FoLiA: Format for Linguistic Annotation

version 0.9.0 - revision 2.6.1

Documentation

ILK Technical Report – ILK 12-03

Maarten van Gompel
Center for Language Studies
Radboud University Nijmegen

P.O. Box 9103, NL-6500 HD Nijmegen, The Netherlands

URL: http://ilk.uvt.nl/folia/ [1]

November 20, 2012

[1]This document is available from http://ilk.uvt.nl/downloads/pub/papers/ilk.1203.pdf.

Contents

Chapter 1

Introduction

FoLiA is a Format for Linguistic Annotation, derived from the D-Coi format[2] developed as part of the D-Coi project by project partner at Polderland Language and Speech Technologies B.V. The D-Coi format was designed for use by the D-Coi corpus, as well as by its successor; the SoNaR corpus [6]. Though being rooted in the D-Coi format, the FoLiA format goes a lot further and introduces a rich generalised framework for linguistic annotation. FoLiA development started at the ILK research group, Tilburg University, and is continued at the Radboud University Nijmegen. It is being adopted in multiple projects in the Dutch and Flemish Natural Language Processing community.

FoLiA is an XML-based[4] annotation format, suitable for the representation of linguistically annotated language resources. FoLiA's intended use is as a format for storing and/or exchanging language resources, including corpora. Our aim is to introduce a single rich format that can accomodate a wide variety of linguistic annotation types through a single generalised paradigm. We do not commit to any label set, language or linguistic theory. This is always left to the developer of the language resource, and provides maximum flexibility.

XML is an inherently hierarchic format. FoLiA does justice to this by maximally utilising a hierarchic, inline, setup. We inherit from the D-Coi format, which posits to be loosely based on a minimal subset of TEI[5]. Because of the introduction of a new and much broader paradigm, FoLiA is *not* backwards-compatible with D-Coi, i.e. validators for D-Coi will not accept FoLiA XML. It is however easy to convert FoLiA to less complex or verbose formats such as the D-Coi format, or plain-text. Converters will be provided. This may entail some loss of information if the simpler format has no provisions for particular types of information specified in the FoLiA format.

The most important characteristics of FoLiA are:

- **Generalised** paradigm - We use a generalised paradigm, with as few adhoc provisions for annotation types as possible.
- **Expressivity** - The format is highly expressive, annotations can be expressed in great detail and with flexibility to the user's needs, without forcing unwanted details. Moreover, FoLiA has generalised support for representing annotation alternatives, and annotation metadata such as information on annotator, time of annotation, and annotation confidence.
- **Extensible** - Due to the generalised paradigm and the fact that the format does not commit to any label set, FoLiA is fairly easily extensible.
- **Formalised** - The format is formalised, and can be validated on both a shallow and a deep level (the latter including tagset validation), and easily machine parsable, for which tools are provided.
- **Practical** - FoLiA has been developed in a bottom-up fashion right alongside applications, libraries, and other toolkits and converters. Whilst the format is rich, we try to maintain it as simple and straightforward as possible, minimising the learning curve and making it easy to adopt FoLiA in practical applications.

The FoLiA format makes mixed-use of inline and stand-off annotation. Inline annotation is used for annotations pertaining to single tokens, whilst stand-off annotation in a separate annotation layers is adopted for annotation types that span over multiple tokens. This provides FoLiA with the necessary flexibility and extensibility to deal with various kinds of annotations. Inspiration for this was in part obtained from the Kyoto Annotation Format [1].

In publication of research that makes use of FoLiA, a citation should be given of: *"Maarten van Gompel (2012). FoLiA: Format for Linguistic Annotation. Documentation. ILK Technical Report 12-03.*
Available from `http://ilk.uvt.nl/downloads/pub/papers/ilk.1203.pdf`".
FoLiA is open-source and all technical resources are licensed under the GNU Public License v3.

Notable features of the FoLiA format include:

- XML-based, validation against RelaxNG schema.
- Full Unicode support; UTF-8 encoded.
- Support for text as well as speech
- Document structure consists of divisions, paragraphs, sentences and words/tokens, and more specific elements.
- Support for annotation of transcribed speech
- Can be used for both tokenised as well as untokenised text, though for meaningful linguistic annotation, tokenisation is mandatory.
- Provenance support for all linguistic annotations: annotator, type (automatic or manual), time.
- Support for alternative annotations, optionally with associated confidence values.
- Support for features using subsets, allowing for more detailed user-defined annotation.
- Not commited to any label set, these are user-defined.
- Agnostic with regard to metadata. External metadata schemes such as CMDI are recommended.

There is support for the following linguistic annotations:

- Part-of-Speech tags (with features)
- Lemmatisation
- Spelling corrections on both a tokenised as well as an untokenised level
- Lexical semantic sense annotation (to be used in DutchSemCor)
- Named Entities / Multi-word units
- Syntactic Parses
- Dependency Relations
- Chunking

- Morphological Analysis
- Subjectivity Annotation
- Semantic Role Labelling
- Co-reference
- Event annotation

FoLiA support is incorporated directly into the following sofware:

- ucto - A tokeniser which can directly output FoLiA XML
- Frog - A PoS-tagger/lemmatiser/parser suite (the successor of Tadpole), will eventually support reading and writing FoLIA.
- CLAM - Computational Linguistics Application Mediator, will eventually have viewers for the FoLiA format.
- PyNLPl - Python Natural Language Processing Library, comes with a library for parsing FoLiA
- libfolia - C++ library for parsing FoLiA

FoLiA is used in the following projects (list may not be complete):

- SoNaR (STEVIN)
- DutchSemCor (NWO)
- TTNWW (CLARIN)
- DU-VNC (CLARIN)
- Ticclops (CLARIN)
- Valkuil.net
- Basilex (NWO)
- LIN (NWO)

To clearly understand this documentation, note that when we speak of "elements" or "attributes", we refer to XML notation, i.e. XML elements and XML attributes.

1.1 Status information

The FoLiA format, this documentation, and the libraries implementing FoLiA are a constant work in progress. In this documentation, the status and implementation of a certain annotation type is indicated as follows:

Status: final since v0.4 · **Implementations:** pynlpl,libfolia

The above example states that the particular section is final since version 0.4 of FoLiA and that it is implemented in the libraries pynlpl (python) and libfolia ($C++$). You may also see portions of this documentation that are proposals, which means the functionality is still open for debate and not final yet. Example:

Status: PROPOSED in v0.9) · **Implementations:** not implemented yet

Any version of FoLiA and its libraries should be compatible with earlier releases. When things have changed between versions, this is indicated in the documentation.

Chapter 2

Document Format

2.1 Global Structure

In FoLiA, each document/text is represented by one XML file. The basic structure of such a FoLiA document is as follows and should always be UTF-8 encoded.

```
<?xml version="1.0" encoding="utf-8"?>
<FoLiA xmlns="http://ilk.uvt.nl/FoLiA"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  version="0.5"
  xml:id="example">
  <metadata>
      <annotations>
          ...
      </annotations>
  </metadata>
  <text xml:id="example.text">
      ...
  </text>
</FoLiA>
```

2.2 Identifiers

Many elements in the FoLiA format specify an identifier by which the element is uniquely identifiable. This makes referring to any part of a FoLiA document easy and follows the lead of the D-Coi format. Identifiers should be unique in the entire document, and can be anything that qualifies as a valid ID according to the XML standard. A well proven convention is of a cumulative nature, in which you append the element name, a period, and a sequence number, to the identifier of a parent element higher in the hierarchy. Identifiers are always encoded in the `xml:id` attribute,

The FoLiA document as a whole also carries an ID.

Identifiers are very important and used throughout the FoLiA format, and mandatory for almost all structural elements. They enable external resources and databases to easily point to a specific part of the document or an annotation therein. FoLiA has been set up in such a way that *identifiers should never change*. Once an identifier is assigned, it should never change, re-numbering is strictly prohibited unless you intentionally want to create a new resource and break compatibility with the old one.

2.3 Paradigm & Terminology

The FoLiA format has a very uniform setup and its XML notation for annotation follows a generalised paradigm. We distinguish several different categories of annotation, three main categories and several higher-order annotation categories.

- **Structural annotation** - Annotations marking global structure, such as chapters, sections, subsections, figures, list items, paragraphs, sentences, words, morphemes, phonemes etc... Section 2.5 will discuss most structure annotation elements in FoLiA. Morphemes and phonemes are discusses in separate sections.
- **Token annotation** - Annotations pertaining to a specific token. These will be elements of the token element (`w`) in inline notation. Linguistic annotations in this category are for example: part-of-speech annotation (lexical categories), lemma annotation, sense annotation. Various token annotation elements may be used on higher levels (e.g. sentence/paragraph) as well and may then be referred to as **Extended Token Annotation**. Section 2.6 will discuss all token annotations.
- **Span annotation** - Annotations spanning over multiple tokens. Each type of annotation will be in a separate **annotation layer** with stand-off notation. These layers are typically embedded on the sentence level, or possibly

also on higher levels (paragraph/division/text) for certain annotation types. Examples in this category are: syntax, syntactic dependencies, chunking, co-references, semantic roles and named entities. Section 2.7 will discuss all span annotations.

- **Higher-order annotation** - Higher-order annotation consists of several categories of annotation. These all have in common that they annotate either other annotations, or in some way modify or point at other annotations.
  - **Feature annotation** - Feature annotation allows for more detailed annotation. It acts as a feature or attribute to an annotation. This category of annotation will be explained in Section 2.10.4.
  - **Alignment annotation** - Allows for associations between arbitrary annotations within or across FoLiA documents.
  - **Corrections** - Allows corrections or suggestions for correction to be associated with annotations.
  - **Alternatives** - Allows annotations to be markes as alternative.

Almost all annotations are associated with what we call a **set**. The set determines the vocabulary of the annotation, i.e. the tags or types of the annotation. An element of such a set is referred to as a **class**. For example, we may have a document with Part-of-Speech annotation according to the CGN set (a tagset for Dutch part-of-speech tags). The CGN set defines main tag classes such as *WW*, *BW*, *ADJ*, *VZ*. FoLiA itself thus never commits to any tagset but leaves you to define this. You can also use multiple tagsets in the same document if so desired, even for the same type of annotation.

Any annotation element may have a `set` attribute, the value of which points to the URL of the set definition file that defines the set. Such an element then also carries a `class` attribute, which selects a particular class from the set.

In addition to this, various other generic FoLiA attributes are available for all annotation elements. These are never mandatory:

1. `annotator` – The name or ID of the system or human annotator that made the annotation.
2. `annotatortype` – "manual" for human annotators, or "auto" for automated systems.

3. confidence – A floating point value between zero and one; expresses the confidence the annotator places in his annotation.

4. datetime – The date and time when this annotation was recorded, the format is YYYY-MM-DDThh:mm:ss (note the T in the middle to separate date from time), as per the XSD Datetime data type.

5. n – A number in a sequence, corresponding to a number in the original document, for example chapter numbers, section numbers, list item numbers.

The following example shows a simple Part-of-Speech annotation without features, but with various generic attributes according:

```
<pos set="http://ilk.uvt.nl/folia/sets/CGN" class="WW"
 annotator="Maarten␣van␣Gompel" annotatortype="manual"
 confidence="0.76" datetime="1982-12-15T19:01" />
```

The FoLiA paradigm is visualised in the following scheme. Note that the more advanced aspects of the FoLiA paradigm, the higher-order annotation categories, will be introduced later in section 2.10.

2.3.1 Speech

Status: PROPOSED in v0.9) · **Implementations:** not implemented yet

FoLiA is also suited for annotation of speech data. The following additional FoLiA attributes are available for *all* structure annotation elements in a speech context:

- src – **source** – Points to a file or full URL of a sound or video file. This attribute is inheritable.
- begintime – **begin time** – A timestamp in HH:MM:SS.MMMM format, indicating the begin time of this speech act. If a sound clip is specified (src); the timestamp refers to a location in the soundclip (minus srcoffset).
- endtime – **end time** – A timestamp in HH:MM:SS.MMMM format, indicating the end time of this speech act. If a sound clip is specified (src); the timestamp refers to a location in the soundclip (minus srcoffset).

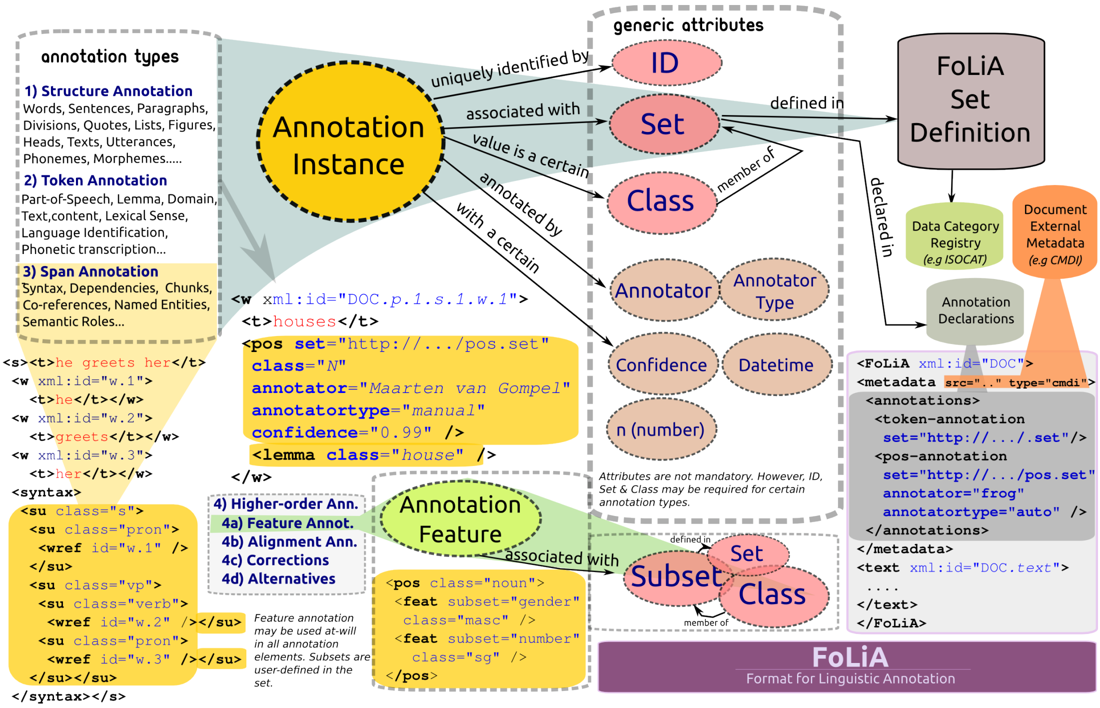

Figure 2.1: The FoLiA Paradigm

- `srcoffset` – **source offset** – A timestamp in `HH:MM:SS.MMMM` format that is *subtracted* from all `begintime`, and `endtime` designations within its scope, to find the position in the audio clip. This attribute enabled the use of global timings in all begintime/endtime attributes, across different sound clips. Defaults to zero.
- `speaker` – **speaker** – A string identifying the speaker. This attribute is inheritable. Multiple speakers are not allowed, simply do not specify a speaker on a certain level if none can be designated.

More about speech annotation in section 2.9.1.

2.4 Annotation Declaration

The annotation declaration is a mandatory part of the metadata that declares all the types of annotation and all sets that are present in the document. Annotations are declared in the annotations block, as shown in the following example. We here define four annotation levels with fictitious sets:

```
<annotations>
        <token-annotation
          set="http://ilk.uvt.nl/folia/sets/ucto-tokconfig-nl"
          annotator="ucto" annotatortype="auto" />
        <pos-annotation set="http://ilk.uvt.nl/folia/sets/CGN"
          annotator="Frog" annotatortype="auto" />
        <lemma-annotation set="http://ilk.uvt.nl/folia/sets/lemmas-nl"
          annotator="Frog" annotatortype="auto" />
        <sense-annotation set="http://ilk.uvt.nl/folia/sets/Cornetto"
         annotator="SupWSD1" annotatortype="auto" />
</annotations>
```

The set attribute is mandatory[1] and refers to a URL of a FoLiA Set Definition file (see chapter 3). In the above example, the set URLs are fictitious, make sure to list your own appropriate sets. A Set Definition specifies exactly what classes are allowed in the set. It for examples specifies exactly what Part-of-Speech tags exist. This information is necessary to completely validate the document at its deepest level. If the sets point to URLs that do not exist or are not URLs at all, warnings will be issued. Validation can still proceed but with the notable exception of deep validation of these sets.

