Lexicon Interchange Format
A Description

Martin Hosken and Stephen McConnel,
SIL Non-Roman Script Initiative (NRSI) and Language Software Development

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 FieldWorks Language Explorer (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 names 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

Core Model Elements

The core model elements are those involved with the primary structure of a lexicon. The description takes a top down approach starting from the root and examines an entry and all that goes to make that up. An overview diagram of these types may be found in the Entry UML diagram.

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

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. 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.

Elements

`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.

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 `dateCreated` and `dateModified` attributes and `field`, `trait`, and `annotation` 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`. For computer generated and processed LIFT files, a "universally unique identifier" (RFC 4122) is highly recommended. Any value that is unique within the LIFT file is acceptable, however. See the Examples section for examples of both approaches.

`guid` [Optional, deprecated, `string`] This gives a unique identifier to this entry in the form of a "universally unique identifier" (RFC 4122). This identifier is unique across all `entry`s and all `sense`s, and across all projects. Since this duplicates the recommended practice for `id`, its use is deprecated.

`order` [Optional, `int`] This is the homograph number. If there are homographs and the `order` attribute is missing. document order will be used. If the `order` attribute is present, but there there are no homographs, it is ignored.

`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.

Elements

`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 referent in the lexicon. The other word is outside the language.

sense

An `entry` is made up of a number of `sense`s.

Inheritance

`Extensible` Adds `dateCreated` and `dateModified` attributes and `field`, `trait`, and `annotation` 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. For computer generated and processed LIFT files, a "universally unique identifier" (RFC 4122) is recommended. Any value that is unique within the LIFT file is acceptable, however.

`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. If the `order` attribute is present, but there there is only one sense, it is ignored.

Elements

`grammatical-info` [Optional, `grammatical-info`] Grammatical information. A sense may only correspond to a single part of speech. Where multiple parts of speech are considered as a single sense, then subsenses should be used.

`gloss` [Optional, Multiple, `Gloss`] Each `gloss` is a single string in a single language and writing system.

`definition` [Optional, `Multitext`] Gives the definition in multiple languages or 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] Senses can form a hierarchy.

variant

variants 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

Multitext Gives the variation to the main lexical form, possibly in multiple writing systems.

Extensible Adds dateCreated and dateModified attributes and field, trait, and annotation elements to the content for extensibility.

Attributes

ref [Optional, refid] Gives the variation as a reference to another entry or sense rather than specifying the form (that is, the Multitext value of the variant).

Elements

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 orthographic 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.

