Lexicon Interchange Format
A Description

Martin Hosken,
SIL Non-Roman Script Initiative (NRSI)

Table of Contents

Introduction

Over the years SIL members have used a variety of markup schemes for representing dictionaries and lexicons. For the most part these have been Standard Format based. A major step forward was taken with the creation of MDF[1] which not only provided a tool for typesetting dictionaries, but also provided a complete schema for their representation. Since then nearly all new dictionaries and lexicons have used an MDF based conceptual model if not the actual schema and markup.

MDF is a Standard Format based schema and with the emergence of the XML standard and newer computing technologies there is a need for an interchange format that can work with these newer technologies. The most important such technology at this time comes from the Fieldworks project in the form of FLEX. But even this has its conceptual model rooted in the MDF model.

This document describes LIFT, a lexicon interchange format. We start by describing the types of various elements including their attributes and possible child elements. UML diagrams are also included to give an overview of how types relate. Following the language description are a number of examples. For those who learn best from examples, this is a good place to start reading just to get a feel for LIFT and then return to the main text to understand the details. The examples are included to show how various key MDF concepts map into LIFT and as such these models for encoding such concepts should be considered normative. Finally there is an implementation section that addresses various implementation issues.

Some of the key features of LIFT are:

- Provides a way of encoding all MDF concepts and is a complete language.
- Handles text represented in multiple scripts (for example a language that has multiple orthographies)
- Can store any number of analysis languages.
- Acknowledges the realities of dictionary development and is extensible to allow the storage of information not currently covered by the main conceptual model.

Since this is an XML format, all data is considered to be in Unicode. For archiving, users should ensure that their data is in true Unicode with no reinterpreted characters arising from non-mapped legacy encoded data.

One aspect of an interchange format is that it is designed to just hold the lexicon data. Although it has a `header` element, this isn't about storing the introduction or rubric of a dictionary. Such information is a straight document and there are plenty of document formats out there and this is not one of them! The practicalities of working with both the lexical data and the rubric in one file are nearly impossible and as such are aimed at different things. The rubric is about the introduction to one dictionary while the lexical data is used for many things including the typesetting of one or more dictionaries.

As such, LIFT is not a document format. A careful analysis of the MDF schema shows it to be halfway between a true database schema and a document format: field order is important; information may be stored in a way that is convenient for printing but not for data analysis. LIFT, on the other hand, goes all the way to a pure database type of storage where, for the most part, element order is not important. Where it is, usually for repeated elements of the same type, then that is described in this document.

Conformance

In addition, this is an interchange format. An interchange format differs slightly from an archival format or from an application specific format. One of the needs is that any application that can put data into this must be able to store all the data it needs in this format and be able to get it back out again without loss. That doesn't mean that other applications have to make sense

[1] Multi-Dictionary Formatter

of it, or even be able to conserve all of it (although that would be a big help). Further still, an interchange format is not primarily about enforcing some higher level of quality on the source data than was already there. LIFT demands a structural integrity just by converting data into an XML based format. Added to that, to ensure that data stored in LIFT is useful over a longer period and across applications, LIFT conformance introduces the following requirements.

- there are no two ids of the same type of object that are identical, including treating senses and subsenses as entries.
- there are corresponding ids for all refids
- there is a definition for all field tags used
- that in a context where only one language data is expected no data from another language is also provided.
- that there are no PUA characters or that PUA characters are only from the corporate PUA area.

It is recognised that it is unlikely that an application can necessarily get to full LIFT conformance in one step. This is discussed further in the section on implementation.

Since the lexicon is often a key database referenced by multiple applications and cross referenced by other data sets, it is necessary to ensure that refids do not change otherwise cross data linkages are broken. Thus, at the application level, LIFT conformance requires that if a LIFT database is read in and the same database is to be output, that any refids not be changed. Once stability of refids is guaranteed, it buys applications many benefits, particularly in the area of data merging.

Stability

An important consideration for a file format that may be considered for use for archiving is that of stability. We have to assume that the first version of the language will not be sufficient for every foreseeable need. Therefore what can be done now to aid forward compatibility?

The first principle is that nothing is ever removed from the language. That is any attributes or elements that are part of the language specification will never be removed in a later version. They may become deprecated and applications may eventually stop supporting them, but it will always be valid for data to use them. This means that no data ever goes out of date. In addition, all changes to the language will involve additional attributes and elements.

For an application to be able to support later versions, therefore, requires that it not restrict the language to only those elements that it knows about. It is not an error to add an unknown attribute. If a stronger level of conformance is required then an application may assume that for a particular file using a particular language version, it will not use anything outside that version of the language specification. Therefore if an application knows the language up to a particular version it can do attribute and element checking, but if it receives a file of a later version it cannot assume to know what extra attributes and elements have been added.

Types

Datatypes

We start with the simplest types which have no attributes or children.

int

This is simply an integer number, stored as a string representation in base 10.

float

A number with integer and decimal parts separated by a period. No scientific notations are supported.

string

This fundamental type is just a sequence of Unicode characters.

PCDATA

The basis of all text in the interchange format is a Unicode string. In implementation terms a `PCDATA` is no different from a `string` but is differentiated in this design to indicate text in a language instead of a representation of some information, e.g. language.

datetime

This is the same type as an XML Schema datetime type. See http://www.w3.org/TR/xmlschema-2/#dateTime for details. In summary a time is a string representing a date and time in the following format: *yyyy*`-`*MM*`-`*dd*`T`*hh*`:`*mm*`:`*sszzzz*. Times are given relative to GMT, thus if no timezone information is included then the time is considered to be in GMT. Likewise if no time is included then it is assumed to be 0, i.e. Midnight GMT at the start of the given day.

yyyy	represents a 4 digit year relative to AD 0 (yes it can be negative)
`-`	separator
MM	represents the month as 2 digits from 01 to 12.
`-`	separator
dd	represents the day of the month as 2 digits from 01 to 31.
`T`	Time separator. This and all following it are optional as a single unit (i.e. if the `T` exists, so must all the time elements unless marked as optional).
hh	represents the hour as 2 digits from 00 to 23. The hour can be 24 if the rest of the time is 0.
`:`	separator
mm	represents the minutes of the hour as two digits from 00 to 59
`:`	separator
ss	represents the seconds as 2 digits from 00 to 59
zzzz	represents optional timezone information in the form: `+`|`-`*hh*`:`*mm* indicating the time zone is ahead or behind GMT by the given number of hours and minutes. Optionally a timezone value of `Z`[1] indicates an explicit zero offset from GMT.

[1] That's capital Z

key

In a number of places in the schema, a key is used to identify a particular item from a list. A key is a string that acts both as a simple identifier that can be used to locate a particular element in a list of elements of the same type, but also may be used as a reserved identifier. Keys are used to identify particular range elements in a range set and also to identify a particular range set. More information will be provided on the sections on range sets.

lang

This is a language tag and follows RFC 4646bis or any superseding document.[1] Full details of how to tag text for language, script, region, etc. is beyond the scope of this document. Language tags should follow the standard wherever they can, if, for example, a particular orthography needs to be marked, that has not been included in the relevant standard's list, it may be specified using a private use extension. For example tpu-Latn-x-testing1.

In order that string comparison may be used for language tags, in addition to conforming to RFC 4646bis, a language tag must be as short as it can be while still representing all the information required. Thus redundant script subtags (due to script suppression) and region tags must be removed.

refid

A refid is an identifier for a lexical entry or a sense. The ambiguity is intended since it allows referrers to refer either to an entry or sense depending on the data need. I.e. if the sense is not known then an entry reference is sufficient otherwise a sense reference is preferred. A refid is a string.

`refid`s are ideal for inter application linkage, or for cross linkage with other data sets. This is particularly true for a lexicon. For this reason, there is an added constraint on a `refi`d that once set, it must not be changed. If it is changed then other applications are at liberty to consider it to identify a different item with no linkage to the original item.

URL

A URL datatype is a string containing a URL (Universal Resource Locator) as specified in RFC 3986.

