicit listing order. An application can imply a listing order from a combination of the headword and the homograph number.
- DMLex Core does not have a concept of ‘subentry’. If you wish to have subentries (ie. entries inside entries) in your lexicographic resource you can use types from the Linking Module for that.

2.3 partOfSpeech

Represents a part-of-speech label.

```
partOfSpeech: <string>
   listingOrder: (1..1) <number>
```

XML

```
<partOfSpeech value="...">
```

JSON

```
"...
```
SQL

```sql
create table partsOfSpeech (  
entryID int foreign key references entries(id),  
value varchar(10),  
listingOrder int,  
id int primary key
)
```

Comments

- `partOfSpeech` is an abbreviation, a code or some other string of text which identifies the part-of-speech label, for example `n` for noun, `v` for verb, `adj` for adjective. You can use the `tag` datatype to explain the meaning of the part-of-speech tags, to constrain which part-of-speech tags are allowed to occur in your lexicographic resource, and to map them onto external inventories and ontologies.

- If you want to model other grammatical properties of the headword besides part of speech, such as gender (of nouns) or aspect (of verbs), the way to do that in DMLex is to conflate them to the part-of-speech label, for example `noun-masc` and `noun-fem`, or `v-perf` and `v-imperf`.

- `listingOrder` is the position of this part-of-speech label among other part-of-speech labels of the same entry. This can be implicit from the serialization.

2.4 sense

Represents one of possibly many meanings (or meaning potentials) of the headword.

**sense**: `<id>`

- `listingOrder`: (1..1) `<number>`
- `indicator`: (0..1) `<string>`
- `label`: (0..n)
- `definition`: (0..n)
- `example`: (0..n)

XML

```xml
<sense id="...">
  <indicator>...</indicator>
  <label.../>
  <definition.../>
  <example.../>
</sense>
```

JSON
SQL

```sql
create table senses (  
    entryID int foreign key references entries(id),  
    id int primary key,  
    indicator nvarchar(50),  
    listingOrder int  
)
```

Comments

- **listingOrder** represents the position of this sense among other senses of the same entry. Can be implicit from the serialization.

- **indicator** is a short statement, in the same language as the headword, that gives an indication of the meaning of a sense and permits its differentiation from other senses in the entry. Indicators are sometimes used in dictionaries instead of or in addition to definitions.

- **definition** is a statement, in the same language as the headword, that describes and/or explains the meaning of a sense. In DMLex, the term definition encompasses not only formal definitions, but also less formal explanations.

Note on the difference between entries and senses:

- An **entry** is a container for formal properties of the headword such as orthography, morphology, syntax and pronunciation. A **sense** is a container for statements about the headword’s semantics. DMLex deliberately makes it impossible to include morphological information at sense level. If you have an entry where each sense has slightly different morphological properties (e.g., a noun has a weak plural in one sense and a strong plural in another) then, in DMLex, you need to treat it as two entries (homographs), and you can use the Linking Module two link the two entries together and to make sure they are always shown together to human users.
2.5 definition

Represents one of possibly several definitions of a sense.

definition: <string>
definitionType: (0..1) <string>
listingOrder: (1..1) <number>

XML

<definition definitionType="...">...

JSON

{
    "text": "....",
    "definitionType": "...

SQL

create table definitions (
    senseID int foreign key references sense(id),
    text nvarchar(255),
    definitionType nvarchar(10),
    listingOrder int,
    id int primary key
)

Comments

- If you have multiple definitions inside a single sense, you can use definitionType to indicate the difference between them, for example that they are intended for different audiences. Optionally, you can use the tag data type to constrain and/or explain the definition types that occur in your lexicographic resource.
- listingOrder is the position of this definition among other definitions of the same sense. This can be implicit from the serialization.

2.6 inflectedForm

Represents one (of possibly many) inflected forms of the headword.
inflectedForm: <string>
  inflectedTag: (0..1) <string>
  listingOrder: (1..1) <number>
  label: (0..n)
  pronunciation: (0..n)

XML

<inflectedForm inflectedTag="...">
  <text>...</text>
  <label.../>
  <pronunciation.../>
</inflectedTag>

JSON

{
  "inflectedTag": "...",
  "text": "...",
  "labels": [...] ,
  "pronunciations": [...] 
}

SQL

create table inflectedForms ( 
  entryID int foreign key references entries(id),
  inflectedTag varchar(10),
  text varchar(255),
  listingOrder int,
  id int primary key
)

Comments

- inflectedTag is an abbreviation, a code or some other string of text which identifies the inflected form, for example pl for plural, gs for genitive singular, com for comparative. You can use the tag datatype to explain the meaning of the inflection tags, to constrain which inflection tags are allowed to occur in your lexicographic resource, and to map them onto external inventories and ontologies.