If multiple sets are used for the same annotation type, they each need a separate declaration:

```
        <pos-annotation set="http://ilk.uvt.nl/folia/sets/CGN"
          annotator="Frog" annotatortype="auto" />
        <pos-annotation set="http://ilk.uvt.nl/folia/sets/brown" />
```

If only one set is declared, then in the document itself you are allowed to skip the set attribute on these specific annotation elements. The declared set will automatically be the default.

The annotator and annotatortype attributes act as defaults for the specific annotation type and set. Unlike set, you do *not* need, and it is in fact prohibited, to declare every possible annotator here!

Annotator defaults can always be overridden at the specific annotation elements. But declaring them allows for the annotation element to be less verbosely expressed. Explicitly referring to a set and annotator for each annotation element

[1] Technically, it can be omitted, but then the set defaults to "undefined". This is allowed for flexibility and less explicit usage of FoLiA in limited settings, but not recommended!

can be cumbersome and pointless in a document with a single set and a single annotator for that particular type of annotation. Declarations and defaults provide a nice way around this problem.

2.5 Structure Annotation

2.5.1 Basic Structural Elements

Basic structural elements for textual documents occur within the `text` element. These are the most basic ones:

- `p` - Paragraph
- `s` - Sentence
- `w` - Word (token)

These are typically nested, the word elements cover the actual tokens. This is the most basic level of annotation; tokenisation. Let's take a look at an example, we have the following text:

```
This is a paragraph containing only one sentence.

This is the second paragraph. This one has two sentences.
```

In FoLiA XML, this will appear as follows after tokenisation. Some parts have been omitted for the sake of brevity:

```
<p xml:id="example.p.1">
   <s xml:id="example.p.1.s.1">
       <w xml:id="example.p.1.s.1.w.1"><t>This</t></w>
       <w xml:id="example.p.1.s.1.w.2"><t>is</t></w>
       ...
       <w xml:id="example.p.1.s.1.w.8" space="no"><t>sentence</t></w>
       <w xml:id="example.p.1.s.1.w.9"><t>.</t></w>
   </s>
</p>
<p xml:id="example.p.2">
```

```
    <s xml:id="example.p.2.s.1">
        <w xml:id="example.p.2.s.1.w.1"><t>This</t></w>
        <w xml:id="example.p.2.s.1.w.2"><t>is</t></w>
        ..
        <w xml:id="example.p.2.s.1.w.5" space="no"><t>paragraph</t></w>
        <w xml:id="example.p.2.s.1.w.6"><t>.</t></w>
    </s>
    <s xml:id="example.p.2.s.2">
        <w xml:id="example.p.2.s.2.w.1"><t>This</t></w>
        <w xml:id="example.p.2.s.2.w.2"><t>one</t></w>
        ..
        <w xml:id="example.p.2.s.2.w.5" space="no"><t>sentences</t></w>
        <w xml:id="example.p.2.s.2.w.6"><t>.</t></w>
    </s>
</p>
```

FoLiA is not just a format for holding tokenised text, although tokenisation is a prerequisite for almost all kinds of annotation. However, FoLiA can also hold untokenised text, on for example paragraph and/or sentence level:

```
<p xml:id="example.p.1">
    <s xml:id="example.p.1.s.1">
        <t>This is a paragraph containing only one sentence.</t>
    </s>
</p>
<p xml:id="example.p.2">
    <s xml:id="example.p.2.s.1">
        <t>This is the second paragraph.</t>
    </s>
    <s xml:id="example.p.2.s.2">
        <t>This one has two sentences.</t>
    </s>
</p>
```

Higher level elements *may* also contain a text element even when the deeper elements do too. It is very important to realise that the sentence/paragraph-level text element *always* contains the text *prior* to tokenisation! Note also that the word element has an attribute space, which defaults to yes, and indicates whether the word was followed by a space in the *untokenised* original. This allows for partial reconstructibility of the sentence in its untokenised form. See section 2.10.10 for a more elaborate overview of this subject.

The following example shows the maximum amount of redundancy, with text elements at every level.

```
<p xml:id="example.p.1">
    <t>This is a paragraph containing only one sentence.</t>
```

```
    <s xml:id="example.p.1.s.1">
        <t>This is a paragraph containing only one sentence.</t>
        <w xml:id="example.p.1.s.1.w.1"><t>This</t></w>
        <w xml:id="example.p.1.s.1.w.2"><t>is</t></w>
         ...
        <w xml:id="example.sp.1.s.1.w.8" space="no"><t>sentence</t></w>
        <w xml:id="example.p.1.s.1.w.9"><t>.</t></w>
    </s>
</p>
<p xml:id="example.p.2">
    <t>This is the second paragraph. This one has two sentences.</t>
    <s xml:id="example.p.2.s.1">
        <t>This is the second paragraph.</t>
        <w xml:id="example.p.2.s.1.w.1"><t>This</t></w>
        <w xml:id="example.p.2.s.1.w.2"><t>is</t></w>
         ..
        <w xml:id="example.p.2.s.1.w.5" space="no"><t>paragraph</t></w>
        <w xml:id="example.p.2.s.1.w.6"><t>.</t></w>
    </s>
    <s xml:id="example.p.2.s.2">
        <t>This one has two sentences.</t>
        <w xml:id="example.p.2.s.2.w.1"><t>This</t></w>
        <w xml:id="example.p.2.s.2.w.2"><t>one</t></w>
         ..
        <w xml:id="example.p.2.s.2.w.5" space="no"><t>sentences</t></w>
        <w xml:id="example.p.2.s.2.w.6"><t>.</t></w>
    </s>
</p>
```

If this kind of redundancy is used (it is not mandatory), you may optionally point back to the text content of its parent by specifying the `offset` attribute:

```
<p xml:id="example.p.1">
    <t>This is a paragraph containing only one sentence.</t>
    <s xml:id="example.p.1.s.1">
        <t offset="0">This is a paragraph containing only one sentence.</t>
        <w xml:id="example.p.1.s.1.w.1">
                <t offset="0">This</t>
        </w>
        <w xml:id="example.p.1.s.1.w.2">
                <t offset="5">is</t>
        </w>
         ...
        <w xml:id="example.p.1.s.1.w.8" space="no">
                <t offset="40">sentence</t>
        </w>
        <w xml:id="example.p.1.s.1.w.9">
                <t offset="48">.</t>
        </w>
```

```
    </s>
</p>
```

Matters can become more complicated as multiple text-content element of different classes may be associated with an element, this will be discussed later on in section 2.10.10.

Paragraph elements may be omitted if a document is described that does not distinguish paragraphs but only sentences. Sentences however may never be omitted; FoLiA documents can never consist of tokens only.

The content element `head` is reserved for headers and captions, it behaves similarly to the paragraph element and holds sentences.

FoLiA also explicitly supports quotes, as demonstrated in the next example, which annotates the following sentence:

```
He said: ``I do not know . I think you are right. ", and left.
```

A quote may consist of one or more sentences, but may also consist of mere tokens:

```
<s xml:id="example.p.1.s.1">
  <w xml:id="example.p.1.s.1.w.1" class="WORD"><t>He</t></w>
  <w xml:id="example.p.1.s.1.w.2" class="WORD"><t>said</t></w>
  <w xml:id="example.p.1.s.1.w.3" class="PUNCTUATION" space="no">
        <t>:</t>
  </w>
  <w xml:id="example.p.1.s.1.w.4" class="PUNCTUATION" space="no">
        <t>''</t>
  </w>
  <quote xml:id="example.p.1.s.1.quote.1">
    <s xml:id="example.p.1.s.1.quote.1.s.1">
       <w xml:id="example.p.1.s.1.w.5" class="WORD"><t>I</t></w>
       <w xml:id="example.p.1.s.1.w.6" class="WORD"><t>do</t></w>
       <w xml:id="example.p.1.s.1.w.7" class="WORD"><t>not</t></w>
       <w xml:id="example.p.1.s.1.w.8" class="WORD"><t>know</t></w>
       <w xml:id="example.p.1.s.1.w.9" class="PUNCTUATION" space="no">
        <t>.</t>
       </w>
    </s>
    <s xml:id="example.p.1.s.1.quote.1.s.2">
       <w xml:id="example.p.1.s.1.w.10" class="WORD"><t>I</t></w>
       <w xml:id="example.p.1.s.1.w.11" class="WORD"><t>think</t></w>
       <w xml:id="example.p.1.s.1.w.12" class="WORD"><t>you</t></w>
```

```
      <w xml:id="example.p.1.s.1.w.13" class="WORD"><t>are</t></w>
      <w xml:id="example.p.1.s.1.w.14" class="WORD"><t>right</t></w>
    </s>
  </quote>
  <w xml:id="example.p.1.s.1.w.15" class="PUNCTUATION" space="no"><t>''</t></w>
  <w xml:id="example.p.1.s.1.w.16" class="PUNCTUATION"><t>,</t></w>
  <w xml:id="example.p.1.s.1.w.17" class="WORD"><t>and</t></w>
  <w xml:id="example.p.1.s.1.w.18" class="WORD"><t>left</t></w>
  <w xml:id="example.p.1.s.1.w.19" class="PUNCTUATION" space="no">
        <t>.</t>
  </w>
 </s>
```

2.5.2 Paragraphs, Sentences and Words

Paragraphs, sentences and words (or tokens) are amongst the most elementary structure elements. As has been seen in a previous section, word elements (`w`) can take a class, pertaining to a certain set, at which point a definition must be present in the metadata, note that the actual set URL is just an example, assign your own:

```
<annotations>
    <token-annotation set="http://ilk.uvt.nl/folia/sets/ucto-tokconfig-nl"
      annotator="ucto" annotatortype="auto" />
</annotations>
```

Being part of a set, this implies that tokens themselves *may* be assigned a class, as is for example done by the tokeniser *ucto*:

```
<s xml:id="example.p.1.s.1">
        <t>I see 2 children.</t>
    <w xml:id="example.p.1.s.1.w.1" class="WORD"><t>I</t></w>
    <w xml:id="example.p.1.s.1.w.2" class="WORD"><t>see</t></w>
    <w xml:id="example.p.1.s.1.w.3" class="NUMBER"><t>2</t></w>
    <w xml:id="example.p.1.s.1.w.4" class="WORD" space="no">
                <t>children</t>
    </w>
    <w xml:id="example.p.1.s.1.w.5" class="PUNCTUATION"><t>.</t></w>
</s>
```

The same can be applied to paragraphs and sentences, which requires a declaration of `paragraph-annotation` and `sentence-annotation` respectively.

2.5.3 Divisions

Within the `text` element, the structure element `div` can be used to create divisions and subdivisions. Each division *may* be of a particular *class* pertaining to a *set* defining all possible classes.

Divisions and other structural units are often numbered, think for example of chapters and sections. The number, as it was in the source document, can be encoded in the `n` attribute of the structure annotation element.

Look at the following example, showing a full FoLiA document with structured divisions. The declared set is a fictitious example:

```xml
<?xml version="1.0" encoding="utf-8"?>
<?xml-stylesheet type="text/xsl" href="http://ilk.uvt.nl/FoLiA/FoLiA.xsl"?>
<FoLiA xmlns="http://ilk.uvt.nl/FoLiA"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  version="0.5"
  xml:id="example">
  <metadata>
      <annotations>
          <div-annotation set="http://ilk.uvt.nl/folia/sets/divisions" />
      </annotations>
  </metadata>
  <text xml:id="example.text">
     <div class="chapter" n="1">
        <head><t>Introduction</t></head>
        <div class="section" n="1">
            <div class="subsection" n="1.1">
                <t>In the beginning ....</t>
            </div>
        </div>
        ...
     </div>
  </text>
</FoLiA>
```

Divisions stem from D-Coi and are modified in FoLiA. These divisions are not mandatory, but may be used to mark extra structure. D-Coi supported the elements `div0`, `div1`, `div2`, etc.., but FoLiA only knows a single `div` element, which can be nested at will and associated with classes. Note that paragraphs, sentences and words have there own explicit tags, as seen earlier, divisions should never be used for marking these, only larger structures can be divisions.

The `head` element may be used to for the header of any division. It may hold `s` and w elements (not p).

2.5.4 Gaps

Status: final since v0.8 (older versions are equal but lack declarations) · **Implementations:** pynlpl, libfolia

Sometimes there are parts of a document you want to skip and not annotate, but include as is. For this purpose the gap element should be used. Gaps may have a particular class indicating the kind of gap it is. Common omissions are for example front-matter and back-matter.

The D-Coi format pre-defined the following "reasons" [2]:

- frontmatter
- backmatter
- illegible
- other-language
- cancelled
- inaudible
- sampling

Due to the flexible nature of FoLiA, we never predefine any classes whatsoever and leave this up to whatever set is declared. The above gives a good indication of what gaps can be used for though.

The gap element may optionally take two elements:

1. `desc` - holding a substitute that may be shown to the user, describing what has been omitted.
2. `content` - The actual raw content of the omission, as it was without further annotations. This is an XML CDATA type element, excluding it from any kind of parsing.

```
  <text xml:id="example.text">
     <gap class="frontmatter" annotator="Maarten van Gompel">
        <desc>This is the cover of the book</desc>
        <content>
<![CDATA[

               SHOW WHITE AND THE SEVEN DWARFS


                     by the Brothers Grimm

                          first edition


               Copyright(c) blah blah
]]>
        </content>
     </gap>
     <div class="chapter" n="1">
        <head><t>Introduction</t></head>
        <div class="section" n="1">
            <div class="subsection" n="1.1">
                <t>In the beginning....</t>
            </div>
        </div>
        ...
     </div>
  </text>
```

Gaps have to be declared, the actual set used here is just a fictitious example:

```
<annotations>
    <gap-annotation set="http://ilk.uvt.nl/folia/sets/dcoi-gaps" />
</annotations>
```

2.5.5 Whitespace and Linebreaks

Status: final · **Implementations:** pynlpl, libfolia

Sometimes you may want to explicitly specify vertical whitespace or line breaks. This can be done using respectively `whitespace` and `br`. Both are simple structural elements that need not be declared. Note that using `p` to denote paragraphs is always strongly preferred over using `whitespace` to mark their boundaries!

```
  <text xml:id="example.text">
```

```
    <s xml:id="example.s.1">
        <w xml:id="example.s.1.w.1">
        <br />
        <w xml:id="example.s.1.w.2">
        <w xml:id="example.s.1.w.3">
    </s>
    <whitespace />
    <s xml:id="example.s.2">
    </s>
  </text>
```

The difference between `br` and `whitespace` is that the former specifies that only a linebreak was present, not forcing any vertical whitespace, whilst the latter actually generates an empty space, which would comparable to two successive `br` statements. Both elements can be used inside divisions, paragraphs, and sentences.

2.5.6 Events

Status: final since v0.7 · **Implementations:** pynlpl, libfolia

Event structure, though uncommon to regular written text, can be useful in certain documents. Divisions, paragraphs, sentences, or even words can be encapsulated in an event element to indicate they somehow form an event entity of a particular class. This kind of structure annotation is especially useful in dealing with written media such as chat logs, tweets, and internet fora, in which chat turns, forum posts, and tweets can be demarcated as particular events.

Below an example of a simple chat log, word tokens omitted for brevity:

```
<event class="message" begindatetime="2011-12-15T19:01"
 enddatetime="2011-12-15T19:05" actor="Jane␣Doe">
    <s>
        <t>Hello John.</t>
    </s>
    <s>
        <t>How are you doing?</t>
    </s>
</event>
<event class="message" begindatetime="2011-12-15T19:06"
 actor="John␣Doe">
    <s>
        <t>I am fine Jane, thanks.</t>
    </s>
```

```
</event>
```

The (optional) features `begindatetime` and `enddatetime` can be used express the exact moment at which an event started or ended. Note that this differs from the generic `datetime` attribute, which would describe the time at which the annotation was recorded, rather than when the event took place! Also, `begindatetime` and `enddatetime` are so-called *features* (see section

The declaration, the actual set is fictitious:

```
<annotations>
    <event-annotation set="http://ilk.uvt.nl/folia/sets/events" />
</annotations>
```

For more fine-grained control over timed events, for example within sentences. It is recommended to use the `timesegment` span annotation element instead! This works in a very similar fashion but uses a stand-off annotation layer. See section 2.7.5.

2.5.7 Lists

Status: final · **Implementations:** pynlpl, libfolia

FoLiA, like D-Coi, allows lists to be explicitly marked as shown in the following example:

```
<head><t>My grocery list</t></head>
<list xml:id="example.list.1">
  <item xml:id="example.list.1.item.1" n="A"><t>Apples</t></item>
  <item xml:id="example.list.1.item.2" n="B"><t>Pears</t></item>
</list>
```

The item element may hold sentences (`s`) and words (`w`). The D-Coi format had a `label` element, this is deprecated in favour of the `n` attribute in the item itself.

