.

The lemon cookbook

John McCrae[1], Guadalupe Aguado-de-Cea[2], Paul Buitelaar[3],
Philipp Cimiano[1],Thierry Declerck[4], Asunción Gómez Pérez[2],
Jorge Gracia[2], Laura Hollink[5], Elena Montiel-Ponsoda[2],
Dennis Spohr[1] and Tobias Wunner[3]

[1] CITEC, Universität Bielefeld
[2] Universidad Politécnica de Madrid
[3] DERI, National University of Ireland, Galway
[4] Deutsches Forschungzentrum für künstliche Intelligenz
[5] Delft University of Technology

monnet

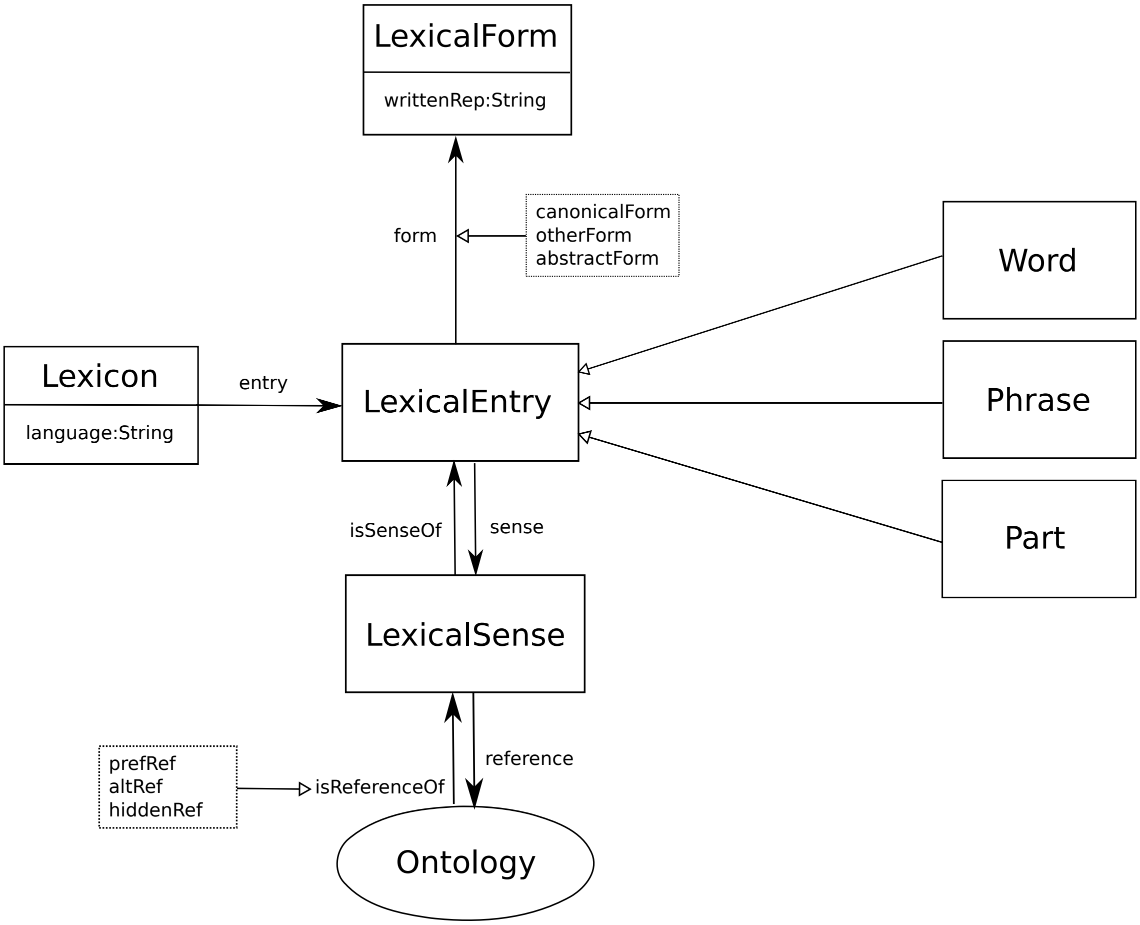
Contents

Introduction

lemon is a model developed in the Monnet project to be a standard for sharing lexical information on the semantic web. lemon draws heavily from earlier work of the Monnet members, in particular from LexInfo (Cimiano et al., 2011), LIR (Montiel-Ponsoda et al., 2008) and LMF (Francopoulo et al., 2006). lemon in contrast to these systems aims to be

- **Concise**: As few classes and definitions as needed.
- **Descriptive not prescriptive**: lemon uses external sources for the majority of its definitions. As such the lemon system can be extended in different ways to handle different purposes, i.e., terminological variation, morpho-syntactic description, translation memory exchange.
- **Modular**: lemon divides into a number of modules, meaning it is not necessary to implement the entire model to create a functional lexicon.
- **RDF-native**: In order to enable sharing on the semantic web, and for interface with tools lemon is based on RDF. This also allows for greater representation of linking between different sections of the lexicon.

1 The lemon core



1.1 Main elements

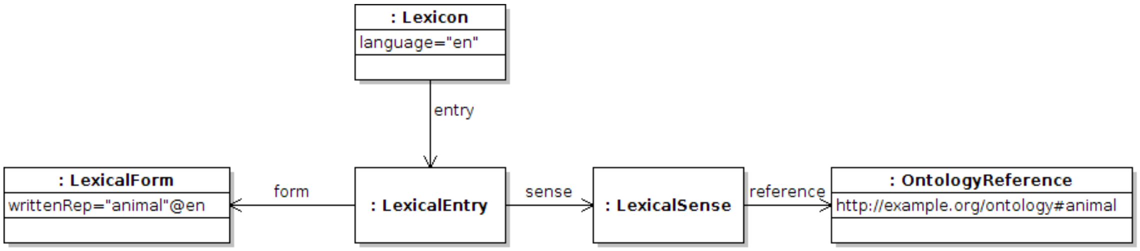
The lemon core module is intended to have a similar expressive power to that of SKOS (Miles and Bechhofer, 2009), while providing distinctions that allow for more powerful linguistic modeling. For example the simplest version of a lemon entry is as follows:[1]

```
@base <http://www.example.org/lexicon>
@prefix ontology: <http://www.example.org/ontology#>
@prefix lemon: <http://www.monnetproject.eu/lemon#>

:myLexicon a lemon:Lexicon ;
   lemon:language "en" ;
   lemon:entry :animal .

:animal a lemon:LexicalEntry ;
   lemon:form [ lemon:writtenRep "animal"@en ] ;
   lemon:sense [ lemon:reference ontology:animal ] .
```

Example 1

Please note, for the course of this book we will use Turtle notation[2] as it is the quickest and most legible way to write lemon (and RDF in general). Furthermore we will mostly omit the type declarations (e.g., `myLexicon` is a `lemon:Lexicon`), as they are generally not necessary and can in all cases be inferred from the RDF Schema document. Also we shall assume in all cases that the prefixes `lemon`, `ontology` and the base prefix are as in the first example. For the sake of completion we will present the above example in RDF/XML.

```
<?xml version="1.0"?>

<rdf:RDF xmlns="http://www.example.org/lexicon#"
     xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
     xmlns:lemon="http://www.monnetproject.eu/lemon#">

  <lemon:Lexicon rdf:about="myLexicon" lemon:language="en">
    <lemon:entry>
      <lemon:LexicalEntry rdf:about="animal">
        <lemon:form rdf:parseType="Resource">
            <lemon:writtenRep xml:lang="en">animal</lemon:writtenRep>
        </lemon:form>
        <lemon:sense rdf:parseType="Resource">
         <lemon:reference rdf:resource="http://www.example.org/ontology#animal"/>
        </lemon:sense>
      </lemon:LexicalEntry>
    </lemon:entry>
  </lemon:Lexicon>
</rdf:RDF>
```



Example 2

This example defines the following entities

1. **Lexicon**: This is the lexicon containing all elements in the lexicon. This approximately corresponds to a SKOS scheme. Each lexicon is marked with a language in the form of an ISO 639 tag.

[1] We include a language tag on the literals even though it can be inferred, for stylistic reasons

[2] `http://www.w3.org/TeamSubmission/turtle/`

2. **Lexical Entry**: This represents the given lexical entry.
3. **Lexical Sense**: Represents the relationship between the lexical entry and the ontology entity.
4. **Reference**: The reference to the resource that can be described by this lexical entry. In the examples above the reference is in a separate ontology, with its own name space.
5. **Form**: A surface realisation of a given lexical entry, typically a written representation.

A second example showing two lexical entries referring to the same ontology entity is as follows:

```
:myLexicon lemon:entry :animal , :creature ;
  lemon:language "en" .

:animal lemon:form [ lemon:writtenRep "animal"@en ] ;
   lemon:sense [ lemon:reference ontology:animal ] .

:creature lemon:form [ lemon:writtenRep "creature"@en ] ;
   lemon:sense [ lemon:reference ontology:animal ] .
```

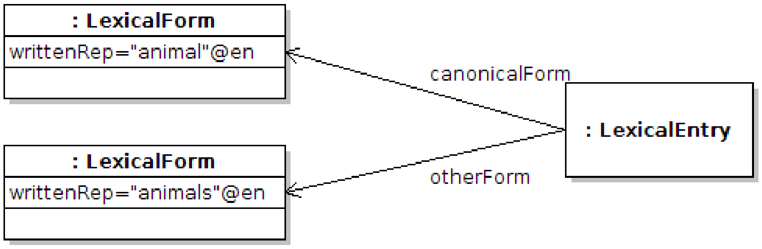
Example 3

This indicates that “animal” and “creature” can refer to the same ontology concept and as such they can be considered to be “synonymous” in a sense. The semantics of the lemon model are further discussed in section 3.2.

1.2 Canonical forms and preferred lexicalizations

lemon allows syntactic variants to be differentiated from preferred forms by sub-properties of `form`.

```
:animal lemon:canonicalForm [ lemon:writtenRep "animal"@en ] ;
   lemon:otherForm [ lemon:writtenRep "animals"@en ] .
```



Example 4

This allows lemon to differentiate between different forms of a word and different words. For example consider we have two labels for an ontology entity, “animal” and “creature”. This is modeled in lemon as follows:

```
:animal lemon:canonicalForm [ lemon:writtenRep "animal"@en ] ;
   lemon:otherForm [ lemon:writtenRep "animals"@en ] ;
   lemon:sense [ lemon:reference ontology:animal ] .

:creature lemon:canonicalForm [ lemon:writtenRep "creature"@en ] ;
   lemon:otherForm [ lemon:writtenRep "creatures"@en ] ;
   lemon:sense [ lemon:reference ontology:animal ] .
```

Example 5

It is also possible to state the lexicon-ontology relationship in the reverse direction because `reference` and `sense` have inverse properties `isReferenceOf` and `isSenseOf`. This allows example 5 to be stated as follows

```
ontology:animal
   lemon:isReferenceOf [
      lemon:isSenseOf [
         lemon:canonicalForm [ lemon:writtenRep "animal"@en ] ;
         lemon:otherForm [ lemon:writtenRep "animals"@en ]
      ]
   ] ;
   lemon:isReferenceOf [
      lemon:isSenseOf [
         lemon:canonicalForm [ lemon:writtenRep "creature"@en ] ;
         lemon:otherForm [ lemon:writtenRep "creatures"@en ]
      ]
   ] .
```

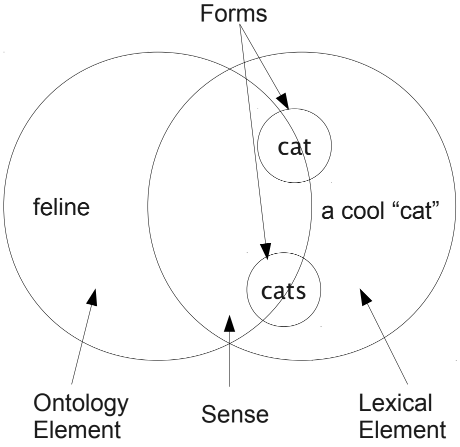
Example 6

It is also possible to state the lexicon without any senses or references and then introduce an ontology mapping layer by creating links such as

```
:animal_sense lemon:isSenseOf :animal ;
   lemon:reference ontology:animal .
```

Example 7

In lemon we assume that each lexical entry is not semantically disambiguated, and that the reference provides the semantics of the term. We introduce sense to represent those occurrences when the lexical entry is used with the given meaning. As such it is assumed that "feline" and "cat" would not share a sense, even though they can be considered as synonyms. Similarly "he is a cool cat" and "cats are mammals" are assumed to have the same lexical entry as they exhibit the same morphological/syntactic behaviors. This is summarized in the following diagram



One of the most important aspects of lemon is that *senses should be unique to a given lexical entry/ontology reference pair*, this means that "creature" and "animal", should not refer to the same sense entity, but can be related using the `equivalent` property. If two lexical entries do share a sense, then it is assumed that they are lexically equivalent, which may be appropriate for example for an initialism or acronym and its full form. Similarly it should be understood that if a sense has two references then these references are equivalent (for example by OWL's `equivalentClass`) property. For more details of this see sections **??**, 3.2 and 4.1.

In lemon for each lexically different element, a different lexical entry and sense is used. This means that both lexical entries are considered possible representations of the ontology entity.

Also like SKOS, we allow the inclusion of partial terms, which we refer to as "abstract", this is useful for representing stems, affices and other morphological units. These are implemented by three sub-properties of `form` can also be used to describe linguistically relevant differences.

- `canonicalForm`: The standard ("dictionary") citation form the entry.
- `otherForm`: A morphological variant of the words.
- `abstractForm`: A stem or other non-realizable morpheme, or other non-realizable forms.

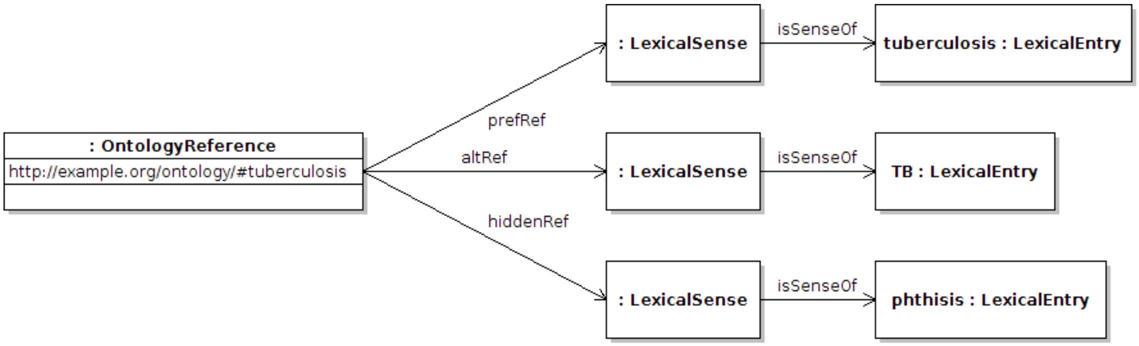
As lemon defines a form as being invariant across different orthographies, different spellings of a word are represented by deriving a sub-property of `representation`. Here it is important to include the `xml:lang` tag to indicate the particular usage of a given term. For example the representation of "color" in US English spelling and "colour" in British English spelling.

```
:color lemon:canonicalForm [
    lemon:writtenRep "color"@en-us ;
    lemon:writtenRep "colour"@en-gb ] .
```

Example 8

It is also important to note that SKOS's `prefLabel`, `altLabel` and `hiddenLabel` do not distinguish between syntactic preference (like `canonicalForm` etc) and pragmatic preference, that is whether the term is preferred for terminological reasons. To cover this pragmatic preference lemon has three sub-properties of `isReferenceOf`, that cover this, namely, `prefRef`, `altRef`, `hiddenRef`. `prefRef` represents the preferred term of an ontology reference (there should be only one such entry), `hiddenRef` represents a term that is not used for various reason (for example it is antiquated) and `altRef` represents any other term. For example the following shows "tuberculosis", with an alternative "TB" and an antiquated term "phthisis".

```
ontology:tuberculosis
   lemon:prefRef [
      lemon:isSenseOf [ :tuberculosis ]
   ] ;
   lemon:altRef [
      lemon:isSenseOf [ :tb ]
   ] ;
   lemon:hiddenRef [
      lemon:isSenseOf [ :phthisis ]
   ] .
```



Example 9

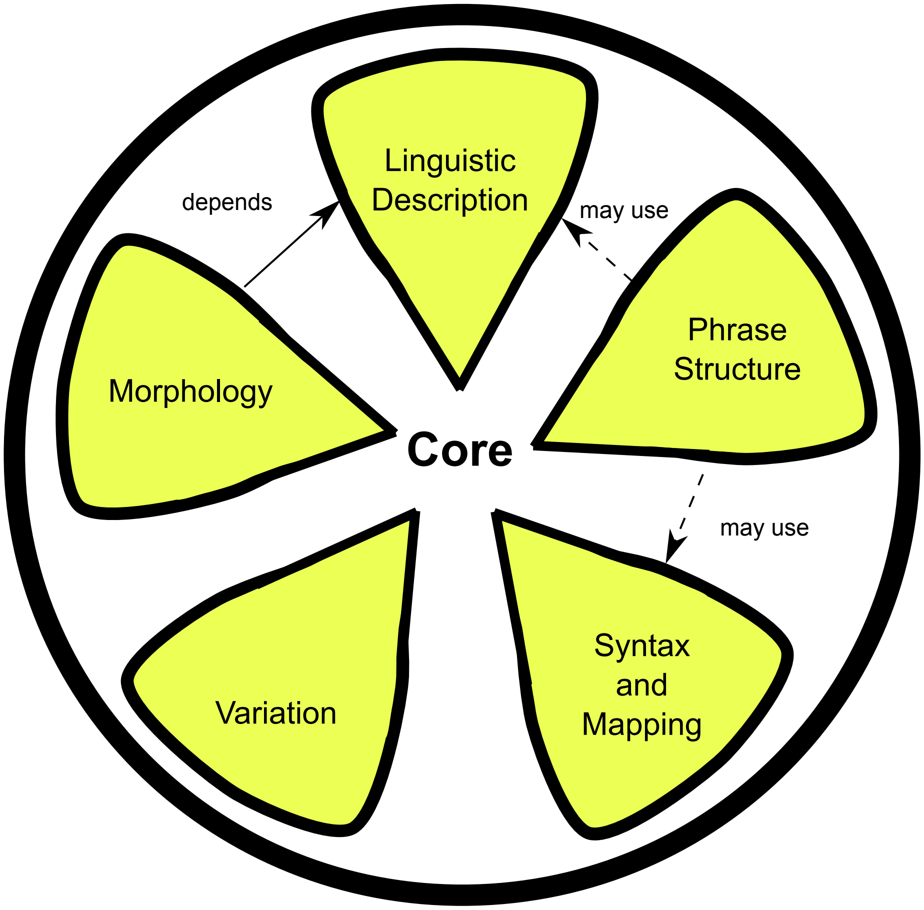
Between the sub-properties of `form` and `isReferenceOf` we can more precisely capture the same semantics as SKOS's `prefLabel`, `altLabel` and `hiddenLabel`. The conversion is as follows:

	Canonical Form	Other Form	Abstract Form
Preferred Reference of	`prefLabel`	`altLabel`	`hiddenLabel`
Alternative Reference of	`altLabel`	`altLabel`	`hiddenLabel`
Hidden Reference of	`hiddenLabel`	`hiddenLabel`	`hiddenLabel`

Summary of vocabulary introduced in this section

lemon element	Description
`entry`	States that a lexical entry is in a given lexicon
`form`	States a realization of a lexical entry
↳`canonicalForm`	Gives the dictionary form of a lexical entry
↳`otherForm`	Gives an alternative form of a lexical entry (Inflectional variant)
↳`abstractForm`	Gives a non-display form of a lexical entry (Non-realizable variant)
`language`	The language of a lexicon, specified as an ISO 639 code.
`sense`	Specifies the lexical entry a sense refers to
`reference`	Specifies a connection from a sense to a resource
`isReferenceOf`	Specifies the sense referred to by an ontology reference. Inverse of `reference`
↳`prefSem`	Indicates the preferred entry and sense for an ontology reference.
↳`altSem`	Indicates an non-preferred but valid entry and sense for an ontology reference.
↳`hiddenSem`	Indicates a generally incorrect entry and sense for an ontology reference.
`isSenseOf`	Indicates the lexical entry for a given sense. Inverse of `sense`.
`representation`	Specifies a specific representation of a form
↳`writtenRep`	Specifies a specific written representation of a form
`Word`	Indicates the lexical entry is a word
`Phrase`	Indicates the lexical entry is composed of multiple words
`Part`	Indicates the lexical entry is a part of a word (e.g., an affix)