- The value of the inflectedForm object is the text of the inflected word itself.

- listingOrder is the position of this inflected form among other inflected forms of the same entry. This can be implicit from the serialization.
• The `inflectedForm` object is intended to model the **inflectional morphology** of a headword. To model derivational morphology, for example feminine forms of masculine nouns, the recommended way to do that in DMLex is to create separate entries for the two words, and link them using the Linking Module.

See Example 1.

### 2.7 label

Represents a restriction on its parent such as temporal (old-fashioned, neologism), regional (dialect), register (formal, colloquial), domain (medicine, politics) or grammar (singular-only).

**label**: `<string>
  listingOrder: (1..1) <number>
</string>`

**XML**

```
<label value="..."/>
```

**JSON**

```
"..."
```

**SQL**

```sql
create table labels (  
  entryID int foreign key references entries(id),
  senseID int foreign key references senses(id),
  inflectedFormID int foreign key references inflectedForms(id),
  pronunciationID int foreign key references pronunciations(id),
  exampleID int foreign key references examples(id),
  value varchar(10),
  listingOrder int,
  id int primary key)
```

**Comments**

• The value of the label object is an abbreviation, a code or some other string of text which identifies the label, for example *neo* for neologism, *colloq* for colloquial, *polit* for politics. You can use the `tag` datatype to explain the meaning of the label tags, to constrain which label tags are allowed to occur in your lexicographic resource, and to map them onto external inventories and ontologies.
• listingOrder is the position of this label among other labels of the same entry. This can be implicit from the serialization.

• A label applies to the object that it is a child of. When the label is a child of entry, then it applies to the headword in all its senses. When the label is a child of sense, then it applies to the headword in that sense only (not including any subsenses linked to it using the Linking Module). When the label is a child of inlectedForm, then it applies only to that inflected form of the headword (in all senses). When the label is a child of pronunciation, then it applies only to that pronunciation of the headword (in all senses).

2.8 pronunciation

Represents the pronunciation of its parent.

pronunciation: <empty>
  soundFile: (0..1) <uri>
  transcription: (0..n)
  listingOrder: (1..1) <number>
  label: (0..n)

XML

<pronunciation soundFile="...">
  <transcription.../>
  <label.../>
</pronunciation>

JSON

{
  "soundFile": "...",
  "transcriptions": [...],
  "labels": [...]
}

SQL

create table pronunciations (  
  entryID int foreign key references entries(id),  
  soundFile varchar(255),  
  listingOrder int,  
  id int primary key
)

Comments
- **transcription** is the transcription of the pronunciation in some notation, such as IPA. If more than transcription is present in a single pronunciation object, then they must be different transcriptions (in different schemes) of the same pronunciation, eg. one in IPA and one in SAMPA.

- **soundFile** is a pointer to a file containing a sound recording of the pronunciation.

- **listingOrder** is the position of this pronunciation object among other pronunciation objects of the same parent. This can be implicit from the serialization.

See Examples 2, 3 and 4.

### 2.9 transcription

Represents the transcription of a pronunciation in some notation such as IPA.

```java
transcription: <string>
    scheme: (0..1) <langCode>
    listingOrder: (1..1) <number>

XML

<transcription scheme="...">...</transcription>

JSON

{
    "text": "...",
    "scheme": "...

}

SQL

create table transcriptions (  
    pronunciationID int foreign key references pronunciation(id),  
    text varchar(255),  
    scheme varchar(10),  
    listingOrder int,  
    id int primary key
)

Comments

- **scheme** object identifies the transcription scheme used here. Example: **en-fonipa** for English IPA. This can be implicit if the lexicographic resource uses only one transcription scheme throughout.
• **listingOrder** is the position of this **transcription** object among other **transcription** objects of the same pronunciation. This can be implicit from the serialization.

### 2.10 example

Represents a sentence or other text fragment which illustrates the headword being used.

**example:** `<string>`