Elements

The core elements are described based on the UML diagram they occur on. We start with the base diagram (leaving some types to the entry diagram) and then examine the entry diagram and finally the header diagram. Any unspecified types are considered to be `string`.

Unless otherwise stated, content elements may occur in any order in the parent.

span

A span is a unicode string that is marked according to its language and formatting information. In addition, spans may occur within spans, allowing changes of formatting within a string. The span is the fundamental string type for textual information including lexical forms, descriptions and glosses. While LIFT supports formatting within such strings, not all applications can handle such enriched text. For this reason, a span may be converted to a simple Unicode string by stripping all embedded markers and retaining the remaining text. A span is the only element in the LIFT schema that has mixed content consisting of Unicode text and other spans.

Space characters within `span`s are significant and are treated as follows. All multiple spacing characters are reduced to a single space. Spaces around the `span` element are significant and are not reduced to below one space if present. LIFT makes no effort to model document structure such as paragraphs and where multiple paragraphs may be required in, for example, a `note`

[1] RFC 4646 supersedes RFC 3066 which in its turn supersedes RFC 1511. RFC 4646 is currently being rewritten and will be superseded in its turn.

field, plain text approaches should be used either using Unicode paragraph characters. For maximum transportability, Unicode paragraph characters should be used.

Outer level spans that merely mark the language as being the same as that of the `form` a `text` element is in, are redundant and should not be output.

Attributes

`lang` [Optional, `lang`] Specifies the language and script of the text. Notice that in some contexts, particularly where vernacular text is expected, if the language component of the language tag is not the same as the expected vernacular language then the data may be ignored by a process and probably not stored in a subsequent saving of the data.

`href` [Optional, `URL`] The text included in the span is to made into a hotlink to the given URL, if the application can do that.

`class` [Optional] Gives the style name or class of the text.

Content

#PCDATA The core content of a `span` is unicode text

`span` [Optional, Multiple, `span`] The content may have other `span` elements embedded in it (which in turn may have other `span` elements embedded in their content).

Naturally, since the content is mixed, order is significant within a `span`.

text

This is a mixed content element that contains textual data mixed with `span`s only. The element takes no attributes inheriting its language information from its parent element, a `form`.

form

A `form` is a representation of a string in a particular language and script as specified by the `lang` attribute. It may optionally contain `trait`s and the textual content is held in the `text` child element.

Attributes

`lang` [Required, `lang`] gives the language tag for the text.

Content

`text` [Required, `text`] holds the text of the `form` in a single language and writing system as specified by the `lang` attribute.

`annotation` [Optional, Multiple, `annotation`] contains various metadata for this textual element including status information.

multitext

There are no occurrences of a `span` being used to store content except as part of a `multitext`. This element allows for different representations of the same information in a given language. For example it allows for different representations of a lexical form, for example in an orthography or in phonemic form.

Inheritance

`text` [Optional] If there is only one `form` the `form` element itself is optional and a `multitext` may consist of a single text node containing the contents of the text. This means that if there is no `form` there is no `span` capability.

Content

`form` [Optional, Multiple, `form`] Each representation of the information is held in a `form` element.

trait [Optional, Multiple, `trait`] These attributes may be referenced by the `form` elements in their `traits` attribute.

URLRef

This is a URL with a caption. It is used to represent media items, for example for pictures in a Sense or a sound file for a phonetic representation.

Attributes

href [Required, `URL`] is the URL of the resource

Content

label [Optional, `multitext`] Gives a multilingual representation of the caption for the media item.

field

A `field` is a generalised element to allow an application to store information in a LIFT file that isn't explicitly described in the LIFT standard. Fields are described as part of the header information so that applications can give some descriptive meaning to the information they add to a file.

Inheritance

multitext Stores the language and multiscript form of the information.

Attributes

type [Required, `key`] The identifying key that gives the field name of the field. Applications may share data by agreeing on the `tag` to use.

dateCreated [Optional, `datetime`] Gives the creation date of the field.

dateModified[Optional, `datetime`] Gives the modification time of the field.

Content

trait [Optional, Multiple, `flag`] Gives additional information about the field.

form [Optional, Multiple, `span`] The multilingual representation of the field content for a particular writing system.

annotation [Optional, Multiple, `annotation`] Adds metainformation describing the element.

trait

A trait is an important mechanism for giving type information to an object or adding binary constraints. There are many ways of interpretting a `trait`.

A trait is simply a reference to a single `range-element` in a `range`. It can be used to give the dialect for a variant or the status of an entry. The semantics of a `trait` in a particular context is given by the parent object and also by the `range` and `range-element` being referred to. Where no `range` is linked the `name` is informal or resolved by name.

Attributes

name [Required, `key`]. This is the identifier of a particular `range`.

value [Required, `key`]. This is the identifier of a partcular `range-element` within the referred `range`. Since `range`s are optional, the `value` attribute must be human readable and usable in the stead of the `range`.

id [Optional, `key`] Gives the particular `trait` an identifier such that it can be referenced by a sibling element. The `id` key only needs to be unique within the parent element, although globale `key`s may be used. There is no requirement that the `key` keeps its value across different versions of the file.

Content

annotation [Optional, Multiple, `annotation`] Contains meta information about the trait. For example it may give a status or an edit history for the trait.

annotation

The `annotation` element provides a mechanism for adding meta-information to almost any element. It includes the option to specify `who` made the annotation, when including a comment. An `annotation` is also a `trait` and gives a historical description of when a particular flag value was set to what when and perhaps by whom. An annotation does not give a current flag value as a `trait` would give. It is purely commentary. It differs from a `note` in that it is designed to hold meta-information about its parent rather than content of the parent.

Inheritance

multitext Gives a comment on the annotation.

Attributes

name [Required, `key`] Gives the range set from which this status value is taken,

value [Required, `key`] Contains the value of the the `type` either now or in the past.

who [Optional, `key`] Specifies are particular element from the `users` range.

when [Optional, `datetime`] Specifies the date/time that the trait was set.

extensible

Many types contain the same set of elements that are used for adding extra information in a controlled extensible way. This type is used to provide that extra information and is only inherited from in order to add those elements.

Content

dateCreated [Optional, `datetime`] Contains a date/timestamp saying when the element was added to the dictionary. Note that this attribute is not required.

dateModified [Optional, `datetime`] Contains a date/timestamp saying when the element was last changed. Note that an application is not required to store this attribute. But if it does, then the semantics of `dateModified` are that if an element is modified then the `dateModified` attribute should be updated or removed. In addition, an element is considered changed if any of its children are modified.

field [Optional, Multiple, `field`] Holds extra textual information.

trait [Optional, Multiple, `flag`] Adds type, filter, trait information.

annotation [Optional, Multiple, `annotation`] Adds metainformation describing the element.

When describing types that inherit from `extensible` for the most part the content elements so added will not be described unless they have a particular meaning in the context of the type being described.

note

A `note` is used for storing descriptive information of many kinds including comments, bibliographic information and domain specific notes. Notes are used to hold informational content rather than meta-information about an element, for which an `annotation` should be used.

Inheritance

multitext Stores the note content, giving its language.

extensible Adds `date`, `field` and `trait` elements to the content for extensibility.

Attributes

type [Optional, `key`] Gives the type of note by reference to a range-element in the `note-type` range. There is only one `note` with a given `type` in any parent element. Thus translations of the `note` are held as different forms of the one `note`.

phonetic

This represents a single phonetic representation.

Inheritance

multitext Allows for storage of different representation forms of the text.

extensible Adds `date`, `field` and `trait` elements to the content for extensibility.

Content

media [Optional, Multiple, `URLRef`] Stores an audio representation of the text.

form [Optional, Multiple, `span`] Stores the phonetic representation using whichever writing system: IPA, Americanist, etc..

etymology

An `etymology` is for describing lexical relations with a word that is not an entry in the lexicon. For example proto forms. As such it holds a representation of the word and a gloss of that word rather than a reference to an `Entry` or `Sense` in the lexicon.

Inheritance

extensible Adds `date`, `field` and `trait` elements to the content for extensibility.