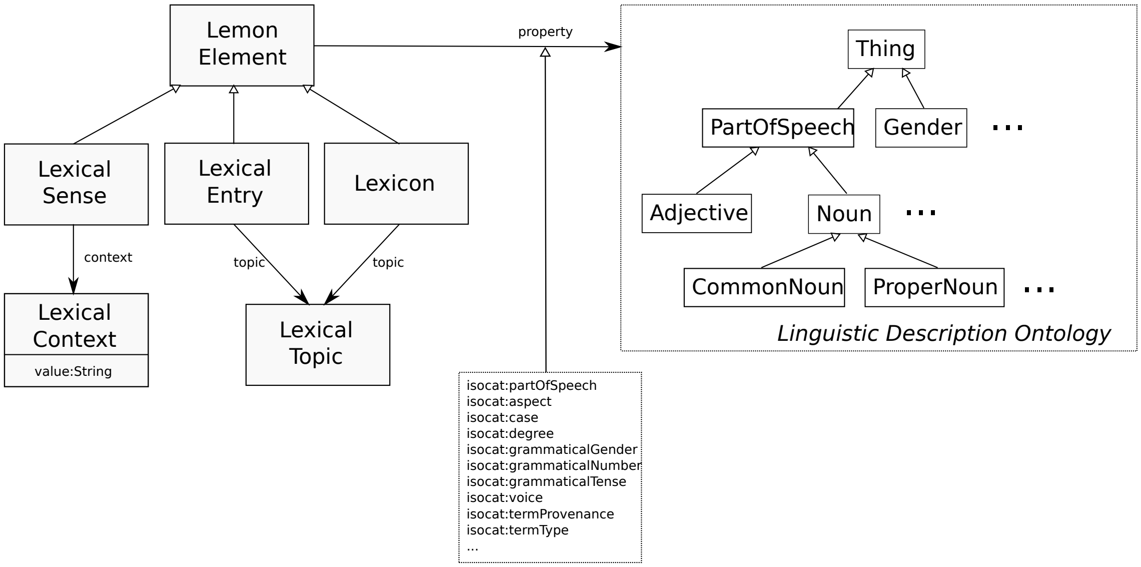
2 Modules



The core of lemon represents a sufficient model for representing a simple lexicon, however for most lexica more sophisticated description is required. This is provided by the following modules:

- **Linguistic Description**: This allows linguistic properties, that are commonly found in dictionaries, may be added to elements in the lexicon
- **Variation**: This allows relationships between different elements of a lexicon to be described
- **Phrase structure**: This describes the representation of multiple word expressions within lexica
- **Syntax and mapping**: This concerns the representation of syntactic frames and mapping these frames to logical predicates in an ontology
- **Morphology**: This module provides compact mapping for a inflected and agglutinative forms of entries.

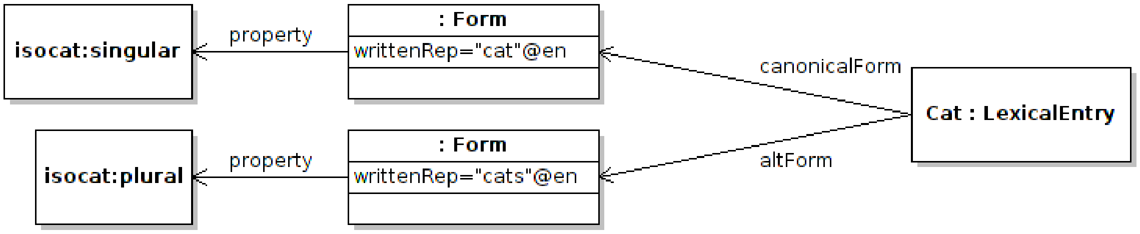
2.1 Linguistic Description Module



2.1.1 Linguistic properties

One of the key challenges that lemon attempts to solve is the attachment of linguistic information to an ontology, which is essential to providing linguistic annotations to the text. This is achieved by using a linguistic taxonomy also known as a *linguistic description ontology* or *data category registry* (Romary, 2010). Examples of these resources include the GOLD ontology or the ISOcat data category registry. For example, indicating that “cat” has a singular canonical form “cat” and a plural form “cats” can be done as follows.[3]

```
:cat lemon:canonicalForm [ lemon:writtenRep "cat"@en ;
                           lemon:property isocat:singular ] .
:cat lemon:otherForm [ lemon:writtenRep "cats"@en ;
                       lemon:property isocat:plural ] .
```



Example 10

In practice it is neither feasible nor desirable to impose a complete set of all possible annotations, thus lemon does not attempt to do so, and is intended to be used with a source of linguistic categories. Also lemon is used to define the structure and relations of the lexicon, and properties from data categories are introduced by asserting them as sub-properties of existing lemon properties (in particular `lemon:property`). For example we will now show an example that marks “ICBM” as a noun and an initialism, using sub-properties of lemon’s property[4]

```
@prefix rdfs: <"http://www.w3.org/2000/01/rdf-schema#">

:ICBM isocat:partOfSpeech isocat:noun ;
```

[3] We base examples on ISOcat and use the “identifier” property as the basis of the URI, so we have the readable description `isocat:noun` instead of `isocat:DC-1333`

[4] We introduce the RDFS name space with its usual form.

```
      isocat:termType isocat:initialism .

isocat:partOfSpeech rdfs:subPropertyOf lemon:property .
isocat:termType rdfs:subPropertyOf lemon:property .
```

Example 11

These sub-property declarations only need to be made once per lexicon. An alternative is to use a controlled vocabulary for lemon, such as LexInfo 2[5] which defines a practical set of data categories for general NLP tasks.

It is of course possible to define a lexical entry or form as having several linguistic properties. For example, in many cases we might wish to introduce this set of data categories again from a source such as ISOcat or GOLD. The use of multiple properties makes it much simpler for applications to query different properties.

```
:eat lemon:otherForm [ lemon:writtenRep "eats"@en ;
                     isocat:person isocat:thirdPerson ;
                     isocat:grammaticalNumber isocat:singular ;
                     isocat:tense isocat:present ] .

isocat:person  rdfs:subPropertyOf lemon:property .
isocat:grammaticalNumber rdfs:subPropertyOf lemon:property .
isocat:tense rdfs:subPropertyOf lemon:property .
```

Example 12

It may also be used to define the linguistic properties of forms, e.g. to indicate whether they are roots or stems etc. For example, the Spanish verb “pescar” has alternative stems “pesc-” and “pesqu-” that may be useful for generating inflectional variants.

```
:pescar lemon:abstractForm
      [ lemon:writtenRep "pesc"@es ;
        isocat:morphologicalUnit isocat:stem ] ;
   lemon:abstractForm
      [ lemon:writtenRep "pesqu"@es ;
        isocat:morphologicalUnit isocat:stem ] .
```

Example 13

Morphology can also be partly handled by assigning the morphological pattern as a type of linguistic annotation. For example the Latin verb “amare” may be stated to have the “first conjugation” morphological pattern as such.

```
:amare isocat:morphologicalPattern :first_conjugation .
```

Example 14

This means that it is possible to use this modelling to avoid stating all inflected forms of a word. The implication for implementation however should still be that stated forms override forms generated from a morphological pattern. This is useful as for example the verb “speak” has a regular third person singular present form “speaks” but irregular simple past and past participle forms “spoke” and “spoken”.

2.1.2 Describing phonetics

For representing transliterations and pronounciations, extra subproperties of `representation` (the super-property of `writtenRep`) can be introduced. For example we model the Japanese word “日本語”, which can be transliterated to either “にほんご” in the phonetic Japanese script Hiragana or “nihongo” in the Latin alphabet

[5] http://www.lexinfo.net/ontology/2.0/lexinfo

```
:nihongo lemon:canonicalForm [
 lemon:writtenRep "日本語"@ja-Jpan ;
    isocat:transliteration "にほんご"@ja-Hira ;
   isocat:transliteration "nihongo"@ja-Latn ] .

isocat:transliteration rdfs:subPropertyOf lemon:representation .
```

Example 15

This sub-property relation also then allows us to develop specific versions of transliteration, for example “神保町” is transliterated as “jimbōchō” in the *Hepburn* romanization scheme (the most widely used method for transliterating Japanese) and “zinbôtyô” in the *Kunrei-shiki* (ISO 3602) romanization scheme. This could be represented as follows.

```
:jimbocho lemon:canonicalForm [
 lemon:writtenRep "神保町"@ja-Jpan ,
     :hepburnTransliteration "jimbōchō"@ja-Latn ;
     :kunreiTransliteration "zinbôtyô"@ja-Latn ] .

:hepburnTransliteration rdfs:subPropertyOf isocat:transliteration .
:kunreiTransliteration rdfs:subPropertyOf isocat:transliteration .
```

Example 16

2.1.3 Topics and contexts

As the size of a lemon lexicon grows it is necessary to manage and organize it according to some principles. One of the first steps to this is the lexicon object, which allows vocabulary to be grouped by creating multiple lexicon objects. This is similar to the role of schemes in SKOS, which allow different topics to be grouped. As such lemon provides `topic` to indicate the topic that is shared between all lexical entries. For example the following shows two lexicons one for animals and one for veterinary anatomy

```
:animal_lexicon lemon:topic :animals ;
                lemon:entry :cat , :mongoose , :seahorse .

:anatomy_lexicon lemon:topic :veterinary_anatomy ;
                 lemon:entry :lung , :fin , :swim_bladder .
```

Example 17

Note that the object of `topic` is again a URI value, which presupposes a set of defined topics (i.e., a topic ontology). It is often the case that this does not exist or for some reason it is better to represent topics with literal strings. In such cases, lemon’s `value` property can be used. Hence the above example can be rewritten as:

```
:animal_lexicon lemon:topic [ lemon:value "animals" ] ;
                lemon:entry :cat , :mongoose , :seahorse .

:anatomy_lexicon lemon:topic [ lemon:value "veterinary anatomy" ] ;
                 lemon:entry :lung , :fin , :swim_bladder .
```

Example 18

It is not recommended that this pattern is used if topics are reused across multiple lexicon objects, as it is more difficult to group lexicons on the same topic and using a string may cause spelling issues.

The lemon `topic` can also be applied to lexical entries to give a term level description of the topic of an entry. Of course, as with all RDF systems, many topics can be attached to a single entry. For example

```
:cat lemon:topic :domestic_animals , :cosmopolitan_species , :felidae_family .
```

Example 19

It is important not to confuse the `topic` of a lexical entry with the `context` of its senses.

Another important feature lemon has for terminological management is the ability to add definitions with the `definition` property. For this the `value` property must always be used to state the value of the definition. For example

```
:cat lemon:sense [
  lemon:reference ontology:cat ;
  lemon:definition [
    lemon:value "The cat is a small domesticated carnivorous animal"@en
  ]
] .
```

Example 20

These definitions are generally intended to be used primarily in terminology management rather than for NLP systems.

In addition a particular meaning of a term may be described with a context, which refers to pragmatic/discourse restrictions on the mapping that may be implied by either the linguistic context. An example of a pragmatic context is given here by the usage of the term “cat” in general discourse and “Felis Catus” in scientific discourse.

```
:felis_catus lemon:sense [ lemon:context isocat:technicalRegister ;
                           lemon:reference ontology:Cat ] .

:cat lemon:sense [ lemon:context isocat:neutralRegister ;
                   lemon:reference ontology:Cat ] .
```

Example 21

In addition contexts are useful for indicating temporal or geographic usage, and this can be done by introducing subproperties to indicate these values. For example we may indicate that the term “blog” has been used since 1998 as follows:

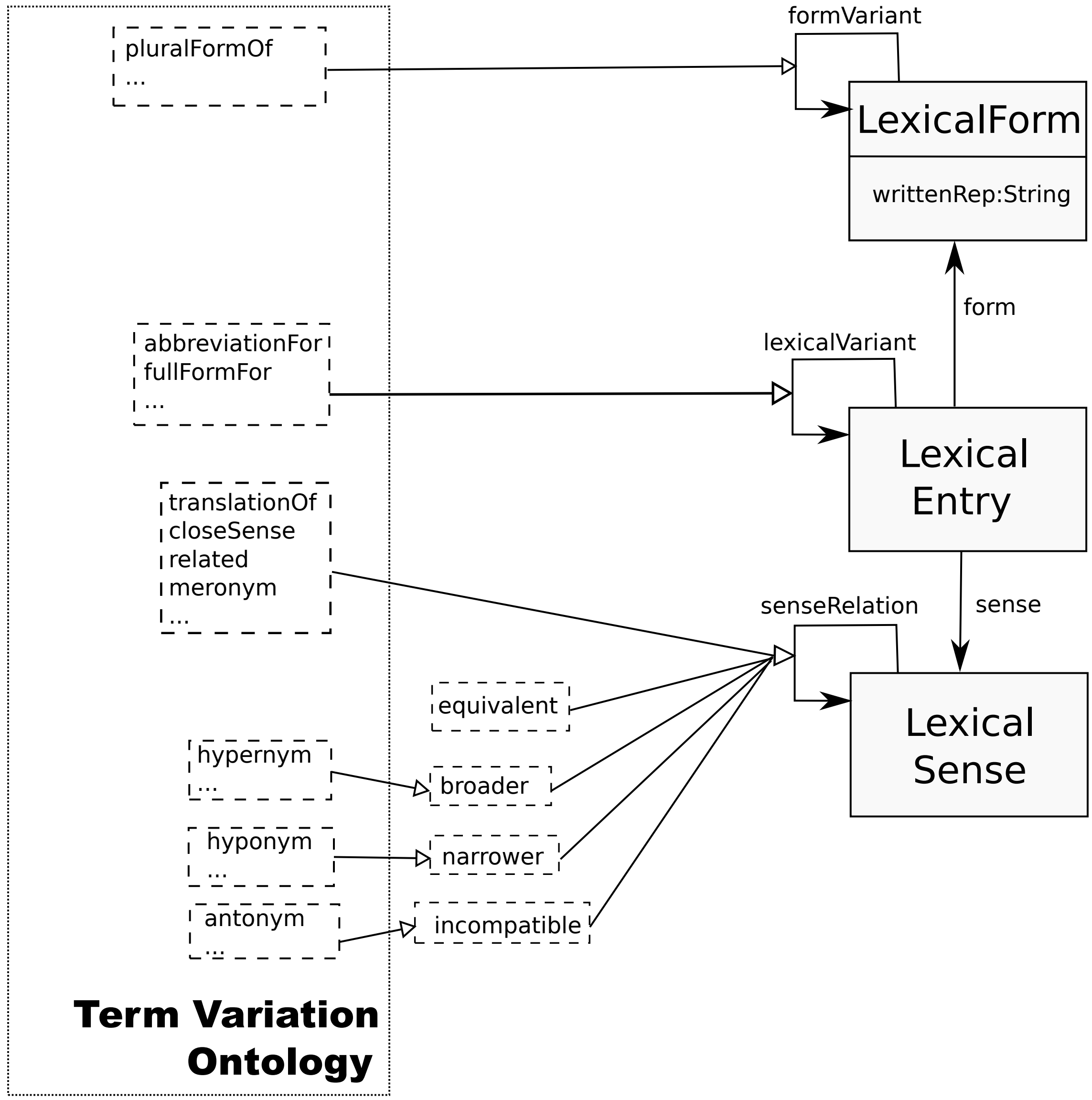
```
:blog lemon:canonicalForm [
    lemon:writtenRep "blog"@en ]
  lemon:sense [
    lemon:reference foaf:weblog ;
    :usedSince [ lemon:value "1998"^^xsd:gYear ] ].

:usedSince rdfs:subPropertyOf lemon:context .
```

Example 22

Summary of vocabulary introduced in this section

Property	Description
`property`	Provides a linguistic annotation. If the subject of this property is a lexical entry this refers to all instances of the lexical entry. If the subject is a form, then it describes the inflection of the given form. If the subject is a component it describes the morphological property of this component within a multi-word expression.
`topic`	Indicates the topic of a lexicon or lexical entry.
`context`	Indicates the context in which a given lexical sense is used

2.2 Variation Module



The variation module is used to describe relationships between objects in a lemon lexicon. This is done primarily by three relations `senseRelation`, `lexicalVariant` and `formVariant`, which describe relationships between senses, entries and forms respectively.

2.2.1 Lexicosemantic relationships

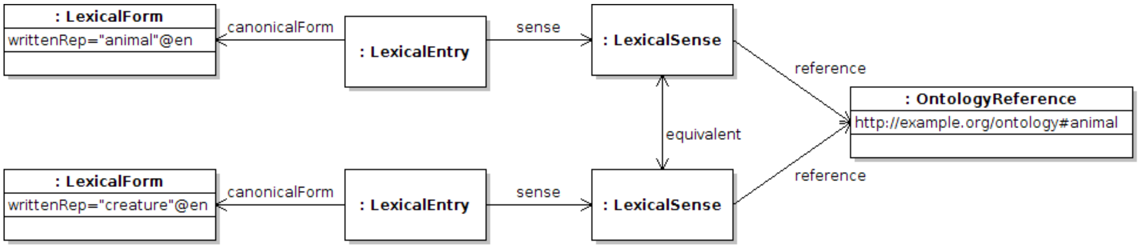
It is possible to relate the sense objects using the predicate `lemon:senseRelation`. This has the following sub-properties defined in lemon:

- `equivalent`
- `incompatible`
- `broader`
- `narrower`[6]

These can be used to state the similarity of the terms "animal" and "creature" as follows

[6] Like SKOS, `broader` and `narrower` are not transitive

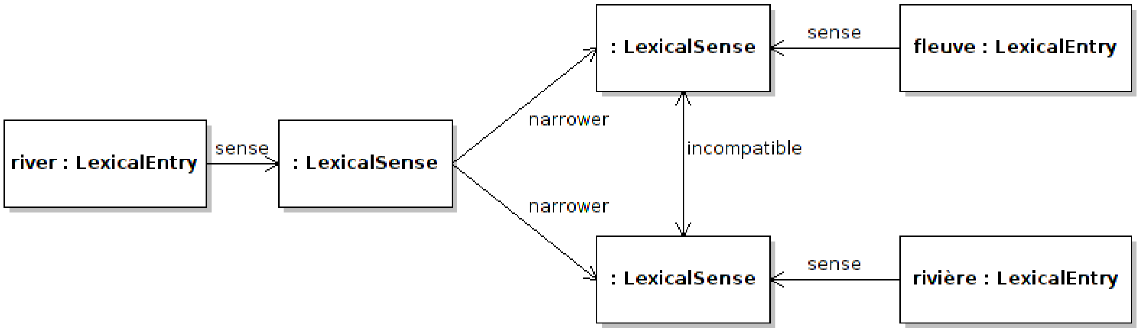
```
:animal lemon:canonicalForm [ lemon:writtenRep "animal"@en ] ;
   lemon:sense :animal_sense .

:animal_sense lemon:reference ontology:animal ;
   lemon:equivalent :creature_sense .

:creature lemon:canonicalForm [ lemon:writtenRep "creature"@en ] ;
   lemon:sense :creature_sense .

:creature_sense lemon:reference ontology:animal ;
   lemon:equivalent :animal_sense .
```



Example 23

For a further example of the usefulness of these similarities consider the case of the French words “rivière” and “fleuve”, these refer to rivers that flow into other rivers and the sea respectively. As such this distinction can be modeled by the following sub-graph.

```
:fleuve lemon:sense :fleuve_sense .
:riviere lemon:sense :riviere_sense .
:river lemon:sense :river_sense .

:fleuve_sense lemon:incompatible :riviere_sense .

:riviere_sense lemon:incompatible :fleuve_sense .

:river_sense lemon:narrower :riviere_sense ,
                            :fleuve_sense .
```



