DMLex: a proposal

This is my proposal for what I think should be in the LEXIDMA standard (DMLex). It is a development from what I had proposed earlier, with one additional twist: I am introducing a new object type, `Segment`, which is an abstraction over entries, senses, subsenses and other such things. My reasoning behind this part of the proposal is outlined below in the chapter *Entry structure*. The rest of my proposal is in line with what has been discussed and "consensused" in the LEXIDMA meetings before.

Some of the text here is text which could go into the actual specification document (DocBook and all), while other bits of the text here are more like explanations of the reasoning *behind* the standard: those probably shouldn't be part of the spec itself but could go into some other, complementary publication *about* the spec (and/or into someone's PhD thesis, ha).

Michal Měchura, May 2012

0. Introduction

In lexicography, we work with things such as *entries*, *headwords*, *senses*, *definitions*, *translations*, *example sentence* and so on. DMLex provides guidance on modelling these things computationally, for example in a relational database or in an XML document. Implementations of DMLex in specific formalisms (XML, relational databases, ...) can be used not only for data interchange but also as internal "native" data formats in dictionary writing systems or in other software agents which process lexicographic data.

0.1. How abstract is DMLex?

As a data model, DMLex is *specific enough to be implementable* but *abstract enough to be implementation-independent*.

DMLex is **implementation-independent**. By this we mean that DMLex does not presuppose any particular implementation, formalism or serialization. DMLex is not a directly usable schema such as LMF, TEI or Ontolex. To use DMLex, you need to *implement* it in some underlying formalism first, for example as an XML Schema, as an OWL ontology, or as a database schema in a relational database management system.

On the other hand, DMLex is **implementable**. By this we mean that DMLex is not a high-level *metamodel* or *reference model*. DMLex is not a vague informal inventory of what "exists" in the lexicography universe. DMLex is a formal and highly explicit schema which can be translated fairly straightforwardly into specific formalisms such as XML Schemas, OWL ontolgies, or the schema of a relational database. In fact, a few suggested serializations are provided at the end of this document.

0.2. DMLex solves the problem of incompatible schemas

We mentioned that DMLex is not a *metamodel*. The metamodel of lexicography is well known: there are *entries*, each entry has a *headword*, entries are subdivided into *senses*, senses contain things such as *definitions* and *translations*, and so on. Broadly speaking, there is consensus on these things in lexicography. The metamodel is not controversial. There is, however, a lack of consensus on the details: How do we model variant headwords? Should part-of-speech labels be allowed on senses? Should subsenses be allowed to exist? How do we model subentries? How do we model cross-references? Individual dictionary projects tend to answer these questions differently and the result is that each dictionary project uses its own, custom-designed schema which is incompatible with schemas used on other projects. So, in lexicography, we currently have *high-level consensus* (there exist *entries*, *senses* etc.) contrasting with *low-level incompatibility* (each dictionary has its own schema).

DMLex is an attempt to remove the low-level incompatibility. DMLex is a data model which can accommodate practically all dictionaries, regardless of how they have chosen to answer the questions of detail. DMLex is sufficiently generic so that the details can be treated as *configuration settings* (enforcable by *business rules*) while the database schema (or XML schema, OWL ontology...) remains the same. With DMLex, it is possible to have only one database schema for multiple dictionary projects, even if each dictionary gives different answers to the detailed questions. For example, one dictionary can allow rich hierarchies of senses and subsenses while another can allow only flat lists of senses. One dictionary can allow grouping senses into "sense groups" while another not. One dictionary can be bilingual while another only monolingual. These details are treated as configuration settings in DMLex, while the data model remains the same.

0.3. How is DMLex formulated?

The DMLex data model is formulated here in terms of **objects**, **relations** between objects, and **attributes** of objects and relations. These three terms (*object, relation, attribute*) must be understood in an abstract, implementation-independent way. How they translate into things in specific implementations, such as database tables or XML elements, is a question the data model is agnostic about.