Attributes

type [Required, `key`] Gives the etymological relationship between this sense and some other word in another language. This is a reference to a range-element in the `etymology` range.

source [Required, `string`] Gives the language for the source language of the etymological relation. Where possible a `lang` type code should be used, but proto languages tend not to appear in the Ethnologue and so a uniquely identifying name may be given here.

Contents

gloss [Optional, Multiple, `form`] Gives glosses of the word that the etymological relationship is with.

form [Required, `form`] Holds the form of the etymological reference.

grammatical-info

The grammatical information of a `Sense` can be a linguistic nightmare, but it is relatively simple as a structural item. It is just a reference to a `range-element` in the `grammatical-info` range.

Attributes

value [Required, `key`] The part of speech tag into the `grammatical-info` range. Notice that generally, the `value` attribute *is* the grammatical information identifier and that an actual `range-element` is only needed if translations of the part of speech is required, or that range set checking is required.

Content

trait [Optional, Multiple, `trait`] Allows grammatical information to have attributes.

reversal

Reverse indexes in a dictionary are a key tool for enabling a wider use of a dictionary.

Inheritance

multitext Stores the reversal entry with its language.

Attributes

type [Optional, key] Gives the type of the reversal as a range-element in the reversal-type range. Generally type is required, but where it is absent, then all such reversals are considered to be of a particular type of blank, unless the reversal is being used as the main for another reversal in which case it takes the type of its containing reversal.

Content

main [Optional, reversal] Reversals may form an entry sub-entry type hierarchy. This gives the parent reversal in any such hierarchy if one is so desired. The full tree is given here. Since the type attribute is shared with the reversal it is not set on any main element.

grammatical-info [Optional, grammatical-info] The mapping between the grammatical information for a sense may not be the same for a particular reversal. This allows a reversal relation to specify what the grammatical information is in the reversal language.

translation

A translation is simply a multitext with an optional translation type attribute. Thus multiple translations of the same type (literal, free, back translation, etc.) but of different languages are merely the same translation with different forms.

Attributes

type [Optional, key] Gives the type of the translation. This is also a key into the translation-types range set.

example

An example gives an example sentence or phrase in the language and glosses in of that example in other languages.

Inheritance

extensible Adds date, field and trait elements to the content for extensibility.

multitext Stores the content of the example in the main language of the dictionary.

Attributes

source [Optional, key] Reference by which another application may refer to this example or is a reference into another database of texts, for example. The key is a reference into an examples range set.

Content

translation [Optional, Multiple, translation] Gives translations of the example into different languages. Each translation is of a single type and contains all the translations of that type into multiple languages and writing systems.

relation

This element is used for lexical relations. The modern understanding of a lexical relation is that it is not owned by any of the senses to which it refers. Instead it is a bidirectional relationship between two sets of senses. For the most part such relations are 1:1 or n:1 (or 1:n depending on how you look at them). This means that for many models, including MDF, ownership of the 1: side of a relation is appropriate, if not strictly accurate. In addition the presence of a relation in

a `Sense` is a strong indication as to whether that relation should be published or not. Further, relations are included here for ease of implementation to facilitate single pass data conversion.

Inheritance

`extensible` Adds `date`, `field` and `trait` elements to the content for extensibility.

Attributes

`type` [Required, `key`] Is the type of the particular lexical relation. It is also a reference into the `lexical-relations range-element`.The `name` is given in terms of the `ref`erenced sense/entry's relation to this entry. For example:

```
<entry id="apple"><sense>
   <relation name="generic" ref="fruit"/>
</sense></entry>
```

`ref` [Required, `refid`] This is the other end of the relation either a `Sense` or an `Entry`.

`order` [Optional, `int`] Gives the relative ordering of relations of a given type when a multiple relation is being described. For example a *component* relation maps to a sequence of `Entry`s or `Sense`s. If no `order` attribute is present, then document order is used.

`usage` [Optional, `multitext`] Gives information on usage in a particular language.

variant

`variant`s are used for all sorts of variation. They are used for free variation in phonemic or orthography, dialectal variants in phonetics, or almost any kind of constraint and combination of constraints one can desire.

Inheritance

`extensible` Adds `date`, `field` and `trait` elements to the content for extensibility.

`multitext` Gives the variation to the main lexical form.

Attributes

`ref` [Optional, `refentry`] Gives the variation as a reference to another entry or sense rather than specifying the `form`.

Content

`pronunciation` [Optional, Multiple, `phonetic`] Holds the phonetic variant whether it is that this is a variation in phonetics only or that the phonetic variation arises because of an orthographiic or phonemic variation.

`relation` [Optional, Multiple, `relation`] Some variants have a lexical relationship with other senses or entries in the lexicon. For example a paradigm variant may have a component relation with a root and suffix in the lexicon.

Sense

An `Entry` is made up of a number of senses. Each sense corresponds to a part of speech. While a Sense may have multiple parts of speech this is only for situations where the language can have an identical sense with multiple parts of speech (e.g. stative verbs which are verbs and adjectives at the same time[1]).

Inheritance

`extensible` Adds `date`, `field` and `trait` elements to the content for extensibility.

Attributes

`id` [Optional, `refid`] This gives an identifier for this `Sense` so that things can refer to it. The `id` is unique across all `Sense`s in the lexicon and all `Entry`s as well.

[1] Is that true. Some linguist help with a better example please

order [Optional `int`] A number that is used to give the relative order of senses within an entry. If there is more than one sense in an entry and no `order` attribute then document order is used.

Content

grammatical-info [Optional, `grammi`] Grammatical information.

gloss [Optional, Multiple, `form`] Each `gloss` is a single string in a single language and writing system. If it is necessary to semantically link glosses either because they are of the same language but different writing systems or because the glosses really are the same semantically across languages, then the `traits` list attribute may be used to link via a `trait`.

definition [Optional, `multitext`] Gives the definition in multiple languages and writing systems.

relation [Optional, Multiple, `relation`] While a lexical relation isn't strictly owned by a sense it is a good place to hold it.

note [Optional, Multiple, `note`] There are lots of different types of notes.

example [Optional, Multiple, `example`] Examples may be used for different target audiences.

reversal [Optional, Multiple, `reversal`] There may be different reversal indexes.

illustration [Optional, Multiple, `URLref`] The picture doesn't have to be static.

subsense [Optional, Multiple, `Sense`] Sense can form a hierarchy.

Entry

This is the core of a lexicon. A Lexicon is made up of a set of Entries. Notice that the entry is not the lexeme. The lexical form is simply an attribute of the entry not the entry an attribute of the lexical form. This allows for a richer entry description.

Inheritance

extensible Adds `date`, `field` and `trait` elements to the content for extensibility.

Attributes

id [Optional, `refid`] This gives a unique identifier to this Entry. Notice that this is unique across all `Entry`s and all `Sense`s. For simple single sense entries, one approach is to use the lexical form as the `id` for the `Entry` and to use the lexical form with a following _ for the `id` of the `Sense`. See the Examples section for examples of this approach.

order [Optional, `int`] This is the homograph number. If there are homographs and the `order` attribute is missing. document order will be used.

guid [Optional, `string`] Holds a unique identifier for an entry primarily for merging purposes. While `guid` is optional, applications should conserve its value if present.

dateDeleted [Optional, `datetime`] If this attribute exists then it indicates that the particular `entry` has been deleted. For security purposes it is wise to delete all the contents of an `entry` when setting this attribute. The primary purpose is to ensure the `id` of entries across versions of the file for merging purposes. There is no requirement for applications to keep deleted entries.

Content

lexical-unit[Optional, `multitext`] The lexical form is the primary lexical form as is found as the primary lexical form in the source data models for this standard.

citation [Optional, `multitext`] This is the form that is to be printed in the dictionary.

pronunciation [Optional, Multiple, `phonetic`] There can be multiple phonetic forms of an entry. Their presence implies free variation.

variant [Optional, Multiple, `variant`] Any constrained variants or free orthographic variants.

sense [Optional, Multiple, `Sense`] This is where the definition goes. A `sense` is not required allowing for word forms which only have relationships with other particular senses and entries but otherwise are not part of the dictionary.

note [Optional, Multiple, `note`] The more notes you keep the better.

relation [Optional, Multiple, `relation`] Gives a lexical relationship between this entry and another `Entry` or `Sense`.

etymology [Optional, Multiple, `etymology`] Differs from a lexical relation in that it has no referant in the lexicon. The other word is outside the language.

field-defn

A field definition gives information about a particular field type that may be used by an application to add information not part of the LIFT standard.