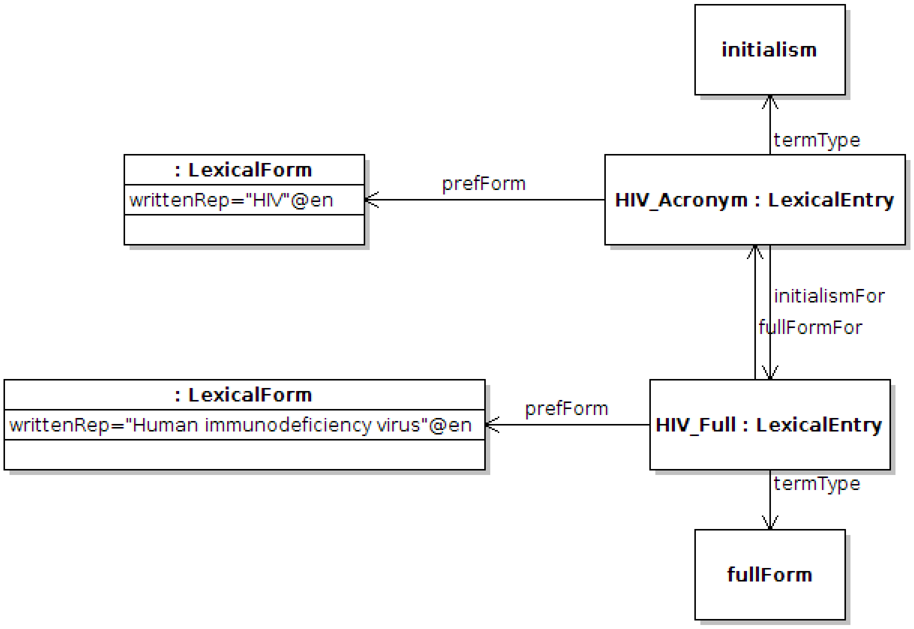
Example 24

2.2.2 Lexical variants

Another important factor of lemon is the ability to model the differences between various words and forms of a word. This is performed by the properties `lexicalVariant`, `formVariant` and `senseRelation`. Again here we handle the large number of potential annotations by referring to a data category registry to define these relations.[7]

[7] Note, we make the technical distinction here between an “initialism” (e.g., USA) and a “acronym” (e.g., NASA)

```
:hiv lemon:canonicalForm [ lemon:writtenRep "HIV"@en ] ;
   isocat:termType isocat:initialism ;
   isocat:initialismFor :human_immunodeficiency_virus .

:human_immunodeficiency_virus
   lemon:canonicalForm [ lemon:writtenRep "Human immunodeficiency virus"@en ] ;
   isocat:termType isocat:fullForm ;
   isocat:fullFormFor :hiv .

isocat:initialismFor rdfs:subPropertyOf lemon:lexicalVariant .
isocat:fullFormFor rdfs:subPropertyOf lemon:lexicalVariant .
```



Example 25

It is also possible to state lexical entries to be syntactic variants of another, if one is somehow derived from another. For example, the words “lexicon”, “lexical” and “lexicalize” are derived from a single root. This is done by means of the `lexicalVariant` relation. E.g.,

```
:lexicon :adjectivalVariant :lexical ;
   :verbalVariant :lexicalize .

:adjectivalVariant rdfs:subPropertyOf lemon:lexicalVariant .
:verbalVariant rdfs:subPropertyOf lemon:lexicalVariant .
```

Example 26

2.2.3 Subphrases as variation

One particular modelling that should be performed as subphrase variation is indicating subphrases. For example we may indicate that “New York” is a subphrase of “New York City” by introducing a property `subphrase` as a subproperty of lexical variant. We may also use this modelling to indicate some syntactic properties such as the head of a phrase.

```
:new_york_city lemon:canonicalForm [ lemon:writtenRep "New York City"@en ] ;
   :subphrase :new_york ;
   :head :city .
```

```
:new_york lemon:canonicalForm [ lemon:writtenRep "New York"@en ] .

:city lemon:canonicalForm [ lemon:writtenRep "City"@en ] .

:subphrase rdfs:subPropertyOf lemon:lexicalVariant .
:head rdfs:subPropertyOf lemon:lexicalVariant .
```

Example 27

Note that this modelling is not intended to be used for deducing the syntactic structure of a multiple word expression, as such it is not easy to deduce which words compose the sub-phrase. Modelling that can provide this information is described in section 4.

2.2.4 Form variants

As we have already seen `senseRelation` to relate different senses of a lexical entry, and `lexicalVariant` to relate different lexical entries, we will also introduce a third property `formVariant`, which relates different forms of the same lexical entries. This property can often be ignored, as the preferred way to represent form variants is by attaching an appropriate set of linguistic properties. However an alternative way to represent the entry "animal" with a plural "animals" is using a `formVariant` link.

```
:animal lemon:canonicalForm :animal_sing_form ;
   lemon:otherForm :animal_plural_form .

:animal_sing_form lemon:writtenRep "animal"@en ;
   :pluralFormOf :animal_plural_form .

:animal_plural_form lemon:writtenRep "animals"@en .

:pluralFormOf rdfs:subPropertyOf lemon:formVariant .
```

Example 28

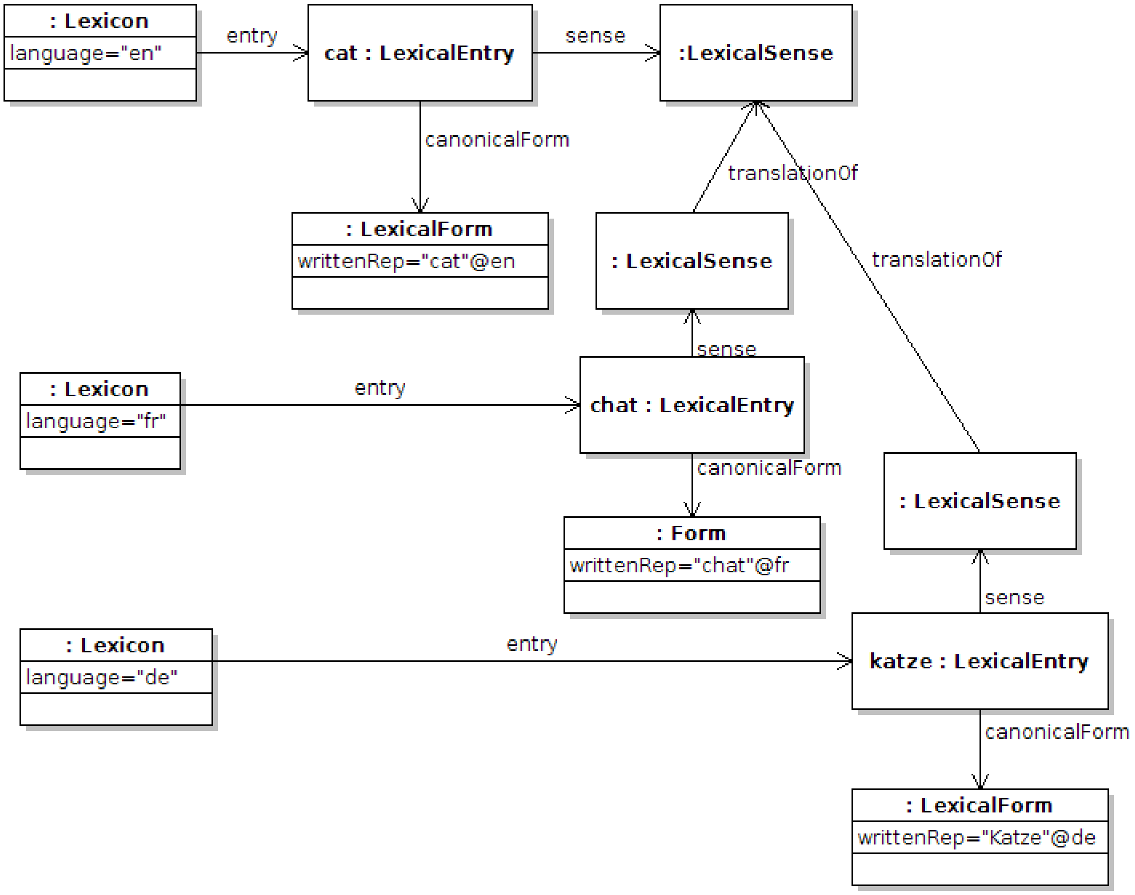
We leave it to the implementation of software using the lemon model to choose which method to use, however we observe that the property/value modelling is generally more compact than form relations.

2.2.5 Translation as variation

Translated variants are for the most part naturally included within the lemon model by the use of references, as translated variants should share the same reference. Hence the ontology should act as an intermediate layer linking lexical entries of different languages. However, in some cases the reference may not be available, or it may be of interest to mark lexical entries as translations if they do not share the same reference. In such cases, sub-properties of `senseRelation` can be used. It should be clear that this translation link is between senses of words as the term should be disambiguated, even if there is not a clear reference for the translation. For example, we will consider the case of "cat" translated into German, and French as "Katze" and "chat" respectively.

```
:lexicon_en lemon:entry :cat ;
  lemon:language "en" .
:lexicon_de lemon:entry :katze ;
  lemon:language "de".
:lexicon_fr lemon:entry :chat ;
  lemon:language "fr".

:cat lemon:canonicalForm [ lemon:writtenRep "cat"@en ] ;
  lemon:sense :cat_sense .
```

```
:chat lemon:canonicalForm [ lemon:writtenRep "chat"@fr ] ;
  lemon:sense [ isocat:translationOf :cat_sense ] .

:katze lemon:canonicalForm [ lemon:writtenRep "katze"@de ] ;
  lemon:sense [ isocat:translationOf :cat_sense ] .

isocat:translationOf rdfs:subPropertyOf lemon:senseRelation .
```

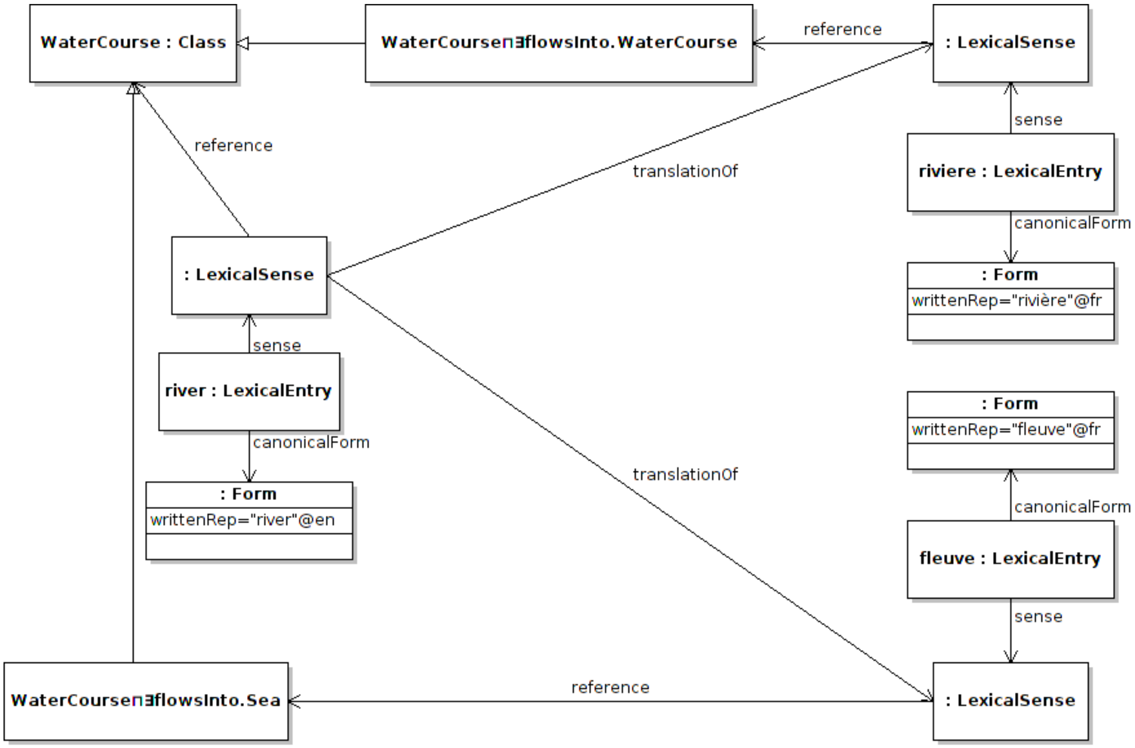


Example 29

As can be seen the translation arc does not in itself state the language pairs in the translation, however the language can be extracted from either the lexica containing the term or from the xml:lang special property.

Modeling translation as variation may be particularly useful if you wish to indicate that a word is a translation of another word. For example, consider the French words “rivière” and “fleuve” which are both translations of the English word “river”. However following the definitions of these words (these are given in section 2.4.7) it may be sensible to map the French words to the anonymous subclasses WaterCourse⊓∃flowsInto.WaterCourse and WaterCourse⊓∃flowsInto.Sea respectively, hence we get the following modeling:

```
:river lemon:canonicalForm [ lemon:writtenRep "river"@en ] ;
  lemon:sense :river_sense .
:river_sense lemon:reference ontology:WaterCourse .

:riviere lemon:canonicalForm [ lemon:writtenRep "rivière"@fr ] ;
  lemon:sense [ lemon:reference [ a owl:Class
      owl:equivalentTo [ owl:intersectionOf (
        ontology:WaterCourse ;
        owl:Restriction [
           owl:onProperty ontology:flowsInto ;
           owl:allValuesForm ontology:WaterCourse
        ]
     )
  isocat:translationOf :river_sense
]]] .

:fleuve lemon:canonicalForm [ lemon:writtenRep "fleuve"@fr ] ;
  lemon:sense [ lemon:reference [ a owl:Class
      owl:equivalentTo [ owl:intersectionOf (
        ontology:WaterCourse ;
        owl:Restriction [
           owl:onProperty ontology:flowsInto ;
           owl:allValuesForm ontology:Sea
        ]
     )
  isocat:translationOf :river_sense
]]] .
```

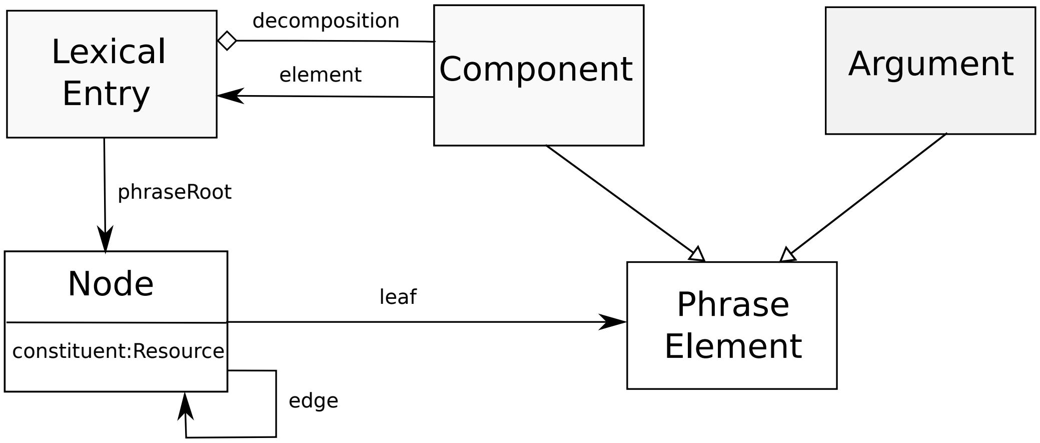


Example 30

Summary of vocabulary introduced in this section

Property	Description
`formVariant`	Describes a variant (inflected form) of a written representation of a lexical entry
`lexicalVariant`	Describes a variant or derived form of a lexical entry
`senseRelation`	States a relations between two senses
`↳equivalent`	States two senses are equivalent (symmetric)
`↳incompatible`	States two senses are disjoint (symmetric)
`↳broader`	States one sense is broader than another. Inverse of `narrower`
`↳narrower`	States one sense is narrower than another. Inverser of `broader`

2.3 Phrase Structure Module



2.3.1 Decomposition of terms

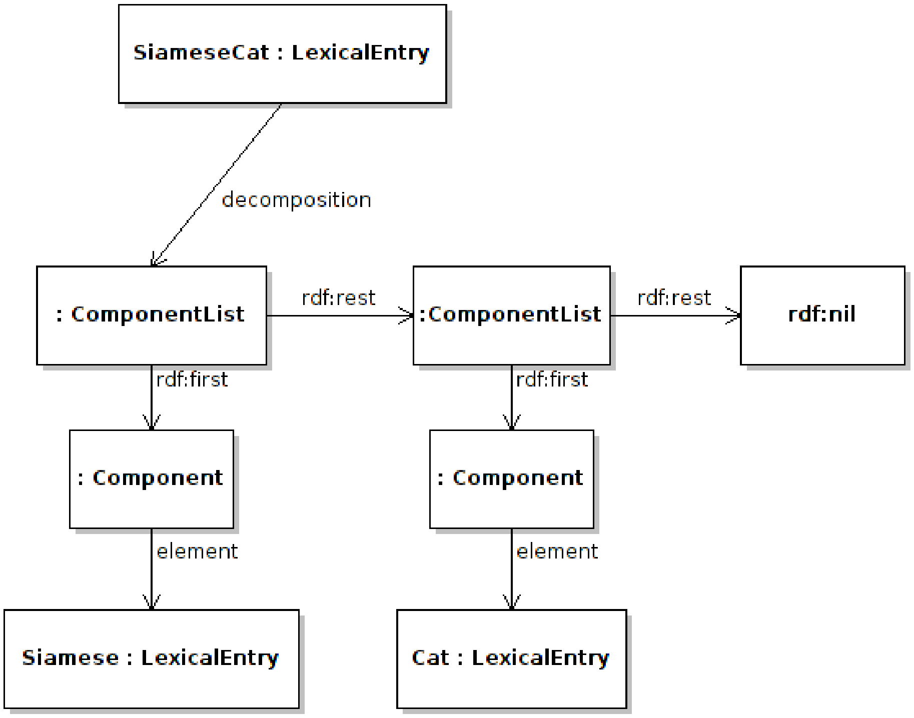
The base method for representing multi-word lexical entries is by decomposing them into their component words, this is done through the usage of RDF lists of component objects. This works as follows.

```
:siamese_cat lemon:decomposition (
   [ lemon:element :siamese ]
   [ lemon:element :cat ] ) .
```

Example 31

Note that the above example uses the RDF list mechanism, which, while relatively compact in Turtle syntax, can get quite large in N-Triples form, e.g.,

```
:siamese_cat lemon:decomposition :siamese_cat_LoC .
:siamese_cat_LoC rdf:first :siamese_cat_C1 .
:siamese_cat_LoC rdf:rest :cat_LoC .
:siamese_cat_C1 lemon:element :siamese .
:cat_LoC rdf:first :siamese_cat_C2 .
:cat_LoC rdf:rest rdf:nil .
:siamese_cat_C2 lemon:element :cat .
```



Example 32

Compound words may also be broken up by the use of components for example, the German word “Schweineschnitzel” is composed of “Schwein” and “Schnitzel”.[8]

```
:schweineschnitzel lemon:decomposition (
   [ lemon:element :schwein ]
   [ lemon:element :schnitzel ] ) .
```

Example 33

lemon indicates the difference between compound words and phrases by the use of the classes `Word` and `Phrase`. As such it is possible to indicate that the composition is a separation of a multi-word expression into words, by adding type statements for the elements.

```
:siamese_cat a lemon:Phrase ;
   lemon:decomposition (
      [ lemon:element :siamese ]
      [ lemon:element :cat ] ) .

:siamese a lemon:Word .
:cat a lemon:Word .
```

Example 34

A third subclass of lexical entry, `Part`, is provided for “part of words”, e.g., affices, which cannot be realised by themselves but may be stored in the lexicon. For example:

```
:preordain lemon:decomposition (
   [ lemon:element :pre ]
   [ lemon:element :ordain ] ) .

:pre a lemon:Part .
:ordain a lemon:Word .
```

Example 35

We must note here the `decomposition` property is not intended for modelling phrase structure, lemon contains a separate mechanism for modelling this described in sections 2 and 4.