Lists, like paragraphs, sentences and headers are content elements that need not be declared and are not associated with a set or class.

2.5.8 Figures

Status: final · **Implementations:** pynlpl, libfolia

Even figures can be encoded in the FoLiA format, although the actual figure itself can only be included as a mere reference to an external image file, but including such a reference (src attribute) is optional.

```
<figure xml:id="example.figure.1" n="1" src="/path/or/url/to/image/file">
  <desc>A textual description of the figure (Like ALT in HTML)</desc>
  <caption><t>The caption for the figure</t></caption>
</figure>
```

Figures are not declared. The caption element may hold sentences (s) and words (w).

2.5.9 Tables

Status: TO BE PROPOSED · **Implementations:** no

Development Notes
There is no provision yet for encoding tables. This may be added later.

2.6 Token Annotation

Token annotations are annotations that are placed within the word (w) element, or by extension within other structure elements, in which case we speak of *extended token annotation*.

All token annotation elements may take all of the generic attributes describes in Section 2.3; this has to be kept in mind when reading this section. Moreover, all token annotations depend on the document being tokenised, i.e. there being w elements.

2.6.1 Part of Speech Annotation

Status: final · **Implementations:** pynlpl, libfolia

Part-of-Speech annotation allows the annotation of lexical categories using the pos element. The following example illustrates a simple Part-of-Speech annotation for the word "boot":

```
<w xml:id="example.p.1.s.1.w.2">
    <t>boot</t>
    <pos class="N" />
</w>
```

Lexical annotation can take more complex forms than assignment of single part-of-speech tag. There may for example be numerous features associated with the part of speech tag, such as gender, number, case, tense, mood, etc... FoLiA introduces a special paradigm for dealing with such features. We will look into this later, in section 2.10.

Whenever Part-of-Speech annotations are used, they should be declared in the `annotations` block as follows. The set you use may differ and all further attributes are optional and are used to set defaults. In the declaration example here it is as if the annotations were made by the software *Frog*, but you will want to use your own. Do note the requirement of a `token-annotation` as well.

```
<annotations>
    <token-annotation set="http://ilk.uvt.nl/folia/sets/ucto-tokconfig-nl"
     annotator="ucto" annotatortype="auto" />
    <pos-annotation set="http://ilk.uvt.nl/folia/sets/CGN"
     annotator="Frog" annotatortype="auto" />
</annotations>
```

As mentioned earlier, the declaration only sets defaults for annotator and annotatortype. They can be overridden in the pos element itself (or any other token annotation element for that matter).

2.6.2 Lemma Annotation

Status: final · **Implementations:** pynlpl, libfolia

In the FoLiA paradigm, lemmas are perceived as classes within the (possibly open) set of all possible lemmas. Their annotation proceeds as follows:

```
<w xml:id="example.p.1.s.1.w.2">
    <t>boot</t>
    <lemma class="boot" />
</w>
```

And the example declaration:

```
<annotations>
    <token-annotation set="http://ilk.uvt.nl/folia/sets/ucto-tokconfig-nl"
     annotator="ucto" annotatortype="auto" />
    <lemma-annotation set="http://ilk.uvt.nl/folia/sets/mblem-nl"
     annotator="Frog" annotatortype="auto" />
</annotations>
```

2.6.3 Language Identification Annotation

Status: final since v0.8.1 · **Implementations:** not yet

Language identification is used to identify a certain element as being in a certain language. In FoLiA, the `lang` element is used to identify language:

```
<w xml:id="example.p.1.s.1.w.2">
    <t>boot</t>
        <lang class="eng" />
</w>
```

This is an extended token annotation element that can also be used directly on other levels, such as a sentence, paragraph, division, or text level

And the example declaration:

```
<annotations>
    <lang-annotation set="http://ilk.uvt.nl/folia/sets/iso639-3" />
</annotations>
```

2.6.4 Lexical Semantic Sense Annotation

Status: final · **Implementations:** pynlpl,libfolia

In semantic sense annotation, the classes in most sets will be a kind of lexical unit ID. In systems that make a distinction between lexical units and synonym sets (synsets), the synset attribute is available for notation of the latter. In systems with only synsets and no other primary form of lexical unit, the class can simply be set to the synset.

A human readable description for the *sense* element, "beeldhouwwerk", could be placed inside a desc element, but this is optional.

```
<w xml:id="example.p.1.s.1.w.2">
    <t>beeld</t>
    <sense class="r_n-6220" synset="d_n-32683">
        <desc>beeldhouwwerk</desc>
    </sense>
</w>
```

The example declaration is as follows:

```
<annotations>
    <token-annotation set="http://ilk.uvt.nl/folia/sets/ucto-tokconfig-nl"
     annotator="ucto" annotatortype="auto" />
    <sense-annotation set="http://ilk.uvt.nl/folia/sets/cornetto" />
</annotations>
```

2.6.5 Domain Tags

Status: final · **Implementations:** pynlpl,libfolia

Domain annotation is used to associate a certain domain with a structural element. This is an extended token annotation element, which means it can also be used directly in any of the content elements, such as sentence (s) and paragraph (p). It can even be used in the text element itself. Example:

```
<w xml:id="example.p.1.s.1.w.2">
    <t>boot</t>
    <domain class="nautical" />
</w>
```

The declaration, the actual set is fictitious:

```
<annotations>
    <token-annotation set="http://ilk.uvt.nl/folia/sets/ucto-tokconfig-nl"
     annotator="ucto" annotatortype="auto" />
    <domain-annotation set="http://ilk.uvt.nl/folia/sets/domains-nl" />
</annotations>
```

2.7 Span Annotation

Not all annotations can be realised as token annotations. Some typically span multiple tokens. For these we introduce a kind of offset notation in separate *annotation layers*. These annotation layers are generally embedded at the sentence level, *after* the word tokens, though they may also be embedded on higher levels. Within these layers, references are made to all of the the word tokens spanned using the `wref` element. Each annotation layer is specific to a kind of span annotation and associated with a set, for which a declaration should be present in the metadata section of the document.

2.7.1 Entities

Status: final · **Implementations:** pynlpl,libfolia

Named entities or other multi-word units can be encoded in the `entities` layer. Below is an example of a full sentence in which one name is tagged. It is recommended for each entity to have a unique identifier.

```
<s xml:id="example.p.1.s.1">
  <t>The Dalai Lama greeted him.</t>
  <w xml:id="example.p.1.s.1.w.1"><t>The</t></w>
  <w xml:id="example.p.1.s.1.w.2"><t>Dalai</t></w>
  <w xml:id="example.p.1.s.1.w.3"><t>Lama</t></w>
  <w xml:id="example.p.1.s.1.w.4"><t>greeted</t></w>
  <w xml:id="example.p.1.s.1.w.5"><t>him</t></w>
  <w xml:id="example.p.1.s.1.w.6"><t>.</t></w>
  <entities>
    <entity xml:id="example.p.1.s.1.entity.1" class="person">
        <wref id="example.p.1.s.1.w.2" t="Dalai" />
        <wref id="example.p.1.s.1.w.3" t="Lama" />
    </entity>
  </entities>
</s>
```

Note that elements that are not part of any span annotation need never be included in the layer. The `wref` element takes an *optional* `t` attribute which contains a copy of the text of the word pointed at. This is to facilitate human readability and prevent the need for resolving words for simple applications in which only the textual content is of interest.

2.7.2 Syntax

Status: final · **Implementations:** pynlpl,libfolia

A very typical form of span annotation is syntax annotation. This is done within the `syntax` layer and introduces a nested hierarchy of syntactic unit (`su`) elements. It is recommended for each syntactic unit to have a unique identifier:

```
<s xml:id="example.p.1.s.1">
  <t>The Dalai Lama greeted him.</t>
  <w xml:id="example.p.1.s.1.w.1"><t>The</t></w>
  <w xml:id="example.p.1.s.1.w.2"><t>Dalai</t></w>
  <w xml:id="example.p.1.s.1.w.3"><t>Lama</t></w>
  <w xml:id="example.p.1.s.1.w.4"><t>greeted</t></w>
  <w xml:id="example.p.1.s.1.w.5"><t>him</t></w>
  <w xml:id="example.p.1.s.1.w.6"><t>.</t></w>
  <syntax>
    <su xml:id="example.p.1.s.1.su.1" class="s">
      <su xml:id="example.p.1.s.1.su.1_1" class="np">
          <su xml:id="example.p.1.s.1.su.1_1_1" class="det">
             <wref id="example.p.1.s.1.w.1" t="The" />
          </su>
          <su xml:id="example.p.1.s.1.su.1_1_2" class="pn">
             <wref id="example.p.1.s.1.w.2" t="Dalai" />
             <wref id="example.p.1.s.1.w.3" t="Lama" />
          </su>
       </su>
     </su>
     <su xml:id="example.p.1.s.1.su.1_2" class="vp">
        <su xml:id="example.p.1.s.1.su.1_1_1" class="v">
            <wref id="example.p.1.s.1.w.4" t="greeted" />
        </su>
        <su xml:id="example.p.1.s.1.su.1_1_2" class="pron">
          <wref id="example.p.1.s.1.w.5" t="him" />
        </su>
     </su>
    </su>
  </syntax>
</s>
```

As is prescribed by the FoLiA paradigm; the classes always depend on the set used. You can use whatever system of syntactic annotation you desire. Moreover, any of the `su` elements can have the common attributes `annotator`, `annotatortype` and `confidence`.

The above example illustrated a fairly simple syntactic parse. Dependency parses are possible too. Dependencies are listed separate from the syntax in an extra annotation layer, see seection 2.7.3.

The declaration is as follows, the actual set is fictitious:

```
<annotations>
    <token-annotation set="http://ilk.uvt.nl/folia/sets/ucto-tokconfig-nl"
     annotator="ucto" annotatortype="auto" />
    <syntax-annotation set="http://ilk.uvt.nl/folia/sets/syntax-nl" />
</annotations>
```

2.7.3 Dependency Relations

Status: slightly revised in v0.8 (no "su" attribute on hd/dep) · **Implementations:** pynlpl,libfolia

Dependency relations are relations between tokens, in most cases equal to syntactic units. A dependency relation is often of a particular class and consists of a single head component and a single dependent component. In the sample "He sees", there is syntactic dependency between the two words: "sees" is the head, and "He" is the dependant, and the relation class is something like "subject", as the dependant is the subject of the head word. Each dependency relation is explicitly noted in FoLiA.

The element `dependencies` introduces this annotation layer. Within it, `dependency` elements describe all dependency pairs.

In the below example, we show a Dutch sentence parsed with the Alpino Parser [3]. We show not only the dependency layer, but also the syntax layer to which it is related. The `dependency` element always contains one head element (`hd`) and one dependant element (`dep`). The words they cover are reiterated in the usual fashion, using `wref`. For a better understanding, the figure below illustrates the syntactic parse with the dependency relations. Both `hd` and `dep` can optionally make extra reference to a syntactic unit (or anything else for that matter) by means of the `aref` element.

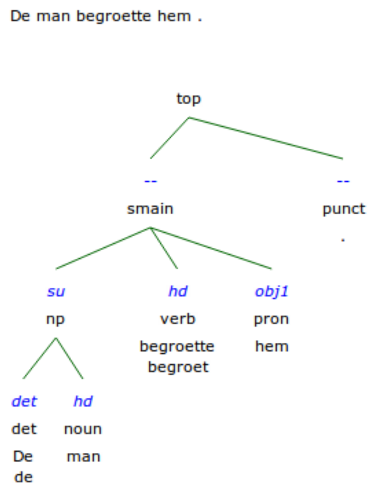

Figure 2.2: Alpino dependency parse for the Dutch sentence “De man begroette hem.”

```
<s xml:id="example.p.1.s.1">
  <t>De man begroette hem.</t>
  <w xml:id="example.p.1.s.1.w.1"><t>De</t></w>
  <w xml:id="example.p.1.s.1.w.2"><t>man</t></w>
  <w xml:id="example.p.1.s.1.w.3"><t>begroette</t></w>
  <w xml:id="example.p.1.s.1.w.4"><t>hem</t></w>
  <w xml:id="example.p.1.s.1.w.5"><t>.</t></w>
  <syntax>
    <su xml:id="example.p.1.s.1.su.1" class="top">
        <su xml:id="example.p.1.s.1.su.1_1" class="smain">
            <su xml:id="example.p.1.s.1.su.1_1_1" class="np">
                <su xml:id="example.p.1.s.1.su.1_1_1_1" class="top">
                    <wref id="example.p.1.s.1.w.1" t="De" />
                </su>
                <su xml:id="example.p.1.s.1.su.1_1_1_2" class="top">
                    <wref id="example.p.1.s.1.w.2" t="man" />
                </su>
            </su>
            <su xml:id="example.p.1.s.1.su.1_1_2" class="verb">
                <wref id="example.p.1.s.1.w.3" t="begroette" />
            </su>
            <su xml:id="example.p.1.s.1.su.1_1_3" class="pron">
                <wref id="example.p.1.s.1.w.4" t="hem" />
            </su>
        </su>
```

```
        <su xml:id="example.p.1.s.1.su.1_2" class="punct">
            <wref id="example.p.1.s.1.w.5" t="." />
        </su>
    </su>
  </syntax>
  <dependencies>
    <dependency xml:id="example.p.1.s.1.dependency.1" class="su">
        <hd>
            <wref id="example.p.1.s.1.w.3" t="begroette">
            <aref id="example.p.1.s.1.su.1_1_2" type="su">
        </hd>
        <dep>
            <wref id="example.p.1.s.1.w.2" t="man" />
            <aref id="example.p.1.s.1.su.1_1_1" type="su">
        </dep>
    </dependency>
    <dependency xml:id="example.p.1.s.1.dependency.3" class="obj1">
        <hd>
            <wref id="example.p.1.s.1.w.3" t="begroette">
            <aref id="example.p.1.s.1.su.1_1_2" type="su">
        </hd>
        <dep>
            <wref id="example.p.1.s.1.w.4" t="hem" />
            <aref id="example.p.1.s.1.su.1_1_3" type="su">
        </dep>
    </dependency>
    <dependency xml:id="example.p.1.s.1.dependency.2" class="det">
        <hd>
           <wref id="example.p.1.s.1.w.2" t="man" />
           <aref id="example.p.1.s.1.su.1_1_1_2" type="su">
        </hd>
        <dep>
            <wref id="example.p.1.s.1.w.1" t="De" />
            <aref id="example.p.1.s.1.su.1_1_1_1" type="su">
        </dep>
    </dependency>
  </dependencies>
</s>
```

Note that in the first dependency relation, the dependent is just "man" rather than "de man" . That is, we point only to the head of dependents, the full scope follows automatically when building the dependency tree.