Each object, relation and attribute in this data model is of a certain **type**. The type of an object or relation determines which attributes it may or must have, and which objects it may or must be related to. The types in this data model have names such as *Entry*, *Headword*, *Sense*.

One very common relation between objects in lexicography is the **parent-child** relation. The organization of lexicographic objects such as entries, headwords and senses into hierarchical trees of parents and children has been a pervasive design pattern in lexicography since digitization began. Consequently, in this data model, the parent-child relation is also very prominent. Many data types in this specification are defined in terms of the (types of) parents and children their instances can have.

On the other hand, parent-child relations are not the only type of relation that appears in this data model. Parent-child hierarchies are only used here to model the basic entry-headword-sense skeleton of a dictionary. Other phenomena, such as cross-references between entries, or the inclusion of enties inside other entries as subentries, are modelled here with other types of relations. These other relations can be said to "break out" of the traditional tree-like hierarchy of a dictionary, and to "annotate" the basic parent-child skeleton with additional information.

0.4. The structure of this document

The DMLex data model is presented in this document in three chapters. The following chapter, *Entry structure*, explains how a lexicographical resource is to be structured into entries, senses, subsenes, subentries and so on. The next chapter after that, *Entry content*, gives you a rich inventory of objects for populating entries and senses with definitions, grammatical labels, example sentenses, translations and other informational objects. The final chapter, *Configuration*, explains how the DMLex data model can be configured for a specific dictionary.

1. Entry structure

In a typical dictionary schema, the structure of the dictionary is defined as a tree-like hierarchy of entries, senses, subsenses, subentries and so on. The schema typically defines which kinds of these objects are allowed to exist, how they are allowed to stack inside each other (for example: a sense is allowed to contain zero or more subsenses), and what data they may or must contain (for example: a sense must contain exactly one definition, a subsense may but must not contain translations). This tree-like hierarchical definition of the dictionary's structure is usually hard-coded in some way, for example as a DTD.

DMLex is radically different from this approach. DMLex does not force its users to adopt any particular pattern of hierarchical structure. DMLex is more abstract, it is able to accommodate an arbitrary tree-like dictionary structure. The motivation and principles of the DMLex approach are explained in the next subchapter, *How DMLex models entry structure*. After that, the rest of this chapter specifies the object types `Expression` and `Segment` and the relations types `Inclusion` and `Link` which are used in DMLex to model the structure of entries.

1.1. How DMLex models entry structure

A dictionary is basically a catalog of *linguistic expressions* (typically: words) in some human language to which it assigns some *informational properties* (part-of-speech labels, pronunciation transcriptions, definitions, translations and others).

When describing an expression, a dictionary divides the description into units such as entries and senses. Typically (but not always), an expression is represented in a dictionary by one entry subdivided into one or more senses. Some of the expression's informational properties are assigned to the entire entry (they have entry-wide scope) while others are assigned only to a specific sense (they have sense-specific scope).

There is a tendency in lexicography for morphosyntactic and phonetic properties to be treated as entry-wide properties, and for semantic and pragmatic properties to be treated as sense-specific properties. So, for example, a part-of-speech label or a pronunciationn transcription would typically be assigned to the entire entry, while a definition, a usage label or a translation would be a assigned to an individual sense. This is motivated by the lexicographer's desire to distinguish *form* from *function*, and to establish a one-to-many mapping between form and function. The form of an expression is defined by its morphosyntax and phonetics (as well as its orthography) and the function of an expressions by its semantics and pragmatics.

However, this is only a tendency and a theoretical ideal, and many deviations occur in practice. There are dictionary schemas which allow the assignment of grammatical labels to individual senses, because there exist (for example) nouns which have different plurals or genders for different senses. Similarly, it is not unusual to see usage labels (such as *vulgar* or *neologism*) applied at entry-wide scope because these are seen as properties of the entire entry, affecting all its senses. And then there are informational properties which apply to a *non-singleton proper subset* of the senses in an entry (that is, they apply to more than one sense but not to all the senses): dictionaries often use something like *sense groups* for this. Finally, there are informational properties which *further specify* or *refine* those assigned already: dictionaries often subdivide senses into subsenses for this purpose.