Inheritance

multitext Contains a multilingual description of this particular field.

Attributes

tag [Required, `key`] This key corresponds to the `tag` attribute found in all `field`s for which this is the definition.

field-defns

This is a simple list of `field-defn`.

Content

field [Optional, Multiple, `field-defn`] The field definitions for all the field types used in this document.

ranges

This is an array of `range`. Details of the class `range` and `range-element` are found in the section on Lift Ranges.

Content

range [Optional, Multiple, `range-ref`] Gives information about where to find the definition of an associated `range`.

header

This holds the header information for a LIFT file including `range-ref` information and added `field-defns` along with any `style-defns`.

Content

The content of a `header` is the definitions and extensions of the various `ranges`, `fields`, `sdomains`, `styles` used by the lexicon. In addition, other files may be referenced from which `header` information will be included. Given that information is additive and no deletion is possible, the only concern is if there is a clash over a definition, for example there are two descriptions in English for a particular `range-element`. The precise result is application specific.

description [Optional, `multitext`] Contains a multilingual description of the lexicon for information purposes only.

ranges [Optional, `ranges`] Contains all the `range-ref` information.

fields [Optional, `field-defns`] Contains definitions for all the `field` types used in the document.

lift

This is the root node of the document and contains a header and all the entries in the database.

Attributes

version [Required] Specifies the lift language version number. This gives an indication of the minimum language version required to fully support this file. The default value is 1.0. Minor version increases imply language changes that merely add to the existing content model. Major version changes imply a change of semantics, probably due to deprecation, such that a file of an earlier major version may lose data if loaded into an application only concerned with supporting the new major version.

producer [Optional, `string`] Identifies the particular producer of this lift file.

Content

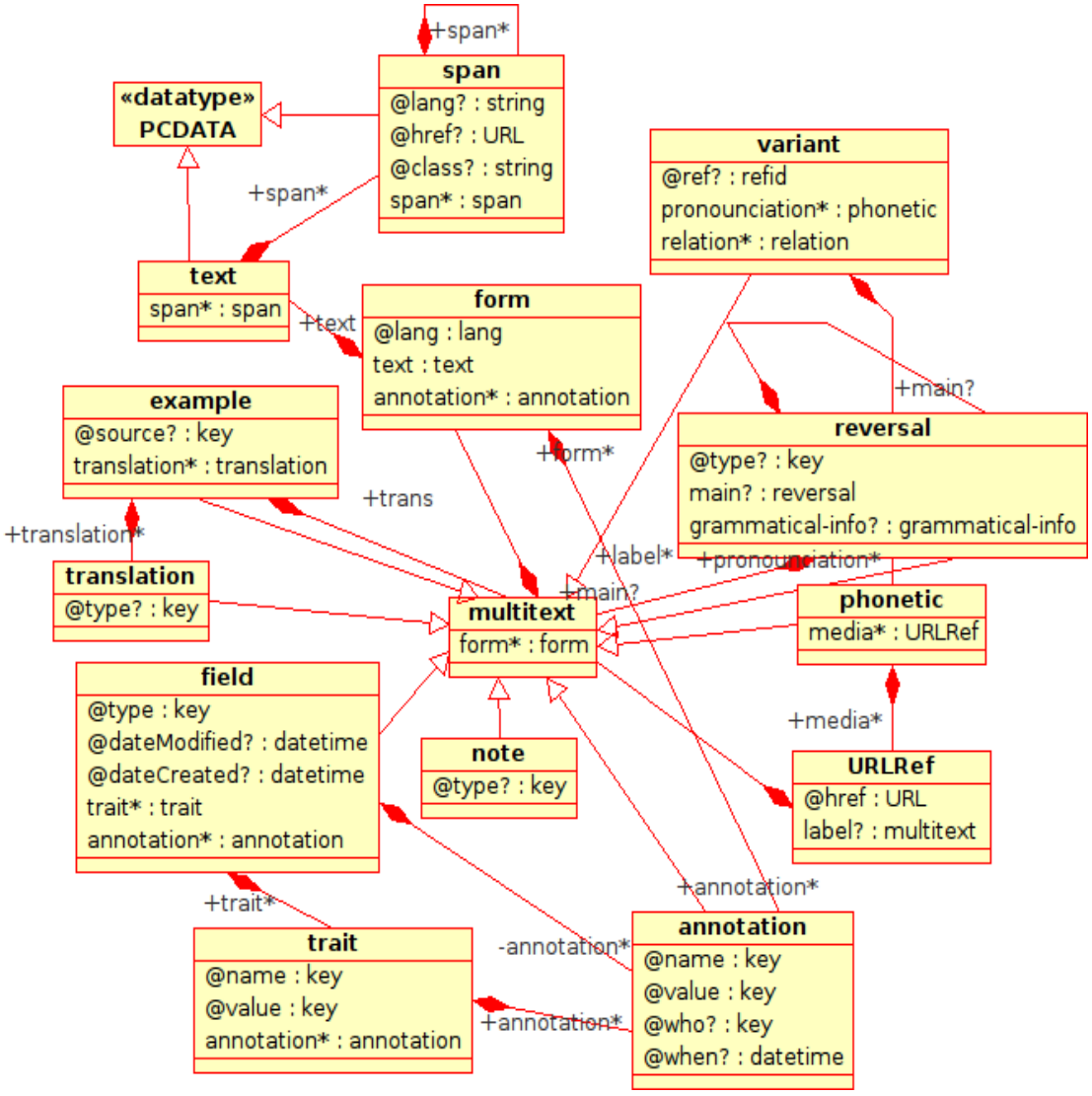
header [Optional, `header`] Contains the header information for the database

entry [Optional, Multiple, `Entry`] Each of the entries in the lexicon. No order is implied.