```xml
<sourceIdentity>: (0..1) <string>
<sourceElaboration>: (0..1) <string>
<label>: (0..n)
<soundFile>: (0..1) <uri>
<listingOrder>: (1..1) <number>
```

**XML**

```xml
<example sourceIdentity="..." sourceElaboration="..." soundFile="...">
    <text>...</text>
    <label.../>
</example>
```

**JSON**

```json
{
    "text": "...",
    "sourceIdentity": "...",
    "sourceElaboration": "...",
    "labels": [...],
    "soundFile": "..."
}
```

**SQL**

```sql
create table examples (senseID int foreign key references senses(id),
    text varchar(255),
    sourceIdentity varchar(50),
    sourceElaboration varchar(255),
    soundFile varchar(255),
    id int primary key)
```

**Comments**
• **sourceIdentity** is an abbreviation, a code or some other string of text which identifies the source. You can use the **tag** datatype to explain the meaning of the source identifiers and to constrain which source identifiers are allowed to occur in your lexicographic resource.

• **sourceElaboration** is a free-form statement about the source of the example. If **source** is present, then **sourceElaboration** can be used for information where in the source the example can be found: page number, chapter and so on. If **sourceIdentity** is absent then **sourceElaboration** can be used to fully name the source.

• **soundFile** is a pointer to a file containing a sound recording of the example.

• **listingOrder** is the position of this example among other examples in the same sense. This can be implicit from the serialization.

2.11 **tag**

Represents one (of many) possible values for **partOfSpeech**, **inflectedTag**, **label**, and **source**.

**tag**: `<string>`
  
  **description**: (0..1) `<string>`
  
  **target**: (0..n) `<symbol>`
  
  **partOfSpeechConstraint**: (0..n) `<string>`
  
  **sameAs**: (0..n)

**XML**

```xml
<tag value="...">
  <description>...</description>
  <target value="..."/>
  <partOfSpeechConstraint value="..."/>
  <sameAs.../>
</tag>
```

**JSON**

```json
{
  "value": "...",
  "description": "...",
  "targets": ["..."],
  "partOfSpeechConstraints": ["..."],
  "sameAs": [...
}
create table tags (  
    lexicographicResourceId int foreign key references lexicographicResource(id),  
    value varchar(10),  
    description varchar(255),  
    targets varchar(255), --comma-separated list  
    partOfSpeechConstraints varchar(255), --comma-separated list  
    id int primary key  
)  

Comments  

• The value is an abbreviation, a code or some other string of text which identifies the source. If you want, you can design your implementation to enforce referential integrity between tag values on the one hand and partOfSpeech, inflectedTag etc. objects on the other hand. In other words, you can make it so that the tags you define in tag objects are the only values allowed for partOfSpeech, inflectedTag etc. However, doing this is optional in DMLex. An implementation of DMLex is compliant regardless of whether it enforces referential integrity on tag values.  

• description is a human-readable description of what the tag means.  

• target tells us where exactly the tag is expected to be used. If omitted, then all four. The possible values are:  
  
  – partOfSpeech: as the value of a partOfSpeech object  
  – inflectedTag: as the value of an inflectedTag object  
  – sourceIdentity: as the value of a sourceIdentity object  
  – label: as the value of a label object  
  – definitionType: as the value of a definitionType object  
  – collocateRole: as the value of a collocateRole object  

• partOfSpeechConstraint, if present, says that this tag is only intended to be used inside entries that are labelled with this part of speech. You can us this to constrain that, for example, only nouns and adjectives can have plurals but other parts of speech cannot.  

• target and partOfSpeechConstraint allow you to specify constraints on which tags are expected to appear where throughout the lexicographic resource. Enforcing these constraints in your implementation is optional.  

See Example 5.
2.12 sameAs

Represents the fact that the parent object is equivalent to an item available from an external authority.

sameAs: <uri>

XML

<sameAs uri="..."/>

JSON

"...

SQL

create table sameAs (  
tagID int foreign key references tags(id),  
uri varchar(255),  
id int primary key
)

Comments

- The value is the URI of an item in an external inventory.

See Example 6.