The result is that every dictionary comes with its own schema for dividing entries into senses, sense groups, subsenses, subentries and so on. The general principle is always the same: first we subdivide an expression into segments (entries, senses, subsenses...), then we assign informational properties to those segmens. What differs from dictionary to dictionary is the kinds of segments that are allowed to exist, and how they are allowed to stack inside each other.

DMLex does not force you to accept any particular "world-view" on these questions. DMLex does not force you to accept any particular inventory of segment kinds or how they stack inside each other. DMLex is more abstract than that. In DMLex, all segments (regardless of whether they are entries, senses or something else) are modelled as instances of the `Segment` object type. Each `Segment` object has a `@role` attribute which specifies what kind of segment it is: permitted values for this attribute are `"entry"`, `"sense"`, `"senseGroup"` and others. Then, individually for each dictionary, DMLex allows you to set up a configuration through which you can constrain the kinds of segments that are allowed to exist in the dictionary, how they are allowed to stack inside each other, and what kinds of informational properties can be assigned to them.

So, for example, you can use DMLex to define a dictionary which consists of entries, sense groups, senses and subsenses, such that: each entry is required to contain at least one sense; senses can optionally be grouped into sense groups; each sense may contain zero or more subsenses; senses and subsenses may contain definitions and translations (but sense groups can't); entries and sense groups can contain part-of-speech labels (but senses and subsensesn can't); and so on. All of these properties of a dictionary, which until now have typically been hard-coded in dictionary schemas, are treated as merely configuration settings in DMlex and are not hard-coded.

1.2. Objects and relations involved in modelling entry structure

In DMLex, the following two object types are available to model the structure of entries:

- `Expression`, where each `Expression` object represents one headword (or some other lingustic expression described in the dictionary).
- `Segment`, where each `Segment` is said to *describe* or *be about* or *belong to* an `Expresion`. Each `Segment` has a `@role` (such as `"entry"` or `"sense"`) and can be a hierarchical child of another `Segment` belonging to the same `Expression`.

These two types are sufficient to model the parent-child hierarchy of entries, senses, subsenses and so on. In addition to that, the following two relation types are available in DMLex to model phenomena which "break out" of the tree-structured hierarchy:

- `Inclusion`, where each `Inclusion` models the fact that one `Segment` is to be included under another `Segment` as a subentry, even though the two `Segments` belong to different `Expressions`.
- `Link`, where each `Link` connects two or more `Segments` into a group of mutual cross-reference or linkage. `Links` are used in DMLex to model sense-to-sense cross-references such as synonyms and antonyms. `Links` are also used in DMLex to model headword-to-headword links, such as links between variants. The `@role` attribute of a `Link` determines the kind of link in question: `"antonyms"`, `"synonyms"`, `"variants"` etc.

1.3. The `Expression` object type

An `Expression` represents an expression being described in the dictionary. Typically, an `Expression` is a headword, but an `Expression` can also be a multi-word expression which heads a phraseological subentry, a collocation which heads the description of a collocation block inside the sense of some entry, and so on.

Attributes

- `@id` (required, can be computable or implicit from `@text`).
- `@text` (required): the text of the `Expression`, in the dictionary's *object language*.
- `@role` (optional): the role this `Expression` plays in the dictionary. Allowed values are `"headword"`, `"variantHeadword"`, `"multiwordUnit"`, `"pattern"`, `"collocation"`, `"idiom"`.

The `@text` attribute

Note that an `Expression` is defined just by its orthography. Any further characteristics, such as its part of speech or its homonym number, will be supplied by `Segments` (typically, by a `Segment` with `@role="entry"`). An `Expression` represents a short string of text which a dictionary user might search for by typing it into the search box of a dictionary website, or which might form part of the URL of a web page on a dictionary website.