The declaration, the actual sets are fictitious:

```
<annotations>
    <token-annotation set="http://ilk.uvt.nl/folia/sets/ucto-tokconfig-nl"
     annotator="ucto" annotatortype="auto" />
```

```
    <syntax-annotation set="http://ilk.uvt.nl/folia/sets/alpino-syntax" />
    <dependency-annotation set="http://ilk.uvt.nl/folia/sets/alpino-dep" />
</annotations>
```

2.7.4 Chunking

Status: final · **Implementations:** pynlpl,libfolia

Unlike a full syntactic parse, chunking is not nested. The layer for this type of linguistic annotation is predictably called `chunking`. The span annotation element itself is `chunk`.

```
<s xml:id="example.p.1.s.1">
  <t>The Dalai Lama greeted him.</t>
  <w xml:id="example.p.1.s.1.w.1"><t>The</t></w>
  <w xml:id="example.p.1.s.1.w.2"><t>Dalai</t></w>
  <w xml:id="example.p.1.s.1.w.3"><t>Lama</t></w>
  <w xml:id="example.p.1.s.1.w.4"><t>greeted</t></w>
  <w xml:id="example.p.1.s.1.w.5"><t>him</t></w>
  <w xml:id="example.p.1.s.1.w.6"><t>.</t></w>
  <chunking>
    <chunk xml:id="example.p.1.s.1.chunk.1">
        <wref id="example.p.1.s.1.w.1" t="The" />
        <wref id="example.p.1.s.1.w.2" t="Dalai" />
        <wref id="example.p.1.s.1.w.3" t="Lama" />
    </chunk>
    <chunk xml:id="example.p.1.s.1.chunk.2">
        <wref id="example.p.1.s.1.w.4" t="greeted" />
    </chunk>
    <chunk xml:id="example.p.1.s.1.chunk.3">
        <wref id="example.p.1.s.1.w.5" t="him" />
        <wref id="example.p.1.s.1.w.6" t="." />
    </chunk>
  </chunking>
</s>
```

The declaration, the actual sets are fictitious:

```
<annotations>
    <chunking-annotation set="http://ilk.uvt.nl/folia/sets/syntax-nl" />
</annotations>
```

2.7.5 Time segmentation

Status: final since v0.8, renamed in v0.9 · **Implementations:** pynlpl

FoLiA supports time segmentation using the `timing` layer and the `timesegment` span annotation element. This element is useful for speech, but can also be used for event annotation. We already saw events as structure annotation in Section 2.5.6, but for more fine-grained control of timing information a span annotation element in an offset layer is more suited. The following example illustrates the usage for event annotation:

```xml
<s>
 <w xml:id="example.p.1.s.1.w.1"><t>I</t></w>
 <w xml:id="example.p.1.s.1.w.2"><t>think</t></w>
 <w xml:id="example.p.1.s.1.w.3"><t>I</t></w>
 <w xml:id="example.p.1.s.1.w.4"><t>have</t></w>
 <w xml:id="example.p.1.s.1.w.5"><t>to</t></w>
 <w xml:id="example.p.1.s.1.w.6"><t>go</t></w>
 <w xml:id="example.p.1.s.1.w.7"><t>.</t></w>
 <timing>
  <timesegment class="utterance" begindatetime="2011-12-15T19:01"
   enddatetime="2011-12-15T19:03" actor="myself">
    <wref id="example.p.1.s.1.w.1" t="I" />
    <wref id="example.p.1.s.1.w.2" t="think" />
  </timesegment>
  <timesegment class="cough" begintime="2011-12-15T19:03"
   endtime="2011-12-15T19:05" actor="myself">
  </timesegment>
  <timesegment class="utterance" begindatetime="2011-12-15T19:05"
   enddatetime="2011-12-15T19:06" actor="myself">
    <wref id="example.p.1.s.1.w.3" t="I" />
    <wref id="example.p.1.s.1.w.4" t="have" />
    <wref id="example.p.1.s.1.w.5" t="to" />
    <wref id="example.p.1.s.1.w.6" t="go" />
  </timesegment>
 </timing>
</s>
```

Time segments may also be nested. As always, the classes in the example are set-defined rather than predefined by FoLiA. The prefefined and optional features `begindatetime` and `enddatetime` can be used express the exact moment at which an event started or ended. These too are set-defined so the format shown here is just an example.

The declaration for time segmentation is as follows:

```
<annotations>
    <timesegment-annotation set="http://ilk.uvt.nl/folia/sets/events" />
</annotations>
```

If you are only interested in a annotation of events, and a more coarser level of annotation suffices, then use the structure annotation element `event` instead. See section 2.5.6.

Note: time segments were known as "timed events" in FoLiA 0.8 and below. They have been renamed to a more appropriate and more generic name. For backward compatibility, libraries should implement `timedevent` as an alias for `timesegment`, and `timedevent-annotation` as an alias for `timesegment-annotation`.

Time segmentation in a speech context

Status: PROPOSED in v0.9 · **Implementations:** pynlpl

If used in a speech context, all the generic speech attributes become available (See section 2.9.1). This introduces `begintime` and `endtime`, which are different from `begindatetime` and `enddatetime` ! The generic attributes `begintime` and `endtime` are not defined by the set, but specify a time location in HH:MM:SS.MMMM format which may refer to the location in an associated sound file. Sound files are associated using the `src` attribute, which is inherited by all lower elements, so we put it on the sentence here:

```
<s src="ithinkihavetogo.mp3">
 <w xml:id="example.p.1.s.1.w.1"><t>I</t></w>
 <w xml:id="example.p.1.s.1.w.2"><t>think</t></w>
 <w xml:id="example.p.1.s.1.w.3"><t>I</t></w>
 <w xml:id="example.p.1.s.1.w.4"><t>have</t></w>
 <w xml:id="example.p.1.s.1.w.5"><t>to</t></w>
 <w xml:id="example.p.1.s.1.w.6"><t>go</t></w>
 <w xml:id="example.p.1.s.1.w.7"><t>.</t></w>
 <timing>
  <timesegment begintime="00:00:00.0000"
   endtime="00:00:00.2500">
    <wref id="example.p.1.s.1.w.1" t="I" />
  </timesegment>
  <timesegment begintime="00:00:00.2500"
   endtime="00:00:00.5000">
    <wref id="example.p.1.s.1.w.2" t="think" />
  </timesegment>
  <timesegment begintime="00:00:00.5000"
   endtime="00:00:00.7500">
```

```
    <wref id="example.p.1.s.1.w.3" t="I" />
  </timesegment>
  <timesegment begintime="00:00:00.7500"
   endtime="00:00:01.000">
    <wref id="example.p.1.s.1.w.4" t="have" />
  </timesegment>
  <timesegment begintime="00:00:01.0000"
   endtime="00:00:01.250">
    <wref id="example.p.1.s.1.w.5" t="to" />
  </timesegment>
  <timesegment begintime="00:00:01.2500"
   endtime="00:00:01.5000">
    <wref id="example.p.1.s.1.w.6" t="go" />
  </timesegment>
 </timing>
</s>
```

In a speech context, all structural elements may carry the generic attributes. So the time segmentation in the previous example, though valid, is not the most intuitive way of accomplishing this. Instead, time segmentation can better be used when actual classes are assigned:

```
<s src="ithinkihavetogo.mp3">
 <w xml:id="example.p.1.s.1.w.1"
  begintime="00:00:00.0000" endtime="00:00:00.2500"><t>I</t></w>
 <w xml:id="example.p.1.s.1.w.2"
  begintime="00:00:00.2500" endtime="00:00:00.5000"><t>think</t></w>
 <w xml:id="example.p.1.s.1.w.3"
  begintime="00:00:00.5000" endtime="00:00:00.7500"><t>I</t></w>
 <w xml:id="example.p.1.s.1.w.4"
  begintime="00:00:00.7500" endtime="00:00:01.0000"><t>have</t></w>
 <w xml:id="example.p.1.s.1.w.5"
  begintime="00:00:01.0000" endtime="00:00:01.2500"><t>to</t></w>
 <w xml:id="example.p.1.s.1.w.6"
  begintime="00:00:01.2500" endtime="00:00:01.5000"><t>go</t></w>
 <w xml:id="example.p.1.s.1.w.7"><t>.</t></w>
 <timing>
        <timesegment class="emphasised">
                <wref id="example.p.1.s.1.w.3" />
                <wref id="example.p.1.s.1.w.4" />
                <wref id="example.p.1.s.1.w.5" />
                <wref id="example.p.1.s.1.w.6" />
        </timesegment>
 </timing>
</s>
```

This usage, and the freedom FoLiA sets offer, opens up possibilities for a wide variety of time-segmented annotations. Moreover, the wref element does not necessarily point at words, but it may also point at phonemes. This will be introduced in section 2.9.3.

2.7.6 Semantic roles

Status: new in 0.9 · **Implementations:** pynlpl

Semantic roles, or thematic roles, are implemented in FoLiA using the span-annotation element `semrole`, within the annotation layer `semroles`, usually embedded at sentence level.

```
<s xml:id="example.p.1.s.1">
  <t>The Dalai Lama greeted him.</t>
  <w xml:id="example.p.1.s.1.w.1"><t>The</t></w>
  <w xml:id="example.p.1.s.1.w.2"><t>Dalai</t></w>
  <w xml:id="example.p.1.s.1.w.3"><t>Lama</t></w>
  <w xml:id="example.p.1.s.1.w.4"><t>greeted</t></w>
  <w xml:id="example.p.1.s.1.w.5"><t>him</t></w>
  <w xml:id="example.p.1.s.1.w.6"><t>.</t></w>
  <semroles>
        <semrole class="agent">
                <wref id="example.p.1.s.1.w.2" />
                <wref id="example.p.1.s.1.w.3" />
        </semrole>
        <semrole class="patient">
                <wref id="example.p.1.s.1.w.5" />
        </semrole>
  </semroles>
</s>
```

Semantic roles commonly correspond with syntactic units. Links between the two can be expressed using FoLiA's facility for alignments (see also section 2.10.8), which were already seen in dependency relations as well. The `aref` element may be used from within the `semrole` element to link to a syntactic unit, or anything else for that matter. The following example illustrates this:

```
<s xml:id="example.p.1.s.1">
  <t>De man begroette hem.</t>
  <w xml:id="example.p.1.s.1.w.1"><t>De</t></w>
  <w xml:id="example.p.1.s.1.w.2"><t>man</t></w>
  <w xml:id="example.p.1.s.1.w.3"><t>begroette</t></w>
  <w xml:id="example.p.1.s.1.w.4"><t>hem</t></w>
  <w xml:id="example.p.1.s.1.w.5"><t>.</t></w>
  <syntax>
    <su xml:id="example.p.1.s.1.su.1" class="top">
```

```
        <su xml:id="example.p.1.s.1.su.1_1" class="smain">
            <su xml:id="example.p.1.s.1.su.1_1_1" class="np">
                <su xml:id="example.p.1.s.1.su.1_1_1_1" class="top">
                    <wref id="example.p.1.s.1.w.1" t="De" />
                </su>
                <su xml:id="example.p.1.s.1.su.1_1_1_2" class="top">
                    <wref id="example.p.1.s.1.w.2" t="man" />
                </su>
            </su>
            <su xml:id="example.p.1.s.1.su.1_1_2" class="verb">
                <wref id="example.p.1.s.1.w.3" t="begroette" />
            </su>
            <su xml:id="example.p.1.s.1.su.1_1_3" class="pron">
                <wref id="example.p.1.s.1.w.4" t="hem" />
            </su>
        </su>
        <su xml:id="example.p.1.s.1.su.1_2" class="punct">
            <wref id="example.p.1.s.1.w.5" t="." />
        </su>
    </su>
  </syntax>
  <semroles>
        <semrole class="agent">
                <wref id="example.p.1.s.1.w.1" />
                <wref id="example.p.1.s.1.w.2" />
                <aref id="example.p.1.s.1.su.1_1_1" type="su">
        </semrole>
        <semrole class="patient">
                <wref id="example.p.1.s.1.w.4" />
                <aref id="example.p.1.s.1.su.1_1_3" type="su">
        </semrole>
  </semroles>
</s>
```

The hd element can optionally be used to mark the head of a semantic role:

```
        <semrole class="agent">
                <wref id="example.p.1.s.1.w.1" t="de" />
                <hd>
                        <wref id="example.p.1.s.1.w.2" t="man" />
                </hd>
                <aref id="example.p.1.s.1.su.1_1_2" type="su">
        </semrole>
```

The mandatory declaration for semantic role annotation is as follows:

```
<annotations>
    <semrole-annotation set="http://ilk.uvt.nl/folia/sets/semroles" />
</annotations>
```

2.7.7 Coreference Relations

Status: new in 0.9 · **Implementations:** pynlpl

Relations between words that refer to the same referent are expressed in FoLiA using the `coreferencechain` and `coreferencelink` span-annotation elements, within the `coreferences` annotation layer. This is done by specifying the entire chain in which all links are coreferent. The head of a coreference may optionally be marked with the `hd` element. This annotation layer itself may be embedded on whatever level is preferred, the following example uses paragraph level, but you can for instance also embed it at sentence level or a global text level:

```
<p xml:id="example.p.1">
<s xml:id="example.p.1.s.1">
  <t>The Dalai Lama greeted him.</t>
  <w xml:id="example.p.1.s.1.w.1"><t>The</t></w>
  <w xml:id="example.p.1.s.1.w.2"><t>Dalai</t></w>
  <w xml:id="example.p.1.s.1.w.3"><t>Lama</t></w>
  <w xml:id="example.p.1.s.1.w.4"><t>greeted</t></w>
  <w xml:id="example.p.1.s.1.w.5"><t>him</t></w>
  <w xml:id="example.p.1.s.1.w.6"><t>.</t></w>
</s>
<s xml:id="example.p.1.s.2">
  <t>He was happy to see him.</t>
  <w xml:id="example.p.1.s.2.w.1"><t>He</t></w>
  <w xml:id="example.p.1.s.2.w.2"><t>was</t></w>
  <w xml:id="example.p.1.s.2.w.3"><t>happy</t></w>
  <w xml:id="example.p.1.s.2.w.4"><t>to</t></w>
  <w xml:id="example.p.1.s.2.w.4"><t>see</t></w>
  <w xml:id="example.p.1.s.2.w.5"><t>him</t></w>
  <w xml:id="example.p.1.s.2.w.6"><t>.</t></w>
</s>
<s xml:id="example.p.1.s.3">
  <t>He was happy to see him.</t>
  <w xml:id="example.p.1.s.3.w.1"><t>He</t></w>
  <w xml:id="example.p.1.s.3.w.2"><t>smiled</t></w>
  <w xml:id="example.p.1.s.3.w.3"><t>.</t></w>
</s>
<coreferences>
        <coreferencechain class="ident">
                <coreferencelink>
                        <wref id="example.p.1.s.1.w.1" t="The" />
                        <hd>
                                <wref id="example.p.1.s.1.w.2" t="Dalai" />
                                <wref id="example.p.1.s.1.w.3" t="Lama" />
                        </hd>
                </coreferencelink>
```

```
                <coreferencelink>
                        <wref id="example.p.1.s.2.w.1" t="he" />
                </coreferencelink>
        </coreferencechain>
</coreferences>
</p>
```

Being a span-annotation element, the `coreference` element may take all of the usual attributes, most notable is the `class` element designating the type of coreference relation. Like its parent, each of the links in the chain may take the standard attributes `annotator`, `annotatortype`, `datetime`, `confidence`. The links or heads do not take a class, only `coreferencechain` does.

`Coreferencelink` may take three attributes, which are actually predefined FoLiA subsets (See Section 2.10.4), their values depend on the set used and are thus user-definable and never predefined:

- `modality` - A subset that can be used for indication that there is modality or negation in this coreference link.
- `time` - A subset used to indicate a time dependency. An example of a time dependency is seen in the sentence: “Bert De Graeve, until recently CEO, will now take up a position as CFO”. Here “Bert De Graeve”, “CEO” and “CFO” would all be part of the same coreference chain, and the second `coreferencelink` (“CEO”) can be marked as being in the past using the “time” attribute.
- `level` - A subset used that can indicate the level on which the coreference holds. A possible value suggestion could be “sense”, indicating that only on sense-level there is a corefence relation, as opposed to an actual reference.

The declaration for coreference relations is done as follows:

```
<annotations>
    <coreference-annotation set="http://ilk.uvt.nl/folia/sets/coref" />
</annotations>
```

2.8 Morphological Annotation

Status: heavily revised since v0.9 · **Implementations:** pynlpl

Tokens can be further segmented into morphemes, a form of structure annotation. Morphemes behave much like `w` elements (tokens). Moreover, morphemes can be referred to from within in span annotation using `wref`, allowing spans to be defined not only over whole words/tokens but also parts thereof. The element for morphemes is `morpheme`, and can only occur within `w` elements. Recall that `t` elements can contain references to higher-level `t` elements. In such cases, the `offset` attribute is used to designate the offset index in the word's associated text element (`t`) (zero being right at the start of the text). Morphemes may do this.

Furthermore, a morpheme may take a class, referring to its type. As always, the classes are defined by the declared set, and not predefined by the FoLiA set. Declaration proceeds as follows, the actual set is fictitious:

```
<annotations>
    <morphology-annotation set="http://ilk.uvt.nl/folia/sets/morphology" />
</annotations>
```

Morphemes are grouped in a `morphology` layer, which itself takes no attributes. An example of morphology in use:

```
<w xml:id="example.p.4.s.2.w.4">
    <t>leest</t>
    <lemma class="lezen" />
    <morphology>
        <morpheme class="stem" function="lexical">
            <t offset="0">lees</t>
        </morpheme>
        <morpheme class="suffix" function="inflexional">
            <t offset="4">t</t>
        </morpheme>
    </morphology>
</w>
```

Note that the attribute `function` is a predefined feature you may use (not mandatory), its values are defined by the set rather than the FoLiA standard, so they are user/set-defined.