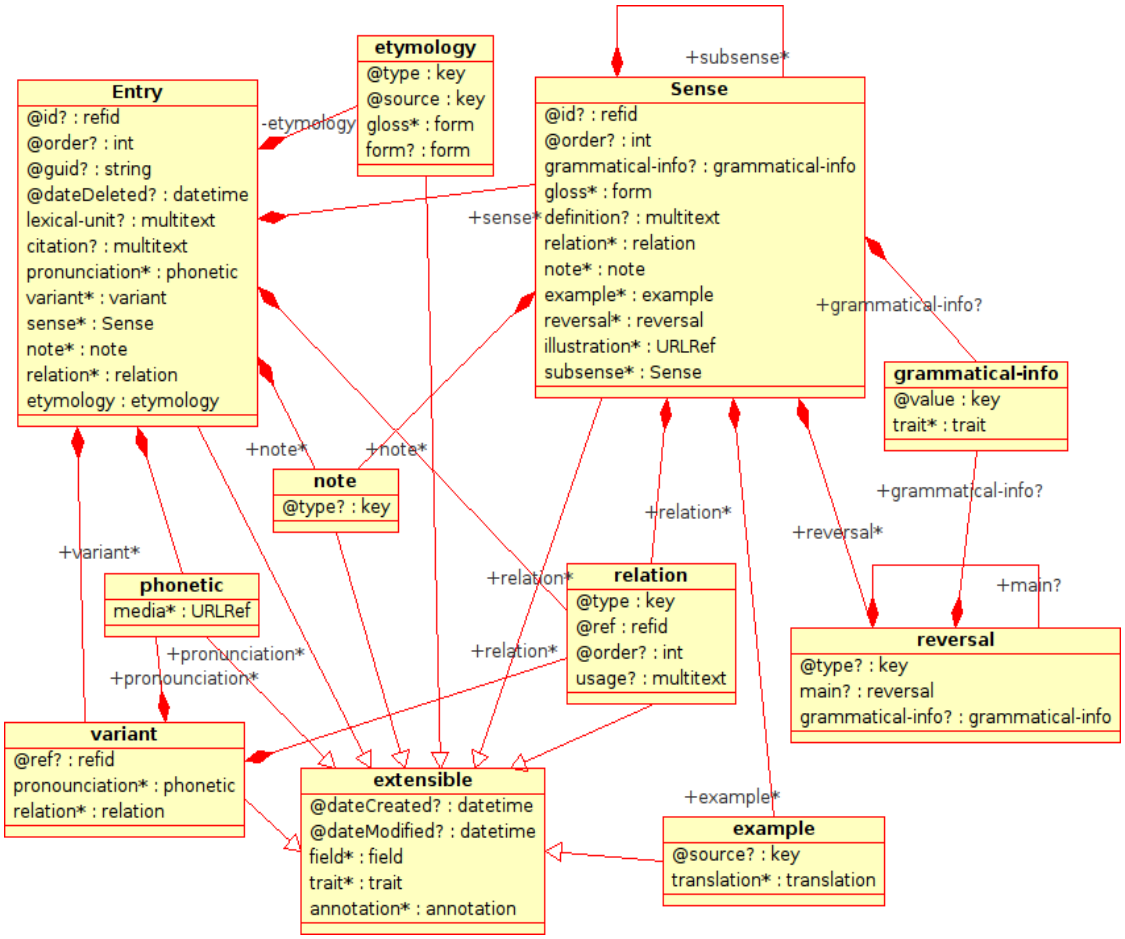
UML Diagrams

The following diagrams show the inter-relationships between the elements in the standard.

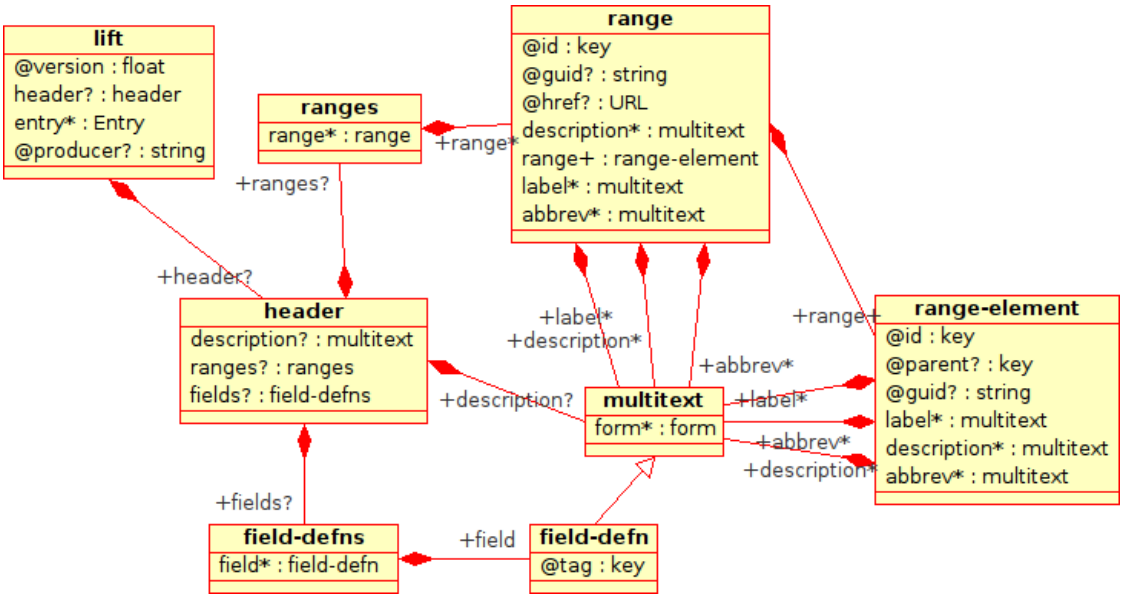
Base Elements



Entry Elements



Header Elements



Lift Ranges

Elements

range-element

A `range-element` is the description of a particular range element found in a particular `range`.

Attributes

`id` [Required, `key`] This is the identifying key for this particular element. `range-element` keys only need to be unique within the one parent `range`.

`parent` [Optional, `key`] Refers to another `range-element` that constitutes a parent in a range hierarchy. This is used for example for semantic domain hierarchies.

`guid` [Optional, `string`] Allows a particular `range-element` to be uniquely identified, particularly across versions of a file.

Content

`description` [Optional, Multiple, `multitext`] Holds the description of the element.

`label` [Optional, Multiple, `multitext`] Holds a caption for the element, typically used in user interfaces when a choice of range values is presented.

`abbrev` [Optional, Multiple, `multitext`] Gives an optional abbreviation for this `range-element` in other languages for GUI purposes.

range

A `range` is a set of `range-elements` and is used to identify both the group of `range-elements` but also to some extent their type.

Attributes

`id` [Required, `key`] This is the identifying key for this particular `range-set` and is used, for example in the `range` attribute of a `flag`. A `range` `id` attribute is unique only among the set of `range`s used in the document.

`guid` [Optional, `string`] Allows a particular `range` to be uniquely identified, particularly when referenced from a lexicon.

href [Optional, `URL`] This attribute may not be used within an external range definition file. In a standard LIFT file, the `href` attribute may be used to reference an external `lift-ranges` file that contains a definition for this range. Any children to this `range` element in the LIFT file override the values (by addition or replacement) in the external range definition.

Content

`description` [Optional, `multitext`] Used to give a multilingual description of the `range`.

`range` [Required, Multiple, `range-element`] This is the list of `range-element`s that make up this `range`. This list is unordered.

`label` [Optional, Multiple, `multitext`] Gives a multilingual label to this `range-set` for GUI purposes.

`abbrev` [Optional, Multiple, `multitext`] Gives an abbreviation for this `range-set` in multiple languages, for GUI purposes.

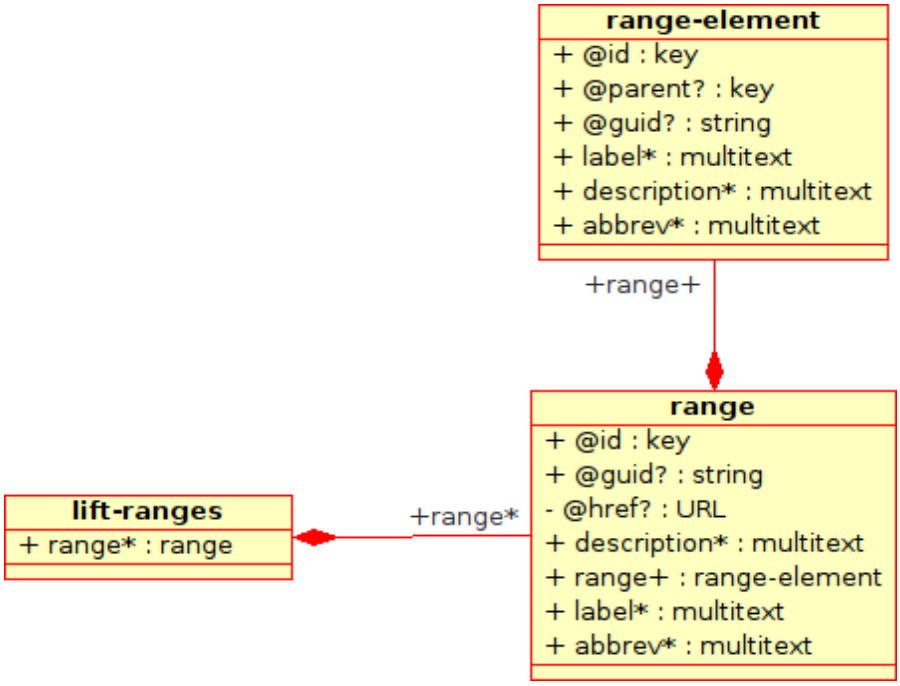
lift-ranges

The root element of the Lift Ranges file type.