3 DMLex Bilingual Module

DMLex’s Bilingual Module extends the DMLex Core so that you can model a bilingual lexicographic resource in it. A bilingual lexicographic resource is a lexicographic resource with two languages: the headwords and the examples are in one language (called the headword language in DMLex) and their translations are in another language (called the translation language in DMLex).

The Bilingual Module is what you should implement if you want to have only one translation language. If you want to have multiple translation languages in your lexicographic resource, then you should implement the Multilingual Module instead. The Bilingual Module and the Multilingual Module are mutually exclusive: you can implement one or the other but not both.
3.1 Extensions to lexicographicResource

Additional children:

\[\text{lexicographicResource}: \ldots\]
\[\text{translationLanguage}: (1..1) <\text{langCode}>\]

XML

\[
<\text{lexicographicResource} \ldots \text{translationLanguage="\ldots"}>
\ldots
</\text{lexicographicResource}>
\]

JSON

\[
\{
\ldots,
"\text{translationLanguage}": "\ldots"
\}
\]

SQL

\[
\text{alter table lexicographicResources (}
\quad \text{add translationLanguage varchar(10)}
\)
\]

See Example 7.

3.2 Extensions to sense

Additional children:

\[\text{sense}: \ldots\]
\[\text{headwordExplanation}: (0..1) <\text{string}>\]
\[\text{headwordTranslation}: (0..n)\]

XML

\[
<\text{sense} \ldots>
\ldots
\quad <\text{headwordExplanation}.../>\]
\[
<\text{headwordTranslation}.../>\]
\ldots
</\text{sense}>
\]

JSON
{  
  "headwordExplanation": "...",
  "headwordTranslations": ["..."],
  ...
}

SQL

```sql
alter table senses (  
  add headwordExplanation varchar(255)
)
```  

Comments

- **headwordExplanation** is a statement in the target language which explains (but does not translate) the meaning of the headword.

### 3.3 headwordTranslation

Represents one of possibly multiple translations of a headword.

**headwordTranslation: <string>**

- **listingOrder: (1..1) <number>**
- **partOfSpeech: (0..n) <string>**
- **label: (0..n)**
- **pronunciation: (0..n)**
- **inflectedForm: (0..n)**

**XML**

```xml
<headwordTranslation>  
  <text>...
  <partOfSpeech.../>  
  <label.../>  
  <pronunciation.../>  
  <inflectedForm.../>  
</headwordTranslation>
```

**JSON**

```json
{
  "text": "...",
  "partOfSpeech": [...],
  "labels": [...],
}  
```
"pronunciations": [...],
"inflectedForms": [...]
}

SQL

create table headwordTranslations (  
    senseID int foreign key references senses(id),  
    text nvarchar(255),  
    listingOrder int,  
    id int primary key  
);
alter table partsOfSpeech (  
    add headwordTranslationID int foreign key references headwordTranslations(id)  
);
alter table labels (  
    add headwordTranslationID int foreign key references headwordTranslations(id)  
);
alter table pronunciations (  
    add headwordTranslationID int foreign key references headwordTranslations(id)  
);
alter table inflectedForms (  
    add headwordTranslationID int foreign key references headwordTranslations(id)  
)

Comments

- listingOrder is the position of this headwordTranslation object among other headwordTranslation objects in the same sense. This can be implicit from the serialization.
- partOfSpeech is an abbreviation, a code or some other string of text which identifies the part-of-speech label, for example n for noun, v for verb, adj for adjective. You can use the tag datatype to explain the meaning of the part-of-speech tags, to constrain which part-of-speech tags are allowed to occur in your lexicographic resource, and to map them onto external inventories and ontologies.

See Examples 8 and 9.

3.4 Extensions to example

Additional children:

extexample: ...
    exampleTranslation: (0..n)
3.5 exampleTranslation

Represents the translation of an example.

```xml
<example ...
  ...
  <exampleTranslation.../>
</example>
```

```json
{
  ...
  "exampleTranslations": [...]
}
```

SQL: no changes needed.
create table exampleTranslations (     exampleID int foreign key references examples(id),     text varchar(255),     soundFile varchar(255),     listingOrder int,     id int primary key     );
alter table labels (     add exampleTranslationID foreign key references exampleTranslations(id)     )

Comments

- **soundFile** is a pointer to a file containing a sound recording of the translation.
- **listingOrder** is the position of this translation among other translations of the same example. This can be implicit from the serialization.