Morphemes allow token annotation just as words do. We can for instance bind lemma annotation to the morpheme representing the word's stem rather than only to the entire word:

```
<w xml:id="example.p.4.s.2.w.4">
    <t>leest</t>
    <lemma class="lezen" />
    <morphology>
```

```
        <morpheme xml:id="example.p.4.s.2.w.4.m.1" class="stem"
         function="lexical">
                    <lemma class="lezen" />
            <t offset="0">lees</t>
        </morpheme>
        <morpheme xml:id="example.p.4.s.2.w.4.m.2" class="suffix"
         function="inflexional">
            <t offset="4">t</t>
        </morpheme>
    </morphology>
</w>
```

Similarly, consider the Spanish word or phrase “Dámelo” (give it to me), written as one entity. If this has not been split during tokenisation, but left as a single token, you can annotate its morphemes, as all morphemes allow token annotation to be placed within its scope:

```
<w xml:id="example.p.1.s.1.w.1">
    <t>dámelo</t>
    <morphology>
        <morpheme class="stem">
            <t offset="0">dá</t>
                    <lemma class="dar" />
                    <pos class="v" />
        </morpheme>
        <morpheme class="suffix">
            <t offset="2">me</t>
                    <lemma class="me" />
                    <pos class="pron" />
        </morpheme>
        <morpheme class="suffix">
            <t offset="4">lo</t>
                    <lemma class="lo" />
                    <pos class="pron" />
        </morpheme>
    </morphology>
</w>
```

Unlike words, morphemes may also be nested, as they can be expressed on multiple levels:

```
<w xml:id="example.p.1.s.1.w.1">
    <t>comfortable</t>
    <morphology>
        <morpheme class="base">
            <t offset="0">comfort</t>
                    <morpheme class="prefix">
                        <t offset="0">com</t>
```

```
                    </morpheme>
                    <morpheme class="morph">
                        <t offset="3">fort</t>
                    </morpheme>
        </morpheme>
        <morpheme class="suffix">
            <t offset="7">able</t>
        </morpheme>
    </morphology>
</w>
```

Note that the annotation of morphology has changed since FoLiA version 0.9. Older versions did not yet assign a class to morphemes themselves, but rather only used features, which were entirely left to the set to define. These documents remain valid in FoLiA 0.9 and above, but this way is no longer the recommended way. The following example illustrates the old style:

```
<w xml:id="example.p.4.s.2.w.4">
    <t>leest</t>
    <lemma class="lezen" />
    <morphology>
        <morpheme>
            <feat subset="type" class="stem">
            <feat subset="function" class="lexical">
            <t offset="0">lees</t>
        </morpheme>
        <morpheme>
            <feat subset="type" class="suffix">
            <feat subset="function" class="inflexional">
            <t offset="4">t</t>
        </morpheme>
    </morphology>
</w>
```

The next example will illustrate how morphemes can be referred to in span annotation. Here we have a morpheme, and not the entire word, which forms a named entity:

```
<w xml:id="example.p.4.s.2.w.4">
    <t>CDA-voorzitter</t>
    <morphemes>
                <morpheme xml:id="example.p.4.s.2.w.1.m.1">
                    <t offset="0">CDA</t>
                </morpheme>
        </morphemes>
        <entities>
                <entity xml:id="entity.1">
                    <wref id="example.p.4.s.2.w.1.m.1" />
```

```
            </entity>
        </entities>
</w>
```

The older FoLiA elements `subentities` and `subentity` are deprecated in favour of this new approach.

The same approach can be followed for other kinds of span annotation. Note that the span annotation layer (`entities` in the example) may be embedded on various levels. Most commonly on sentence level, but also on word level, paragraph level or the global text level.

2.9 Speech Annotation

Status: Proposed in v0.9 · **Implementations:** not implemented yet

2.9.1 Speech Structure Annotation

Status: Proposed in v0.9 · **Implementations:** not implemented yet

FoLiA is not just suited for the annotation of text, but also accommodates annotation of transcribed speech. This generally asks for a different document structure than text documents. The top-level element for speech-centred resources is `speech`. Certain elements described in the section on text structure may be used under `speech` as well; such as divisions (`div`), sentences (`s`) and words (`w`). Notions such as paragraphs and figures make less sense in a speech context.

All structure elements in a speech context may take the extra FoLiA attributes for speech, as laid out in section 2.3.1. These include attributes for referring to associating sound clips.

Utterances

Status: Proposed in v0.9 · **Implementations:** not implemented yet

An utterance may consist out of words or sentences, which in turn may contain words. The opposite is also true, a sentence may consists out of multiple utterances. The utterance element in FoLiA is `utt`.

Utterances need not be declared necessarily, but may be if classes are assigned. The actual set in the example is fictitious:

```
<annotations>
    <utterance-annotation set="http://ilk.uvt.nl/folia/sets/utterances" />
</annotations>
```

An actual example of utterances is shown later in the section on phonetic content.

Non-speech events

Non-speech events are simply covered by `event` annotation as seen in section 2.5.6. Consider the following small example, with speech-context attributes associated:

```
<event class="cough" src="soundclip.mp3" begintime="..." endtime="..." />
```

2.9.2 Phonetic Content

Status: Proposed in v0.9 · **Implementations:** not implemented yet

Written text is always contained in the text content element (`t`), for phonetics there is a similar counterpart: `ph`. The `ph` element holds a phonetic transcription. It is used in a very similar fashion:

```
<utt src="helloworld.mp3"  begintime="..." endtime="...">
        <ph>hel'oʊ wɔ:ld</ph>
        <w xml:id="example.utt.1.w.1" begintime="..." endtime="...">
                <ph>hel'oʊ</ph>
        </w>
        <w xml:id="example.utt.1.w.2"  begintime="..." endtime="...">
                <ph>wɔ:ld</ph>
        </w>
</utt>
```

Like the `t` element, the `ph` element supports the `offset` attribute, referring to the offset in the phonetic transcription. The first index being zero. Phonetic transcription and text content can also go together without problem:

```
<utt>
        <ph>hel'oʊ wɔ:ld</ph>
        <t>hello world</t>
        <w xml:id="example.utt.1.w.1">
                <ph offset="0">hel'oʊ</ph>
                <t offset="0">hello</t>
        </w>
        <w xml:id="example.utt.1.w.2">
                <ph offset="8">wɔ:ld</ph>
                <t offset="6">world</t>
        </w>
</utt>
```

2.9.3 Phoneme Annotation

Status: PROPOSED in v0.9 · **Implementations:** not implemented yet

The smallest unit of annotatable speech is the phoneme level. The `phoneme` element is a form of structure annotation used for phonemes. Alike to morphology, it is embedded within a layer `phonetics` which can be used within word/token elements (`w`) or directly within `utt` if no words are distinguished:

```
<utt>
        <w xml:id="word" src="book.wav">
                <t>book</t>
                <ph>bʊk</ph>
                <phonetics>
                        <phoneme begintime="..."  endtime="...">
                                <ph>b</ph>
                        </phoneme>
                        <phoneme begintime="..." endtime="...">
                                <ph>ʊ</ph>
                        </phoneme>
                        <phoneme begintime="..." endtime="...">
                                <ph>k</ph>
                        </phoneme>
                </phonetics>
        </w>
</utt>
```

Phoneme annotation needs to be declared if classes are assigned to the phonemes:

```
<annotations>
    <phoneme-annotation set="http://ilk.uvt.nl/folia/sets/utterances" />
</annotations>
```

2.9.4 Distortion

Status: Proposed in v0.9 · **Implementations:** not implemented yet

FoLiA has a token annotation element `distortion` which can be used in a speech context. It indicates that a certain distortion of change in the sound speech has taken place. It can be used for background sounds. The classes are of course not predefined by the FoLiA format but depend on the class used:

```
<utt>
        <ph>hel'oʊ wɔːld</ph>
        <distortion class="windnoise" />
</utt>
```

The distortion element is also useful to mark specific accents or dialects, depending of course on the set used:

```
<utt>
        <t>day</t>
        <ph>dæi</ph>
        <distortion class="cockney" />
</utt>
```

The mandatory declaration goes as follows, the set is fictitious:

```
<annotations>
    <distortion-annotation set="http://ilk.uvt.nl/folia/sets/distortion" />
</annotations>
```

2.10 Higher-order Annotation

We introduced the FoLiA paradigm in section 2.3 and listed the four categories of annotation: structure annotation, token annotation, span annotation and higher-order annotation. In this section we will discuss the higher-order annotation elements and the more advanced aspects of FoLiA. The higher-order annotation category forms less of a unity than the other categories. All annotations in this category have in common that they all are annotations about other annotations, relating to other annotations, or enhancing other annotations.

In our discussion of the various types of higher-order annotation, we will encounter the more advanced aspects of the FoLiA paradigm.

2.10.1 Human readable descriptions

Status: final since v0.6 · **Implementations:** pynlpl,libfolia

This is one of the simplest forms of higher-order annotation. Any annotation element may hold a `desc` element containing in its body a human readable description for the annotation. An example of this has been already shown for the `sense` and `gap` elements

```
<w xml:id="example.p.1.s.1.w.1">
        <t>boot</t>
        <pos class="n">
                <desc>Noun</desc>
        </pos>
        <desc>boot</desc>
</w>
```

2.10.2 Alternative Token Annotations

Status: final · **Implementations:** pynlpl,libfolia

The FoLiA format does not just allow for a single authoritative annotation per token; it allows the representation of *alternative* annotations. Alternative token annotations are grouped within one or more `alt` elements. If multiple annotations are grouped together under the same `alt` element, then they are deemed *dependent* and form a single set of alternatives.

Each alternative preferably is given a unique identifier. In the following example we see the Dutch word "bank" in the sense of a sofa, alternatively we see two alternative annotations with a different sense and domain. Any annotation element within an *alt* block by definition needs to be marked as non-authoritative by setting `auth="no"`. This facilitates the job of parsers and queriers.

```
<w xml:id="example.p.1.s.1.w.1">
    <t>bank</t>
    <domain class="furniture" />
    <sense class="r_n-5918" synset="d_n-21410"
     annotator="John␣Doe" annotatortype="manual"
     confidence="1.0">zitmeubel</sense>
    <alt xml:id="example.p.1.s.1.w.1.alt.1">
        <domain auth="no" class="finance" />
        <sense auth="no" class="r_n-5919" synset="d_n-27025"
         annotator="Jane␣Doe" annotatortype="manual"
```

```
        confidence="0.6">geldverlenende instelling</sense>
    </alt>
    <alt xml:id="example.p.1.s.1.w.1.alt.2">
        <domain auth="no" class="geology" />
        <sense auth="no" class="r_n-5920" synset="d_n-38257"
         annotator="Jim Doe" annotatortype="manual"
         confidence="0.1">zandbank</sense>
    </alt>
</w>
```

Sometimes, an alternative is concerned only with a portion of the annotations. By default, annotations not mentioned are applicable to the alternative as well, unless the alternative is set as being *exclusive*. Consider the following expanded example in which we added a part of speech tag and a lemma.

```
<w xml:id="example.p.1.s.1.w.1">
    <t>bank</t>
    <domain class="furniture" />
    <sense class="r_n-5918" synset="d_n-21410"
     annotator="John Doe" annotatortype="manual"
     confidence="1.0">furniture</sense>
    <pos class="n" />
    <lemma class="bank" />
    <alt xml:id="example.p.1.s.1.w.1.alt.1">
        <domain auth="no" class="finance" />
        <sense auth="no" class="r_n-5919" synset="d_n-27025"
         annotator="Jane Doe" annotatortype="manual"
         confidence="0.6">financial institution</sense>
    </alt>
    <alt xml:id="example.p.1.s.1.w.1.alt.2">
        <domain auth="no" class="geology" />
        <sense auth="no" class="r_n-5920" synset="d_n-38257"
         annotator="Jim Doe" annotatortype="manual"
         confidence="0.1">river bank</sense>
    </alt>
    <alt xml:id="example.p.1.s.1.w.1.alt.2" exclusive="yes">
                <t>bank</t>
        <domain auth="no" class="navigation" />
        <sense auth="no" class="r_n-1234">to turn</sense>
                <pos class="v" />
            <lemma class="bank" />
    </alt>
</w>
```

The first two alternatives are inclusive, which is the default. This means that the pos tag "n" and the lemma "bank" apply to them as well. The last alternative is set as exclusive, using the `exclusive` attribute. It has been given a different pos tag and the lemma and even te text content have been repeated even though they are equal to the higher-level annotation, otherwise there would be no lemma nor text associated with the exclusive alternative.

Alternatives can be used as a great way of postponing actual annotation, due to their non-authoritative nature. When used in this way, they can be regarded as "options". They can be used even when there are no authoritative annotations of the type. Consider the following example in which domain and sense annotations are presented as alternatives and there is no authoritative annotations of these types whatsoever:

```
<w xml:id="example.p.1.s.1.w.1">
    <t>bank</t>
    <alt xml:id="example.p.1.s.1.w.1.alt.1">
        <domain auth="no" class="finance" />
        <sense auth="no" class="r_n-5919" synset="d_n-27025"
         annotator="Jane Doe" annotatortype="manual"
         confidence="0.6">geldverlenende instelling</sense>
    </alt>
    <alt xml:id="example.p.1.s.1.w.1.alt.2">
        <domain auth="no" class="geology" />
        <sense auth="no" class="r_n-5920" synset="d_n-38257"
         annotator="Jim Doe" annotatortype="manual"
         confidence="0.1">zandbank</sense>
    </alt>
</w>
```

2.10.3 Alternative Span Annotations

With token annotations one can specify an unbounded number of alternative annotations. This functionality is available for span annotations as well, but due to the different nature of span annotations this happens in a slightly different way.

Where we used `alt` for token annotations, we now use `altlayers` for span annotations. Under this element several alternative layers can be presented. Analogous to `alt`, any layers grouped together are assumed to be somehow dependent. Multiple `altlayers` can be added to introduce independent alternatives. Each alternative may be associated with a unique identifier. The layers withing `altlayers` need to be marked as non-autoritative using `auth="no"`.

Below is an example of a sentence that is chunked in two ways:

```xml
<s xml:id="example.p.1.s.1">
  <t>The Dalai Lama greeted him.</t>
  <w xml:id="example.p.1.s.1.w.1"><t>The</t></w>
  <w xml:id="example.p.1.s.1.w.2"><t>Dalai</t></w>
  <w xml:id="example.p.1.s.1.w.3"><t>Lama</t></w>
  <w xml:id="example.p.1.s.1.w.4"><t>greeted</t></w>
  <w xml:id="example.p.1.s.1.w.5"><t>him</t></w>
  <w xml:id="example.p.1.s.1.w.6"><t>.</t></w>
  <chunking>
    <chunk xml:id="example.p.1.s.1.chunk.1">
        <wref id="example.p.1.s.1.w.1" t="The" />
        <wref id="example.p.1.s.1.w.2" t="Dalai" />
        <wref id="example.p.1.s.1.w.3" t="Lama" />
    </chunk>
    <chunk xml:id="example.p.1.s.1.chunk.2">
        <wref id="example.p.1.s.1.w.4" t="greeted" />
    </chunk>
    <chunk xml:id="example.p.1.s.1.chunk.3">
        <wref id="example.p.1.s.1.w.5" t="him" />
        <wref id="example.p.1.s.1.w.6" t="." />
    </chunk>
  </chunking>
  <altlayers xml:id="example.p.1.s.1.alt.1">
       <chunking annotator="John Doe"
         annotatortype="manual" confidence="0.0001" auth="no">
        <chunk xml:id="example.p.1.s.1.alt.1.chunk.1">
            <wref id="example.p.1.s.1.w.1" t="The" />
            <wref id="example.p.1.s.1.w.2" t="Dalai" />
        </chunk>
        <chunk xml:id="example.p.1.s.1.alt.1.chunk.2">
            <wref id="example.p.1.s.1.w.2" t="Lama" />
            <wref id="example.p.1.s.1.w.4" t="greeted" />
        </chunk>
        <chunk xml:id="example.p.1.s.1.alt.1.chunk.3">
            <wref id="example.p.1.s.1.w.5" t="him" />
            <wref id="example.p.1.s.1.w.6" t="." />
        </chunk>
      </chunking>
  </altlayers>
</s>
```

The support for alternatives and the fact that multiple layers (including those of different types) cannot be nested in a single inline structure, should make clear why FoLiA uses a stand-off notation alongside an inline notation.

2.10.4 Feature Annotation

Status: revised in v0.8 · **Implementations:** pynlpl,libfolia

In addition to a main class, an arbitrary number of *features* can be added to *any* annotation element. Each feature pertains to a specific *subset*. Subsets and the classes within them can be invented at will as they are part of the set definition, which is left entirely to the user. However, certain annotation elements also have some predefined subsets you may use.

The element `feat` is used to add features to any kind of annotation. In the following example we make use of a subset we invented which ties a lemma to a page number in some dictionary where the lemma can be found.

```
<lemma class="house">
  <feat subset="dictionary_page" class="45" />
</lemma>
```

A more thorough example for Part-of-Speech tags with features will be explained in section 2.10.5.