Content

range [Required, Multiple, `range`] A `range` definition.

UML Diagram



Ranges

Ranges are a powerful way to add normalising information to a lexicon. Rather than repeating information at every occurrence of its use it can be shared by storing the information against a key in a particular dictionary. If one considers the key to be a `range-element` and the dictionary to be a `range` the mechanism used in LIFT is precisely that.

By making the actual range sets optional in a LIFT file we obviously allow the existence of far from complete data sets. Therefore we introduce the concept of a normalised and non-normalised LIFT file. In effect a non-normalised form of a data set may be a transitional step between a legacy data set and an archive quality file. Providing such a form allows for much simpler implementation and the ability to create more generalised tools to improve the quality of data.

In this section we look at the set of range sets that are defined as part of LIFT. Notice that all range sets can be extended for a particular lexicon or group of lexicons.

dialect

While LIFT makes no required reference to a `dialect` range set, it is an important concept particularly in `variant`s. Because LIFT has no idea which dialects a lexicon may want to refer to, the range set is left empty. Each lexicon should add the dialects it refers to, to it.

etymology

This range set lists all the etymological relations needed.

Elements

`borrowed` The word is borrowed from another language

`proto` The proto form of the word in another language

grammatical-info

This range-set contains a standard set of grammatical information identifiers.

lexical

This set lists various lexical relations. Users may add others that they need.

Elements

`ref` General cross reference.

`main` Reference to a main entry from a minor entry.

`isa` The gen-spec relation where the special relates to the general.

`kindof` The kind-of relation in which the `Sense` is a kind of another sense.

`actor` The actor of this verb

`undergoer` The undergoer of this verb

`component` This word is grammatically built from these components.

note-type

This lists all the different note types that are in the various standards

Elements

anthropology Gives anthropological information.

biblography Biblographic information.

comment This note is an arbitrary comment not for publication

discourse Gives discourse information about a sense.

encyclopedic This note gives encyclopedic information.

general General notes that do not fall in another clear category

grammar Gives grammatical information about a word.

phonology Gives phonological information about a word.

questions Contains questions yet to be answered

restrictions Gives information on the restriction of usage of a word.

scientific_name Gives the scientific name of a sense

sociolinguistics Gives sociolinguistic information about a sense.

source Contains information on sources

usage Gives information on usage

paradigm

LIFT makes no explicit reference to the `paradigm` range-set, but it is an important enough concept as to be centrally named.

Elements

1d 1st dual

1e 1st exclusive

1i 1st inclusive

1p 1st person plural

1s 1st person singular

2d 2nd dual

2p 2nd plural

2s 2nd singular

3d 3rd dual

3p 3rd plural

3s 3rd singular

non-dual non-human or inanimate dual

non-plural non-human or inanimate plural

non-sing non-human or inanimate singulare

plural plural form

redup reduplication form

sing singular

reversal-type

While LIFT reserves the name of this range set, it is empty and should be added to either at the area/entity level or also for a particular project.

semantic_domain

This is the primary semantic domain range set. Other range sets may be used for specific semantic domain classifications.

status

Gives the status of a particular element, for example whether a certain check has been applied.

users

Gives a list of users as used by annotations.

Examples

This section contains various examples, most of which use the MDF SFM schema, and how they would be stored in LIFT.

Simple Records

\lx srapa[1] \ps vt \ge slap \de slap with open hand \dt 27/Aug/91 **srapa** *vt.* slap with open hand	<?xml version="1.0"?> <lift lang="und-Latn"> <!--unknown language, latin script--> <entry id=”srapa” dateModified=”1991-08-27”> <lexical-unit> <form lang=”und-Latn”><text>srapa</text></form> </lexical-unit> <sense id="srapa_"> <!--id can't be the same as entry id--> <grammatical-info type="vt"/> <gloss lang="en"><text>slap</text></gloss> <definition> <!--or here--> <form lang="en"><text>slap with open hand</text></form> </definition> </sense> </entry> </lift>

[1] Making Dictionaries, p29

```
\lx abat[1]
\ps n
\ge grove
\gn dusun
\rf d2.077.03
\xv Kbwai abatke ti ksweruk
nurare.
\xe I went to the coconut
groves to clear the grass.
\xn Saya pergi menyiangi dusun
kelapa.
\rf d4.079.16
\xv Kbwa ti ktwan nurke o
abatke.
\xe I'm going to plant coconut
trees in the grove.
\xn Saya pergi tanam kelapa di
dusun.
\ee This is uc:not limited to
coconut groves but is used for
mangoes, etc.
\sg abatke
\dt 26/Feb/90
```

abat *n.* grove; *dusun.* Kbwai abatke ti ksweruk nurare. I went to the coconut groves to clear the grass. *Saya pergi menyiangi dusun kelapa.* Kbwa ti ktwan nurke o abatke. I'm going to plant coconut trees in the grove. *Saya pergi tanam kelapa di dusun.* This is not limited to coconut groves but is used for mangoes, etc. *Sg:*abatke.

```
<entry id="abat" dateModified="1990-02-26">
  <lexical-unit>
    <form lang="und-Latn"><text>abat</text></form>
  </lexical-unit>
  <variant>                              <!--This is a paradigm-->
    <trait name="paradigm" value="sing"/>
    <form lang="und-Latn"><text>abatke</text></form>
  </variant>
  <sense>
    <grammatical-info value="n"/>
    <gloss lang="en"><text>grove</text></glass>
    <gloss lang="id"><text>dusun</text></gloss>
    <example source="d2.077.03">
      <form lang="und-Latn"><text>Kbwai abatke to ksweruk
nurare.</text></form>
      <translation><form lang="en"><text>
        I went to the coconut groves to clear the grass.
      </text></form>
      <form lang="id"><text>
        Saya pergi menyiangi dusun kelapa.
      </text></form></translation>
    </example>
    <example source="d4.079.16">
      <form lang="und-Latn"><text>Kbwa ti ktwan nurke o
abatke.</text></form>
      <translation><form lang="en"><text>
        I'm going to plant coconut trees in the grove.
      </text></form>
      <form lang="id"><text>
        Saya pergi tanam kelapa di dusun.
      </text></form></translation>
    </example>
    <note type="encyclopedic">
      <form lang="en"><text>
        This is <span class="underline">not</span> limited
to coconut groves but is used for mangoes, etc.
      </text></form>
    </note>
  </sense>
</entry>
```

This example shows a fairly full entry including two examples and an encyclopedic note. Notice the use of `span` for handling the underlining. Notice also the use of `variant` to represent a paradigm form, which makes it merely a variant constrained according to a paradigm form.

Subentries

Subentries are really a document artefact. They are used to present various entries in direct relation to another entry. The actual lexical relation being represented may be anything from a component-whole or paradigm to a shared semantic domain. Different presentations of a lexicon may present these relations in different ways even to the extent of inverting the subentry-subhead relation and giving the parent entry as a subentry of its subentry, in say an online dictionary.

Therefore, rather than storing an explicit subentry element we use lexical relations to model the precise relationship and then leave the typesetter to resolve the precise presentation of those relationships.

But since subentries have been around for so long and are an important, if informal, relation, we present here how various subentry relationships can be modelled.