2.3.2 Phrase structures

The multi-word expression extensions of lemon are intended to model phrase structures of multi-word expressions. These are done through the use of the properties `phraseRoot`, `edge` and `leaf`, which allows arbitrary graphs to be created and related. For example, the decomposition of “human immunodeficiency virus” into “[human [immunodeficiency [virus]]]”, can be represented as follows.

```
:human_immunodeficiency_virus
  lemon:decomposition ( :human_component
                        :immunodeficiency_component
                        :virus_component ) ;
  lemon:phraseRoot [
      lemon:edge [ lemon:leaf :human_component ] ;
      lemon:edge [
        lemon:edge [ lemon:leaf :immunodeficiency_component ]
        lemon:edge [
          lemon:edge [ lemon:leaf :virus_component ]
```

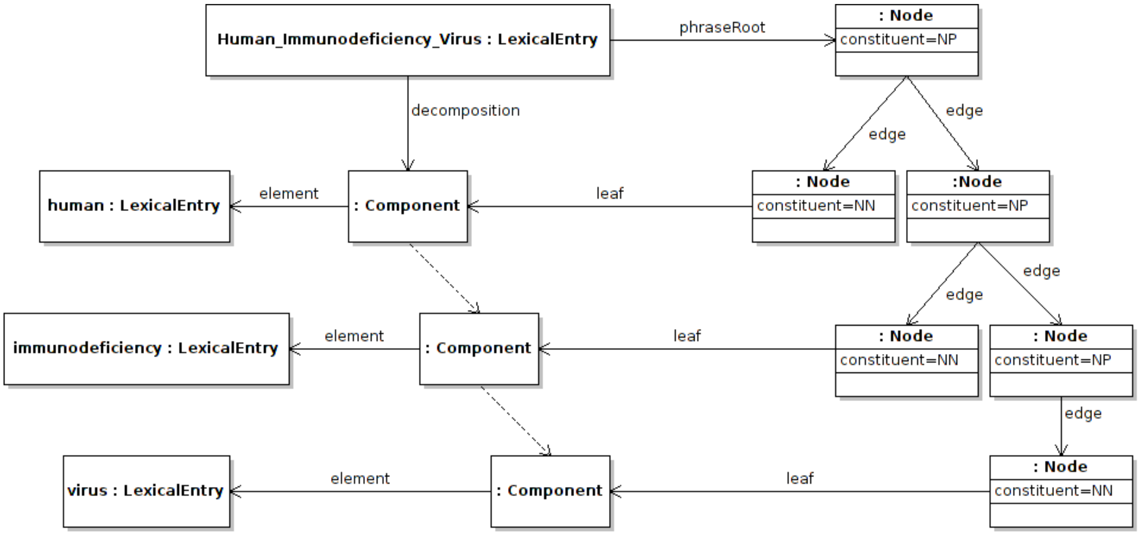
[8] Note the extra “e” is a dative inflection so not modelled as a component

```
        ]
      ]
    ] .

:human_component lemon:element :human .
:immondeficiency_component lemon:element :immunodeficiency .
:virus_component lemon:element :virus .
```

Example 36

It is important to note here that the phrase structure tree is not itself ordered, but the decomposition is, hence the order is obtained this way. It is further possible to name the arcs and nodes, by means of the property `constituent`. For example, we could extend the above example by denoting the (sub)phrases as follows:

```
:human_immunodeficiency_virus
  lemon:decomposition ( :human_component
                        :immunodeficiency_component
                        :virus_component ) ;
    lemon:phraseRoot [ lemon:constituent :NP ;
      lemon:edge [ lemon:constituent :NN ;
                   lemon:leaf :human_component ]   ;
      lemon:edge [ lemon:constituent :NP ;
        lemon:edge [ lemon:constituent :NN ;
                     lemon:leaf :immunodeficiency_component ] ;
        lemon:edge [ lemon:constituent :NP ;
          lemon:edge [ lemon:constituent :NN ;
                       lemon:leaf :virus_component ]
          ]
        ]
      ] .

:human_component lemon:element :human .
:immondeficiency_component lemon:element :immunodeficiency .
:virus_component lemon:element :virus .
```

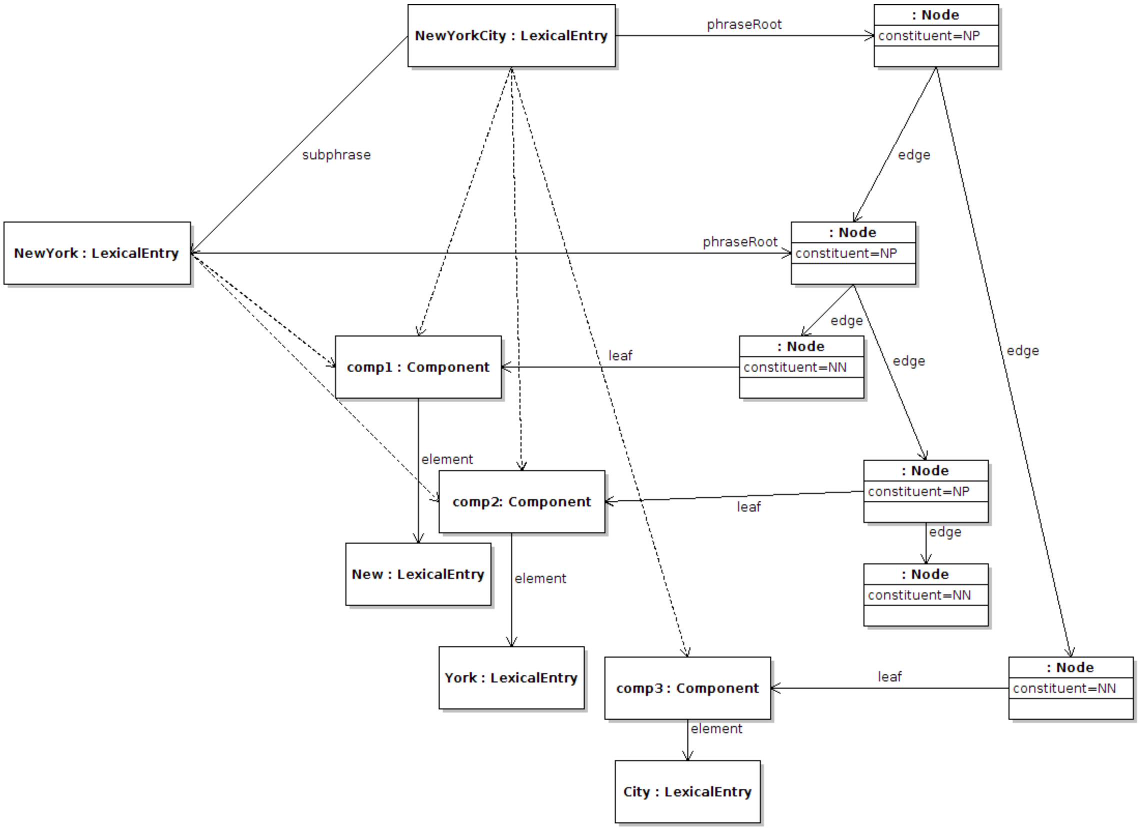


Example 37

Note that in the example all the constituents are marked as resources in the model. This means that there must exist some description already of the valid phrase constituents that exist for a certain grammar. Hence, for each grammar, there must be a *grammar description ontology*.

It is of course possible for multiple lexical entries to share the same phrase structure and this provides a more principled modelling of the decomposition than in section 1. This is as follows:

```
:new_york_city
  lemon:decomposition ( :comp1
                        :comp2
                        :comp3 ) ;
  lemon:phraseRoot [
    lemon:edge :new_york_node ;
    lemon:edge [ lemon:constituent :NN ;
                 lemon:leaf :comp3 ]
  ] .

:new_york
  lemon:decomposition ( :comp1
                        :comp2 ) ;
  lemon:phraseRoot :new_york_node .

:new_york_node
  lemon:edge [ lemon:constituent :NN ;
               lemon:leaf :comp1 ] ;
  lemon:edge [ lemon:constituent :NP ;
    lemon:edge [ lemon:constituent :NN ;
                 lemon:leaf :comp2 ] ] .

:comp1 lemon:element :new .
:comp2 lemon:element :york .
:comp3 lemon:element :city .
```



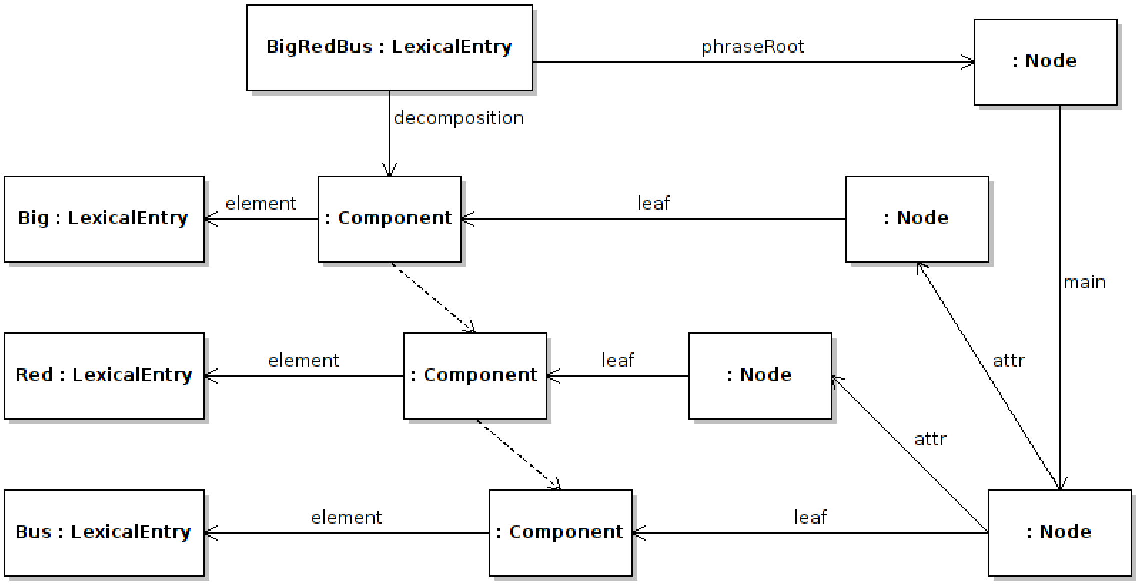
Example 38

Here :new_york_node is reused from the “New York City” tree as the root of the “New York” tree.

2.3.3 Dependency relations

Dependency grammars focus on the links between different words and so lemon is a very suitable representation. In fact all a dependency grammar needs to do is create sub-properties of `lemon:edge` in its grammar description ontology. For example we could create a dependency parse of "big red bus" as follows:

```
:big_red_bus
   lemon:decomposition ( :big_component
                         :red_component
                         :bus_component ) ;
   lemon:phraseRoot [
     :main [ lemon:leaf :bus_component ;
         :attr [ lemon:leaf :red_component ]
         :attr [ lemon:leaf :big_component ]
         ] ] .

:main rdfs:subPropertyOf lemon:edge .
:attr rdfs:subPropertyOf lemon:edge .
```



Example 39

It is of course possible to include both the dependency and the phrase structure parse simultaneously.

These parses can also be useful for identifying elements in frames. For example phrasal verbs in English, such as "switch off" can have multiple frames, namely "X switches Y off" and "X switches off Y". To perform this we need to be able to state a decomposition of the phrase, which could be done as follows:

```
:switch_off lemon:decomposition ( :switch_component
                                  :off_component ) .
   lemon:phraseRoot [ lemon:leaf :switch_component ;
      :particle [ lemon:leaf :off_component ] ] .

:particle rdfs:subPropertyOf lemon:edge .
```

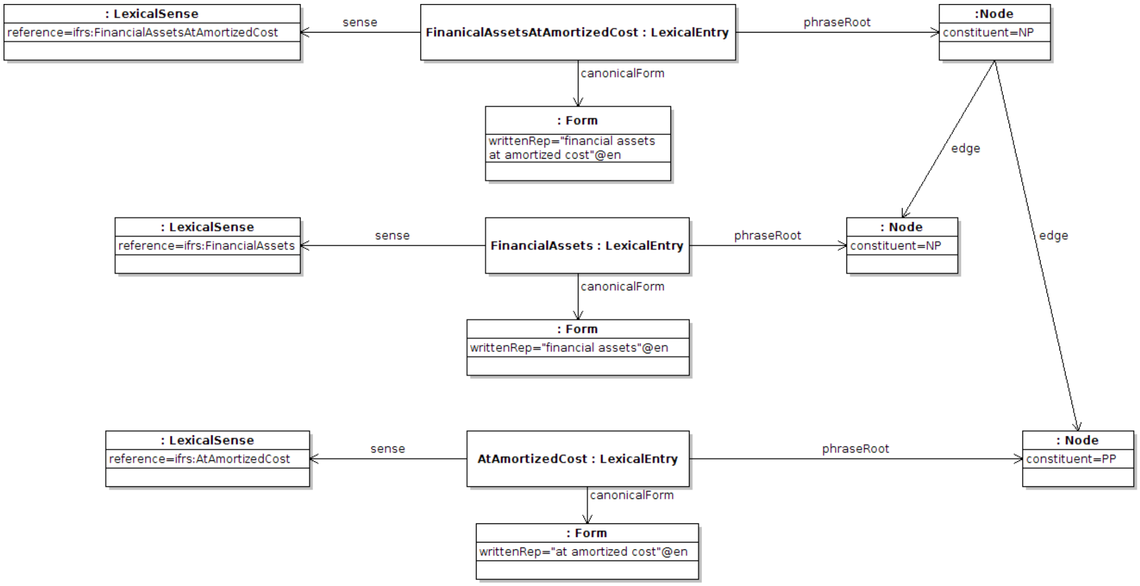
Example 40

We will present an alternative approach to this in the next section that uses the full parse tree (example 43).

2.3.4 Noun phrase chunks

It is often case that the full parse tree is not needed and only a section is required, for example the noun phrase chunks within the entry. This is modelled the same way as other phrase structure analyses in lemon, however such trees are not complete, i.e., they do not have `leaf` relations. An example of this is given for the phrase "Financial assets at amortized cost", which is chunked into "[Finanical Assets] [at amortized cost]":

```
:FinancialAssetsAtAmortizedCost a lemon:LexicalEntry ;
  lemon:sense [
    lemon:reference ifrs:FinancialAssetsAtAmortizedCost ] ;
  lemon:canonicalForm [
    lemon:writtenRep "finanical assets at amortized cost"@en ] ;
  lemon:phraseRoot [
    lemon:constituent :NP ;
    lemon:edge :NPChunk , :PPChunk ] .

:FinancialAssets a lemon:LexicalEntry ;
  lemon:sense [
    lemon:reference ifrs:FinancialAssets ] ;
  lemon:canonicalForm [
    lemon:writtenRep "finanical assets"@en ] ;
  lemon:phraseRoot :NPChunk .

:AtAmortizedCost a lemon:LexicalEntry ;
  lemon:sense [
    lemon:reference ifrs:ValueAtAmortizedCost ] ;
  lemon:canonicalForm [
    lemon:writtenRep "at amortized cost"@en ] ;
  lemon:phraseRoot :PPChunk  .

:NPChunk lemon:constituent :NP .
:PPChunk lemon:constituent :PP .
```

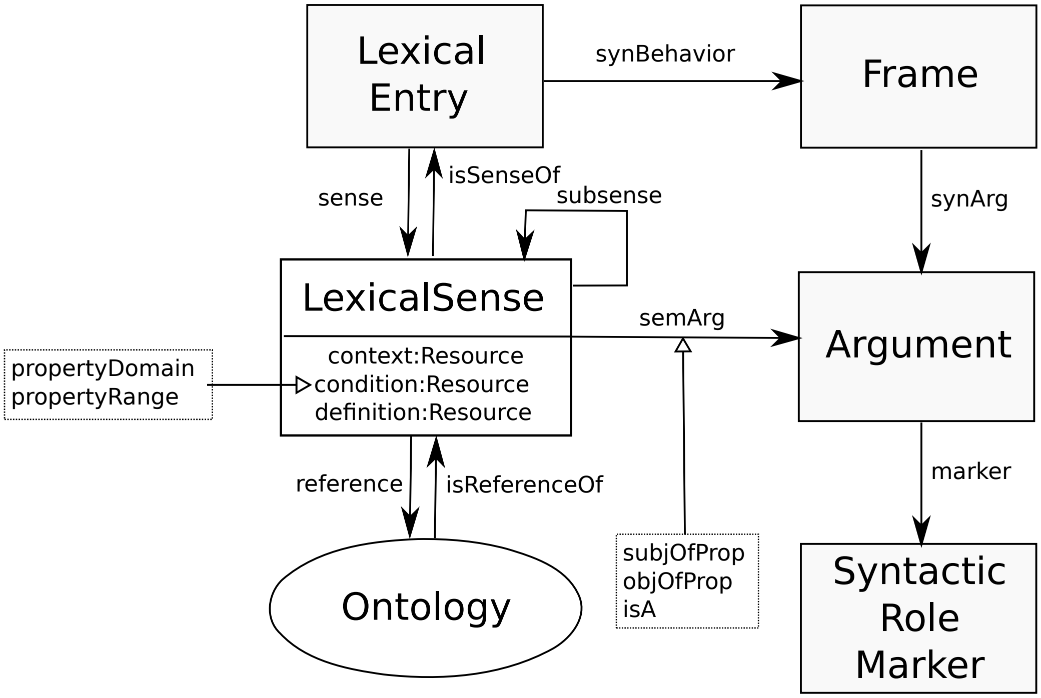

: LexicalSense
reference=ifrs:FinancialAssetsAtAmortizedCost
sense
FinanicalAssetsAtAmortizedCost : LexicalEntry
phraseRoot
:Node
constituent=NP
canonicalForm
: Form
writtenRep="financial assets at amortized cost"@en
edge
: LexicalSense
reference=ifrs:FinancialAssets
sense
FinancialAssets : LexicalEntry
phraseRoot
: Node
constituent=NP
edge
canonicalForm
: Form
writtenRep="financial assets"@en
: LexicalSense
reference=ifrs:AtAmortizedCost
sense
AtAmortizedCost : LexicalEntry
phraseRoot
: Node
constituent=PP
canonicalForm
: Form
writtenRep="at amortized cost"@en

Example 41

Summary of vocabulary introduced in this section

Property	Description
`decomposition`	Defines a list of components of a lexical entry
`element`	Defines the lexical entry of the given component
`phraseRoot`	A pattern describing the phrase structure of a lexical entry
`edge`	An edge between some nodes
`leaf`	A node’s reference to a component of a lexical entry
`constituent`	The phrase type of a node

2.4 Syntax and Mapping Module



2.4.1 Frames

lemon's approach to syntax builds on the notion of a "frame", which represents a stereotypical occurrence of a lexical entry, with a set of arguments. The frame is indicated with the property `synBehavior`, and each of the argument with the property `synArg`. For example, modeling a simple transitive frame in English (X eats Y) is as follows:

```
:eat lemon:synBehavior [ lemon:synArg :arg1 ;
                         lemon:synArg :arg2 ] .
```

Example 42

It is of course possible to relate these instances of by a subcategorization ontology such as LexInfo[9]. For example, it is possible to create subproperties of `synArg` to represent the syntactic functions of these arguments. Hence, the above example could be represented as follows.

```
:eat lemon:synBehavior [ a lexinfo:TransitiveFrame ;
                         lexinfo:subject :eat_subject ;
                         lexinfo:object :eat_object] .

lexinfo:TransitiveFrame rdfs:subClassOf lemon:Frame .
lexinfo:subject rdfs:subPropertyOf lemon:synArg .
lexinfo:object rdfs:subPropertyOf lemon:synArg .
```

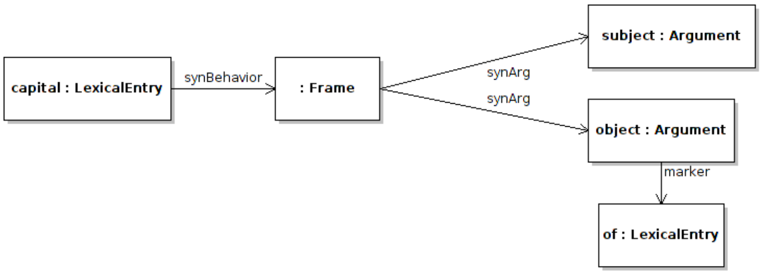
Example 43

In many cases frames require a marker for the syntactic argument, this is generally an adposition, particle or case inflection. These can be modeled as follows.