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 (or entries) to which it refers. Instead it is a bidirectional relationship between two sets of senses (or entries). 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 (or entry) 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 dateCreated and dateModified attributes and field, trait, and annotation 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 type is given in terms of the referenced sense/entry's relation to this sense/entry. For example:

```
<entry id="apple"><sense>
   <relation type="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 entrys or senses. If no order attribute is present, then document order is used.

usage [Optional, `Multitext`] Gives information on usage in one or more languages or writing systems.

example

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

Inheritance

Multitext Stores the content of the example in the main language of the dictionary, possibly in multiple writing systems.

Extensible Adds `dateCreated` and `dateModified` attributes and `field`, `trait`, and `annotation` elements to the content for extensibility.

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.

Elements

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

note [Optional, Multiple, `note`] Holds notes on this example.

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 different `form`s in the same `translation`.

Inheritance

Multitext Stores the content of the translation, possibly in multiple languages or writing systems.

Attributes

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

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, possibly in multiple writing systems.

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 `reversal`s 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`.

Elements

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 path to the root of the 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.

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.

Elements

trait [Optional, Multiple, trait] Allows the grammatical information for a given sense to have more information than just the part of speech given by the value attribute.

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 dateCreated and dateModified attributes and field, trait, and annotation 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 (RFC 5646) 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 [Optional, Form] Holds the form of the etymological reference.

Phonetic

This represents a single pronunciation in phonetic form.

Inheritance

Multitext Allows for storage of different representation forms of the phonetic text.

Extensible Adds dateCreated and dateModified attributes and field, trait, and annotation elements to the content for extensibility.

Elements

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

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, possibly in multiple languages or writing systems.

`Extensible` Adds `dateCreated` and `dateModified` attributes and `field`, `trait`, and `annotation` 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`.

Header Elements

In addition to entries, a LIFT database contains header information designed to make the database somewhat self documenting. Elements designed for extensibility, like fields and trait ranges, are described so that even if an original application that generated the data is lost, the meaning of extensible data can be recovered.

header

This holds the header information for a LIFT file including range information and added field definitions.

Elements

The content of a `header` is the definitions and extensions of the various `ranges` and `fields` 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` information.

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

ranges

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

Elements

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

fields

This is a simple list of `field-definition` elements.

Elements

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

field-definition

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. The goal is that data can fully transferred between copies of the same program that reads or writes LIFT files. A different program may or may not be able to make full use of data provided by `field` (or `trait`) elements. (Despite its name, a `field-definition` may apply to a `trait` as well as to a `field`.)