### 3.6 Extensions to tag

Redefinition of partOfSpeechConstraint:

- If present, says that:
  - If this tag is used inside a headwordTranslation, then it is intended to be used only inside a headwordTranslation labelled with this part of speech.
  - If this tag is used outside a headwordTranslation, then it is intended to be used only inside entries that are labelled with this part of speech.

### 4 DMLex Multilingual Module

DMLex’s Multilingual Module extends the Core and turns a monolingual lexicographic resource into a multilingual one. A multilingual lexicographic resource is a lexicographic resource with multiple (typically three or more) languages: the headwords and the examples are in one language (called the headword language in DMLex) and their translations are in multiple other languages (called the translation languages in DMLex).

The Multilingual Module is what you should implement if you want to have more than one translation language. If you want to have only one translation language in your lexicographic resource, then you should implement the Bilingual Module instead. The Multilingual Module and the Bilingual Module are mutually exclusive: you can implement one or the other but not both.
4.1 Extensions to lexicographicResource

Additional children:

```plaintext
lexicographicResource: ...
   translationLanguage: (1...n)
```

**XML**

```xml
<lexicographicResource ...>
   ...
   <translationLanguage.../>
</lexicographicResource>
```

**JSON**

```json
{
   ...
   "translationLanguage": [...]
}
```

**SQL:** no change needed.

4.2 translationLanguage

Represents one of the languages in which translations are given in this lexicographic resource.

```plaintext
translationLanguage: <langCode>
   listingOrder: (1..1) <number>
```

**XML**

```xml
<translationLanguage langCode=""/>
```

**JSON**

```
"...
```

**SQL**

```sql
create table translationLanguage (  
   lexicographicResourceId int foreign key references lexicographicResources(id),  
   langCode varchar(10) primary key,  
   listingOrder int,
)
```
Comments

- **listingOrder** sets the order in which translations (of headwords and examples) should be shown. It outranks the listing order given in `headwordTranslation`, `headwordExplanation` and `exampleTranslation` objects.

See Example 10.

### 4.3 Extensions to sense

Additional children:

```plaintext
sense: ...
    headwordExplanation: (0..n)
    headwordTranslation: (0..n)
```

**XML**

```xml
<sense ...
    ...
    <headwordExplanation.../>
    <headwordTranslation.../>
    ...
</sense>
```

**JSON**

```json
{
    ...
    "headwordExplanations": {...},
    "headwordTranslations": {...},
    ...
}
```

**SQL**: no changes needed

### 4.4 headwordExplanation

Represents a statement in the target language which explains (but does not translate) the meaning of the headword.

```plaintext
headwordExplanation: <string>
    language: (1..1) <langCode>
```

**XML**
<headwordExplanation language="...">...</headwordExplanation>

**JSON**

"headwordExplanations": {
  language: "...",
  ...
}

**SQL**

```sql
create table headwordExplanations (
    senseID int foreign key references senses(id),
    language nvarchar(10) foreign key references translationLanguage(langCode),
    text nvarchar(255),
    id int primary key
)
```

**Comments**

- `language` indicates the language of this explanation. You can use the `translationLanguage` datatype to explain the meaning of the language codes that appear here and/or to constrain which language codes are allowed.
- It is assumed that there will always be a maximum of one `headwordExplanation` per language.

### 4.5 headwordTranslation

Represents one of possibly multiple translations of a headword.

```xml
<headwordTranslation language="...">
  <text>...</text>
  <partOfSpeech.../>
  <label.../>
  <pronunciation.../>
</headwordTranslation>
```
<inflectedForm.../>
</headwordTranslation>

JSON

"headwordTranslations": {
  language: [{
    "text": "...",
    "partsOfSpeech": [...],
    "labels": [...],
    "pronunciations": [...],
    "inflectedForms": [...]
  }, ...],
...
}

SQL

cREATE TABLE headwordTranslations (
  senseID int foreign key references senses(id),
  language nvarchar(10) foreign key references translationLanguage(langCode),
  text nvarchar(255),
  listingOrder int,
  id int primary key
);