Some elements have predefined subsets because some features are very commonly used. It however still depends on the set on whether these can be used, and which values these take. Whenever subsets are predefined in the FoLiA standard they can be assigned using XML attributes. Consider the following example of lexical semantic sense annotation, in which subset "synset" is a predefined subset:

```
<sense class="X" synset="Y" />
```

This is semantically equivalent to:

```
<sense class="X">
    <feat subset="synset" class="Y" />
</sense>
```

The following example of event annotation with the feature with predefined subset "actor" is similar:

```
<event class="tweet" actor="John␣Doe">
  ...
</event>
```

This is semantically equivalent to:

```
<event class="tweet">
```

```
 <feat subset="actor" class="John␣Doe" />
  ...
</event>
```

Features can also be used to assign multiple classes within the same subset, which is impossible with main classes. In the following example the event is associated with a list of two actors. In this case the XML attribute shortcut no longer suffices, and the feat element must be used explicitly.

```
<event class="conversation">
 <feat subset="actor" class="John␣Doe" />
 <feat subset="actor" class="Jane␣Doe" />
 <p>...</p>
</event>
```

To recap: the feat element can always be used to freely to associate any additional classes of *any* designed subset with *any* annotation element. For certain elements, there are predefined subsets, in which case you can assign them using the XML attribute shortcut. This however only applies to the predefined subsets.

2.10.5 Part-of-Speech tags with features

Status: final · **Implementations:** pynlpl,libfolia

Part-of-speech tags are a good example of the scenario outlined above. Part-of-speech tags may consist of multiple features, which in turn *may* be associated with specific subsets. There are two scenarios envisionable, one in which the class of the pos element combines all features, and one in which it is the foundation upon which is expanded. Which one is used is entirely up to the defined set.

Option one:

```
<w xml:id="example.p.1.s.1.w.2">
    <t>boot</t>
    <pos head="N" class="N(soort,ev,basis,zijd,stan)">
        <desc>Noun, singular, neuter</desc>
        <feat subset="ntype" class="soort" />
        <feat subset="number" class="ev" />
        <feat subset="degree" class="basis" />
        <feat subset="gender" class="zijd" />
        <feat subset="case" class="stan" />
    </pos>
</w>
```

In FoLiA, this attribute head is a predefined subset "head" of whatever set you defined. This would thus be equal to:

```
    <feat subset="head" class="N" />
```

Option two:

```
<w xml:id="example.p.1.s.1.w.2">
    <t>boot</t>
    <pos class="N">
        <desc>Noun, singular, neuter</desc>
        <feat subset="ntype" class="soort" />
        <feat subset="number" class="ev" />
        <feat subset="degree" class="basis" />
        <feat subset="gender" class="zijd" />
        <feat subset="case" class="stan" />
    </pos>
</w>
```

2.10.6 Metrics

Status: final since v0.9 · **Implementations:** pynlpl,libfolia

The metric element allows annotation of some kind of measurement for the element it is a member of. It may be used with any kind of annotation.

Example:

```
<w xml:id="example.p.1.s.1.w.2">
    <t>boot</t>
    <metric class="charlength" value="4" />
    <metric class="frequency" value="0.00232" />
</w>
```

Example:

```
<su class="np"
    <wref id="..." />
    <wref id="..." />
    <metric class="length" value="2" />
</w>
```

The declaration. The actual sets are fictitious examples:

```
<annotations>
    <token-annotation set="http://ilk.uvt.nl/folia/sets/ucto-tokconfig-nl"
```

```
    annotator="ucto" annotatortype="auto" />
  <metric-annotation set="http://ilk.uvt.nl/folia/sets/metrics" />
</annotations>
```

2.10.7 Corrections

Status: final since v0.4 · **Implementations:** pynlpl,libfolia

Corrections, including but not limited to spelling corrections, can be annotated using the `correction` element. The following example shows a spelling correction of the misspelled word "treee" to its corrected form "tree".

```
<w xml:id="example.p.1.s.1.w.1">
    <correction xml:id="TEST-000000001.p.1.s.1.w.1.c.1" class="spelling">
        <new>
            <t>tree</t>
        </new>
        <original auth="no">
            <t>treee</t>
        </original>
    </correction>
</w>
```

The class indicates the kind of correction, according to the set used. The `new` element holds the actual content of the correction. The `original` element holds the content prior to correction. Note that all corrections must carry a unique identifier. In this example, what we are correcting is the actual textual content, the text element (`t`). To facilitate the job of parsers and queriers, the original element has to be marked as being non-authoritative, using `auth="no"`. This states that this element and anything below it is not authoritative, meaning that any text or annotations within do not affect the text or annotations of the structure element (the word in this case) of which it is a part.

Corrections can be nested and we want to retain a full back-log. The following example illustrates the word "treee" that has been first mis-corrected to "three" and subsequently corrected again to "tree":

```
<w xml:id="example.p.1.s.1.w.1">
    <correction xml:id="TEST-000000001.p.1.s.1.w.1.c.2" class="spelling"
      annotator="Jane Doe" annotatortype="manual" confidence="1.0">
        <new>
            <t>tree</t>
        </new>
```

```
        <original auth="no">
            <correction xml:id="TEST-000000001.p.1.s.1.w.1.c.1" class="spelling"
               annotator="John Doe" annotatortype="manual" confidence="0.6">
                <new>
                    <t>three</t>
                </new>
                <original auth="no">
                    <t>treee</t>
                </original>
            </correction>
        </original>
    </correction>
</w>
```

In the examples above what we corrected was the actual textual content (`t`). It is however also possible to correct other annotations: The next example corrects a part-of-speech tag; in such cases, there is no `t` element in the correction, but simply another token annotation element, or group thereof.

```
<w xml:id="example.p.1.s.1.w.1">
    <t>tree</t>
    <correction xml:id="TEST-000000001.p.1.s.1.w.1.c.1">
        <new>
            <pos class="n" />
        </new>
        <original auth="no">
            <pos class="v" />
        </original>
    </correction>

</w>
```

Error detection and correction with suggestions

Status: Revised in v0.8.2, no error attribute · **Implementations:** pynlpl,libfolia

The correction of an error implies the detection of an error. In some cases, detection comes without correction, for instance when the generation of correction suggestions is postponed to a later processing stage. The `errordetection` element is a very simple element that serves this purpose. It signals the existence of errors and is a normal token annotation element:

```
<w xml:id="example.p.1.s.1.w.1">
    <t>treee</t>
    <errordetection class="spelling" annotator="errorlistX" />
```

```
</w>
```

We can also imagine it specifically marking something as *not* being an error, in which case a class could be used that denotes the absence of an error. Note that this class is in no way predefined, but always up to the user and set.

```
<w xml:id="example.p.1.s.1.w.1">
    <t>tree</t>
    <errordetection class="noerror" />
</w>
```

This kind of error detection is very simple and does not provide actual correction nor suggestions for correction. In some cases, it is desirable to record suggestions for correction, but without making the actual correction.

The correction tag can also be used in such situations in which you want to list suggestions for correction, but not yet commit to any single one. You may for example want to postponed this actual selection to another module or human annotator. Recall that the actual correction is always included in the "new" tag, non-committing suggestions are included in the "suggestion" tag. All suggestions may take an ID and may specify an annotator, if no annotator is specified it will be inherited from the `correction` element itself. Suggestions never take sets or classes by themselves, the class and set pertain to the correction as a whole, and apply to all suggestions within. This implies that you will need multiple correction elements if you want to make suggestions of very distinct types. The following example shows two suggestions for correction:

```
<w xml:id="example.p.1.s.1.w.1">
    <t>treee</t>
    <correction xml:id="example.p.1.s.1.w.1.c.1"
       class="spelling" annotator="errorlistX">
        <suggestion confidence="0.8" auth="no">
            <t>tree</t>
        </suggestion>
        <suggestion confidence="0.2" auth="no">
            <t>three</t>
        </suggestion>
    </correction>
</w>
```

In the situation above we have a possible correction with two suggestions, none of which has been selected yet. The actual text remains unmodified so there are no `new` or `original` tags. Note that anything in the scope of a suggestion is by definition non-authoritative and suggestions have to be marked as such using `auth="no"` to facilitate the job of parsers.

When an actual correction is made, the correction element changes. It may still retain the list of suggestions. In the following example, a human annotator named John Doe took one of the suggestions and made the actual correction:

```
<w xml:id="example.p.1.s.1.w.1">
    <correction xml:id="example.p.1.s.1.w.1.c.1"
       class="spelling" annotator="John␣Doe"
        annotatortype="human">
        <new>
            <t>tree</t>
        </new>
        <suggestion annotator="errorlistX" auth="no"
          annotatortype="auto" confidence="0.8">
            <t>tree</t>
        </suggestion>
        <suggestion annotator="errorlistX" auth="no"
          annotatortype="auto" confidence="0.2">
            <t>three</t>
        </suggestion>
        <original auth="no">
            <t>treee</t>
        </original>
    </correction>
</w>
```

Something similar may happen when a correction is made *on the basis of* one or more kinds of error detection, the `correction` element directly embeds the `errordetection` element:

```
<w xml:id="example.p.1.s.1.w.1">
    <correction class="spelling" annotator="John␣Doe">
        <new>
            <t>tree</t>
        </new>
        <original auth="no">
            <t>treee</t>
        </original>
        <errordetection class="spelling"
         annotator="errorlist" annotatortype="auto" />
    </correction>
</w>
```

In the above example, "treee" was detected by an automated error list as being an error, and was corrected to "tree" by human annotator John Doe.

Like everything, corrections and error detection have to be declared, and have to be declared separately. Nothing stops you from pointing them both to the same set however. Suggestions fall under the scope of corrections and need not be declared separately.

```
<annotations>
    <token-annotation set="http://ilk.uvt.nl/folia/sets/ucto-tokconfig-nl"
     annotator="ucto" annotatortype="auto" />
    <errordetection-annotation set="http://ilk.uvt.nl/folia/sets/corrections" />
    <correction-annotation set="http://ilk.uvt.nl/folia/sets/corrections" />
</annotations>
```

Merges, Splits and Swaps

Sometimes, one wants to merge multiple tokens into one single new token, or the other way around; split one token into multiple new ones. The FoLiA format does not allow you to simply create new tokens and reassign identifiers. Identifiers are by definition permanent and should never change, as this would break backward compatibility. So such a change is therefore by definition a correction, and one uses the `correction` tag to merge and split tokens.

We will first demonstrate a merge of two tokens ("on line") into one ("online"), the original tokens are always retained as `w-original` elements. First a peek at the XML prior to merging:

```
<s xml:id="example.p.1.s.1">
    <w xml:id="example.p.1.s.1.w.1">
        <t>on</t>
    </w>
    <w xml:id="example.p.1.s.1.w.2">
        <t>line</t>
    </w>
</s>
```

And after merging:

```
<s xml:id="example.p.1.s.1">
 <correction xml:id="example.p.1.s.1.c.1" class="merge">
    <new>
        <w xml:id="example.p.1.s.1.w.1-2">
            <t>online</t>
        </w>
    </new>
    <original auth="no">
        <w xml:id="example.p.1.s.1.w.1">
            <t>on</t>
        </w>
```

```
        <w xml:id="example.p.1.s.1.w.2">
            <t>line</t>
        </w>
    </original>
 </correction>
</s>
```

Note that the correction element here is a member of the sentence (`s`), rather than the word token (`w`) as in all previous examples. The new identifier denotes the span of the merge, but this is mere convention.

Now we will look at a split, the reverse of the above situation. Prior to splitting, assume we have:

```
<s xml:id="example.p.1.s.1">
 <w xml:id="example.p.1.s.1.w.1">
    <t>online</t>
 </w>
</s>
```

After splitting:

```
<s xml:id="example.p.1.s.1">
 <correction xml:id="example.p.1.s.1.c.1" class="split">
    <new>
        <w xml:id="example.p.1.s.1.w.1_1">
            <t>on</t>
        </w>
        <w xml:id="example.p.1.s.1.w.1_2">
            <t>line</t>
        </w>
    </new>
    <original auth="no">
        <w xml:id="example.p.1.s.1.w.1">
            <t>online</t>
        </w>
    </original>
 </correction>
</s>
```

The same principle as used for merges and splits can also be used for performing "swap" corrections:

```
<s xml:id="example.p.1.s.1">
 <correction xml:id="example.p.1.s.1.c.1" class="split">
    <new>
```

```
        <w xml:id="example.p.1.s.1.w.2">
            <t>on</t>
        </w>
        <w xml:id="example.p.1.s.1.w.1">
            <t>line</t>
        </w>
    </new>
    <original auth="no">
        <w xml:id="example.p.1.s.1.w.1">
            <t>line</t>
        </w>
        <w xml:id="example.p.1.s.1.w.2">
            <t>on</t>
        </w>
    </original>
 </correction>
</s>
```

Note that in such a swap situation, the identifiers of the word tokens will appear out of sequence after correction, due to the principle that identifiers never change once set.

Omissions and Insertions

Omissions are words that are omitted in the original and have to be inserted in correction, insertions are words that are erroneously inserted in the original and have to be removed in coorrection. FoLiA deals with these in a similar way to merges, splits and swaps. For insertions, the `new` element is simply empty. In the following example the word "the" was duplicated and removed in correction:

```
<s xml:id="example.p.1.s.1">
 <w xml:id="example.p.1.s.1.w.1">
    <t>the</t>
 </w>
 <correction xml:id="example.p.1.s.1.c.1" class="duplicate">
    <new>
    </new>
    <original auth="no">
        <w xml:id="example.p.1.s.1.w.2">
            <t>the</t>
        </w>
    </original>
 </correction>
 <w xml:id="example.p.1.s.1.w.3">
    <t>man</t>
```

```
  </w>
</s>
```

For omissions, the original element is empty.

```
<s xml:id="example.p.1.s.1">
  <w xml:id="example.p.1.s.1.w.1">
      <t>the</t>
  </w>
  <correction xml:id="example.p.1.s.1.c.1" class="duplicate">
      <new>
          <w xml:id="example.p.1.s.1.w.1_1">
              <t>old</t>
          </w>
      </new>
      <original auth="no">
      </original>
  </correction>
  <w xml:id="example.p.1.s.1.w.2">
      <t>man</t>
  </w>
</s>
```

2.10.8 Alignments

Status: revised in v0.8 · **Implementations:** not implemented yet

FoLiA provides a facility to align parts of your document with other parts of your document, or even with parts of other FoLiA documents. These are called *alignments* and are implemented using the alignment element. Within this context, the aref element is used to refer to the aligned FoLiA elements.

Consider the two following aligned sentences from excerpts of two *distinct* FoLiA documents in different languages:

```
<s xml:id="example-english.p.1.s.1">
  <t>The Dalai Lama greeted him.</t>
  <alignment class="french-translation" xlink:href="doc-french.xml"
    xlink:type="simple">
     <aref id="doc-french.p.1.s.1" t="Le␣Dalai␣Lama␣le␣saluait." type="s" />
  </alignment>
</s>

<s xml:id="example-french.p.1.s.1">
  <t>Le Dalai Lama le saluait.</t>
```

```
  <alignment class="english-translation" xlink:href="doc-english.xml"
    xlink:type="simple">
      <aref id="doc-english.p.1.s.1" t="The␣Dalai␣Lama␣greeted␣him." type="s" />
  </alignment>
  <alignment class="dutch-translation" xlink:href="doc-dutch.xml"
     xlink:type="simple">
      <aref id="doc-dutch.p.1.s.1" t="De␣Dalai␣Lama␣begroette␣hem." type="s" />
  </alignment>
</s>
```

The `t` attribute to the `aref` element is merely optional and this overhead is added simply to facilitate the job of limited FoLiA parsers and provides a quick reference to the target text for both parsers and human users. The `xlink:href` attribute is used to link to the target document, if any. If the alignment is within the same document then it should be simply omitted. The `type` attribute in `aref` is mandatory and specifies the type of element the alignment points too, i.e. its value is equal to the tagname it points to.

Although the above example has a single alignment reference (`aref`), it is not forbidden to specify multiple references within the `alignment` block. For more complex alignments however, such as word alignments that include many-to-one, one-to-many or many-to-many alignments, the element `complexalignment` may be more suitable, which behaves similarly to a span annotation element. This element groups `alignment` elements together, effectively creating a many-to-many alignment. The following example illustrates an example similar to the one above. All this takes place within the `complexalignments` annotation layer.

```
<s xml:id="example-english.p.1.s.1">
  <t>The Dalai Lama greeted him.</t>
  <w xml:id="example-english.p.1.s.1.w.1"><t>The</t></w>
  <w xml:id="example-english.p.1.s.1.w.2"><t>Dalai</t></w>
  <w xml:id="example-english.p.1.s.1.w.3"><t>Lama</t></w>
  <w xml:id="example-english.p.1.s.1.w.4"><t>greeted</t></w>
  <w xml:id="example-english.p.1.s.1.w.5"><t>him</t></w>
  <w xml:id="example-english.p.1.s.1.w.6"><t>.</t></w>
  <complexalignments>
    <complexalignment>
              <alignment>
          <aref id="example-english.p.1.s.1.w.2" t="Dalai" type="w">
              <aref id="example-english.p.1.s.1.w.3" t="Lama" type="w">
              </alignment>
              <alignment class="french-translation" xlink:href="doc-french.xml"
        xlink:type="simple">
              <aref id="example-french.p.1.s.1.w.2" t="Dalai" type="w">
              <aref id="example-french.p.1.s.1.w.3" t="Lama" type="w">
              </alignment>
```

```
    </complexalignment>
  </complexalignments>
</s>
```

Here aref is used instead of wref, as despite similarities alignments are technically not exactly span annotation elements. You can in fact align anything that can carry an ID, within the same document and across multiple documents. Moreover, the notion of alignments is not limited to just to words, and it can be used for more than specifying translations.