Attributes

`name` [Required, `key`] This key corresponds to the `name` attribute found in all `field`s (or `trait`s) for which this is the definition.

`class` [Optional, `string`] This attribute provides the name of the LIFT element that contains all `field`s (or `trait`s) for which this is the definition. If more than one LIFT element may contain such `field`s or `trait`s, the value of **`class`** may be a space-separated list of element names.

`type` [Optional, `string`] This attribute defines the basic data type of the data. For a `field-definition` which describes a `trait`, the **`type`** attribute tells us about the contents of the `trait`'s `value` attribute. While any non-empty string may be used for the **`type`**, standard values are:

- `"datetime"` (ISO 8601) (zero or one `trait`)
- `"integer"` (zero or one `trait`)
- `"option"` (zero or one `trait`)
- `"option-collection"` (unordered references) (zero or more `trait`s)
- `"option-sequence"` (ordered references) (zero or more `trait`s)

Note, for the "option" choices, the range referenced is specified by an accompanying `option-range` attribute, described next.

For a `field-definition` which describes a `field`, the `type` attribute tells us about the contents of the element. Standard values are:

- `"multistring"` (0 or more parallel strings in different writing systems, each a single paragraph of text or less)
- `"multitext"` (0 or more parallel strings in different writing systems, each possibly containing multiple paragraphs)

`option-range` [Optional, `key`] This attribute is valid only for a `field-definition` that contains one of the option type values. Its value must match against an `id` attribute of a `range` element in the header.

`writing-system` [Optional, `string`] Provides the default list of writing systems for displaying the information in this field. These are given as a space delimited list of language tags (RFC 5646). This list does not limit which languages or writing systems may contain data in the field, just which ones are displayed by default.

Note: at this time, it is not possible to declare a multiplicity other than that implied by the **`type`** attribute. For example, it is not currently possible to declare that a text field may be repeated.

Elements

`label` [Optional, `Multitext`] Gives the UI label of the field, possibly in multiple languages and writing systems.

`description` [Optional, `Multitext`] Contains a multilingual description of this particular field's content.

Base Elements

The core base types are described here and may be found on the base types overview UML diagram.

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.

Attributes

`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.

Elements

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

`trait` [Optional, Multiple, `trait`] Adds type or constraint information.

`annotation` [Optional, Multiple, `annotation`] Adds meta-information describing the element.

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

annotation

The `annotation` element provides a mechanism for adding meta-information to almost any element. An `annotation` is also a `trait`, but includes the option to specify `who` made the annotation and `when` it was made. 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 the textual meta-information of the annotation, possibly in multiple languages or writing systems.

Attributes

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

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

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

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

trait

A trait is an important mechanism for giving type information to an object or adding binary constraints. There are many ways of interpreting 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 are 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 its use in a `field-definition`.

Attributes

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

value [Required, `key`]. This is the identifier of a particular `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 global `key`s may be used. There is no requirement that the `key` keeps its value across different versions of the file.

Elements