ALTER TABLE partsOfSpeech (  
  add headwordTranslationID int foreign key references headwordTranslations(id)
);

ALTER TABLE labels (  
  add headwordTranslationID int foreign key references headwordTranslations(id)
);

ALTER TABLE pronunciations (  
  add headwordTranslationID int foreign key references headwordTranslations(id)
);

ALTER TABLE inflectedForms (  
  add headwordTranslationID int foreign key references headwordTranslations(id)
)

Comments

- **language** indicates the language of this translation. You can use the *translationLanguage* datatype to explain the meaning of the language codes that appear here and/or to constrain which language codes are allowed.
- For more comments see comments under *headwordTranslation* in the Bilingual Module.
4.6 Extensions to example

Additional children:

\[\text{sense:\ldots} \]
\[\begin{align*}
\text{exampleTranslation: } & (0..n) \\
\text{XML} & \\
<\text{example} \ldots> & \\
\ldots & \\
<\text{exampleTranslation} \ldots/> & \\
</\text{example}> & \\
\text{JSON} & \\
& \\
& \{ \\
& \ldots, \\
& "\text{exampleTranslations}" : \{\ldots\} \\
& \} \\
\text{SQL: no changes needed.}
\]

4.7 exampleTranslation

Represents the translation of an example.

\[\begin{align*}
\text{exampleTranslation: } & <\text{string}> \\
& \text{language: } (1..1) <\text{langCode}> \\
& \text{soundFile: } (0..1) <\text{uri}> \\
& \text{listingOrder: } (1..1) <\text{number}> \\
\text{XML} & \\
<\text{exampleTranslation} \text{ language="\ldots" soundFile="\ldots">} & \\
& \text{text}\ldots</\text{text>} \\
& \text{<label\ldots/>} & \\
</\text{exampleTranslation}> & \\
\text{JSON} & \\
\]

27
"exampleTranslations": {
    "language": [
      {
        "text": "...",
        "labels": [...],
        "soundFile": "...
      }, ...
    ],
    ...
  }

SQL

```sql
CREATE TABLE exampleTranslations (  
  exampleID int FOREIGN KEY REFERENCES examples(id),  
  language varchar(10) FOREIGN KEY REFERENCES translationLanguage(langCode),  
  text varchar(255),  
  soundFile varchar(255),  
  listingOrder int,  
  id int PRIMARY KEY
)
```  

**Comments**

- **language** indicates the language of this translation. You can use the `translationLanguage` datatype to explain the meaning of the language codes that appear here and/or to constrain which language codes are allowed.
- For more comments see comments under `exampleTranslation` in the Bilingual Module.

### 4.8 Extensions to `tag`

Additional child:

```xml
<tag ...>  
  <translationLanguageConstraint langCode="..."/>
</tag>
```
JSON

{
  ...
  "translationLanguageConstraint": ["..."]
}

SQL

`alter table tags (add translationLanguageConstraints varchar(255), --comma-separated list)`

Comments

- `translationLanguageConstraint`, if present, says that if this tag is being used inside a `headwordTranslation` or an `exampleTranslation`, then it is intended to be used only inside `headwordTranslation` and `exampleTranslation` objects labelled with this language.

- For a redefinition of `partOfSpeechConstraint`, see see `tag` in the Bilingual Module.

5 DMLex Linking Module

DMLex’s Linking Module can be used to construct *relations* between objects which “break out” of the tree-like parent-and-child hierarchy constructed from datatypes from the Core and from other modules. The Linking Module can be used to create relations between senses which are synonyms or antonyms, between entries whose headwords are homonyms or spelling variants, between senses which represent superordinate and subordinate concepts (eg. hypernyms and hyponyms, holonyms and meronyms), between entries and subentries, between senses and subsenses, and many others.

Each relation is represented in DMLex by an instance of the `relation` datatype. A relation brings two or more `members` together. The fact that an object (such as a sense or an entry) is a member of a relation is represented in DMLex by an instance of the `member` datatype.

The Linking Module can be used to set up relations between objects inside the same lexicographic resource, or between objects residing in different lexicographic resources.

Relations themselves can be members of other relations.