[1] Making Dictionaries, p59

There are at least three ways of storing subentry relationships between a main entry and its subentry:

- Store the subentry in the entry as a sub-element. LIFT does not do this. All entries are full `entry` elements.
- Store a marker in the subentry referring back to the main entry under which this subentry occurs. This can be done using a `subhead` relation.
- Store a marker in the main entry to the subentry at the point you want it output. This can be done using a `subentry` relation.

```
\lx brush[1]
\ps n
\ge bristly_instrument
\de bristly instrument used for
cleaning, arranging or applying
a liquid to something
\se hairbrush
\ps n
\de kind of brush typically with
stiff one inch long bristles
loosely spaced arranged
perpendicularly to the handle
for rearranging hair
\se paintbrush
\ps n
\de kind of brush of varying
sizes and varying lengths and
textures of bristles arranged as
an extension of the handle used
to apply paint and similar
materials
```

brush *n.* bristly instrument used for cleaning, arranging, or applying a liquid to something

hairbrush *n.* kind of brush typically with stiff one inch long bristles loosely spaced arranged perpendicularly to the handle for rearranging hair.

paintbrush *n.* kind of brush of varying sizes and varying lngths and textures of bristles arranged as an extension of the handle used to apply paint and similar materials.

```
<entry id="brush">
  <lexical-unit>
    <form lang=”en”><text>brush</text></form>
  </lexical-unit>
  <sense id="brush_">
    <grammatical-info value="n"/>
    <gloss lang="en"><text>bristly
instrument</text></gloss>
    <definition><form lang="en"><text>
      bristly instrument used for cleaning, arranging or
applying a liquid to something
    </text></form></definition>
    <relation type=”subentry” ref="hairbrush"/>
    <relation type=”subentry” ref="paintbrush"/>
  </sense>
</entry>

<entry id=hairbrush">
  <lexical-unit>
    <form lang=”en”><text>hairbrush</text></form>
  </lexical-unit>
  <sense id="hairbrush_">
    <grammatical-info value="n"/>
    <definition><form lang="en"><text>
      kind of brush typically with stiff one inch long
bristles loosely spaced arranged perpendicularly to the
handle for rearranging hair
    </text></form></definition>
  </sense>
</entry>

<entry id="paintbrush">
  <lexixal-unit>
    <form lang=”en”><text>paintbrush</text></form>
  </lexical-unit>
  <sense id="paintbrush_">
    <grammatical-info value="n"/>
    <definition><form lang="en"><text>
      kind of brush of varying sizes and varying lengths
and textures of bristles arranged as an extension of the
handle used to apply paint and similar materials
    </text></form></definition>
  </sense>
</entry>
```

Notice here how we have made the subentries refer back to the sense rather than the entry. The advantage of doing this in the lexical database is that at least then one has the option of how to typeset them.

[1] Making Dictionaries, p80

Reverse Index

It is not possible for LIFT to know all the different types of reverse index that may be required in the dictionaries around the world, so we need to add a list of reversal indexes to the file.

\lx utan[1] \ps n \sd Nplant \ge veg \gn sayur ; jamu \re vegetable ; mushroom \de non-bulbous edible leafy and stalky plant and fungi **utan** *n.* non-bulbous edible leafy and stalky plant and fungi.	<entry id="utan"> <lexical-unit> <form lang=”und-Latn”><text>utan</text></form> </lexical-unit> <sense> <grammatical-info value="n"/> <trait name="semantic-domain" value="Nplant"/> <gloss lang="en"><text>veg</text></gloss> <gloss lang="id"><text>sayur</text></gloss> <gloss lang="id"><text>jamu</text></gloss> <reversal> <form lang="en"><text>vegetable</text></form> </reversal> <reversal> <form lang="en"><text>mushroom</text></form> </reversal> <definition><form lang="en"><text> non-bulbous edible leafy and stalky plant and fungi </text></form></definition> </sense> </entry>

Notice that the two reversals in this case could have formed a hierarchy:

```
<reversal>
  <form lang=”en”>mushroom</form>
  <main><form lang="en">vegetable</form></main>
</reversal>
```

Notice also that it is difficult often to align glosses from different languages.

Lexical Relations

Lexical relations are relatively straightforward to encode.

\lx hete[2] \ps vt \ge cut \de cut into sections for use \lf Gen = lata \le cut \pd -k **hete** *vt.* cut into sections for use. *Gen*: lata ‘cut’. *Prdm*: -k	<entry id="hete"> <lexical-unit> <form lang=”und-Latn”><text>hete</text></form> </lexical-unit> <sense id="hete_"> <grammatical-info value=”vt”/> <gloss lang="en"><text>cut</text></gloss> <definition><form lang="en"><text> cut into sectios for use </text></form></definition> <relation type="gen" ref="lata_"/> </sense> <variant> <form lang=”und-Latn”><text>hetek</text></form> </variant> <!--no idea what kind of paradigm, so really a free variant--> </entry>

But when you add dialect into the mix, things can get more complicated.

[1] Making Dictionaries, p89

[2] Making Dictionaries, p116

\lx apu[1] \ps n \ge lime \re lime ; chalk \de lime slaked from burning seashells and used as an ingredient in chewing betelnut \lf synD = ahul \le Lisela, Rana dialects \et *apuR \eg lime, chalk **ahul** *n.* lime slaked from burning seashells and used as an ingredient in chewing betelnut. *SynD*: ahul 'Lisela, Rana dialects'. *Etym*: *apuR 'lime, chalk'.	<entry id="apu"> <lexical-unit> <form lang="und-Latn"><text>apu</text></form> </lexical-unit> <sense id="apu_"> <grammatical-info value="n"/> <gloss lang="en"><text>lime</text></gloss> <reversal> <form lang="en"><text>lime</text></form> </reversal> <definition><form lang="en"><text> lime slaked from burning seashells and used as an ingredient in chewing betelnut </text></form></definition> <variant> <trait name="dialect" value="lisela"/> <trait name="dialect" value="rana"/> <form lang="und-Latn"><text>ahul</text></form> </variant> </sense> <etymology type="proto"> <form lang="x-proto-ind"><text>apuR</text></form> <gloss><form lang="eng"><text>lime, chalk</text></form></gloss> </etymology> </entry>

Notice how the lexical function in the MDF data has been transformed into a variant in LIFT. An alternative which is more probable from an automatic conversion might be:

```
<relation key="syn" sense="ahul_">
  <type value="dialects" value="lisela"/>
  <type value="dialects" value="rana"/>
</relation>
```

The problem with this is that `ahul` would need to be in the lexicon with its own entry and sense. But if it is a dialectal variant, it probably has no entry of its own.

Hierarchies

Since sense can both form a hierarchy and also not be labelled, it is possible to model the various sense hierarchies that exist.

[1] Making Dictionaries, p119

\lx opon[1]
\ps n
\sn 1a
\ge grand_kin
\de grandparent, grandchild; reciprocal term of plus or minus two generations
\sn 1b
\ge ancestor
\de ancestor, descendent
\sn 2
\ge master
\de master, lord, owner; the one with the say over someone or something
	<entry id="opon">
	  <lexical-unit>
	  <form lang="und-Latn"><text>opon</text></form>
	  </lexical-unit>
	  <sense id="opon_1a" order="1">
	  <grammatical-info value="n"/>
	  <gloss lang="en"><text>grand kin</text></gloss>
	  <definition><form lang="en"><text>
	  grandparent, grandchild; reciprocal term of plus or minus two generations
	  </text></form></definition>
	  <sense id="opon_1b">
	  <grammatical-info value="n"/>
	  <gloss lang="en"><text>ancestor</text></gloss>
	  <definition><form lang="en">
	  <text>ancestor, descendent</text>
	  </form></definition>
	  </sense>
	  </sense>
	  <sense id="opon_2" order="2">
	  <grammatical-info value="n"/>
	  <gloss lang="en"><text>master</text></gloss>
	  <definition><form lang="en"><text>
	  master, lord, owner; the one with the say over someone or something
	  </text></form></definition>
	  </sense>
	</entry>

Notice how subsenses are treated as full senses when referenced. Also notice how each sense has its own `grammatical-info`.

Multiple Scripts

Why do we have all these seemingly redundant `<form>` tags around the place? They are needed for the situation where one word may be written in multiple scripts. This isn't a case of different languages or even just extra glosses. We need to acknowledge that the text is identical but is being stored in two or more writing systems. So, for example if a gloss were written with two writing systems, there would not be two glosses but just the one gloss written two different ways. This is different from glosses in two languages where they are effectively two different glosses in two different languages.

Note that the Toolbox markup is not pure MDF but it is MDF motivated so you can probably follow along.