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

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. The goal is that enough information can be given in the header to allow full data transfer between copies of the same program that both reads and writes LIFT files.

Inheritance

Multitext Stores the textual information of the field, possibly in multiple languages or writing systems.

Attributes

name [Required, `key`] The identifying key that gives the field name of the field. Applications may share data by agreeing on the `name` to use. This should normally match the `name` of a `field-definition` in the header.

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

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

Elements

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

annotation [Optional, Multiple, `annotation`] Adds meta-information describing the field.

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, or possibly a relative path to a file (relative to the location of the LIFT file itself).

Elements

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

Multitext

This element allows for different representations of the same information in a given language, or in multiple languages. For example it allows for different representations of a lexical form, as in an orthography and also in phonemic form. It also allows for the literal translation of an example into multiple languages.

Elements

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

text [Optional, deprecated] 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. Note that `text` does not allow for `annotation`s, but `form` does. Also, `form` specifies the language or writing system, but `text` does not.

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 `annotation`s and the textual content is held in the `text` child element.

Attributes

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

Elements

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 meta-information for this textual element including status information.

Gloss

A `Gloss` is a representation of a sense's gloss 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.

Inheritance

Form Provides the `lang` attribute, and the `text` and `annotation` elements

Elements

trait [Optional, Multiple, `trait`] contains additional information for this gloss. If it is necessary to semantically link glosses either because they are of the same language (but different writing systems) or because they really are the same semantically across languages, then the glosses are linked by containing a `trait` with the `name="linkage"` and the `value` being the same.

text

This is a mixed content element containing textual data mixed with `span`s only. The language information is inherited from its parent element, or provided explicitly in an embedded `span`.

Elements

#PCDATA The core content of `text` is Unicode text. (This isn't really an element as such, but rather simple textual content.)

span [Optional, Multiple, `span`] The content may have `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.

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` field, plain text approaches should be used either using Unicode paragraph characters or newline 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, `string`] Gives the style name or class of the text.

Elements

`#PCDATA` The core content of a `span` is Unicode text. (This isn't really an element as such, but rather simple textual content.)

`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`.

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.

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 is provided in the sections on range sets.

lang

This is a language tag and follows RFC 5646 or any superseding document.[2] 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 5646, 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.

[1] That's capital Z

[2] RFC 5646 supersedes 4646 which supersedes RFC 3066 which in its turn supersedes RFC 1511.

`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 `refid` 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. In the context where multiple people may be independently adding entries to a lexicon, and then merging, this constraint in effect requires that a `refid` be globally unique. Therefore it is strongly recommended that all `refid`s be 128 bit "universally unique identifiers" (RFC 4122) since those are easily generated on all common computer systems. These are typically written as hexadecimal numbers grouped like xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx. (These are also called "globally unique identifiers", or guids.)