See Examples 12-18.
5.1 Extensions to lexicographicResource

Additional children:

**lexicographicResource:** ...

- **relation:** (0..n)
- **relationType:** (0..n)

**XML**

```xml
<lexicographicResource ...>
...
<relation.../>
<relationType.../>
</lexicographicResource>
```

**JSON**

```json
{
...
"relations": [...],
"relationTypes": [...]
}
```

**SQL:** no change needed

5.2 relation

Represents the fact that a relation exists between two or more objects.

**relation:** <string>

- **description:** (0..1) <string>
- **member:** (2..n)

**XML**

```xml
<relation type="...">
  <description>...</description>
  <member.../>
</relation>
```

**JSON**
{  
  "type": "...",  
  "description": "...",  
  "members": [...]  
}

SQL

create table relations (  
id int primary key,  
type varchar(10),  
description nvarchar(255)  
)

Comments

- The value of a relation specifies what type of relation it is, for example a relation between synonyms or a relation between a sense and a subsense. Optionally, you can use relationType objects to explain those types and to constrain which types of relations are allowed to exist in your lexicographic resource.
- description is an optional human-readable explanation of this relation.

5.3 member

Represents the fact that an object is a member of a relation.

member: <idref>  
  role: (0..1) <string>  
  listingOrder: (1..1) <number>  
  reverseListingOrder: (1..1) <number>

XML

<member idref="..." role="..." reverseListingOrder="..."/>

JSON

{
  "idref": "...",
  "role": "...",
  "reverseListingOrder": "..."
}

SQL
create table members (  
lexicographicResourceID int foreign key references lexicographicResources(id),  
relationID int foreign key references relations(id),  
memberEntryID int foreign key references entries(id),  
memberSenseID int foreign key references senses(id),  
memberCollocateMarkerID int foreign key references collocateMarkers(id),  
role nvarchar(50),  
listingOrder int,  
reverseListingOrder int,  
id int primary key  
)  

Comments  

- The value of member is the ID of an object, such as an entry or a sense.  
- role is an indication of the role the member has in this relation: whether it is the hypernym or the hyponym (in a hyperonymy/hyponymy relation), or whether it is one of the synonyms (in a synonymy relation), and so on. You can use membershipRole objects to explain those roles and to constrain which relations are allowed to contain which roles, what their object types are allowed to be (eg. entries or senses) and how many members with this role each relation is allowed to have.  
- listingOrder is the position of this member among other members of the same relation. It should be respected when showing members of the relation to human users. This can be implicit from the serialization.  
- reverseListingOrder is the position of this relation among other relations this member is involved in. It should be respected when showing the relations of this member to a human user. This can be implicit from the serialization.  

5.4 relationType  

Represents one of possible values for relation.  

relationType: <string>  
  description: (0..1) <string>  
  scope: (0..1) <symbol>  
  sameAs: (0..n)  
  memberRole: <0..n>  

XML
<relationType type="..." scope="...">
    <description>...</description>
    <sameAs.../>
    <memberRole.../>
</relationType>

JSON

{
    "type": "...",
    "scope": "...",
    "sameAs": [
        "..."]
    "memberRoles": [...]
}

SQL

create table relationTypes (  
    lexicographicResourceId int foreign key references lexicographicResources(id),  
    type varchar(10),  
    scope varchar(50),  
    id int primary key  
);
alter table sameAs (  
    add relationTypeID int foreign key references relationTypes(id)  
)

Comments

• description is a human-readable explanation of this relation type.

• scope specifies restrictions on member of relations of this type. The possible values are:
  
  – sameEntry: members must be within of the same entry
  – sameResource: members must be within the same lexicographicResource
  – any: no restriction

• memberRole objects define roles for members of relations of this type.

5.5 memberRole

memberRole: <stringOrEmpty>
    description: (1..1) <string>
    memberType: (1..1) <symbol>
    min: (0..1) <number>
max: (0..1) <number>
action: (1..1) <symbol>
sameAs: (0..n)

**XML**

```xml
<memberRole role="..." memberType="..." min="..." max="..." action="...">
    <description/>
</sameAs... />
</memberRole>
```

**JSON**

```json
{
    "role": "...",
    "description": "...",
    "memberType": "...",
    "min": "...",
    "max": "...",
    "action": "...",
    "sameAs": [...]
}
```

**SQL**

```sql
create table memberRoles ( 
    relationTypeID int foreign key references relationTypes(id),
    role varchar(50),
    description varchar(255),
    memberType varchar(50),
    min int,
    max int,
    action varchar(50)
);
alter table sameAs ( 
    add memberRoleID int foreign key references memberRoles(id)
)
```

**Comments**

- If the value is empty, then members having this role do not need to have a `role` property.
- `description` is a human-readable explanation of this member role.
- `memberType` is restrictions on the types of objects that can have this role. The possible values are:
– **sense**: the object that has this role must be a sense.
– **entry**: the object that has this role must be an entry.
– **collocateMarker**: the object that has this role must be a collocateMarker.