The `@role` attribute

The `@role` attribute can be used as a hint to software agents on what to do with the `Expression`. For example, when an `Expression` has `@role="heaword"`, the software agent can display it among other headwords in an alphabetical list, or display it in large bold font when showing an entry to the human user.

The `role` attribute can also be used to constrain the dictionary's structure. It is possible to declare in DMLex (using the configuration formalism described later in this document) that, for example, each `Expression` with `@role="headword"` must be attached to at least one `Segment` with `@role="entry"`, and that each `Segment` with `@role="entry"` must have a part-of-speech label, may have a pronunciation transcription, and so on.

Implementing the `Expression` object type

When implementing the `Expression` type in a specific environment, such as a relational database or an XML Schema, it is possible to do it in two ways:

- **The abstract approach:** create just one "type" (i.e. just one database table, or just one XML element name) for all `Expressions`, and then distinguish between the different roles (`"headword"`, `"multiwordUnit"`, ...) by means of a column in the table or by means of an XML attribute.
- **The concrete approach:** create separate "types" for the different roles of `Expressions` in your dictionary, for example one database table for headwords, another database table for multi-word units, and so on.

Both approaches are valid implementations of DMLex. The difference is in the level of abstraction, and consequently in the amount of flexibility. In the abstract approach, the advantage is that you can accommodate different dictionaries in the same schema, but the disadvantage is that any constraints on the entry structure must be formulated as business rules and must be enforced at the application level (= outside the schema). In the concrete approach, the disadvantage is that your schema can accommodate only one hard-coded entry structure, but the advantage is that any constraints on the entry structure are part of the schema and therefore enforced directly at the data level (no business rules needed).

1.4. The `Segment` object type

A `Segment` represent an entry, a sense, a subsense, a subentry, a sense group, or some other object of this kind. A `Segment` is a container for informational objects which describe an `Expression`.

Attributes

- `@id` (required, can be computable).
- `@expression` (required): a reference to the `Expression` which this `Segment` describes, to which this `Segment` belongs.
- `@role` (required): the role this `Segment` has in describing the expression. Allowed values are `"entry"`, `"sense"`, `"subsense"`, `"senseGroup"`, `"subentry"`.
- `@parent` (optional): a reference to another `Segment` (belonging to the same `Expression`) which is the hierarchical parent of this `Segment`.
- `@listingOrder` (required, can be implicit): a sortkey which determines the position of this `Segment` among its siblings when showing an entry to the human user.

The `@role` attribute

The `@role` attribute can be used as a hint to software agents on what to do with the `Segment`. For example, when a `Segment` has `@role="entry"`, the agent should format it as a dictionary entry, starting with the `Expression` it belongs to. When a `Segment` has `@role="sense"` then the agent should format it like a conventional dictionary sense, with a bullet point or an (automatically generated) sense number, but *without* the `Expression`.

More importantly, the `role` attribute can be used to constrain the dictionary's structure: to constrain which kinds of `Segments` can form parent-child relationships (by means of the `@parent` attribute). It is possible to declare in DMLex (using the configuration formalism described later in this document) that, for example, a `Segment` with `@role="entry"` must have at least one child `Segment` with `@role="sense"`, and that a `Segment` with `@role="sense"` may have zero or more child `Segments` with `@role="subsense"`. It is also posssible to declare in DMLex (using the same configuration mechanism) which informational properties each kind of `Segment` may or must contain: for example that each `Segment` with `@role="sense"` must have a definition but a `Segment` with `@role="senseGroup"` may not, and so on.

Implementing the `Segment` object type

When implementing the `Segment` type in a specific environment, such as a relational database or an XML Schema, there are once again two possible approaches to doing that, the abstract approach and the concerete approach.

- **The abstract approach:** create just one "type" (i.e. just one database table, or just one XML element name) for all `Segments`, and then distinguish between the different roles (`"entry"`, `"sense"`, ...) by means of a column in the table or by means of an XML attribute.
- **The concrete approach:** create separate "types" for the different roles of `Segments` in your dictionary, for example one database table for entries, another database table for senses, and so on.