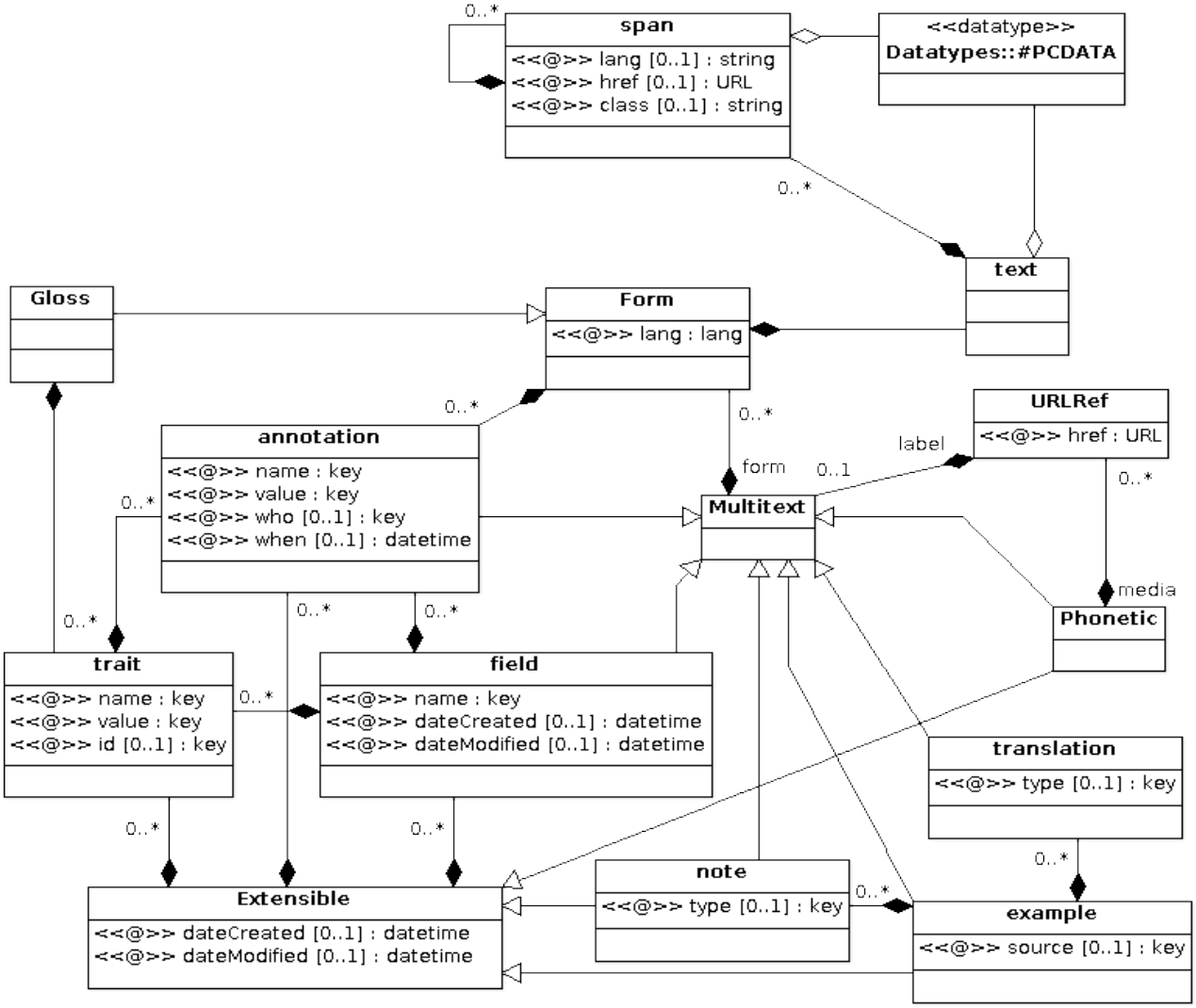
URL

A URL datatype is a string containing a URI (Universal Resource Indicator) reference as specified in RFC 3986. Note that this includes relative references that are relative to the containing lift document.

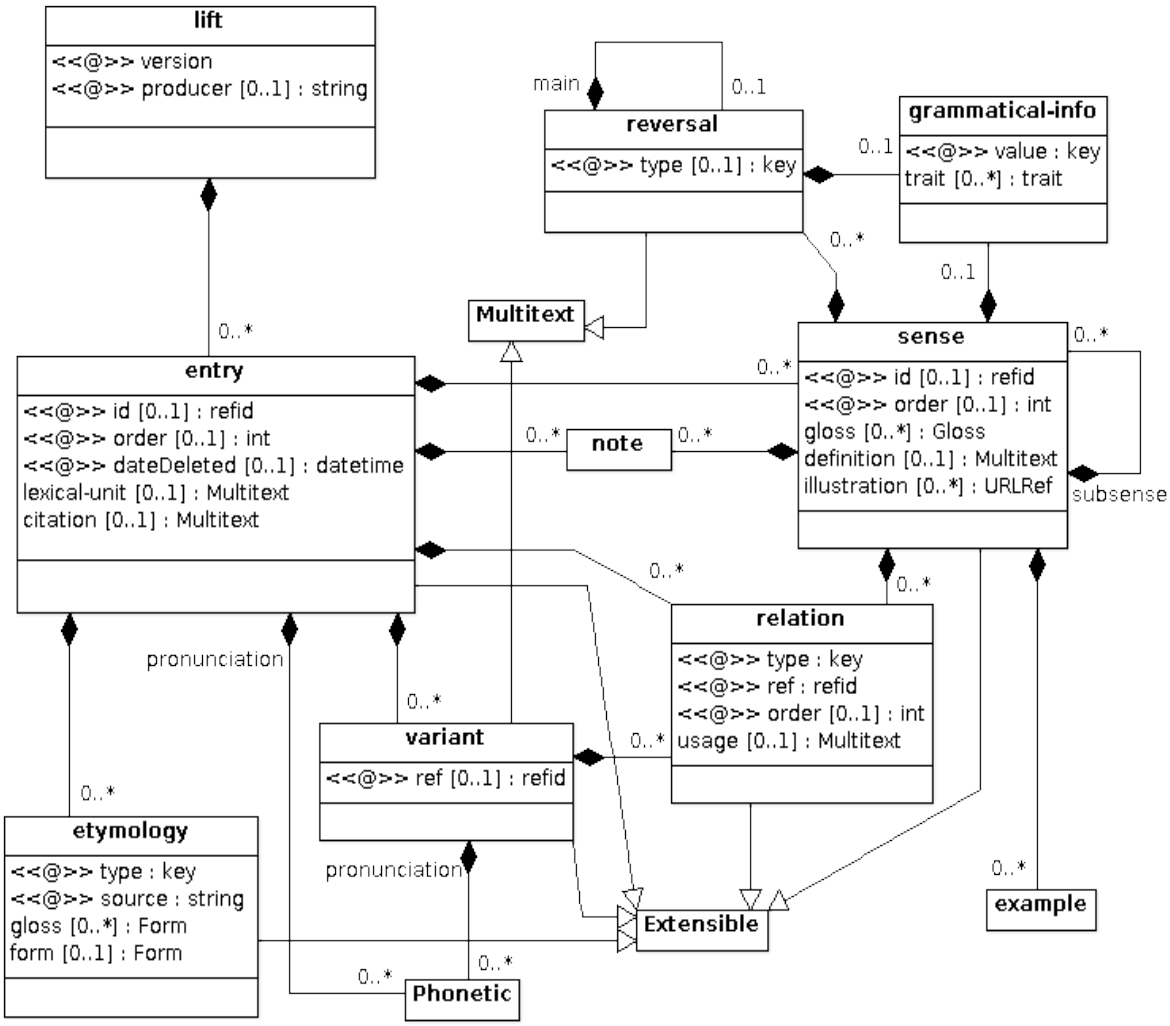
UML Diagrams

The following diagrams show the inter-relationships between the elements in the standard. Associations between classes show which elements are part of an element of a given class or type. If an association is named, then the element is given that name, otherwise it takes the name of the class to which it associates.

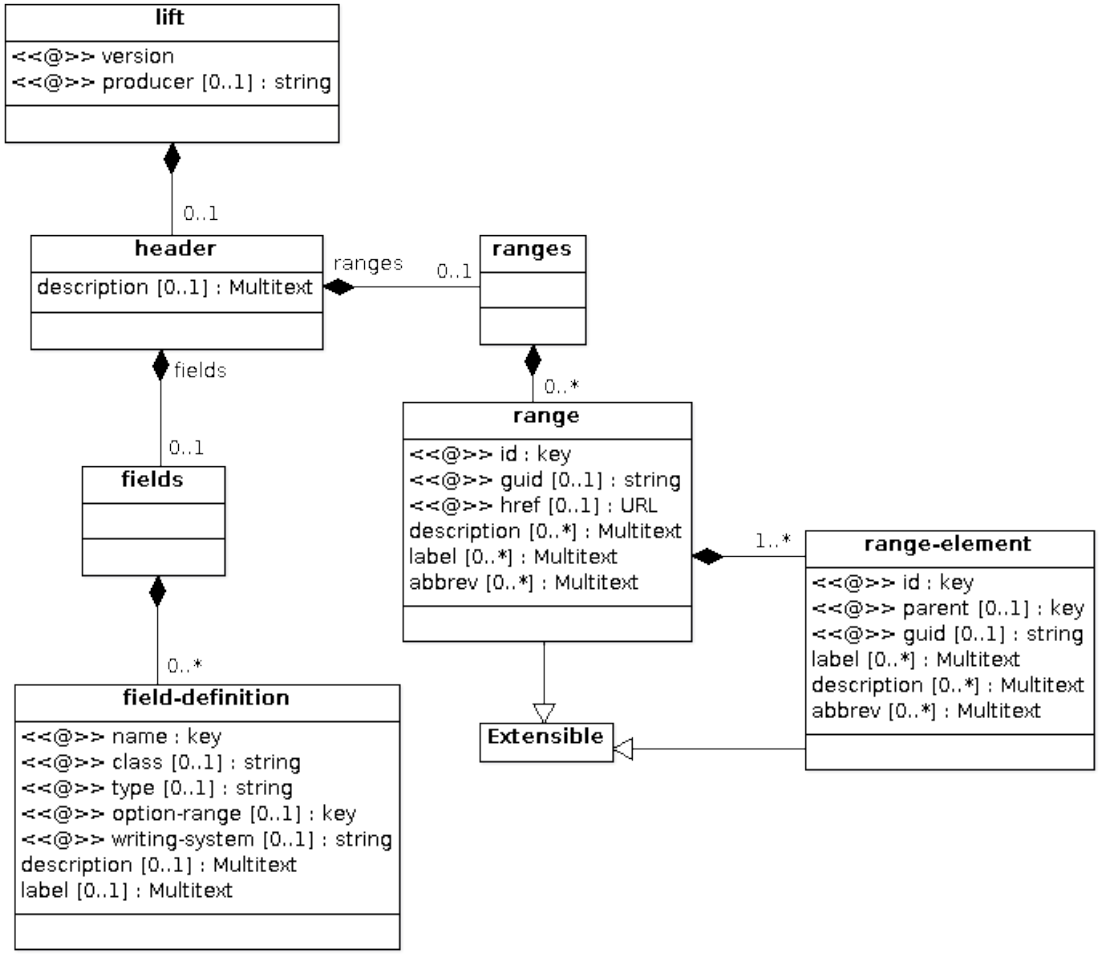
Base Elements



Entry Elements



Header Elements



Lift Ranges

Elements

lift-ranges

Lift ranges can be stored either as part of the header material of the LIFT file, or as one or more separate files referenced by the header of the LIFT file. The top element for ranges in the header section of a LIFT file is `ranges`. The root element in a Lift Ranges file is `lift-ranges`. The content inside this element is the same for both files unless otherwise indicated in the description below.

Elements

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

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.

Inheritance

`Extensible` Adds `dateCreated` and `dateModified` attributes and `field`, `trait`, and `annotation` elements to the content for extensibility.

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. In practice, if the `range` element has an `href` attribute the `id` and `href` values are usually the only information given in the LIFT file: all other information is given in the referenced Lift Ranges file, with the `id` attribute repeated to link from the Lift Ranges file back to the LIFT file..

Elements

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

`range-element` [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.

range-element

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

Inheritance

`Extensible` Adds `dateCreated` and `dateModified` attributes and `field`, `trait`, and `annotation` elements to the content for extensibility.

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.

Elements

`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.

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. In addition, the values specified here correspond to the values used in the FieldWorks application. For specifics of some of the larger sets, the reader is referred to that application. A summary example is listed here in those situations.

anthro-code

This creates a hierarchy of range-elements based on the numeric anthropological codes. For example **`630`** Territorial Organization.

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. Notice that elements in a dialect range are not expected to be full languages or writing systems, but refer to linguistic subgroups of some kind.

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. The following are the core set used in FieldWorks.

Elements

`Adverb` An adverb, narrowly defined, is a part of speech whose members modify verbs for such categories as time, manner, place, or direction. An adverb, broadly defined, is a part of speech whose members modify any constituent class of words other than nouns, such as verbs, adjectives, adverbs, phrases, clauses, or sentences. Under this definition, the possible type of modification depends on the class of the constituent being modified.

`Noun` A noun is a broad classification of parts of speech which include substantives and nominals.

Pro-form	A pro-form is a part of speech whose members usually substitute for other constituents, including phrases, clauses, or sentences, and whose meaning is recoverable from the linguistic or extralinguistic context.
Pronoun	A pronoun is a pro-form which functions like a noun and substitutes for a noun or noun phrase.