English: "X is the capital of Y"

```
:capital_of lemon:synBehavior [ lemon:synArg :subject ;
                                lemon:synArg :pobject ] .

:noun_pp_pobject lemon:marker :of .
```

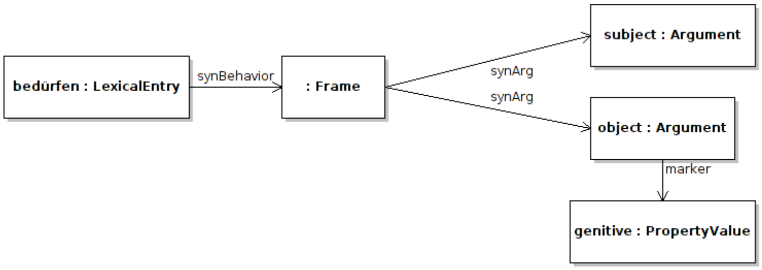
[9] http://www.lexinfo.net/ontology/2.0/lexinfo



Example 44

German: “X bedarf Y [genitive case]”[10]

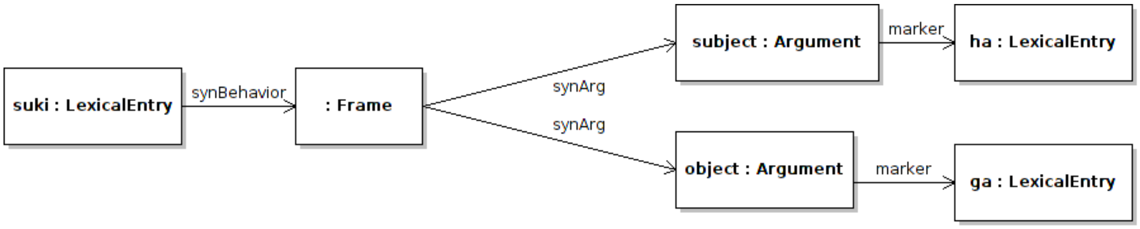
```
:beduerfen lemon:synBehavior [ lemon:synArg :subject ;
                               lemon:synArg :object ] .

:genitiv_verb_object lemon:marker isocat:genitiveCase .
```



Example 45

Japanese: “XはYが好きだ” (double subject construction) (“X ha Y ga suki da” = “X likes Y” lit. “X (*primary subject*) Y (*secondary subject*) is pleasing”)

```
:suki lemon:synBehavior [ lemon:synArg :ha_subject ;
                          lemon:synArg :ga_subject ] .

:ha_subject lemon:marker :ha .
:ga_subject lemon:marker :ga .
```



Example 46

lemon also allows for modeling using optional arguments in the frame, this is often the case with, for example, prepositionally marked arguments in English. For example the verb “move” can have prepositional arguments such “from”,“to”,“by”,“for”,“past”,“away”,“along” etc. These are modeled as extra arguments to the frame but are not necessary for the frame to give a valid syntactic realization. This is performed as follows.

```
:move_frame lemon:synArg :move_frame_subject ,
                         :move_frame_to_obj ,
                         :move_frame_from_obj ,
                         :move_frame_by_obj .
:move_frame_to_obj  lemon:marker :to ;
```

[10] `PropertyValue` is the class for the range of `property`

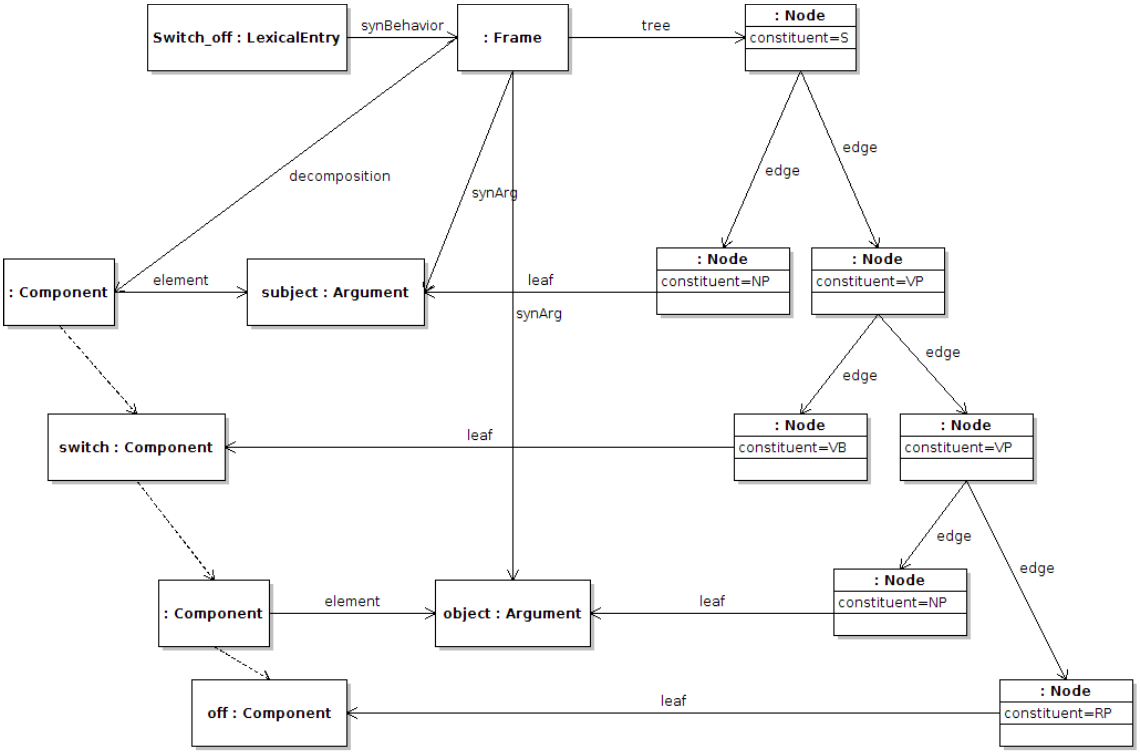
```
   lemon:optional "true"^^xsd:boolean .
:move_frame_from_obj  lemon:marker :from ;
   lemon:optional "true"^^xsd:boolean .
:move_frame_by_obj  lemon:marker :by ;
   lemon:optional "true"^^xsd:boolean .
```

Example 47

2.4.2 Phrase structure and frames

lemon frames may also be combined with the multi-word expressions extension to give valid parse trees for a particular frame, creating something similar to the substitution trees used in Tree-Adjoining Grammars. Here the trees are attached to the frame instead of to the lexical entry, and use the property `tree`, in addition we need to indicate the order of the components in the phrase structure, which is done by adding a `decomposition` to the frame. Note that this `decomposition` unlike decompositions of lexical entries may (and in fact must) include the arguments of the frame. For example “X switches Y off” may generate a parse structure as follows:

```
:switch_off
   lemon:synBehavior
     [ lemon:synArg :switch_off_subject ,
       lemon:synArg :switch_off_object
       lemon:tree [
         lemon:edge [ lemon:leaf :switch_off_subject ] ;
         lemon:edge [
           lemon:edge [ lemon:leaf :switch_comp ] ;
           lemon:edge [
             lemon:edge [ lemon:leaf :switch_off_object ] ;
             lemon:edge [
               lemon:edge [ lemon:leaf :off_component ]
             ]
           ]
         ]
       ] ;
       lemon:decomposition (
         [ lemon:element :switch_off_subject ]
         [ lemon:element :switch_comp ]
         [ lemon:element :switch_off_object ]
         [ lemon:element :off_comp ]
       )
     ] .
```



Example 48

2.4.3 Predicate mapping

The mapping part of this module is primarily focussed on linking the entities in the ontology with the linguistic descriptions. This consists of primarily two tasks: describing how the arguments of an ontology predicate map to those of a syntactic frame and adding extra conditions for the applicability of a given mapping.

We shall tackle the first part of this, which requires reifying the arguments of a property. Although RDF supports the reification of statements, this is not a reification over all triples using a given property, rather over a single property. We instead introduce the idea of a semantic argument, which can be related to the model via three properties, namely `subjOfProp`, `objOfProp` and `isA`. The former two are used if a lexical sense has a reference to a property in the ontology, for example

```
:capital lemon:sense [ lemon:reference ontology:capital ;
                       lemon:subjOfProp :capital_sem_subj ;
                       lemon:objOfProp :capital_sem_obj ] .
```

Example 49

Similarly lemon views classes as unary predicates like in other formalisms such as SWRL, the argument is then specified with the property `isA`

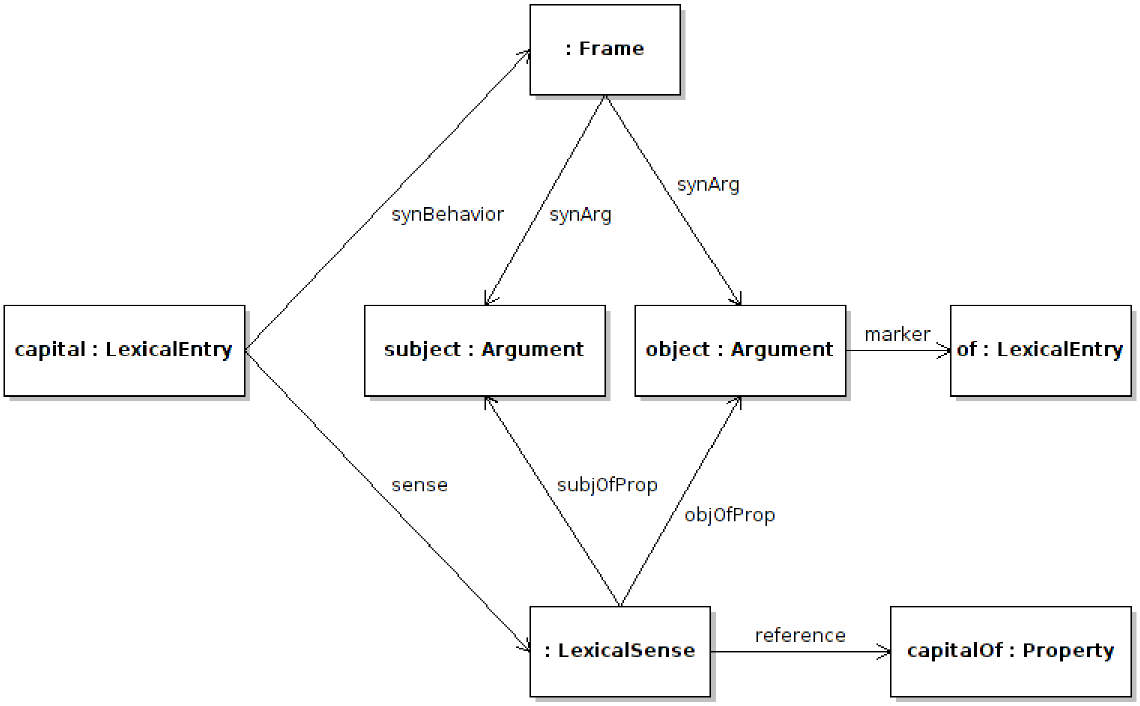
```
:cat lemon:sense [ lemon:reference ontology:Cat ;
                   lemon:isA :isa_cat ] .
```

Example 50

In fact, lemon does not use separate elements for the syntactic and semantic arguments. This allows for much more economical encoding of arguments than in LMF, without any loss in expressivity (see section 4.1).

```
:capital lemon:sense [ lemon:reference ontology:capital ;
                       lemon:subjOfProp :subject ;
                       lemon:objOfProp :object ] ;
```

```
   lemon:synBehavior [ lemon:synArg :subject ;
                       lemon:synArg :object ] .

:capital_obj lemon:marker :of .
```



Example 51

2.4.4 Conditions

Conditions are another mechanism applied to senses to restrict the mapping between the syntactic and semantic layers. Conditions are something that can be generally tested to check whether a given lexical entry is appropriate to an ontology reference or vica versa. One of the main usage of conditions is to describe the necessary constructs that are beyond the true "sense" of the lexical entry, but are essential to understanding the mapping. For example the German verb "essen" is generally used for humans, whereas "fressen" is generally used when the subject is an animal. This can be modeled by saying the two verbs are restricted with the `propertyDomain` and `propertyRange`. This is a condition that applies only if the object/subject of the triple in question is in the appropriate class. In fact, as lemon is based on RDF(S) and OWL and thus makes an *open world assumption*, such conditions are only false if it is provable that the object/subject is not in the appropriate class.

```
:essen lemon:sense [ lemon:reference ontology:eat ;
   lemon:propertyDomain ontology:Human ] .
:fressen lemon:sense [ lemon:reference ontology:eat ;
   lemon:propertyDomain ontology:NonHuman ] .
```

Example 52

lemon only contains these two properties however we may also extends the `condition` property to new conditions as is required. For example, we take the Spanish "párajo", which means "bird" but is used only for small birds that can fly. So we may introduce a property `notUsedFor` to indicate particular subclasses that the term may not be used for, e.g., here we indicate that it is not used for *ratites* (order *struthioniformes*), the taxon containing ostriches, emus, rheas and kiwis:

```
:parajo lemon:canonicalForm [
    lemon:writtenRep "bird"@es ];
  lemon:sense [
    lemon:reference ontology:Bird ;
    :notUsedFor ontology:Ratite
  ] .

:notUsedFor rdfs:subPropertyOf lemon:condition .
```

Example 53

We can even include full logical conditions that can be used to test if a sense should be used by specifying a rule in some logic that can be evaluated. We present such an example below; it should be read as "does there exist a ?y such that for some individual ?x, flowsInto(?x,?y) and River(?y) hold)/

```
:riviere lemon:sense [ lemon:reference ontology:River ;
   lemon:condition [ lemon:value "exists ?y : flowsInto(?x,?y), River(?y)" ] ] .
:fleuve lemon:sense [ lemon:reference ontology:River ;
   lemon:condition [ lemon:value "exists ?y : flowsInto(?x,?y), Sea(?y)" ] ] .
```

Example 54

Readers should note we use a new property value here with a blank node. This is because we require that the range of condition are individuals in order to guarantee the model is OWL DL compatible. We discuss value more in section 2.1.3.

There is some overlap between conditions and contexts, however as a rule of thumb, the use of a lexical entry for an ontology entity that violates the context would be considered "inappropriate" by a reader, where as if the condition is violated it would be considered "incorrect."

2.4.5 Mapping to more complex representations

As OWL only supports unary (class), and binary (property) predicates, it is not possible to simply map more complex syntactic structures to OWL predicates. This requires mapping these complex predicates to structures in the ontology composed over several predicates. For example, consider we had the following simple ontology to describe a giving event, and a lexical entry for "give" in the lexicon as below.

```
ontology:giver a owl:ObjectProperty ;
   rdfs:domain ontology:GivingEvent ;
   rdfs:range ontology:Giver .

ontology:givenObject a owl:ObjectProperty ;
   rdfs:domain ontology:GivingEvent ;
   rdfs:range ontology:Given .

ontology:givenTo a owl:ObjectProperty ;
   rdfs:domain ontology:GivingEvent ;
   rdfs:range ontology:Recipient .

ontology:Giver a owl:Class .
ontology:GivingEvent a owl:Class .
ontology:Given a owl:Class .
ontology:Recipient a owl:Class .
```

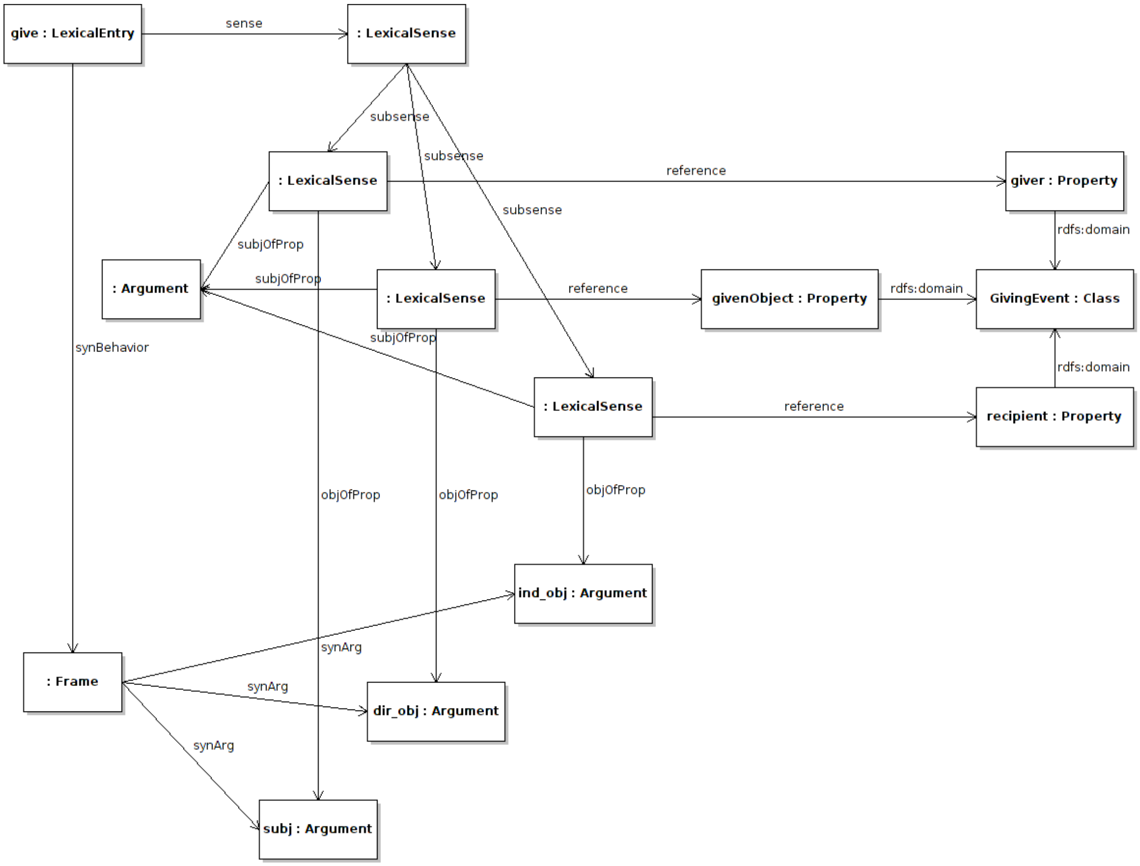
Example 55

```
:give a lemon:Word ;
      lemon:synBehavior [ lemon:synArg :give_subj ;
                          lemon:synArg :give_dir_obj ;
                          lemon:synArg :give_ind_obj ] .
```

Example 56

lemon solves such mapping by creating a *compound sense*, which is composed of *atomic senses* which we have described above. As such the lexical entry has a single compound sense, which is composed of atomic senses that map to the ontology reference. The composition of these senses is given by the subsense property. We will now demonstrate how this is used to represent the "giving" ontology represented above.

```
:give
   lemon:sense [
      lemon:subsense [
                 lemon:reference ontology:giver ;
                 lemon:objOfProp :give_subj ;
                 lemon:subjOfProp :event ] ,
               [ lemon:reference ontology:givenObject ;
                 lemon:objOfProp :give_dir_obj ;
                 lemon:subjOfProp :event ] ,
               [ lemon:reference ontology:givenTo ;
                 lemon:objOfProp :give_ind_obj ;
                 lemon:subjOfProp :event ]
             ].
```



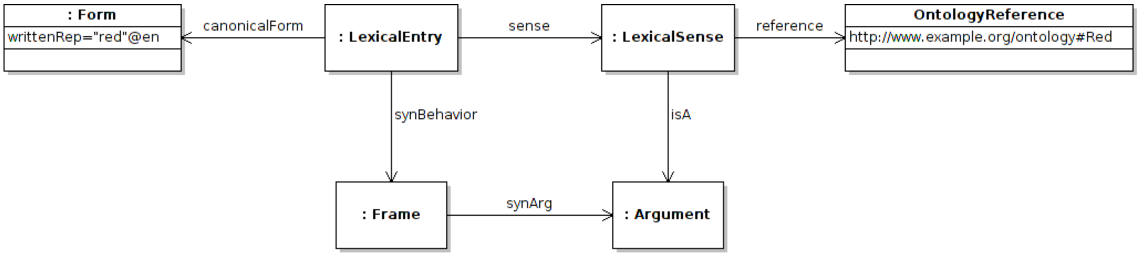
Example 57

Here the lexical entry "give" has a single sense composed of three senses, one for each property in the ontology. Each of these sub-senses has an argument mapped to the syntactic frame and one to a new argument element :event, which is not bound to any syntactic frame, but instead indicates that the subject of all properties should be the same individual.