The first alignment element has no xlink reference, and therefore simply refers to the current document. The second alignment element links to the foreign document. This notation is powerful as it allows you to specify a large number of alignments in a concise matter. Consider the next example in which we added german and italian. Effectively specifying what can be perceived as 16 relationships over four different documents:

```
<s xml:id="example-english.p.1.s.1">
  <t>The Dalai Lama greeted him.</t>
  <w xml:id="example-english.p.1.s.1.w.1"><t>The</t></w>
  <w xml:id="example-english.p.1.s.1.w.2"><t>Dalai</t></w>
  <w xml:id="example-english.p.1.s.1.w.3"><t>Lama</t></w>
  <w xml:id="example-english.p.1.s.1.w.4"><t>greeted</t></w>
  <w xml:id="example-english.p.1.s.1.w.5"><t>him</t></w>
  <w xml:id="example-english.p.1.s.1.w.6"><t>.</t></w>
  <complexalignments>
    <complexalignment>
            <alignment class="english-translation">
        <aref id="example-english.p.1.s.1.w.2" t="Dalai" type="w">
            <aref id="example-english.p.1.s.1.w.3" t="Lama" type="w">
            </alignment>
            <alignment class="french-translation" xlink:href="doc-french.xml"
      xlink:type="simple">
            <aref id="example-french.p.1.s.1.w.2" t="Dalai" type="w">
            <aref id="example-french.p.1.s.1.w.3" t="Lama" type="w">
            </alignment>
            <alignment class="german-translation" xlink:href="doc-german.xml"
      xlink:type="simple">
            <aref id="example-german.p.1.s.1.w.2" t="Dalai" type="w">
            <aref id="example-german.p.1.s.1.w.3" t="Lama" type="w">
            </alignment>
            <alignment class="italian-translation" xlink:href="doc-italian.xml"
      xlink:type="simple">
            <aref id="example-italian.p.1.s.1.w.2" t="Dalai" type="w">
            <aref id="example-italian.p.1.s.1.w.3" t="Lama" type="w">
            </alignment>
    </complexalignment>
```

```
  </complexalignments>
</s>
```

Now you can even envision a FoLiA document that does not hold actual content, but acts merely as a document containing all alignments between for example different translations of the document. Allowing for all relations to be contained in a single document rather than having to be made explicit in each language version.

The `complexalignment` element itself may also take a set, which is *independent* from the alignment set. They thus also have two separate declarations.

It should also be noted that all *span* annotation elements can directly take `aref` elements, without `alignment` elements, to facilitate alignments of span annotation elements with other span annotation elements. An example of this was already seen in section 2.7.3.

2.10.9 Aligned corrections

Status: PROPOSAL in FoLiA v0.9 · **Implementations:** no

The element `alignedcorrection`, within the annotation layer `alignedcorrections`, is a specific kind of alignment that allows you to specify dependency relations between two or more corrections, or their suggestions. Consider the erroneous dutch sentence "Toen ik naar binnen gingen", which has a concordancy error and could be either "Toen ik naar binnen ging" or "Toen wij naar binnen gingen":

```
<s>
  <w xml:id="example.s.1.w.1"><t>Toen</t></w>
  <correction xml:id="correction.1a" class="persoonsvorm_onderwerp_mismatch">
        <original>
          <w xml:id="example.s.1.w.2"><t>ik</t></w>
    </original>
        <suggestion xml:id="correction.1a.suggestion.A">
          <w xml:id="example.s.1.w.2a"><t>ik</t></w>
        </suggestion>
        <suggestion xml:id="correction.1a.suggestion.B">
          <w xml:id="example.s.1.w.2a"><t>wij</t></w>
        </suggestion>
  </correction>
  <w xml:id="example.s.1.w.3"><t>naar</t></w>
  <w xml:id="example.s.1.w.4"><t>binnen</t></w>
  <correction xml:id="correction.1b" class="persoonsvorm_onderwerp_mismatch" >
```

```
        <original>
          <w xml:id="example.s.1.w.5"><t>gingen</t></w>
        </original>
        <suggestion xml:id="correction.1b.suggestion.A">
          <w xml:id="example.s.1.w.2a"><t>ging</t></w>
        </suggestion>
        <suggestion xml:id="correction.1b.suggestion.B">
          <w xml:id="example.s.1.w.2a"><t>gingen</t></w>
        </suggestion>
  </correction>
  <alignedcorrections>
        <alignedcorrection class="persoonsvorm_onderwerp_mismatch">
                <aref id="correction.1a" type="correction" />
                <aref id="correction.1b" type="correction" />
                <alignedsuggestion>
                        <aref id="correction.1a.suggestion.A" type="suggestion" />
                        <aref id="correction.1b.suggestion.A" type="suggestion" />
                </alignedsuggestion>
                <alignedsuggestion>
                        <aref id="correction.1a.suggestion.B" type="suggestion" />
                        <aref id="correction.1b.suggestion.B" type="suggestion" />
                </alignedsuggestion>
   </alignedcorrection>
  </alignedcorrections
</s>
```

The metacorrection has alignment references the correction elements that form a part of it. It can optionally also include `alignedsuggestion` elements which in turn contain alignment references the parts that form the suggestions.

2.10.10 Text content

Status: final since v0.6 · **Implementations:** pynlpl,libfolia

In section 2.5.1 we have seen the text content element `t`. This element can be associated with structural elements such as `w`, `s`, and `p`. The `offset` attribute may be used to explicitly link the text between child and parent. This is demonstrated on three levels in the following example:

```
<p xml:id="example.p.1">
   <t>Hello. This is a sentence. Bye!</t>
   <s xml:id="example.p.1.s.1">
        <t offset="7">This is a sentence.</t>
        <w xml:id="example.p.1.s.1.w.1"><t offset="0">This</t></w>
        <w xml:id="example.p.1.s.1.w.2"><t offset="5">is</t></w>
```

```
        <w xml:id="example.p.1.s.1.w.3"><t offset="8">a</t></w>
        <w xml:id="example.p.1.s.1.w.4" space="no"><t offset="10">sentence</t></w>
        <w xml:id="example.p.1.s.1.w.5"><t offset="18">.</t></w>
    </s>
</p>
```

Moreover, we have seen the space attribute, which is a simple alternative that can be used to reconstruct the untokenised text if it is not explicitly provided in a parent's `t` element. Allowed values for `space` are:

- "`yes`" or " " (a space) – This is the default and says that the token is followed by a single space.
- "`no`" or "" (empty) – This states that the token is not followed by a space.
- any other character or string – This states that the token is followed by another character or string that acts as a token separator.

When explicit text content on sentence/paragraph level is provided, offsets can be used to refer back to it from deeper text-content elements. This does imply that there are some challenges to solve. First of all, by default, the offset refers to the direct parent of whatever text-supporting element the text content (`t`) is a member of. If a level is missing we have to explicitly specify this reference using the `ref` attribute. Note that there is no text content for the sentence in the following example, and we refer directly to the paragraph's text:

```
<p xml:id="example.p.1">
    <t>Hello. This is a sentence. Bye!</t>
    <s xml:id="example.p.1.s.1">
        <w xml:id="example.p.1.s.1.w.1">
          <t ref="example.p.1" offset="7">This</t>
        </w>
        <w xml:id="example.p.1.s.1.w.2">
          <t ref="example.p.1" offset="12">is</t>
        </w>
        <w xml:id="example.p.1.s.1.w.3">
          <t ref="example.p.1" offset="15">a</t>
        </w>
        <w xml:id="example.p.1.s.1.w.4" space="no">
          <t ref="example.p.1" offset="17">sentence</t>
        </w>
        <w xml:id="example.p.1.s.1.w.5">
          <t ref="example.p.1" offset="25">.</t>
        </w>
```

```
    </s>
</p>
```

Note that text content is always expected to be untokenised, except in `w` tags as it by definition is the tokenisation layer.

It is possible to associate *multiple text-content* elements with the same element, and thus associating multiple texts with the same element. You may wonder what could possibly be the point of such extra complexity. But there is a clear use-case when dealing with for example corrections, or wanting to associate the text version just prior or after a processing step such as Optical Character Recognition or another kind of normalisation.

Corrections are challenging because they can be applied to text content and thus change the text. Corrections are often applied on the token level (within `w` tags), but you may want them propagated to the text content of sentences or paragraphs whilst at the same time wanting to retain the text how it originally was. This can be accomplished by introducing text content of a different class. Text content that has no associated class obtains the "current" class by default, it is expected to always be up-to-date. There is a notable exception: text content that appears within the scope of `original` elements within a `correction` element automatically adopts the "original" class.[2] This thus implies that in this rare case, FoLiA actually pre-defines classes (i.e: "original" and "current")! In addition to these two pre-defined classes, any other custom classes may be added as you see fit. If you add custom classes, you need a declaration, otherwise it may be omitted:

```
<annotations>
    <text-annotation set="http://ilk.uvt.nl/folia/sets/text-annotation" />
</annotations>
```

Below is an example illustrating the usage of multiple classes. To show the flexibility, offsets are added, but these are of course always optional. Note that when an offset is specified, it always refers to a text-content element of the same class!

```
<p xml:id="example.p.1">
    <t>Hello. This is a sentence. Bye!</t>
    <t class="original">Hello. This iz a sentence. Bye!</t>
    <s xml:id="example.p.1.s.1">
        <t offset="7">This is a sentence.</t>
```

[2] For more deeply nested original elements, you will have to assign your own classes if you do not want them to take the "original" class.

```
        <t class="original" offset="7">This is a sentence.</t>
        <w xml:id="example.p.1.s.1.w.1">
         <t offset="0">This</t>
        </w>
        <w xml:id="example.p.1.s.1.w.2">
         <correction>
          <new>
           <t offset="5">is</t>
          </new>
          <original auth="no">
           <t offset="5">iz</t>
           <!-- Note that this element has class 'original' by definition! -->
          </original>
         </correction>
        </w>
        <w xml:id="example.p.1.s.1.w.3">
         <t offset="8">a</t>
        </w>
        <w xml:id="example.p.1.s.1.w.4" space="no">
         <t offset="10">sentence</t>
        </w>
        <w xml:id="example.p.1.s.1.w.5">
         <t offset="48">.</t>
        </w>
    </s>
</p>
```

In the above example, the correction is explicit, in the next example, it is implicit. Furthermore, to illustrate how you could use other custom classes, the next example introduces an custom “ocroutput” class that shows the (fictitious) output of an OCR system prior to some implicit correction stage.

```
<p xml:id="example.p.1">
    <t>Hello. This is a sentence. Bye!</t>
    <t class="original">Hello. This iz a sentence. Bye!</t>
    <t class="ocroutput">Hell0 Th1s iz a sentence, Bye1</t>
    <s xml:id="example.p.1.s.1">
        <t offset="7">This is a sentence.</t>
        <t class="original" offset="7">This is a sentence.</t>
        <t class="ocroutput" offset="6">Th1s iz a sentence,</t>
        <w xml:id="example.p.1.s.1.w.1">
         <t offset="0">This</t>
         <t class="ocroutput" offset="0">Th1s</t>
        </w>
        <w xml:id="example.p.1.s.1.w.2">
           <t offset="5">is</t>
           <t offset="5" class="original">iz</t>
           <t offset="5" class="ocroutput">iz</t>
```

```
        </w>
        <w xml:id="example.p.1.s.1.w.3">
         <t offset="8">a</t>
        </w>
        <w xml:id="example.p.1.s.1.w.4" space="no">
         <t offset="10">sentence</t>
        </w>
        <w xml:id="example.p.1.s.1.w.5">
         <t offset="48">.</t>
         <t offset="48" class="original">.</t>
         <t offset="48" class="ocroutput">,</t>
        </w>
    </s>
</p>
```

Last, an important note regarding offsets; all offset values are measured in unicode code-points, the first character having index zero. Take special care with combining diacritical marks versus codepoints that directly integrate the diacritical mark.

2.11 Metadata

Status: final since v0.4 · **Implementations:** pynlpl,libfolia

FoLiA has support for metadata, most notably the extensive and mandatory declaration section for all used annotations which you have seen throughout this documentation. To complement this, there is FoLiA's native metadata system, in which simple metadata fields can be defined and used at will. FoLiA is also able to operate with IMDI or CMDI metadata, either in external file or stored inline. Note however that storing CMDI or IMDI inside your FoLiA document may cause problems when you want to validate your FoLiA document. It is also incompatible with CMDI or IMDI editors that are unaware of FoLiA.

Reference to CMDI in external file is recommended and proceeds in the following simple fashion:

```
<metadata type="cmdi" src="/path/or/url/to/metadata.cmdi">
 ...
</metadata>
```

The procedure for IMDI is the same:

```
<metadata type="imdi" src="/path/or/url/to/metadata.imdi">
```

```
  ...
</metadata>
```

If you use neither CMDI nor IMDI, then you can use FoLiA's native system, which is very simple: It introduces the `meta` element that allows you to define key value pairs as follows:

```
<metadata type="native">
    <annotations>
     ..
    </annotations>
    <meta id="title">Title to my document</meta>
    <meta id="language">eng</meta>
</metadata>
```

You can simply define fields with custom IDs, but the following fields are predefined and recommended to be filled:

- **title** – The title of the FoLiA document
- **language** – An ISO-639-3 language code identifying the language the document is
- **date** – The date of publication in `YYYY-MM-DD` format
- **publisher** – The publishing institution or individual
- **license** – The type of license of the document (for example: *GNU Free Documentation License*)

Chapter 3

Set Definition Format

Status: PROPOSED since v0.7 · **Implementations:** pynlpl (partly)

3.1 Introduction

The FoLiA format consists not just out of the Document Format discussed in the previous chapter, but also of a Set Definition Format. The document format is agnostic about all sets and the classes therein, it is the Set Definition Format that defines precisely what classes are allowed in a certain set, including any subsets.

Recall from section 2.4 that all used sets need to be declared in the document header and that they point to URLs holding a FoLiA set definition. If no set definition files are associated, then a full in-depth validation can not take place.

3.2 Types and classes

The set definition format is fairly straightforward, each set definition file represents one set, including all of its subsets.

Here is a simple example:

```
<set xml:id="simplepos" type="closed">
```

```
    <class xml:id="N" label="Noun" />
    <class xml:id="V" label="Verb" />
    <class xml:id="A" label="Adjective" />
</set>
```

The ID of the class determines a value the class attribute may take in the FoLiA document, for elements of this set. The label attribute carries a human readable description for presentational purposes, this is optional but highly recommended.

There are three possible types for sets and subsets:

1. **open**: classes may be anything and are not defined
2. **closed**: classes are defined strictly
3. **mixed**: classes may be anything, but some are predefined

A set definition file for an open type set definition may be as concise as:

```
<set xml:id="lemmas-nl" type="open" />
```

3.3 Concept link

You may want to associate classes, or even sets themselves, with some kind of semantic web or category registry. This link can be made using the conceptlink attribute which may be placed on classes, sets and subset elements.

```
<set xml:id="simplepos" type="closed" conceptlink="http://some/host/simplepos">
    <class xml:id="N" label="Noun" conceptlink="http://some/host/noun" />
    <class xml:id="V" label="Verb" conceptlink="http://some/host/verb" />
    <class xml:id="A" label="Adjective" conceptlink="http://some/host/adj" />
</set>
```

FoLiA does not dictate any format requirements for conceptual links, it can be anything, such as an RDF resource, or any other kind. If you want something more specific, or you want to link to multiple semantic resources, simply use your own "conceptlink" attribute in a different custom XML namespace.

3.4 Subsets

Section 2.10.4 introduced subsets. These can be defined in a similar fashion to sets and also carry a `type` attribute:

```
<set xml:id="simplepos" type="closed">
    <class xml:id="N" label="Noun" />
    <class xml:id="V" label="Verb" />
    <class xml:id="A" label="Adjective" />
    <subset xml:id="gender" class="closed">
        <class xml:id="m" label="Masculine" />
        <class xml:id="f" label="Feminine" />
        <class xml:id="n" label="Neuter" />
    </subset>
</set>
```

It is possible for subsets to be used multiple times if the subset is declared with the attribute `multi` set to `true` (defaults to false). This allows multiple classes to be associated with a subset. Subsets can be made mandatory by setting the attribute `required` to `true`.