Verb	A verb is a part of speech whose members typically signal events and actions; constitute, singly or in a phrase, a minimal predicate in a clause; govern the number and types of other constituents which may occur in the clause; and, in inflectional languages, may be inflected for tense, aspect, voice, modality, or agreement with other constituents in person, number, or grammatical gender.

lexical-relation

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.
Parts	Identify parts in a parts/whole relation.
Specifics	Reference the specific elements from the general in a gen-spec relation.
Synonyms	Marks the relation to other synonyms.
Antonym	Relates to an antonym
Calendar	Used to mark calendar type elements: days, weeks, etc.
Compare	For general compare and contrast relations.
Classifier	Use this to reference words this word classifies.
subentry	Reference from a main entry to a sub entry.
minorentry	Reference to a minor entry from a main entry.

note-type

This lists all the different note types that are in the various elements.

Elements

anthropology	Anthropological information.
Bibliography	Bibliographic information.
comment	This note is an arbitrary comment not for publication
cv-pattern	The pattern of consonants and vowels in a pronunciation.
discourse	Discourse information about a sense.
encyclopedic	This note gives encyclopedic information.

general	General notes that do not fall in another clear category
grammar	Grammatical information about a word.
literal-meaning	The literal meaning of the entry.
phonology	Phonological information about a word.
questions	Contains questions yet to be answered
restrictions	Information on the restriction of usage of a word.
scientific-name	The scientific name of a sense
semantics	Semantic information on the word.
sociolinguistics	Sociolinguistic information about a sense.
source	Information on sources
summary-definition	The summary definition for an entry.
tone	Tone information for a pronunciation.
usage	Information on usage

owner

This range set consists of each of the element types in lift along with fields which take the form field:type where type is the type attribute of the field.

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. There are various sets of semantic domains.

semantic-domain-ddp4

This semantic domain set corresponds to the numeric codes for the Dictionary Discovery Process version 4. For example `1.1.2.1` for words that are about causing air to move.

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 version="0.15"> <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 value="vt"/> <gloss lang="en"><text>slap</text></gloss> <definition> <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="6b7537f0-31c8-42f9-811c-1da97cc907b4"
       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 id="c71333e0-be6c-434b-adda-bdc6c154d562">
    <grammatical-info value="n"/>
    <gloss lang="en"><text>grove</text></gloss>
    <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 artifact. 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

[1] Making Dictionaries, p59

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 modeled.

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="562156e0-bc5e-444a-b699-6c1a7df95851">
  <lexical-unit>
    <form lang="en"><text>brush</text></form>
  </lexical-unit>
  <sense id="aca2cc02-75ad-4e09-909f-8974efc17fd4">
    <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="51e84fed-bd53-4c62-af55-f994a496ee94"/>
    <relation type="subentry"
            ref="c8fa1b21-5769-4e24-82f6-ccc8122c4ed1"/>
  </sense>
</entry>

<entry id="51e84fed-bd53-4c62-af55-f994a496ee94">
  <lexical-unit>
    <form lang="en"><text>hairbrush</text></form>
  </lexical-unit>
  <sense id="27ad1b86-cceb-4a7b-9c93-ccf227ae69bf">
    <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="c8fa1b21-5769-4e24-82f6-ccc8122c4ed1">
  <lexical-unit>
    <form lang="en"><text>paintbrush</text></form>
  </lexical-unit>
  <sense id="bf9f7677-0314-4880-a200-818d43fde490">
    <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.

<table>
<tr>
<td>\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.</td>
<td><pre>
<entry id="db24e454-307d-4939-a517-7852376185c2">
  <lexical-unit>
    <form lang="und-Latn"><text>utan</text></form>
  </lexical-unit>
  <sense id="ae081346-aa0a-4e79-86db-1ebcf94d5a53">
    <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>
</pre></td>
</tr>
</table>

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.

[1] Making Dictionaries, p89