2.4.6 Mapping adjectives

Adjectives present particular challenges in mapping and while the lemon vocabulary presented so far is suitable for most mappings, we will present design patterns explaining how mapping with adjectives should be performed. For the most part adjectives are involved in unary syntactic frames (that is a frame with a single argument), except for the case where they have a prepositional complement, e.g., “X is similar to Y” or are used postpositively, e.g., “X ist Y ähnlich” in German. We shall leave these binary cases as an example to the user as they function like other examples presentend above. As such we shall only handle the case where adjectives are unary. Given that we have assumed adjectives are unary then it is clear that on the ontology side they correspond to classes. For example assume `Red` is a class in our ontology then we can map the lexical entry “red” as follows

```
:red lemon:canonicalForm [ lemon:writtenRep "red"@en ] ;
  lemon:sense [ lemon:reference ontology:Red ;
                lemon:isA :attr ] ;
  lemon:synBehavior [ lemon:synArg :attr ] .
```



Example 58

In fact the inclusion of adjectives as classes in the ontology is rare in general, but lemon cannot semantically map lexical entries to semantics that is not extant in the ontology. As such it is often necessary to the ontology vocabulary, i.e., OWL, to include these semantics to map to a lexicon. For example consider the example of an ontology with a property `color` with values {`green`, `red`, `blue`}. As the color words are individuals it is not possible to map the syntactic predicates of “red”, “green” and “blue“ to them without first making them into classes. This is simply done as follows

```
ontology:Green a owl:Class ;
  owl:equivalentTo [ a owl:Restriction ;
    owl:onProperty ontology:color ;
    owl:hasValue ontology:green
  ] .
```

DL Syntax:

$$Green\equiv\ni color.green$$

Example 59

Then these classes are mappable like the previous example. An even more complex example exists if you wish to map an adjective to a datatype property, for example “big” to a property `size`. As of OWL2 there exists vocabulary to define classes in terms of data type properties, so we may define a class of big cities, `BigCity`, as all cities of size greater than 500,000.

```
ontology:BigCity a owl:Class
  owl:equivalentTo [ owl:intersectionOf (
    ontology:City
    [ a owl:Restriction ;
      owl:onProperty ontology:size ;
      owl:someValueFrom [ a rdfs:Datatype ;
                          owl:onDatatype xsd:integer ;
                          owl:withRestrictions ( [ xsd:minExclusive 500000 ] )
                        ]
             ] )
    ] .
```

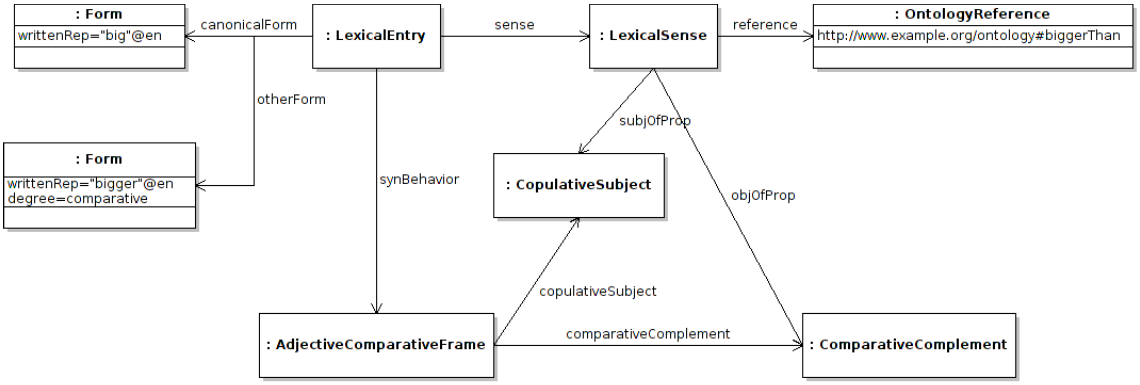
DLSyntax:

$$BigCity \equiv City \sqcap \exists size.integer[> 500000]$$

Example 60

Another aspect of adjectives is that they are frequently used comparatively or superlatively, that is with “more” and “most.” Again this behavior can only be analyzed if the appropriate semantics exist within the ontology. For example, consider we have “big” and we wish to define “bigger”, this clearly gives a binary frame “X is bigger than Y” and hence must be matched to a property in the ontology as follows, the frame should then be marked in someway as a comparative frame, for example in LexInfo 2 this is done by the `AdjectiveComparativeFrame`. So we could map as follows.

```
:big lemon:canonicalForm [ lemon:writtenRep "big"@en ] ;
     lemon:otherForm [ lemon:writtenRep "bigger"@en ;
                       isocat:degree isocat:comparative ] ;
     lemon:synBehavior [ a lexinfo:AdjectiveComparativeFrame ;
                         lexinfo:copulativeSubj :subj ;
                         lexinfo:comparativeComplement :comp ]
     lemon:sense [ lemon:reference ontology:biggerThan ;
                   lemon:subjOfProp :subj ;
                   lemon:objOfProp :comp ] .

lexinfo:AdjectiveComparativeFrame rdfs:subClassOf lemon:Frame .
lexinfo:copulativeSubj rdfs:subPropertyOf lemon:synArg .
lexinfo:comparativeComplement rdfs:subPropertyOf lemon:synArg .
```



Example 61

This comparative property is not currently representable in OWL, however it is possible to express it in SWRL, for example we could define the `biggerThan` property in the following way

```
:y rdf:type swrl:Variable .
:x rdf:type swrl:Variable .
:s1 rdf:type swrl:Variable .
:s2 rdf:type swrl:Variable .
[ rdf:type swrl:Imp ;
  swrl:body ( [ rdf:type swrl:ClassAtom ;
                swrl:classPredicate :City ;
                swrl:argument1 :x
              ]
              [ rdf:type swrl:ClassAtom ;
                swrl:classPredicate :City ;
                swrl:argument1 :y
              ]
              [ rdf:type swrl:DatavaluedPropertyAtom ;
                swrl:argument2 :s1 ;
                swrl:propertyPredicate :size ;
                swrl:argument1 :x
              ]
              [ rdf:type swrl:DatavaluedPropertyAtom ;
                swrl:argument2 :s2 ;
                swrl:propertyPredicate :size ;
                swrl:argument1 :y
              ]
              [ rdf:type swrl:BuiltinAtom ;
                swrl:builtin swrlb:greaterThan ;
                swrl:arguments ( :s1
                                 :s2 )
              ]
            ) ;
  swrl:head ( [ rdf:type swrl:IndividualPropertyAtom ;
                         swrl:propertyPredicate :biggerThan ;
                         swrl:argument1 :x ;
                         swrl:argument2 :y
              ]
            ) ] .
```

SWRL Rule:
City(?x) , City(?y) , size(?x, ?s1) , size(?y, ?s2) , greaterThan(?s1, ?s2) → biggerThan(?x, ?y)

Example 62

2.4.7 Correspondence

The key understanding of a sense is as a correspondence betwen an ontology entity and a lexical entry and as such the set of senses in the lexicon constitute a many-to-many mapping between lexical entries and ontology entities. This means that each sense must apply to exactly one lexical entry and one ontology element. Thus, for any ontology-based disambiguation task the challenge can be simply specified as selecting the correct sense for the task. For this reason lemon provides a number of features that should prove useful for this task, which are detailed in section **??**.

There are several reasons for defining the correspondence as an element of the lexicon. Firstly, to allow further description of pragmatic usage between the two terms: for example, “cats and dogs” would be considered a natural usage whereas “Felis Catus and dogs” would be considered odd as the pragmatic register of the two terms is different. Although lemon also includes features to assign words to a pragmatic context, it does require the definition of a pragmatic taxonomy, which must be done for any particular application of the model.

The second case is where some terms are closely lexically bound, i.e., they generally occur together. An example from French is “rivière et fleuve”, which are often used together to express the same concept in English as “river”. It is natural that we should wish to connect these two terms by indicating they have disjoint senses that together constitute the idea of “river”. As the meaning of “rivière” in French is defined by Larousse as

> Cours d’eau de faible ou moyenne importance qui se jette dans un autre cours d’eau (Watercourse of small size or little importance, which flows into another watercourse)

This can be considered in a semantic sense to be nearly equivalent to the French term “affluent”, defined as

> Un cours d’eau qui se jette dans un autre. (A watercourse which flows into another)

However, a system generating “fleuve et affluent” as the equivalent concept to that of “river” would generally not be considered to be correct, due to lexical collocations. In fact for reference resolution it may be better to assert “affluent” as a broader term for “rivière”, as every usage of “rivière” is necessarily an “affluent”.

The final case is where terms are used in a metaphoric or metonymic manner. For example,

> The White House said today, ...

Here “the White House” does not refer to the literal building but the spokesperson and other members of the organization in that building. As such the correspondence would give an ontology entity with the properties of being a human or organization; although reasoning using this information, may generate odd statements such as

> *The White House is a member of the species Homo Sapiens

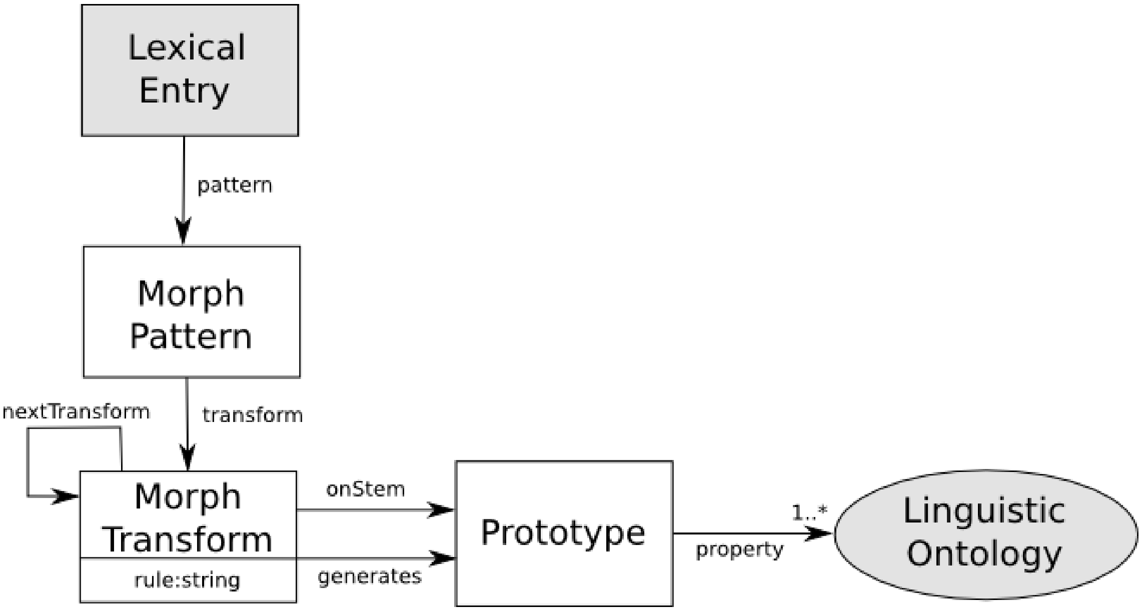
Thus we need to define the lexical semantics as clearly separate from those of the ontological element. This could be done for example by asserting something as a metonymic raising of another sense e.g.,

```
:white_house
  lemon:canonicalForm [ lemon:writtenRep "The White House"@en ] ;
  lemon:sense :white_house_sense .
:white_house_spokesperson
  lemon:canonicalForm [ lemon:writtenRep
     "White House Spokesperson"@ en ] ;
  lemon:sense :white_house_spokesperson_sense .

:white_house_sense :metonymicRaisingOf :white_house_spokesperson_sense.

:metonymicRaisingOf rdfs:subPropertyOf lemon:senseRelation.
```

Summary of vocabulary introduced in this section

Property	Description
synBehavior	The frame of a lexical entry.
synArg	The syntactic slots of a frame.
marker	The case or lexical element that marks a semantic argument.
optional	Indicates an argument is not required in order to realize a frame.
tree	Indicates the a lexicalized grammar tree attached to a frame.
semArg	A semantic argument
↳subjOfProp	The subject of property
↳objOfProp	The object of a property
↳isA	The argument of a class predicate
context	A contextual parameter on the mapping
condition	A condition that can be checked to see if the given lexical entry and reference correspond in the given context
↳propertyDomain	A condition on the domain of the property being mapped to
↳propertyRange	A condition on the range of the property being mapped to
subsense	Indicates an atomic sense that composes a compound sense

2.5 Morphology Module



The usual lemon method for representing different forms of the same lexical entry is to indicate all non-canonical forms by the use of the `otherForm` property. While this may be reasonably practical for English as there are only four, three and two forms for verbs, adjectives and nouns respectively, there can be many forms for items of more synthetic languages. For example, a regular verb in Italian, such as "amare", has 43 distinct forms, and encoding all of these as seperate forms for every word would greatly increase the size of lexicon. This problem is even more acute for languages exhibiting *polysynthesis*, where multiple inflections may be combined. For example in Japanese the following (among other) inflections of the regular verb exist (here "taberu", to eat) .

	Positive	Negative
Present	taberu	tabenai
Past	tabeta	tabenakatta

The same verb also can be inflected to give a passive form, "taberareru", and a causative form, "tabesaseru"[11]. These forms are in fact verbs, that have the same inflections as the regular verb. To further complicate things, both the passive and causative may be combined to give a passive-causative form "tabesaserareru". As such the following form of the verb exists, which may be decomposed as follows (the meaning is "(he) was not made to eat").

tabe	-	sase	-	rare	-	naka	-	tta
tabe		saseru		rareru		nai		ta
stem		*causative*		*passive*		*negative*		*past*

Example 63

This leads to a very large number of inflections and as such it is difficult to state how many word forms exist for a Japanese verb, although in practice it is unlikely that verbs composed of more than 5 morphemes occur. Therefore, a minimal encoding is needed that allows regular and semi-regular verbs to be represented economically.

In difference to the majority of the vocabulary in lemon, this morphology is not specific to a given lexicon, in fact the same morphological rules can be used for any lexicon within the same language. As such, it should not be necessary for most lexica to represent the morphology of their given language, and for most major language bespoke systems will often in practice be more useful than the modules described here. However, there are use cases where this is useful, for example for resource-poor languages it may be preferable to have a common representation of morphology. Furthermore, bespoke systems are often opaque about the methods they use for generating inflectional morphology and this can make it difficult with words that have unusual forms (a

[11] The meaning of the causative is approximately "to cause to do"

simple example in English is that nouns ending in “y” normally have a plural in “ies”, e.g., “cherries”, but in some odd cases has a plural in “ys”, e.g., “the two Germanys”).

2.5.1 Inflection

The morphology of a word is given by a *pattern*, which is composed of a set of *transforms*, that is in turn composed of *rules* and produces a form described as a *prototype*. To give a simple example we will look at how we represent that English nouns form their plural in ”s”

```
:english_noun a lemon:MorphPattern ;
  lemon:transform [
    lemon:rule "~s" ;
    lemon:generates [
      isocat:number isocat:plural
    ]
  ] .
```

Example 64

This gives a pattern, with a single rule defined as `"˜s"` that generates a prototype with the property number with value plural. The rules are specified in a simple manner where the tilde symbol represents the canonical form and then the “s” is simply appended. A lexical entry may be indicated to have a given morphological pattern by the `pattern` property as follows.

```
:cat lemon:pattern :english_noun
```

Example 65

The rule above is clearly not sufficient: for example for the word “cherry” it would generate the erroneous plural “cherrys.” Another rule can be added to handle this case, for example

```
:english_noun a lemon:MorphPattern ;
  lemon:transform [
    lemon:rule "~s" ;
    lemon:rule "~y/~ies" ;
    lemon:generates [
      isocat:number isocat:plural
    ]
  ] .
```

Example 66

In this example the slash indicates the difference between the *matcher* and the *replacer* as such for the rule to applies the canonical form must first match the matcher and then the matched text is replaced with the replacer. The first rule has no matcher and so is assumed to match all canonical forms that aren’t matched by a more specific rule.[12] This rule is still not correct as it generates for “play” the form “plaies”. As such we should further modify the rule to check for a preceding vowel this is done as follows

```
:english_noun a lemon:MorphPattern ;
  lemon:transform [
    lemon:rule "~s" ;
    lemon:rule "~(<?![aeiou])y/~ies" ;
    lemon:generates [
      isocat:number isocat:plural
    ]
  ] .
```

[12] It is up to the lexicon author to ensure no two matchers can match the same string

Example 67

In fact, as this example may have hinted the underlying representation of the rules used by this module are in fact Perl-like regular expressions[13]. In fact as such an alternative rule that performs the same task `"˜([ˆaeiou])y/˜$1ies"`; note that here the penultimate constonant must be reintroduced as it gets removed by being matched, and this is done with `"$1"`. We think it is clearer to state rules using the look-ahead and look-behind assertions (namely `(?=X)`, `(?!X)`, `(<?=X)`, and `(<?!X)`), as they do not cause vowels to be lost. Another reason for prefering these zero-width assertions is that in implementation the tildes must be translated to `(.*)` and `$1` respectively, and it makes it easier to do this if there are no groups on the left hand side.

The use of the tilde is useful is as it makes it easier to represent both prefices and suffices, for example consider the case of the German perfect participle, this is formed for regular weak verbs by prefixing “ge” and changing the inflectional suffix to “t”. We can model this as follows

```
:german_weak_verb a lemon:MorphPattern ;
  lemon:transform [
    lemon:rule "~e?n/ge~t" ;
    lemon:generates [
      isocat:tense isocat:past ;
      isocat:aspect isocat:perfective ;
      isocat:verbFormMood isocat:participle
     ]
   ] .
```

Example 68

Of course it is natural it is possible for a pattern to have multiple transforms and even for a transform to have multiple generations. For example, we shall show an example for Italian generating the present singular first and third person forms and then using the third person singular to get the singular imperative

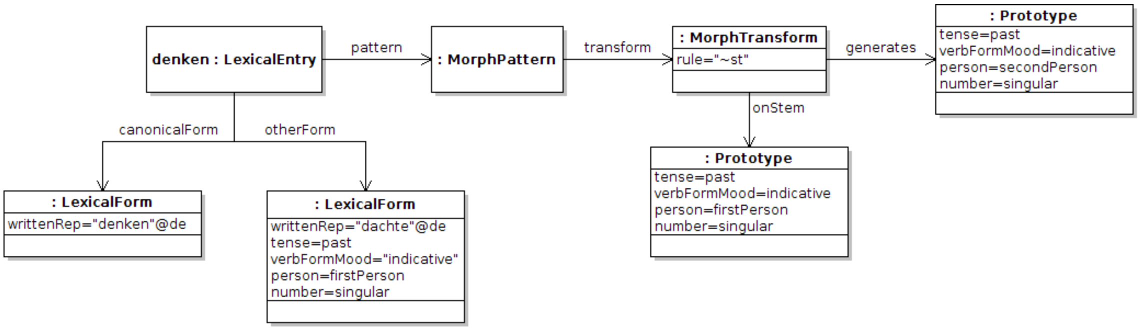
```
:italian_are_verb a lemon:MorphPattern ;
  lemon:transform [
    lemon:rule "~are/~o" ;
    lemon:generates [
      isocat:tense isocat:present ;
      isocat:person isocat:firstPerson ;
      isocat:number isocat:singular
    ]
  ] , [
    lemon:rule "~are/~a" ;
    lemon:generates [
       isocat:tense isocat:present ;
       isocat:person isocat:firstPerson ;
       isocat:number isocat:singular
     ] , [
       isocat:verbFormMoode isocat:imperative ;
       isocat:number isocat:singular
     ]
  ] .
```

Example 69

[13] There is no standard for Perl-like regular expressions, here we assume that these rules must support the functionality of POSIX regular expressions, and in addition the + operator, negated character classes, non-greedy quantifiers, shy groups, lookahead and lookbehind assertions, backreferences, directives and conditionals. This condition is satisfied by the standard regular expressions libraries for Java (and other JVM Languages), the .NET platform, Perl, PHP and Python, the Boost.Regex library for C/C++ and the Unix `grep` command.