Both approaches are valid implementations of DMLex. The difference is in the level of abstraction, and consequently in the amount of flexibility. In the abstract approach, the advantage is that you can accommodate different dictionaries in the same schema, but the disadvantage is that any constraints on the entry structure must be formulated as business rules and must be enforced at the application level (= outside the schema). In the concrete approach, the disadvantage is that your schema can accommodate only one hard-coded entry structure, but the advantage is that any constraints on the entry structure are part of the schema and therefore enforced directly at the data level (no business rules needed).

Implementing the `@expression` attribute

Each `Segment`, regardles of its role, "belongs" to an `Expression`, and the `Segment`'s `@expression` attribute tells you which `Expression` that is. But the `@xpression` attribute does not always need to be specified explicitly: for `Segments` which are children of other `Segments`, the `Expression` can be worked out by following the chain of child-to-parent links up to the top. Only the top-level `Segment` (which would typically have `@role="entry"` and which has no parent `Segment`) needs to have its `Expression` specified explicitly.

When implementing DMLex, you are free to implement the `@expression` attribute in any way you want. If you are going with the concrete approach, you can make `@expression` explicit for some kinds of `Segments` (typicaly: entries) and leave it implicit for others. If you are going with the abstract approach, you may or may not need to make `@expression` explicit for all `Segments`, depending on the environment you are in (for example, a relational database). If `@expression` is explicit for all `Segments` in your implementation, then you may also need some business rules for enforcing that parent `Segments` and child `Segments` belong to the same `Expression`.

1.5. The `Inclusion` relation type

An `Inclusion` represents the fact that one `Segment` (called the *included* segment) should be shown to human users as a child of another `Segment` (called the *includer* segment), even though the two `Segments` belong to different

`Expressions`.

The `Inclusion` relation is useful for modelling situations when, for example, a multiword subentry is to be included somewhere inside the entry for a single-word headword. In DMLex, the headword would be represented by an `Expression`, and the phraseological multiword unit would be represented by another `Expression`. Each of these `Expressions` would have at least one `Segment`. Then, using the `Inclusion` relation, one of the multiword `Expression`'s `Segments` would be included under one of the headword `Expression`'s `Segments`.

The consequence is that the multiword subentry exists in the dictionary as an independent `Expression`, with one or more `Segments` describing it. Users can search for, it can be findable, and a software agent can show it to human users. But, at the same time and without any duplication of data, the same multiword unit is shown inside the entry for the headword.

Attributes

- `@includerSegment` (required): a reference to the `Segment` inside of which the other `Segment` is to be included.
- `@includedSegment` (required): a reference to the `Segment` which is to be included inside the other `Segment`.
- `@listingOrder` (required, can be implicit): a sortkey which determines the position of the included `Segment` among the children of the includer `Segment`.

The `@listingOrder` attribute

When a software agent is showing a `Segment` to a human user, one of the things the agent needs to do is to obtain a list of the `Segment`'s children, and show them in some order.

To obtain a list of a `Segment`'s children, the agent needs to combine its **internal children** (= `Segments` who belong to the same `Expression` and who are children through the `@parent` attribute) with its **external children** (= `Segments` who belong to other `Expressions` and who are children through the `Inclusion` relation). Together, the internal children and the external children are the **display-time children** of the parent `Segment`.

To put the display-time children in an order, the `@listingOrder` attribute is to be used. But which `@listingOrder`? For internal children, the `@listingOrder` of the `Segment` is to be used. For external children, the `@listingOrder` of the `Inclusion` relation is to be used.

1.6. The `Link` relation type

A `Link` represents the fact that two or more `Segments` are "linked": the actual meaning of the linkage is given in the `Link`'s `@role` attribute. When a software agent is showing a `Segment` to a human user, and when that `Segment` is involved in a `Link` relation with one or more other `Segments`, the agent should provide a clickable hyperlink to go to the other `Segments`.