```
\lx hete[1]
\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="5eb49840-eb3b-4f1e-8622-071a10ca30f5">
  <lexical-unit>
    <form lang="und-Latn"><text>hete</text></form>
  </lexical-unit>
  <sense id="b24fbde8-ecad-45ef-a2d4-2ed438aff378">
    <grammatical-info value="vt"/>
    <gloss lang="en"><text>cut</text></gloss>
    <definition>
      <form lang="en">
        <text>cut into sections for use</text>
      </form>
    </definition>
    <relation type="gen"
              ref="131692cd-97c2-428f-bfa4-19e3cb40860f"/>
  </sense>
  <variant>
    <form lang="und-Latn"><text>hetek</text></form>
  </variant><!--no idea what kind of paradigm, so really a
free variant-->
</entry>

<!-- This implies another entry which isn't given in SFM.-->
<entry id="89b76e95-2c77-46fa-8b1f-2a75c00d0944">
  <lexical-unit>
    <form lang="und-Latn"><text>lata</text></form>
  </lexical-unit>
  <sense id="131692cd-97c2-428f-bfa4-19e3cb40860f">
  </sense>
</entry>
```

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

```
\lx apu[2]
\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="b24fbde8-ecad-45ef-a2d4-2ed438aff378">
  <lexical-unit>
    <form lang="und-Latn"><text>apu</text></form>
  </lexical-unit>
  <variant>
    <trait name="dialect" value="lisela"/>
    <trait name="dialect" value="rana"/>
    <form lang="und-Latn"><text>ahul</text></form>
  </variant>
  <sense id="a0fc19e4-fa0d-484c-b500-58fe8268cd0b">
    <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>
  </sense>
  <etymology type="proto" source="und-x-proto">
    <form lang="und-x-proto"><text>apuR</text></form>
    <gloss lang="en"><text>lime, chalk</text></gloss>
  </etymology>
</entry>
```

[1] Making Dictionaries, p116

[2] Making Dictionaries, p119

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="a0fc19e4-fa0d-484c-b500-58fe8268cd0b"><!--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 labeled, it is possible to model the various sense hierarchies that exist.