3.5 Constraints

Not all classes in subsets can be combined with others. Often the need arises to put constraints on which classes can go together. The previous example already illustrates this. For many languages, the "gender" subset does not make sense on verbs. We can put a constraint on the usage of this subset, limiting its usage to nouns and adjectives:

```
<set xml:id="simplepos" type="closed">
    <class xml:id="N" label="Noun" />
    <class xml:id="V" label="Verb" />
    <class xml:id="A" label="Adjective" />
    <subset xml:id="gender" class="closed">
        <constraint>
            <restrict class="N" />
            <restrict class="A" />
        </constraint>
        <class xml:id="m" label="Masculine" />
        <class xml:id="f" label="Feminine" />
        <class xml:id="n" label="Neuter" />
    </subset>
</set>
```

For sake of brevity, constraints can be named and referred to when they are needed multiple times.

```
<set xml:id="simplepos" type="closed">
    <class xml:id="N" label="Noun" />
    <class xml:id="V" label="Verb" />
    <class xml:id="A" label="Adjective" />
    <subset xml:id="gender" class="closed">
        <constraint name="constraint.1">
            <restrict class="N" />
            <restrict class="A" />
        </constraint>
        <class xml:id="m" label="Masculine" />
        <class xml:id="f" label="Feminine" />
        <class xml:id="n" label="Neuter" />
    </subset>
    <subset xml:id="case" class="closed">
        <constraint ref="constraint.1" />
        <class xml:id="nom" label="Nominative" />
        <class xml:id="gen" label="Genitive" />
        <class xml:id="dat" label="Dative" />
        <class xml:id="acc" label="Accusative" />
    </subset>
</set>
```

Constraints can be used within subsets, but also within classes:

```
<set xml:id="simplepos" type="closed">
    <class xml:id="N" label="Noun" />
    <class xml:id="V" label="Verb" />
    <class xml:id="A" label="Adjective" />
    <subset xml:id="gender" class="closed">
        <class xml:id="m" label="Masculine">
            <constraint name="constraint.1">
                <restrict class="N" />
                <restrict class="A" />
            </constraint>
        </class>
    </subset>
</set>
```

Using the restrict element, you force a certain class from the main set or any subset, thus enumerating all the allowed classes. For example, the following constraint demands masculine or feminine nouns in either nominative or accusative case. All of the restrictions must be satisfied for the constraint to match, restrictions on the same subset (or the main set if no subset is specified) are automatically considered as disjunctions.

```
<constraint>
    <restrict class="N" />
    <restrict subset="gender" class="f" />
    <restrict subset="gender" class="m" />
    <restrict subset="case" class="nom" />
    <restrict subset="case" class="acc" />
</constraint>
```

Disjunctions, over different subsets, can be made explicitly using the `disjunction` element. The following would be constrained to feminine nouns *or* plural nouns, rather than feminine plural nouns if the disjunction element were not present:

```
<constraint>
    <restrict class="N" />
        <disjunction>
            <restrict subset="gender" class="f" />
            <restrict subset="number" class="plural" />
        </disjunction>
</constraint>
```

You can also opt to specify the "forbidden" classes using `except`. Only if not a single one of the exceptions applies, the constraint is met.

```
<constraint>
    <except class="V" />
    <except class="A" />
    <except subset="gender" class="n" />
    <except subset="case" class="gen" />
    <except subset="case" class="dat" />
</constraint>
```

Restrict and except elements can also be mixed, in which case the constraint matches if all of the restrictions do, and if none of the exceptions do. Moreover, disjunctions can be nested, to form complex constraints.

The *required* attribute can be used on subsets to indicate whether they are mandatory or optional, but a more powerful mechanism is available using constraints and the `require` element (or its complement: `forbid`). The following example adds a constraint on nouns and requires it to have the gender and number subsets specified, whereas for verbs, tense and number are required.

```
<class="N">
        <constraint>
                <require subset="gender" />
                <require subset="number" />
        </constraint>
</class>
```

```
<class="V">
	<constraint>
		<require subset="tense" />
		<require subset="number" />
	</constraint>
</class>
```

Appendix A

Common Queries

Considering the fact that FoLiA is an XML-based format, XPath and its derivatives are the designated tools for searching in a FoLiA document.

A very common XPath predicate found in many XPath expressions for FoLiA is `not(ancestor-or-self::*/@auth)`. This exploits the notion of authoritativeness. Certain elements in FoLiA are non-authoritative, which means that they have no direct bearing on the actual state of the element they describe. The most notable elements that are non-authoritative are alternatives, suggestions for correction, and the original part of a correction. The predicate `not(ancestor-or-self::*/@auth)` guarantees that no elements can be selected that occur within the scope of any non-authoritative element. This prevents selecting for example alternative annotations or annotations that were superseded by a correction step. This is in most cases what the user wants and why you will find this predicate appended to almost every XPath expression for FoLiA.

Some common XPath queries are listed below, note that for brevity and readability the namespace prefix is omitted. In actual situations you will have to specify the FoLiA namespace with each element, as XPath unfortunately has no notion of a default namespace.

- XPath query for all paragraphs: `//p[not(ancestor-or-self::*/@auth)]`
- XPath query for all sentences: `//s[not(ancestor-or-self::*/@auth) and not(ancestor::quote)]`
  Explanation: When selecting sentences, you often do not want sub-sentences

that are part of a quote, since they may overlap with the larger sentence they form a part of. The `not(ancestor::quote)` predicate guarantees this can not happen.

- XPath query for all words: `//w[not(ancestor-or-self::*/@auth)]`

- XPath query for the text of all words/tokens: `//w//t[not(ancestor-or-self::*/@auth) and not(ancestor-or-self::*/morpheme) and not(@class)]/text()` *Explanation: The `[not(@class)]` predicate is important here and makes sure to select only the text content element in case there are multiple. (See also Section 2.10.10). The `not(ancestor-or-self::*/morpheme` makes sure morphemes are excluded.*

- XPath query for all words with lemma X: `//w[.//lemma[@class="X" and not(ancestor-or-self::*/@auth) and not(ancestor-or-self::*/morpheme] ]` *Note: This query assumes there is only one declaration for lemma annotation, and the set has been verified. It furthermore excludes morphemes.*

- XPath query for all words with PoS-tag A in set S: `//w[.//pos[@set="S" and @class="A" and not(ancestor-or-self::*/@auth)] ].` *Note: This query assumes the set attribute was set explicitly, i.e. there are multiple possible sets in the document for this annotation type. This query does not exclude morphemes.*

- XPath query for the text of all words with PoS-tag A in set S: `//w[.//pos[@set="S" and @class="A" and not(ancestor-or-self::*/@auth)] ]//t[not(ancestor-or-sel and not(@class)]/text()` *Note: The predicate for non-authoritativeness here needs to be applied both to the pos element and the text content element `t`, otherwise you may accidentally select the text of words which have the desired pos tag only as an alternative.*

- XPath query to select all alternative PoS tags for all words: `//w/alt/pos`

When selecting text elements (`t`), you generally want to add `not(@class)` to the constraint, to select only the text content elements that have not been assigned an alternative class. Recall that multiple text content may be present, bearing another class. Omitting this constraint will prevent you from properly retrieving the current text of a document, as it will also retrieve all this differently typed text content.

Before you release XPath queries on FoLiA documents, make sure to first parse the declarations present in the metadata (the `annotations` block). Verify that the annotation type with the desired set you are looking for is actually present, otherwise you need not bother running a query at all. Note that the XPath expression differs based on whether there is only one set defined for the sought annotation type, or if there are multiple. In the former case, you can't use the @set attribute to select, and in the latter case, you must.

Appendix B

Validation

Validation proceeds at two levels: shallow validation and deep validation. Shallow validation considers only the structure of the FoLiA document, without validating the sets and classes used. Deep validation checks the sets and classes for their validity using the set definition files.

Shallow validation is performed using a RelaxNG schema, to be found at `https://github.com/proycon/folia/blob/master/schemas/folia.rng` .

You can validate your document using standard XML tools such as `xmllint` or `jing`, the latter is known to produce friendlier error output in case of validation errors.

```
$ xmllint -relaxng folia.rng document.xml
$ jing folia.rng document.xml
```

Alternatively, you can use the simpler `foliavalidator` tool available in the FoLiA tools (see appendix D).

Development Notes
Deep validation is still being worked on and will most likely use Schematron.

Appendix C

Implementations

The following FoLiA implementations exist currently, both follow a highly object-oriented model in which FoLiA XML elements correspond with classes.

1. `pynlpl.formats.folia` - A FoLiA library in Python. Part of the Python Natural Language Processing Library. Documentation can can be found at http://ilk.uvt.nl/folia/
2. `libfolia` - A FoLiA library in $C++$. (Still under heavily development, September 2011)

Information regarding implementation of certain elements for these two libraries in present in the status boxes throughout this documentation. The following table shows the level of FoLiA support for advanced deatures in these libraries:

	PyNLPl	libfolia	
Programming Language	python	$C++$	
Query facility (findwords)	partial[1]	partial[2]	
IMDI intepretation	partial[3]	no	
RelaxNG schema generation	yes[4]	no	
RelaxNG validation	yes	no	
Set definition support	partial[5]	no	
Deep validation (Schematron)	not yet	no	
D-Coi read compatibility	partial[6]	no	

[1] for token annotation only

[2] for token annotation only

[3] and only for in-document IMDI

[4] The RelaxNG schema is generated dynamically by the library itself

[5] under development

[6] only basic elements, no List, Figure, etc..

Appendix D

FoLiA Tools

D.1 Introduction

A number of command-line tools are readily available for working with FoLiA, to various ends. The following tools are currently available:

- `folia2txt` – Convert FoLiA XML to plain text (pure text, without any annotations)
- `foliaquery` – A query tool that searches FoLiA documents for a specified pattern, or combination of patterns. Supports various token annotations.
- `foliavalidator` – Tests if documents are valid FoLiA XML.
- `folia2dcoi` – Convert FoLiA XML to D-Coi XML (only for annotations supported by D-Coi)
- `dcoi2folia` – Convert D-Coi XML to FoLiA XML
- `folia2html` – Converts a FoLiA document to a semi-interactive HTML document, with limited support for certain token annotations.

All of these tools are written in Python, and thus require a Python (2.6 or higher) installation to run. As FoLiA matures and is adopted more widely, more tools are likely to be added.

D.2 Installation

The FoLiA tools are published to the Python Package Index and can be installed effortlessly using `easy install`, on any Linux/BSD system:

```
$ sudo easy_install folia-tools
```

If `easy install` is not yet available, install it as follows:

On Debian/Ubuntu-based systems:

```
$ sudo apt-get install python-setuptools
```

On RedHat-based systems:

```
$ yum install python-setuptools
```

Alternatively, the FoLiA tools can be obtained from github (`https://github.com/proycon/folia`), and once downloaded and extracted, installed using `sudo ./setup.py install`.

D.3 Usage

To obtain help regarding the usage of any of the available FoLiA tools, please pass the $-h$ option on the command line to the tool you intend to use. This will provide a summary on available options and usage examples. Most of the tools can run on both a single FoLiA document, as well as a whole directory of documents, allowing also for recursion. The tools generally take one or more file names or directory names as parameters.

D.4 FoLiA Query tool

The query tool, `foliaquery`, deserves some extra attention. This can be used out-of-the-box to search one or more FoLiA documents for certain patterns. These patterns can be textual, but can also be patterns of (token) annotation.

Consider a use case in which you have several FoLiA documents in a directory /tmp/foliadocs/. If you are interested in for instance finding all occurrences of the Shakespearean phrase "to be or not to be" in these documents, then this can be accomplished using the following command:

```
$ foliaquery --text="to be or not to be" /tmp/foliadocs/
```

The query tool also allows you to specify this using wildcards of either fixed or variable length, as illustrated respectively in the next example:

```
$ foliaquery --text="to be ^ ^ to be" /tmp/foliadocs/
$ foliaquery --text="to be * to be" /tmp/foliadocs/
```

In addition to querying on mere text, you may query on annotations such as for example part-of-speech tags or lemmata:

```
$ foliaquery --pos="ADJ N" /tmp/foliadocs/
$ foliaquery --pos="ADJ ^ N" /tmp/foliadocs/
$ foliaquery --lemma="be happy" /tmp/foliadocs/
```

Various patterns may be combined, and matches have to satisfy all:

```
$ foliaquery --pos="ADJ N" --lemma="beautiful house" /tmp/foliadocs/
$ foliaquery --pos="ADJ N" --lemma="beautiful ^" /tmp/foliadocs/
```

Disjunctions, on the level of single tokens, may be specified using the pipe character:

```
$ foliaquery --pos="ADJ|DET N" --lemma="the|beautiful house|houses" \
/tmp/foliadocs/
```

As you see, the query tool is a powerful tool for finding desired patterns in your corpus data. The output will invariably consist of the filename in which the occurrence was found, the ID of the first matching token, and the text of the entire match.

Bibliography

[1] Eneko Agirre1, Xabier Artola1, Arantza Diaz de Ilarraza1, German Rigau1, Aitor Soroa1, and Wauter Bosma. Kyoto annotation format, 2009.

[2] Wilko Apperloo. XML basisformaat D-Coi: Voorstel XML formaat presentational markup. Technical report, Polderland Language and Speech Technology, 2006.

[3] Gosse Bouma, Gertjan van Noord, and Rob Malouf. Alpino: Wide-coverage computational analysis of dutch. In Walter Daelemans, Khalil Sima'an, Jorn Veenstra, and Jakub Zavrel, editors, *CLIN*, volume 37 of *Language and Computers - Studies in Practical Linguistics*, pages 45–59. Rodopi, 2000.

[4] Tim Bray, Jean Paoli Paoli, and C.M. Sperberg-McQueen. Extensible markup language (XML) 1.0. Technical report, W3C, 2 1998.

[5] Nancy Ide and Jean Véronis. *The Text Encoding Initiative : Background and Context*. Kluwer Academic Publishers, Dordrecht, 1995.

[6] Nelleke Oostdijk, Martin Reynaert, Paola Monachesi, Gert-Jan Van Noord, Roeland Ordelman, Ineke Schuurman, and Vincent Vandeghinste. From D-Coi to SoNaR: A reference corpus for dutch. In *Proceedings of the Sixth International Language Resources and Evaluation (LREC'08)*, Marrakech, Morocco, 2008.

Appendix E

Comparison

There are numerous annotation formats in the field, so one might wonder why we felt the need to create yet another one. First of all, there are many old and ad-hoc formats in the computational linguistics community, formats that are often specific to one particular application and lack the expressive power for other kind of annotations than initially designed for. Often, such formats are simple text or column-based formats without any kind of formal scheme. FoLiA is intended as a solution for replacing such formats for the storage and exchange of language resources, and as such to facilitate inter-connectivity of NLP tools and sharing of data.

There are also formats with are modern, XML-based and formalised. In the remainder of this appendix we will draw a quick comparison with some notable formats:

E.0.1 TCF v0.4 – Text Corpus Format

The Text Corpus Format[?] is specifically designed as a language resource exchange format for webservices chained in a workflow, which is also one of the intended uses of FoLiA. The format is a modern, XML-based, unicode encoded, and supports a fair subset of the annotation types supported by FoLiA. It is one of the formats closest to FoLiA and therefore most interesting to compare. All annotations in TCF are represented in a stand-off fashion within the same XML-file, although there seems to be a provision for some inline annotations as well. The format is simpler and less verbose than FoLiA, slimness and processing efficiency also explicitly being one of the aims of the authors. This however does compromise expressibility and makes it less expressive than FoLiA. For example, there seems to be no support for corrections, alternatives and the encoding of annotators, time of annotation, and confidence level, something that in FoLiA is deeply integrated into the paradigm. TCF is tagset independent and tagsets can be specified by the user, but it does not generalise or formalise the "set/class" paradigm to the extent FoLiA does. The level of generalisation does not go as far as it does in FoLiA.

In annotation formats there is always a balance between expressibility and efficiency/simplicity. FoLiA is clearly on the more expressive side here, offering more features but therefore making higher demands on memory usage, whereas TCF is more on the efficiency side.

E.0.2 KAF – Kyoto Annotation Format

(Yet to write)

E.0.3 LAF – Linguistic Annotation Framework, and GraF

(Yet to write)

E.0.4 TEI P5 – Text Encoding Initiative

(Yet to write)