- **min** is a number which says that relations of this type must have at least this many members with this role. If omitted then there is no lower limit (effectively, zero).
- **max** is a number which says that relations of this type may have at most this many members with this role. If omitted then there is no upper limit.
- **action** gives instructions on what machine agents should do when showing this relation to a human user (either on its own or in the context of one of its members). The possible values are:
  - **embed**: Members that have this role should be shown in their entirety, i.e. the entire entry or the entire sense. This is suitable for the relation between entries and subentries, or senses and subsenses.
  - **navigate**: Members that have this role should not be shown in their entirety, but a navigable (e.g. clickable) link should be provided. This is suitable for the relation between synonyms, for example.
  - **none**: Members that have this role should not shown.

### 6 DMLex Inline Markup Module

This module makes it possible to mark up substrings inside the string values of certain objects and to attach properties to them.

It is up to the implementer to decide how to implement inline markup in an implementation of the DMLex Inline Markup module, whether *in-place* (as in XML) or as *stand-off* markup (for example through start and end indexes).

#### 6.1 Extensions to headword

Additional children:

`headword: ...`

`placeholderMarker: (0..n)`

XML: TBD
JSON: TBD
SQL: TBD
6.2 Extensions to headwordTranslation

Additional children:

headwordTranslation: ...
    placeholderMarker: (0..n)

XML: TBD
JSON: TBD
SQL: TBD

6.3 Extensions to example

Additional children:

element: ...
    headwordMarker: (0..n)
    collocateMarker: (0..n)

XML: TBD
JSON: TBD
SQL: TBD

6.4 Extensions to exampleTranslation

Additional children:

exampleTranslation: ...
    headwordMarker: (0..n)
    collocateMarker: (0..n)

XML: TBD
JSON: TBD
SQL: TBD
6.5 Extensions to definition

Additional children:

definition: ...
    headwordMarker: (0..n)
    collocateMarker: (0..n)

XML: TBD
JSON: TBD
SQL: TBD

6.6 placeholderMarker

Marks up a substring inside a headword or inside a headword translation which is not part of the expression itself but stands for things that can take its place. An application can use the inline markup to format the placeholders differently from the rest of the text, to ignore the placeholder in full-text search, and so on.

placeholderMarker: <string>

XML: TBD
JSON: TBD
SQL: TBD
See Examples 19 and 20.

6.7 headwordMarker

Marks up a substring inside an example, inside an example translation or inside a definition which corresponds to the headword (or to a translation of the headword). An application can use the inline markup to highlight the occurrence of the headword for human readers through formatting.

headwordMarker: <string>

XML: TBD
JSON: TBD
SQL: TBD
See Example 21.
6.8 collocateMarker

Marks up a substring inside an example, inside an example translation or inside a definition where an important collocate of the headword (or of its translation) occurs. An application can use the inline markup to highlight the collocate for human readers through formatting.

collocateMarker: <string>
    lemma: (0..1) <string>
    collocateRole: (0..n) <string>

XML: TBD
JSON: TBD
SQL: TBD

Comments

- **lemma** is the lemmatized form of the collocate. An application can use it to provide a clickable link for the user to search for the lemma in the rest of the lexicographic resource or on the web. (If you want to link the collocate explicitly to a specific entry or to a specific sense in your lexicographic resource, or even in an external lexicographic resource, you can use the Linking Module for that.)

- **collocateRole** can be used to communicate facts about the role of the collocate in the sentence, for example its syntactic role (subject, direct object etc.), its semantic role (agent, affected etc) or its semantic type (human, institution etc.) Optionally, you use the **tag** datatype to explain and/or constrain the collocate types that are allowed to appear in your lexicographic resource.

See Example 22.