```
\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="f6434298-b2c2-4be3-98f8-cb8230d90a81">
  <lexical-unit>
    <form lang="und-Latn"><text>opon</text></form>
  </lexical-unit>
  <sense id="4eea7c36-c37a-4f1d-afbe-77e2e8731868" 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>
    <subsense id="a14e3ac9-1a57-4cda-89ca-3d376fb7b8ea">
      <grammatical-info value="n"/>
      <gloss lang="en"><text>ancestor</text></gloss>
      <definition>
        <form lang="en">
          <text>ancestor, descendent</text>
        </form>
      </definition>
    </subsense>
  </sense>
  <sense id="c9eac5cb-a386-4e0f-9650-37c6397f3f5b" 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 something may be written in multiple scripts. This isn't always a case of different languages. We may need to acknowledge that the text is identical but is being stored in two or more writing systems. So, for example if a lexeme were written with two writing systems, there would not be two lexemes but just the one lexeme written two different ways.

[1] Making Dictionaries, p47

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.

```
\lx jɔ̃dɛ̌
\lxt ยองเด ่ง
\dia Ratburi
\a jə̀ŋɗɛ̌ʲ
\ps N
\gt เอว
\ge waist
\so lang1.42.6
\sd body
\dat 21/Feb/2003
```

```
<entry id="0656d719-24c7-467c-afab-949d8c52e4c6">
  <lexical-unit>
    <form lang="und-fonipa"><text>jɔ̃dɛ</text></form>
    <form lang="und-Thai"><text>ยองเด<่ง</text></form>
  </lexical-unit>
  <variant>
    <trait name="dialects" value="Ratburi"/>
    <form lang="und-fonipa"><text>jə̀ŋɗɛ̌ʲ</text></form>
  </variant>
  <sense id="ec1c8b03-658c-4a82-9411-6063d7b8a54e"
         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 available 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 modeling 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

Round-trip 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 round-trips 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`.
		6/May/2008	Remove `range-set` type in favour of `range`.
0.13	SRMc	31/Mar/2009	Changed version number to force automatic XSLT updates of old files. Changed `semantic_domain` to `semantic-domain`, and `scientific_name` to `scientific-name`.
0.14	MJPH	8/Apr/2009	Make `field`-defn, range and `range-element` inherit from `extensible`. Add `literal-meaning` to `entry`. Describe standard ranges so as to be the same as Fieldworks.
0.14.1	MJPH	1/Oct/2009	Restructure document. Add `note` to `example` to make document and relax grammar the same.
0.15	MJPH	26/Apr/2010	Remove `@guid` and change `refid` to be a guid. Add owner range set.
0.15	SRMc	28/Sep/2011	Bring document into conformance with the proposed changes that have been approved by the LIFT community for version 0.15 (LIFT RFC 1 and LIFT RFC 2). Add some additional explanatory material in several places.
		5/Oct/2011	Fix a number of minor errors, including in the examples and the diagrams.