[1] Making Dictionaries, p47

\lx jɔ̃dɛ̃ \lxt ยองเด่ง \dia Ratburi \a jə̀ŋdɛ̃ʲ \ps N \gt เอว \ge waist \so lang1.42.6 \sd body \dat 21/Feb/2003	<entry id="jɔ̃dɛ̃"> <lexical-unit> <form lang="und-fonipa"><text>jɔ̃de</text></form> <form lang="und-Thai"><text>ยองเด่ง</text></form> </lexical-unit> <variant> <trait name="dialects" value="Ratburi"/> <form lang="und-fonipa"><text>jə̀ŋdɛ̃ʲ</text></form> </variant> <sense id="jɔ̃dɛ̃_" dateCreated="2003-02-21"> <grammatical-info value="N"/> <gloss lang="th"><text>เอว</text></gloss> <gloss lang="en"><text>waist</text></gloss> <note type="source"><form lang="en"><text> lang1.42.6 </text></form></note> <trait name="semantic-domain" value="body"/> </sense> </entry>

Implementation

This section examines various issues regarding the implementation of applications that may use LIFT.

Lift Conformance

Conformance to LIFT calls for a relatively high level of structural and semantic integrity from a lexical database. It is unlikely that a source database will be structured to allow for a single pass generation of LIFT. We discuss how feasible generating LIFT in a single pass is. Then we look at approaches for a two stage process. Following that we examine some parsing issues with LIFT when converting back to say an MDF based database.

Single Pass Generation

Given all the `id`s and `refid`s, is it possible to generate a LIFT file in a single pass from some kind of database processing each record in sequence (rather than making random access into the database)? This question raises a number of issues we will discuss here.

Refid generation

The generalised approach to `refid` generation is to hold the `id` for each entry and sense in the entry or sense and then to look it up when one needs to refer to it. For a single pass system, this can work if `refid` is created when first referred to or the entry or sense is output. But keeping track of `refid`s can be problematic in some systems.

One powerful way of working with `refid`s is via `refid` munging. This is where there is an algorithmic relationship between the primary lexical form and the homograph number of an entry and its `refid`. For example `"test:1"` or if there is no homograph number then remove the `":"` as well, resulting in: `"test"`. Moving on to a sense based `refid`, we can simply add the `sense` label after a `"_"`, as in: `"test:1_2"`. This way when converting between data sets that do not have specific references but do store the information necessary to build such references there is no need to store a map during conversion or to deal with forward references by multiple passes over the data. Notice also that it is only during file generation that such munging is needed, a reader just uses the `refid`s it is given. Therefore a particular application may use any system of `refid` generation it wants. For example it could just be a record number or GUID.

Generating LIFT

Given that the header information and list of range-elements, etc. has to go at the beginning of a file, is it possible to generate a fully specified LIFT file in a single pass?

It would certainly be possible to generate such a file if the header information were stored at the end of the file rather than the beginning, but it needs to be at the beginning to aid applications reading LIFT. One approach, though, is to store the header information in another file that is referenced via an `include` element with a fixed name, that can be generated easily during the main output. The header information may then be output at the end into the referenced file and everyone is happy!

Subentries

As stated in the example there are different ways of modelling subentry relationships. The one used in LIFT allows for the greatest flexibility whilst also keeping subentries as full entries. In cases where the source data has subentries stored with the main entry, generation of the list of subentry keys is not difficult in a single pass. In the case where subentries reference their main entry and the main entry has no knowledge of the subentry, it is not possible to generate a full model in a single pass, instead a program will need to generate the necessary lists and add them to the senses of the main entries.

Multiple Passes

While it is possible in a single pass to generate full LIFT, it is probable that there will be something that is not achievable in a single pass. Instead one approach is for the primary export to generate something as close to LIFT as it can and for it to pass other information using fields. Then a second process can take this intermediate format and generate full LIFT from it. The two processes will have to be designed to work together. But it should be possible to design the backend process fairly generically and make it useful for various export routines and intermediate models.

For example, an export process from Toolbox might generate nearly complete LIFT but with the following functions passed to a second process.

- Creation of subentry reference lists
- Split morphological segments into separate lexical relational elements

Roundtrip Requirements

It is impossible to have a file format that can at the same time store anything that an application may potentially want to store using that file format, and that can be completely stored by most applications, interpreted to something meaningful and regenerated in a helpful way. This is why the specification of LIFT has started out with a limited number of extensible types. The aim is that all applications will be able to store unexpected information that use these types (`field`, `flag`, `date-class`) even if they make no effort to interpret the information. It is also designed that the semantics of these types are not dependent on other content in the parent changing. I'm sure someone can come up with such semantics, but they should be aware that if they do that then any other program that roundtrips their data may well break their semantics.

This specifically precludes complex linkages between elements beyond those specified within LIFT itself. If you need some more linkages you will need to negotiate for an improved LIFT spec. This is an area of LIFT that could do with some more work, unsurprisingly.

Merging XML

Merging XML files is a notoriously difficult thing to do. In addition, since there is nowhere in LIFT where element order is significant, except perhaps that the `header` occurs first and within `span`s, this can be both a blessing and a curse. LIFT elements are designed to be keyed off their attributes with only a few elements having problems in this area: `phonetic` and `variant`. Merging involves synchronizing key elements across versions of the file you are merging. So it is important that `id` attributes keep the same values across versions of the data files.

Change History

Initial Development

0.2	MJPH	18/Jul/2006	Added versioning and change history
0.2.1	MJPH	19/Jul/2006	Tidy up refid description, add partial conformance requirements.
0.3	MJPH	24/Jul/2006	Add `style` and friends. Move `include` from `ranges` to `header`. Tidied up issues. Time is optional.
0.3.1	MJPH	25/Jul/2006	Move borrowed to its own range-set.
0.4	MJPH	27/Jul/2006	renamed `time` to `datetime`, `unicode` to `text`. Removed pictures from `span`s.
0.5	MJPH	2/Aug/2006	Remove `paradigm`, add `extensible` and refactor. Add implementation section. Remove `LIFT.meta`. Create `sdomains`. Remove `pattern` and `tone` from `phonetic`. `field` is beefier now.
0.6	MJPH	14/Sep/2006	Remove `allomorph`, `style` stuff and `sdomains`. `multitext` is now single language. Make `form` optional.
0.7	MJPH	18/Dec/2006	Remove `xml:lang`, `script`, `gloss`, `date` add `annotation`, `@dateCreated`, `@dateModified` change semantics of `@lang`.
0.7.1	MJPH	19/Dec/2006	`datetime` changed ZZZZ to zzzz.
0.8	MJPH	9/Mar/2007	Remove header into an optionally referenced section file. Remove `@script` and just use `@lang` everywhere.
0.9	MJPH	14/Mar/2007	Lots of minor changes to element names. `form` no longer optional. Tighten up ambiguities and looseness, particularly around `span`. `trait`s contain `annotation`s. Hopefully ready for public review now and for testing against real, hard data. Removed `subentry`.
0.9.1	MJPH	21/Mar/2007	`gloss` is now a `form`. `multitext` takes `trait`. Other minor tidy ups. Add `entry/@guid` and `lift/@producer`.
0.10	MJPH	27/Mar/2007	Rename `text` to `PCDATA` and add the `text` element allowing `form`s to take `trait`s. Tidy up some inconsistencies between the diagrams and the text. Use `und` for undefined language not `zxx` in language tags.
0.10.1	MJPH	26/Oct/2007	Change `sense/@picture` to be `sense/@illustration`.
0.11	MJPH	29/Oct/2007	`@status` becomes `@annotation`. Add `field/@annotation`.
0.11.1	MJPH	18/Dec/2007	Make `refid`s invariant
0.11.2	MJPH	20/Dec/2007	Change grammi type to grammatical-info. No change to actual grammar.
0.11.3	MJPH	11/Jan/2008	Fix UML for `etymology` and `example`s; make outer level `spans` redundant; fix `relation`s in examples. Fix `trait/status` to become `trait/annotation`. Also `field/annotation` is no longer an attribute.

0.12	MJPH	15/Jan/2008	Allow full `range` definitions within a LIFT file. Move `etymology` to `entry` from `sense`. Add `reversal/grammatical-info`.