Sometimes the canonical form alone is not sufficient to generate all forms of the word, for this reason other forms must be used as the base for generating some forms, and these forms should be given in the lexicon. An example is German mixed verbs, for example “denken”, to think, that has an irregular past “dachte”. This form is the first (and third) person singular preterite, and the second person singular is created by adding “st” to this form. For these cases the `onStem` property is provided, for example:

```
:german_mixed_verb a lemon:MorphPattern ;
  lemon:transform [
    lemon:onStem [
      isocat:tense isocat:past ;
      isocat:verbFormMood isocat:indicative ;
      isocat:person isocat:firstPerson ;
      isocat:number isocat:singular
    ] ;
    lemon:rule "~st" ;
    lemon:generates [
      isocat:tense isocat:past ;
      isocat:verbFormMood isocat:indicative ;
      isocat:person isocat:secondPerson ;
      isocat:number isocat:singular
    ]
  ] .
```



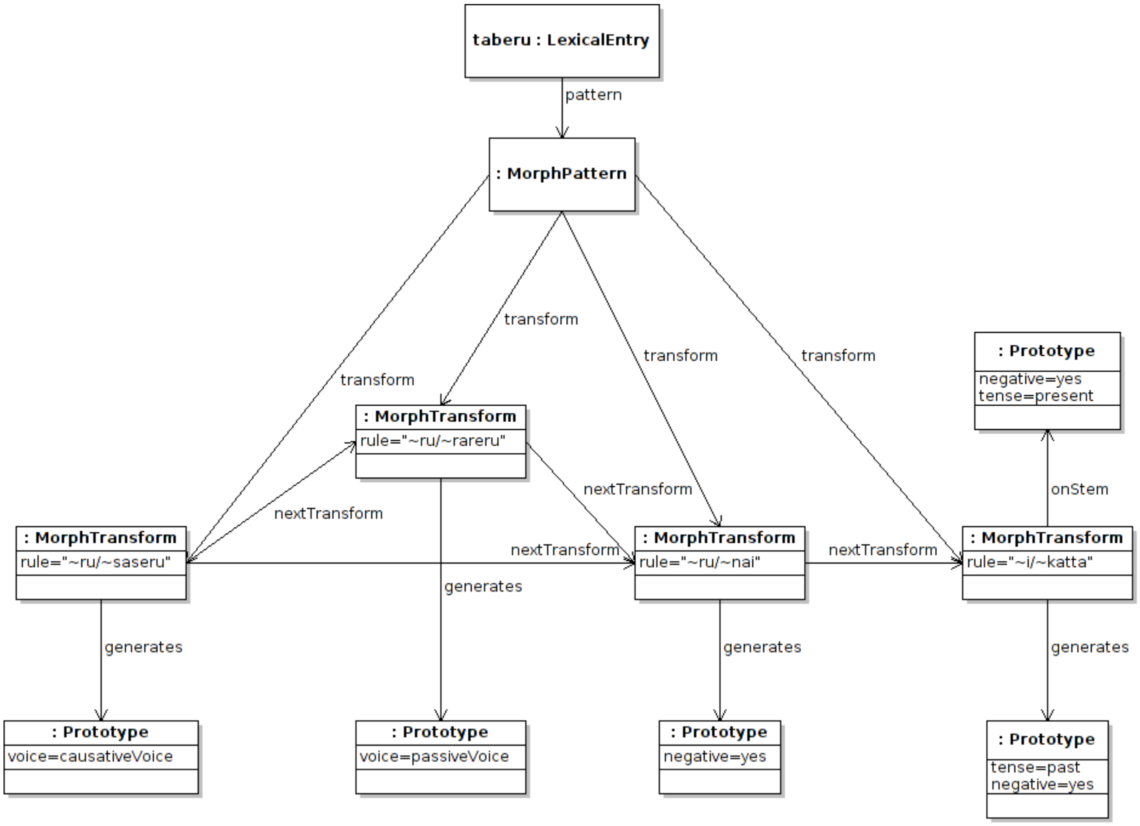
Example 70

For polysynthesis as in example 60,we introduce a property that indicates how multiple inflections can be composed called `nextScope`. So the inflections that compose the example are as follows[14]

```
:japanese_vowel_stem_verb a lemon:MorphPattern ;
  lemon:transform :causative_transform ,
                  :passive_transform ,
                  :negative_transform ,
                  :past_transform_on_negative .

:causative_transform lemon:rule "~ru/~saseru" ;
  lemon:nextScope :passive_transform, :negative_transform ;
  lemon:generates [
    isocat:voice isocat:causativeVoice
  ] .

:passive_transform lemon:rule "~ru/~rareru" ;
  lemon:nextScope :negative_tranform ;
  lemon:generates [
    isocat:voice isocat:passiveVoice
  ] .
```

[14] For readability we will continue to use Hepburn romanization, examples should in fact be represented in native script.

```
:negative_transform lemon:rule "~ru/~nai" ;
  lemon:nextScope :past_transform_on_negative ;
  lemon:generates [
    isocat:negative isocat:yes
  ] .

:past_transform_on_negative lemon:onStem [
     isocat:negative isocat:yes ;
     isocat:tense isocat:present
   ] ;
   lemon:rule "~i/~katta" ;
   lemon:generates [
     isocat:negative isocat:yes ;
     isocat:tense isocat:past
   ] .
```



Example 71

2.5.2 Agglutination

Many languages uses agglutination where the form of a word is dependent on neighboring words and we would like to be able to represent this as well. We use agglutination as a word that covers many distinct phenomena such as assimilation, liasion, sindha and vowel harmony. For example, in Maltese the definite article “il-” is a proclitic that assimilates with the first phoneme of the following word if it is a *coronal constonant* or vowel. For example

il- + missier	il-missier	the father
il- + omm	l-omm	the mother
il- + tifel	it-tifel	the boy

Example 72

The rules for doing this are similar to inflection rules but include the symbol “+” to indicate the word with which the agglutination occurs. So we describe maltese agglutination as follows

```
:maltese_il_assimilation a lemon:MorphPattern ;
  lemon:transform [
     lemon:rule "~l+([ċdnrstxżz])/~$2-$2" ;
     lemon:rule "~l+([aeiou])/l-$2"
   ] .
```

Example 73

Note that this pattern only applies to one word, it is written in a way that could be used elsewhere if the pattern occurs differently. Similarly `$2` is used as the matcher (as `$1` corresponds to ~).

Hungarian (as well as all Uralic and many Altaic langauges) exhibit an interesting agglutinative property known as vowel harmony, in which the vowels in a suffix must agree with the vowels in the preceding word. Hungarian groups the vowels into three categories back vowels (a, o, u), front vowels (ö, ü) and intermediate vowels (e, i). Prepositions and cases in Hungarian are prefices which must agree in terms of their vowel for example

lakás + hoz	lakáshoz	to the house
szem + hoz	szemhez	to the eye
kör + hoz	körhöz	to the circle

Example 74

This can be modeled using different stem forms as in example 67,so we would model this as follows[15]

```
:hungarian_vowel_harmony a lemon:MorphPattern ;
  lemon:transform [
    lemon:onStem [
      dcr:vowelHarmony dcr:back
    ]
    lemon:rule "^([^öőüűeéií]*[aáoóuú].*)+~/$1+~" ;
  ] , [
    lemon:onStem [
      dcr:vowelHarmony dcr:intermediate
    ]
    lemon:rule "^([^aáoóuúeéií]*[öőüű].*)+~/$1+~" ;
  ] , [
    lemon:onStem [
      dcr:vowelHarmony dcr:front
    ]
    lemon:rule "^([^aáoóuúöőüű]*[eéií].*)+~/$1+~" ;
  ] .
```

Example 75

Here the regular expression matches the hole of the agglutination, indicated by the ^ indicating the start of string.

[15] Vowel harmony is currently not a property expressed in ISOcat so we will use the prefix `dcr` here. We also include the Hungarian long vowels (á, é, í, ó, ú, ő, ű) in this example.

Summary of vocabulary introduced in this section

Property	Description
`pattern`	Indicates the morphological pattern used by a lexical entry
`transform`	Indicates a single possible transform used by a morphological pattern
`rule`	The rule used by a particular transform
`generates`	The prototype for a form generated by a transform
`onStem`	The prototype for a from used as input for a transform
`nextTransform`	Indicates a potential sequence in a compound transform

3 Advanced Issues

3.1 Annotations and Global Restrictions

lemon does not implement a model for storing other meta-data and applying global restrictions on the lexicon, instead reusing the methodologies from RDF and OWL to achieve this.

3.1.1 Annotation schemes

One of the most important issues in terminology management is the ability to express the ownership and creation date of a lexicon and its entries. lemon has no mechanism of its own for performing this task instead, like many other RDF models it uses the Dublin Core vocabulary for this. The Dublin Core vocabulary allows the following to be stated:

- Creator: The creator of the given resource
- Description: A description of the resource. `lemon:definition` should be used for defining senses of lexical entries, `dublincore:description` for describing other parts of the model e.g., the lexicon object.
- Publisher: The publisher of the data
- Contributor: Any contributors to the given lexicon
- Date: The date a given resource was created
- Format: Generally should be specified as MIME. Use with care, as the format of the ontology may change.
- Identifier: An identifier of the lexicon or entry (useful if converting from another format)
- Source: The source of a given resource
- Coverage: The time and location covered by the resource (if relevant)
- Rights: Any copyright information

In addition, there are several annotations available in the OWL and RDF schemas that are useful for lexicon management

- `owl:backwardCompatibleWith`: The lexicon is compatible with some previous lexicon
- `rdfs:comment`: A comment about a resource
- `owl:deprecated`: Indicates the resource should not be used any more
- `owl:incompatibleWith`: The lexicon is incompatible with some previous lexicon
- `rdfs:isDefinedBy`: The resource is fully defined else where
- `owl:priorVersion`: Indicates an earlier version of the lexicon
- `rdfs:seeAlso`: Refer to another web resource
- `owl:versionInfo`: The version number of the lexicon

This information can of course simply be added as triples

```
@prefix dublincore: <http://purl.org/dc/elements/1.1/>
@prefix rdfs: <http://www.w3.org/2001/02/rdf-schema#>
@prefix owl: <http://www.w3.org/2002/07/owl#>

:myLexicon a lemon:Lexicon ;
   dublincore:creator "John McCrae" ;
   dublincore:date "2010-07-24"^^xsd:date ;
   rdfs:comment "An example lexicon from the lemon cookbook"@en ;
   lemon:language "en" ;
   lemon:entry :cat .

:cat a lemon:Word ;
   dublincore:creator "John McCrae" ;
   dublincore:date "2010-07-25"^^xsd:date ;
   rdfs:seeAlso "http://en.wikipedia.org/wiki/Cat" ;
   lemon:canonicalForm [ lemon:writtenRep "cat"@en ] .
   .
```

Example 76

The use of these annotations is another reason why the ranges of many properties are resources instead of literals as expected. For example we could now easily state the source of a definition as follows.

```
:cat lemon:sense [ lemon:reference ontology:cat ;
   lemon:definition [
      lemon:value "The cat is a small domesticated carnivorous animal"@en ;
      dublincore:source "http://en.wikipedia.org/wiki/Cat" ]
    ] .
```

Example 77

3.1.2 Global Information

LMF provides several classes for describing global information about the lexicon and constraints. lemon, however, does not need to do this as it can use the OWL language to specify decidable constraints on the lexicon. OWL will not be fully described here however there are many excellent resources available. A good place to start may be the W3C documents available at http://www.w3.org/standards/techs/owl#w3c_all. Here we shall show an example of how OWL can be used to specify that French nouns are only masculine or feminine.

```
:FrenchLexicon a owl:Class ;
  owl:equivalentTo [ owl:intersectionOf (
    lemon:Lexicon
    [ a owl:Restriction ;
       owl:onProperty lemon:language ;
       owl:hasValue "fr" ]
  ) ] ;
  rdfs:subClassOf [ a owl:Restriction ;
    owl:onProperty lemon:entry ;
    owl:allValuesForm :FrenchWord ] .

:Noun a owl:Class ;
  owl:equivalentTo [ a owl:Restriction ;
       owl:onProperty isocat:partOfSpeech ;
       owl:hasValue isocat:noun ] .

:FrenchWord a owl:Class .
```

```
:FrenchNoun a owl:Class ;
  owl:equivalentTo [ owl:intersectionOf (
    :FrenchWord :Noun ) ] ;
  rdfs:subClassOf [ a owl:Restriction ;
    owl:onProperty isocat:grammaticalGender ;
    owl:allValuesFrom [ owl:oneOf (
      isocat:masculine isocat:feminine ) ] ] .
```

The OWL restrictions are as follows in DL syntax:

$$FrenchLexicon \equiv Lexicon \sqcap \exists language."fr"$$

$$FrenchLexicon \sqsubseteq \forall entry.FrenchWord$$

$$Noun \equiv \exists partOfSpeech.noun$$

$$FrenchNoun \equiv FrenchWord \sqcap Noun$$

$$FrenchNoun \sqsubseteq \forall grammaticalGender.\{masculine, feminine\}$$

We first state that `FrenchLexicon`s are those lexica that have the value “fr” for the `language` property, and that all entries of `FrenchLexicon`s are `FrenchWord`s. We then define the class `Noun` as all things that have the value `noun` for the `partOfSpeech` property and then that `FrenchNoun`s are `FrenchWord`s that are also `Noun`s. Finally the key restriction is that every `FrenchNoun` has its `grammaticalGender` from the set {`masculine, feminine`}. (Note we do not assert here that the gender cannot be both masculine and feminine, as some words may be both, such as “après-midi”, however it is of course possible to do so).

Property	Description
`topic`	The topic of a lexicon or lexical entry
`definition`	The definition of a lexical entry relative to a sense.

3.2 The semantics of the lemon model

One of the most important aspects of a lemon model is the connection between the semantics contained within the lexicon, and the semantics in the ontology. Before we can hope to handle this difficult task we divide the tasks that are intended to be performed into three main groups

- **Lexical**: These are tasks that belong purely to the lexical layer of lemon and can be processed without strong semantic knowledge
  - *Tokenization*: Identifying words and components
  - *Chunking*: Identifying multi-word elements
  - *POS-Tagging*: Identifying labels for each token/chunk.
  - *Lemmatization*: Decomposing terms into inflectional components.
  - *Parsing*: Building a parse tree over a sentence.
  - *Coreference*: Deducing which elements in a sentence share a reference. (N.B., this task can often be considered to include weak semantic knowledge as well).
  - *Inflection*: Inverse of lemmatization
  - *Grammar Generation*: Inverse of parsing.
- **Correspondence**: Involves tasks that primarily focus on the task of connecting lexical realizations to conceptual objects.
  - *Annotation/Word Sense Disambiguation*: Converting the lexical representation to a semantic (ontological) relation
  - *Selection*: Inverse of grounding
- **Reasoning**: Making higher level deductions based on an ontological representation without lexical information.

3.2.1 Formal model of lemon senses

The primary model that we use to describe the semantics of lemon is inspired by this and views these tasks is that of an aligned semantic interpretation. Assume we have a language, L, on a vocabulary, Σ, i.e., $L \subseteq \Sigma^*$. Say we have a lexicon, $\mathcal{X}$, and we can define a language, $X \subseteq \mathcal{X}^*$, that constitutes all description of sentences in terms of the lexical entries it uses and the dependencies between the entries. Furthermore, we assume we have a lexicalization function l that maps a sentence in the language to a list of lexical representations taken from a lexicon, $\mathcal{X}$, that is that l is a function with the signature:

$$l : L \to X$$

We call such an l a *lexical interpretation* and represents the result of the lexical parsing stages applied above, we simplify the definition here by assuming that this results in a single unambiguous representation relative to the lexicon.

Similarly assume we have an ontology $\mathcal{O}$, which can be used with a logic, $\mathcal{L}$, such that we have a language $O \subseteq (\mathcal{O} \cup \mathcal{L})^*$. Again, we define a function s that maps a sentence in the language to its semantic representations, this function is called the *semantic interpretation* and has the signature:

$$s : L \to \mathcal{P}(O)$$

This function is indicated to input the set of correct results from all the lexical, correspondence and reasoning based analysis processes.

For example we take a simple sentence "The man bites the dog," we assume that the lexicon contains the entities $\{\text{man}, \text{dog}, \text{bite}(\cdot, \cdot)\}$ and the ontology contains two classes

Man and Dog and a property $bite$. Now, for example, we could have the folllowing interpretations:

$$l(\text{“The man bites the dog”}) = \text{bite}(\text{man}, \text{dog})$$

$$s(\text{“The man bites the dog”}) = \exists x, y : Man(x) \wedge Dog(y) \wedge bite(x, y)$$

We may then define an aligment, $A \subseteq (\mathcal{X} \times \mathcal{O})^*$ such that if we have $\mu \in X$, then for all $\sigma \in L$ such that $l(\sigma) = \mu$, we have $a_{\mu,p} \in A$ for each $p \in s(\sigma)$. Furthermore we require that if $\mu = \lambda_1 \ldots \lambda_n$, $p = o_1 \ldots o_m$ then we have $a_{\mu,p} = \alpha_1 \ldots \alpha_n$ such that $\alpha_i = (\lambda_i, o_j)$ for some $j : 1 \leq j \leq m$.

We then define the set of senses in the lexicon, $S \subseteq (\mathcal{X} \times \mathcal{O})$, as the set satisfying

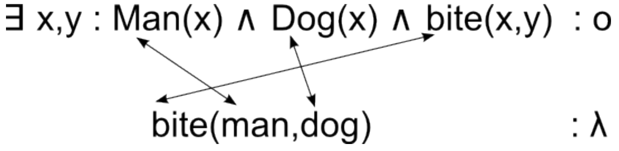
$$(\lambda, o) \in S \Leftrightarrow \exists \alpha_1 \ldots \alpha_n \in A, i \text{ such that } 1 \leq i \leq n \wedge \alpha_i = (\lambda, o)$$

As such our example may be explained by the following:

$$a_{\mu,p} = (\text{bite}(\cdot,\cdot), bite)(\text{man}, Man)(\text{dog}, Dog)$$

$$\{(\text{man}, Man), (\text{dog}, Dog), (\text{bite}(\cdot,\cdot), bite)\} \subseteq S$$

This can be illustrated as follows:



We view the sense as having three aspects:

- From the lexical point of view s indicates a disambiguated usage of a term, we call this a *Disambiguated Lexical Entry* or DALE. We can formally define the set of DALEs, D_λ for a lexical entry λ as follows:

$$D_\lambda = \{d_{\lambda,o} | (\lambda, o) \in S\}$$

- From the semantic view, s represents a sub-entity representing only the usage of a given class or property or individual when the given lexical entry is used to describe it. We call this an *projected ontology sub-entity* or POSE. We can formally define this as an ontology entity π_s such that:

$$\pi_s \sqsubseteq o$$

$$\pi_s \equiv \bot \Leftrightarrow o \equiv \bot$$

- Finally we can view S as a pair representing a possible alignment between the lexicographic and the semantic representations. We call this a *correspondence*.

We define the lemon `senseRelations`, `equivalent`, `narrower`, `broader` and `incompatible` and their relationship to the ontology as follows:

$$s_1 = s_2 \Leftrightarrow \pi_{s_1} \equiv \pi_{s_2} \tag{1}$$

$$s_1 < s_2 \Leftrightarrow \pi_{s_1} \sqsubset \pi_{s_2} \tag{2}$$

$$s_1 > s_2 \Leftrightarrow \pi_{s_1} \sqsupset \pi_{s_2} \tag{3}$$

$$s_1 \otimes s_2 \Leftrightarrow \pi_{s_1} \sqcap \pi_{s_2} = \bot \tag{4}$$

Theorem

If we have $o_1, o_2 \in \mathcal{O}$ and some corresponding senses $s_1, s_2 \in S$ then the following hold:

1. $o_1 \sqcap o_2 \equiv \bot \vdash s_1 \otimes s_2$

2. $s_1 = s_2 \vdash o_1 \sqcap o_2 \not\equiv \bot \vee o_2 \equiv o_1 \equiv \bot$

3. $s_1 > s_2 \vdash o_1 \sqcap o_2 \not\equiv \bot \vee o_2 \equiv \bot$

Proof

1. We have $\pi_{s_1} \sqsubseteq o_1$ and $\pi_{s_2} \sqsubseteq o_2$ hence $\pi_{s_1} \sqcap \pi_{s_2} \sqsubseteq o_1 \sqcap o_2$ hence we have $\pi_{s_1} \sqcap \pi_{s_2} \equiv \bot$ hence $s_1 \otimes s_2$

2. Assume $o_2 \not\equiv \bot$ hence it follows that $\pi_{s_2} \not\equiv \bot$, but we have $\pi_{s_2} \equiv \pi_{s_1}$ hence $o_1 \sqcap o_2 \sqsupseteq \pi_{s_1} \sqcap \pi_{s_2} \equiv \pi_{s_2} \not\equiv \bot$. Similarly for $o_1 \not\equiv \bot$.