Attributes

- `@role` (required): the nature of the link. Allowed values are `"synonyms"`, `"antonyms"`, `"nearSynonyms"`, `"opposites"`, `"seeAlso"`, `"variants"`.
- `@members` (required): references to two or more `Segments` which are to be conneced through this `Link`.

Implementing the `@members` attribute

The `@members` attribute is an abstractly defined attribute which can take multiple values: two or more references to `Segments`. To implement this in a specific environment you may need to create more than one "entity". For example, in a relational database, you would need two tables: one which records information about each `Link` (its `@role` plus probably some database-internal ID) and one which lists each `Link`'s member `Segments`.

Uses of the `Link` relation

The `Link` relation is to be used in DMLex to model cross-references from one sense of one entry to another sense of another entry, such as synonymy. Two or more `Segments` -- typically those with `@role="sense"` -- can be connected through a `Link` with `@role="synonyms"`.

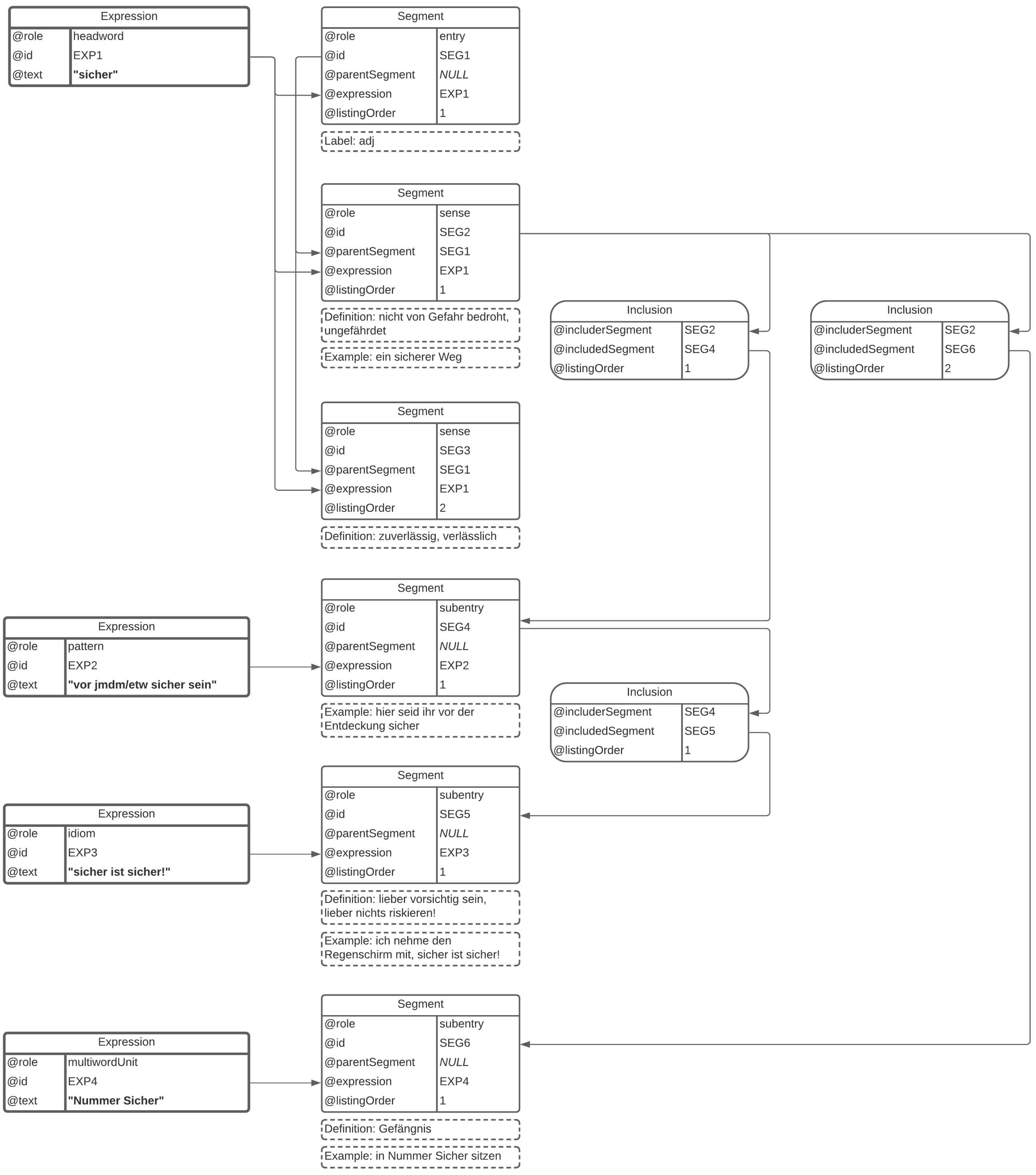
Also, the `Link` relation is to be used in DMLex to model entry-level relations, such as variance between headwords. The "main" headword and the variant headword would exist in the dictionary as separate `Expressions`. Each would have a `Segment` (with `@role="entry"`) while the variant's `Segment` would probaly be very sparce, containing perhaps only its part of speech and having no child `Segments` (no senses). Finally, the `Segments` would be connected through a `Link` with `@role="variants"`.