3. (As 2.)

As a motivating example for making this distinction consider the example of “student” and “person”, these are asserted to be hypernyms in WordNet however in a strict ontology sense are not in a subsumption (subclass) relation. In fact “student” is a role of a “person” and would not be modeled the same in a well-reasoned ontology (i.e., DOLCE, which would distinguish “student” as anti-rigid and “person” as rigid). However for certain tasks, such as coreference resolution, it is necessary to have these “lexical” semantics, and as such we include them within the lexicon. As such it is possible to state the “student“ is a narrower sense of “person”, without affecting the correctness or the ontology.

In the context of coreference this modelling can be used for word sense disambiguation and coreference as it contains lexical semantics that are weaker than those in the ontology. In particular, there are four relations defined in lemon that are used to aid these tasks: equivalent, disjoint, broader and narrower. Each of these can be understood in the following manner.

- equivalent: Two DALEs which have equivalent senses can be substituted for each other without changing the semantics of the text. This should be assumed to be a positive feature for coreference.

- incompatible: Two DALEs which have incompatible (disjoint) senses cannot be substituted for each other without changing the semantics of the text. This should be considered to be a negative feature for coreference.

- broader/narrower: If a DALE is asserted to be broader than another DALE, then it can be substituted to give a more general semantics. This should be considered a positive feature for coreference, if the preceding term is narrower.

4 Relation to existing systems

4.1 LMF

LMF (Francopoulo et al., 2006) is a framework for describing lexicons and it has a strong influence on lemon. LMF was primarily intended for XML representation, and it has some constraints and factors we do not have in our RDF-based model

- No named links: Arcs in LMF are not named, in contrast to lemon where they must be named.
- Composition/Association: LMF models relationships between elements as either being compositional or associational, this is not considered important for our RDF framework
- No ontology: LMF does not have an existing ontological framework to build on like OWL.
- Little external linkage: In contrast to lemon, LMF does not refer to other frameworks, or link to other descriptive frameworks (with one exception see below).

As such, although we declare lemon to be highly compliant with LMF, we need now to describe the steps in converting a lemon model to a LMF lexicon.

lemon drops many elements

lemon has significantly fewer elements than LMF. This is partly due to the use of RDF which generally allows for a much more compact representation of complex interlinked data. We have also chosen to remove all nodes from LMF that are purely structural and carry no information such as SyntacticBehaviour and PredicativeRepresentation (it also appears that these elements may be artifacts of the XML representation). Due to the RDF representation it is not necessary to have nodes that describe links between elements. This means that elements such as RelatedForm, SenseRelation and MWEEdge are replaced with properties in lemon (formVariant, senseRelation and edge respectively). Finally, we found that by using RDF the mapping from syntax to semantics could be represented much more compactly so we have only senses and arguments covering LMF's SemanticPredicate, PredicativeRepresentation, SemanticArgument, SyntacticArgument, SynSemArgMap and SynSemCorrespondence.

lemon modifies some element names

Many lemon class names are different from their LMF equivalents. This is primarily to keep the names short and to avoid close naming between the classes and their associated properties. lemon also uses US English spelling, so the syntactic behavior has been respelled appropriately.

lemon	LMF
Frame	SubcategorizationFrame
LexicalSense	Sense
Node	MWENode
OntologyReference	MonolingualExternalRef
synBehavior	SyntacticBehaviour

Senses and semantics

One of the primary differences between LMF and lemon is the handling of semantics. In LMF, similarly to other resources such as WordNet, there are a fixed set of senses, however in lemon the sense ties the lexical entry to an ontology. There are several reasons for this: It is difficult to be certain of a fixed set of senses that fully capture all meanings of a word. Words are frequently used with meanings beyond the dictionary set of senses (e.g., metonymic raising). Different languages frequently divide senses in ways that are not natural in the other language (e.g., "river" in English and "rivière" and "fleuve" in French). Senses have no inherent meaning, i.e., practical systems can only understand the meaning of a word by grounding it in some ontologically described formalism. Ontology languages provide better modeling of semantics than a lexicon-based system can. Note that although lemon is intended to and likely to be used primarily with OWL, it is not necessary that the ontology system is OWL.

For these reasons the semantic modeling in lemon is more lightweight than that of LMF and so we have far fewer classes, as we do not wish to significantly duplicate the semantic modeling that should exist in the ontology.

Differences in the linguistic description

Most elements of the LMF vocabulary are defined by data values attached to the given instances, for example LMF says that a lexical entry may have properties such as partOfSpeech and grammaticalGender. In lemon such annotations are obtained by use of linguistic description ontologies, such as ISOcat. LMF's models ideally use ISOcat to provide its data categories so alignment in this case should be a trivial matter.

What lemon has that LMF doesn't

- Description of syntactic frames (subcategorization) by use of phrase structure
- Mapping to ontology predicates
- Pragmatic context
- Sense conditions and contexts

What lemon omits from LMF

- Morphology: lemon does not provide process descriptions of inflection generation.
- Lexeme properties in subcategorization frames: this is replaced by MWE/Frame links
- Synsets: lemon handles this through the use of ontology reference, i.e., the synset is the set of lexical senses that share a reference.
- Tests of cross-lingual mappings: replaced by sense conditions
- Global lexicon constraints: replaced by OWL restrictions

4.2 SKOS

lemon is designed to subsume most of the features of SKOS (Miles and Bechhofer, 2009), in particular the ability to state preferred labels and represent soft semantic relations. lemon uses the sub-properties of `lexicalForm` and `isReferenceOf` to more precisely capture the same semantics as SKOS's prefLabel, altLabel and hiddenLabel. The conversion is as follows:

	Canonical Form	Other Form	Abstract Form
Preferred Reference of	prefLabel	altLabel	hiddenLabel
Alternative Reference of	altLabel	altLabel	hiddenLabel
Hidden Reference of	hiddenLabel	hiddenLabel	hiddenLabel

As such it should be easy to convert a lemon model into a SKOS model, but converting the other way is harder as it involves deducing if the reason for an alternative or hidden label is pragmatic or morphosyntactic.

lemon also allows for SKOS's semantic properties broader, narrower and related to be mapped in a one-to-one manner to lemon's broader, narrower and senseRelation. Although Like the lemon properties are – like their SKOS counterparts – not transitive, lemon's concept of senses being related (i.e. `senseRelation`) is a super property of broader and narrower.

4.3 TBX

TBX (Term Base eXchange, ISO 30042) is a standard that is used for exchanging terminology databases and in particular translation memories. Therefore, a certain degree of interoperability with lemon would be useful. As with LMF, TBX contains much header information and this can mostly be either discarded or mapped onto the descriptions given in 4.7.2. The body of a TBX document consists primarily of a set of termEntrys each of which is a concept, these roughly correspond to sense/reference pairs. The conversion is trivial if there is a known ontology referenced by the term base (ideally indicated by a ref tag), however if this is not the case either the id attribute of termEntry can be used as a URI or some invented URI should be used. Doing this effectively raises the concepts expressed in the TBX document into a pseudo-ontology, so they can be used with lemon. In TBX each termEntry is first split into languages and then into terms, as such each langSet should be mapped to a lexicon in lemon. Finally, the terms should be mapped to lexical entries with a given form. As TBX does not have a simple method for indicating the standard form, either the forms should be guessed from properties such as lemma, or all just marked with the general form.

Conclusion

lemon is a model that allows for lexica to be shared on the semantic web and provides an efficient and simple representation. Nevertheless, the representation is capable of presenting a very wide range of lexical information through both its own properties and the easy integration of data categories from other sources. By incorporating other semantic web standards lemon can gain many basic features for expressing meta-data about the lexica and for describing constraints on the nature of the lexicon. lemon's innovative use of sense allows for lexical semantics to be represented in harmony with more powerful semantic representation provided by ontology languages such as OWL. Finally lemon maintains a high degree of interoperability with other standards, most importantly SKOS and LMF.

Thanks

We would like to thank the following people for their contribution to the model and discussion leading to its creation: Axel Pollers (DERI), Antoine Zimmermann (DERI), Dimitra Anastasiou (CNGL), Susan Marie Thomas (SAP), Christina Unger (CITEC), Sue Ellen Wright (Kent State University), Menzo Windhouwer (Universiteit van Amsterdam)

A FAQ

Q What is a form and what is a lexical entry? For example should initialisms be considered different forms?

A In general a form is considered to be "orthography-invariant", that is that the form should be a single entity across different orthographies or media. In contrast a lexical entry is "syntax-invariant", this means that it should always become the same entity once syntactic processing is complete. Initialism are clearly not "syntax-invariant" (as they consist of a different number of words), so should be different lexical entries.

Q Regarding the relations between lexical entries, if I understood it correctly, what are formVariant and lexicalVariant?

A In general this follows from the question of what is a form and what is a lexical entry?

Q Regarding the relations to be established between Senses, I am not so sure of the advantages in using the broarderSense and narrower relations compared to underspecification of these relations? Does introducing such relations not duplicate information already in an ontology?

A In fact the senses are quite underspecified, and the referenced ontology is intended to contain the precise and rigorous semantics. The use of sense relations is intended primarily for lexical processing using the lexicon.

Q Regarding the morphological information, is all this information needed for my application? Do we really need so much detail?

A No, of course not. In fact the minimal lemon model needs use only forms and lexical entries (and senses and references if referring to an ontology). But many applications may need something more than already exists in the model and lemon aims to be extensible; we do not require that the entire model is always implemented!

Q I do not understand the sentence "Although lemon also includes features to assign words to a pragmatic context, as this requires defining a pragmatic taxonomy it may not be a wise idea for many applications". What do you mean by a pragmatic taxonomy? I personally believe that the pragmatic sense of the word is already given by the word, but that we may want to make it explicit for practical reasons, in order to be able to ask the ontology to give us the most appropriate vocabulary for a certain context.

A In fact pragmatic context is really about the mapping, it is not even strictly necessary if there is no mapping. There are also examples where the meaning of the word is defined by its pragmatic context, for example “bitch” refers to a “female dog” in veterinary context, but has a vulgar meaning in another context.

Q Regarding the ”three-faceted understanding of sense”, is this not too subtle to be incorporated into the model?

A For many applications it may be, but for some applications a strong definition of how the mappings correspond to implementations is useful. Much like any format different levels of understanding can be used, i.e., RDF manages to be used quite effectively in contexts where its extensional model semantics are not relevant.

Q Components are used to describe both word tokenization and term decomposition. How should I represent “deutsches Schweineschnitzel”

A This should generally be represented by decomposing it first into “deutsch(es)” and “Schweineschnitzel” and marking both of these as `lemon:Word`s, then applying a decomposition of “Schweineschnitzel” to “Schwein(e)” and “Schnitzel”.

Q Can a property of a lexical entry have multiple values? E.g., can I assign a lexical entry to multiple subject fields?

A Sure, RDF also you to assert multiple triples with the same subject and property

Q Could I have a hierachical value for the property of a lexical entry?. E.g., if I say my lexical entry is a proper noun, could I deduce it is also a noun?

A Limitations of RDF/OWL make this very difficult to do in general. There are three solutions

1. Handle it yourself. Especially if you are referring to a lexicon standard in a non-RDF format (e.g., ISOcat’s DCIF) then this may be the best approach
2. Use punning. OWL2 allows classes to be punned to individuals, this is suitable for some modelling (e.g., GOLD), but will not deduce the desired triples automatically
3. Add OWL axioms to the description ontology. For example the above case could be handled with

$$\exists partOfSpeech.properNoun \sqsubseteq \exists partOfSpeech.noun$$

Q You use blank nodes frequently through out this document is this required in lemon?

A No. It just keeps the examples more readable

Q How do I represent how superlatives (e.g., “biggest”) map to an ontology

A Superlatives are difficult to represent well, as they clash with the open world semantics inherent to OWL and RDF. In particular it is difficult to say what the greatest element of a non-closed set would in fact be. You could of course map it to an object property, whose range is some finite set (e.g., RDF’s List or an OWL enumeration), and then it could be mapped like a comparative.

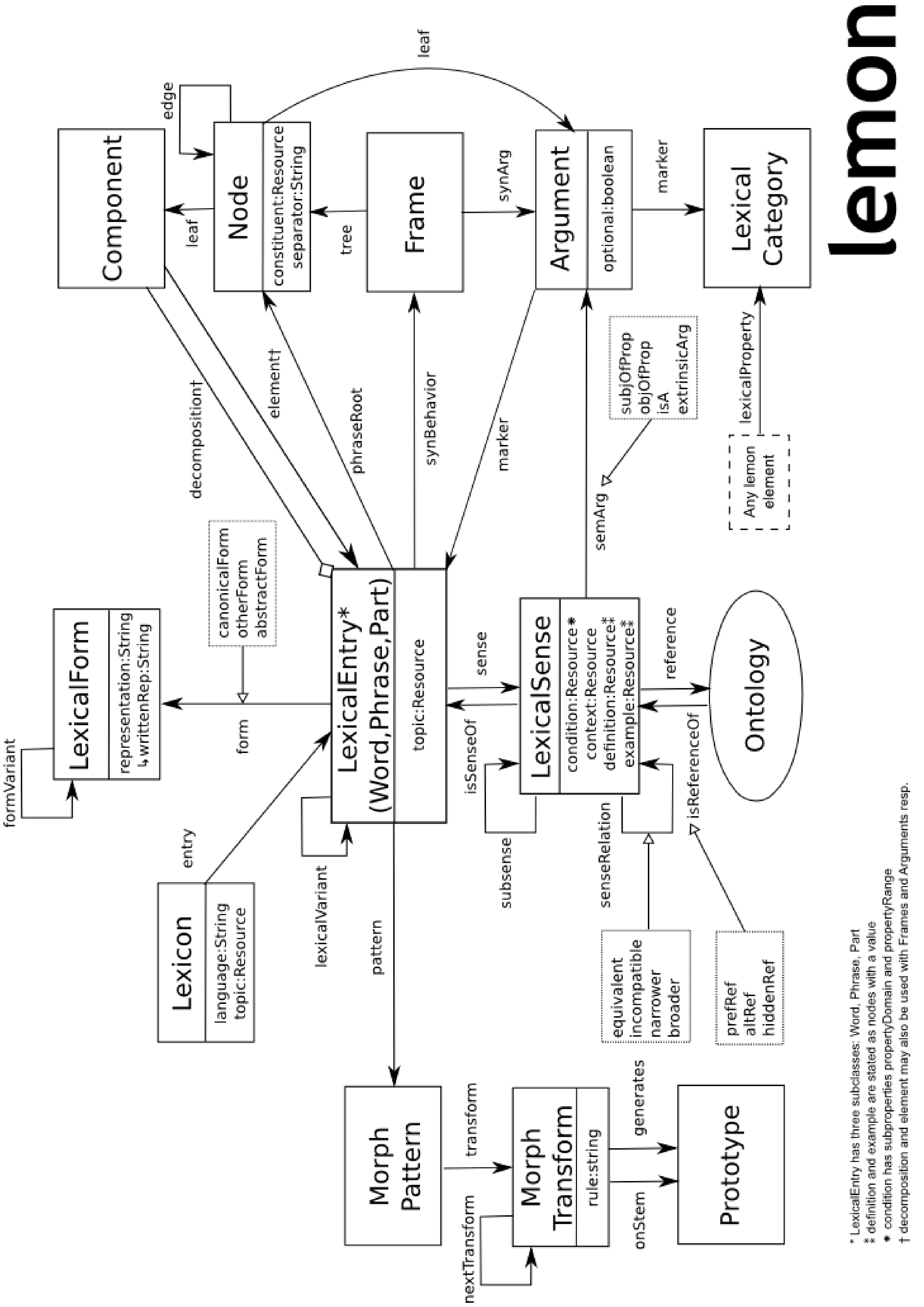
Q As in multi-word expressions elements are inflected shouldn’t decomposition be by forms not lexical entries?

A We have chosen to place it at the lexical entry level and use properties for several reasons, but in particular that the decomposition should be the same for all forms of the lexical entry. For example in the case of “number of employees” it is clear that both that canonical form and the plural form “numbers of employees” should have the same decomposition, even though they both use the inflected form “employees.” Doing otherwise would require that the decomposition be repeated (potentially many times). Also it should be noted that this decomposition is semantically disambiguated with “number” strictly in the sense of an amount and not as a word or symbol. For general compactness and the ability to interface with the phrase structure model we limit decomposition to the lexical entry level and then assume that this decomposition can also be soundly applied at the lexical and semantic level.

B LMF comparison

LMF	lemon	Notes
LexicalResource GlobalInformation	`rdf:RDF`	Root element of RDF/XML document is `rdf:RDF`
Lexicon	Lexicon	
LexicalEntry	LexicalEntry	
Form	LexicalForm	
Representation FormRepresentation	representation	
TextRepresentation	writtenRep	
Definition	definition	
Statement	*no equivalent*	Use `rdfs:comment` on the definition
Lemma WordForm Stem	canonicalForm otherForm $\sim$ abstractForm	See discussion in section 1
RelatedForm	formRelation	
ListOfComponents	ComponentList	
Component	Component	
Equivalent	equivalent	
Context	context	
Subject Field	topic	
Syntactic Behaviour	synBehavior	
SubcategorizationFrame	Frame	
LexemeProperty	*no equivalent*	Generally should be modelled at the lexical entry level
Subcategorization-FrameSet SynArgMap	*not necessary*	Instead use the same argument entity with multiple frames to model sets of frames
SyntacticArgument SemanticArgument SynSemArgMap SynSemCorrespondence	synArg semArg *not necessary*	By using the same entity as both syntactic and semantic argument the encoding in lemon is more economical; see section **??**
MonolingualExternalRef	reference	Plays a slightly different and significantly more important role in lemon
Sense SenseAxis TransferAxis PredicativeRepresentation SemanticPredicate Synset	LexicalSense	As predicates are assumed to exist the ontology, they are not modelled in the lexicon. Synsets given implicitly as the set of senses that have the same reference
SenseRelation PredicateRelation SynsetRelation SenseAxisRelation TransferAxisRelation	senseRelation	
SourceTest TargetTest	condition	Condition on the sense instead of on the transfer axis, as senses are more precise
InterlingualExternalRef	*no equivalent*	Can be modelled with `dublincore:source`
ArgumentRelation	*no equivalent*	It is not clear what argument relations exist
ContextAxis ContextAxisRelation	*no equivalent*	Should be modelled with an ontology of contexts, not in the lexicon
morphological pattern extensions		Replace by lemon morphology extensions, section 2.5.
MWEPattern	phraseRoot	
MWENode	Node	
MWEEdge	edge	
MWELex	leaf	
constraint expression extensions	*no equivalent*	Use OWL

C lemon **model diagram**



* LexicalEntry has three subclasses: Word, Phrase, Part
※ definition and example are stated as nodes with a value
✱ condition has subproperties propertyDomain and propertyRange
† decomposition and element may also be used with Frames and Arguments resp.

References

P. Cimiano, P. Buitelaar, J. McCrae, and M. Sintek. LexInfo: A Declarative Model for the Lexicon-Ontology Interface. *Web Semantics: Science, Services and Agents on the World Wide Web*, 9(1):29 – 51, 2011.

G. Francopoulo, M. George, N. Calzolari, M. Monachini, N. Bel, M. Pet, and C. Soria. Lexical markup framework (LMF). In *Proceedings of the Fifth International Conference on Language Resource and Evaluation (LREC'O6)*, 2006.

A. Miles and S. Bechhofer. *SKOS Simple Knowledge Organization System Reference*, 2009. URL `http://www.w3.org/TR/skos-reference/`. Accessed 19 October 2010.

E. Montiel-Ponsoda, G. Aguado de Cea, A. Gómez-Pérez, and W. Peters. Modelling multilinguality in ontologies. In *Proceedings of the 21st International Conference on Computational Linguistics (COLING)*, 2008.

L. Romary. Standardization of the formal representation of lexical information for NLP. In *Dictionaries: An International Encyclopedia of Lexicography.* Mouton de Gruyter, 2010.