1.7. Example

To conclude the chapter on entry structure, let's have a look at an example of how the structure of a (non-trivial) dictionary entry could be expressed in DMLex. This entry for sicher comes from a digitised version of *Wörterbuch der deutschen Gegenwartssprache*. Some of the larger entries in this dictionary have a very "branchy" hierarchy of senses, subsenses and subentries, and 'sicher' is one of them. Only the first few senses from 'sicher' are shown here.

```
- entry
    headword: sicher
    pos: adj
    - sense
        definition: nicht von Gefahr bedroht, ungefährdet
        example: ein sicherer Weg
        - subsense
            pattern: vor etw/jmdm sicher sein
            example: hier seid ihr vor der Entdeckung sicher
            - subentry
                idiom: sicher ist sicher!
                definition: lieber vorsichtig sein, lieber nichts riskieren!
                example: ich nehme den Regenschirm mit, sicher ist sicher!
        - subentry
            expression: Nummer Sicher
            definition: Gefängnis
            example: in Nummer Sicher sitzen
    - sense
        definition: zuverlässig, verlässlich
        ...
```

The following diagram shows how the structure of this entry would be expressed in DMLex by means of `Expressions`, `Segments` and `Inclusions` (there is no need for `Links` in this example).



The diagram is close to how DMLex could be implemented in a relational database. For comparison, the following code shows how the same content could be expressed in an XML implementation of DMLex. Notice that some attributes, such as `@listingOrder` and `@parentSegment`, are not present in the XML because they can be inferred from the XML itself (`@listingOrder` from the position of elements, `@parentSegment` by traversing the document hierarchy upwards).

```
<expression role="headword" id="EXP1">
  <text>sicher</text>
  <segment role="entry" id="SEG1">
    <label>adj</label>
    <segment role="sense" id="SEG2">
      <definition>nicht von Gefahr bedroht, ungefährdet</definition>
      <example>ein sicherer Weg</example>
      <includeHere segmentID="SEG4"/>
      <includeHere segmentID="SEG6"/>
    </segment>
    <segment role="sense" id="SEG3">
      <definition>zuverlässig, verlässlich</definition>
    </segment>
  </segment>
</expression>

<expression role="pattern" id="EXP2">
  <text>vor etw/jmdm sicher sein</text>
  <segment role="subentry" id="SEG4">
    <example>hier seid ihr vor der Entdeckung sicher</example>
    <includeHere segmentID="SEG5"/>
  </segment>
</expression>

<expression role="idiom" id="EXP3">
  <text>sicher ist sicher!</text>
  <segment role="subentry" id="SEG5">
    <definition>lieber vorsichtig sein, lieber nichts riskieren!</definition>
    <example>ich nehme den Regenschirm mit, sicher ist sicher!</example>
  </segment>
</expression>

<expression role="idiom" id="EXP4">
  <text>Nummer Sicher</text>
  <segment role="subentry" id="SEG6">
    <definition>Gefängnis</definition>
    <example>in Nummer Sicher sitzen</example>
  </segment>
</expression>
```

2. Entry content

The previous chapter explained how DMLex structures dictionaries into entries, senses and other such units, and how the `Segment` object type serves as an abstract "superclass": entries and senses are understood as special cases of `Segments`.

Each `Segment` is said to "belong" to some `Expression`, and the purpose of the `Segment` is to communicate to human users some useful information about the `Expression`. To fulfill this purpuse, `Segments` contain **informational objects** such as definitions, part-of-speech labels and translations. In this chapter we present an inventory of object types which are available in DMLex to encode these informational properties.

The types for informational objects are defined here not only in terms of their attributes but also in terms the **parents** and **children** they can have. Most object types introduced in this chapter are children of `Segments`. Some object types, in particular `Label`, can also be children of other informational objects.

DMLex allows informational objects to be children of `Segment`, regardless of the kind of `Segment` (entry, sense, ...). But, when implementing DMLex for a particular dictionary, you can constrain this further using DMLex's configuration mechanism, which is decribed later in this document.

As a guide to what follows in this chapter, here is a tree-like sumamry of the informational objects types and how they are allowed to stack inside each other.

```
- Segment
    - Indicator
    - Label
    - Definition
    - Pronunciation
        - Label
    - InflectedForm
        - Label
        - Pronunciation
            - Label
    - ExpressionTranslation
        - Label
        - Pronunciation
            - Label
        - InflectedForm
            - Label
            - Pronunciation
                - Label
    - Example
        - Label
        - ExampleTranslation
            - Label
```

2.1. The `Definition` object type

TBD...

2.2. The `Indicator` object type

TBD...

2.3. The `Label` object type

TBD...

2.4. The `Pronunciation` object type

TBD...

2.5. The `InflectedForm` object type

TBD...

2.6. The `ExpressionTranslation` object type

TBD...

2.7. The `Example` object type

TBD...

2.8. The `ExampleTranslation` object type

TBD...

2.9. Implementing the informational object types

Taking the object types introduced in this chapter and implementing them in a specific environment, such as in a relational database or in an XML Schema, should be mostly straightfoward. The only possible complication are those types which camn have different types of parents. An example is the `Label` type, instances of which can be children of `Segments`, `Pronunciations`, `InflectedForms` and many others.

To implement these types on your application you have, once again, two options: the abstract approach or the concrete approach.

- **The abstract approach:** create just one "type" (i.e. just one database table, or just one XML element name) for all `Labels`, and then distinguish between the parent types by means of a column in the table, or --if in XML -- leave it for the application to figure out by traversing the document hierarchy upwards.
- **The concrete approach:** create separate "types" for the different subtypes of `Label` in your dictionary, for example one for `Labels` when they are children of (certain kinds of) `Segments`, another for `Labels` when they are children of `Definition`, and so on. A good naming convention is to prefix (some of) the parenthood path to type's name, creating names such as `SenseLabel`, `SegmentDefinitionLabel`, and so on.

Both approaches (abstract and concrete) are valid implementations of DMLex. The difference is in the level of abstraction, and consequently in the amount of flexibility. In the abstract approach, the advantage is that you can accommodate different dictionaries in the same schema, but the disadvantage is that any constraints on the entry content must be formulated as business rules and must be enforced at the application level (= outside the schema). In the concrete approach, the disadvantage is that your schema can accommodate only one hard-coded content model, but the advantage is that any constraints on the entry content are part of the schema and therefore enforced directly at the data level (no business rules needed).

3. Configuration

TBD...

4. A recommended implementation as a relational database

Using the abstract approach throughout. TBD...

5. A recommended serialization into XML

Using the abstract approach throughout. TBD...

6. A recommended serialization into RDF

Using the abstract approach throughout. TBD...