Language and Speech Technology
Technical Report Series

Report Number LST-14-01

FoLiA: Format for Linguistic Annotation

version 1.1.0– Revision 4.4

Documentation

Maarten van Gompel

January 2nd, 2014 (published) – April 21st, 2016 (last revision)

PI Group Language and Speech Technology
Centre for Language Studies
Radboud University Nijmegen
P.O. Box 9103
NL-6500 HD Nijmegen
The Netherlands
http://www.ru.nl/lst

Series editors:
Nelleke Oostdijk
Antal van den Bosch
David van Leeuwen
ISSN 2352-3107

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 successorr, the SoNaR corpus [7]. 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 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 accommodate 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[6]. 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 ad-hoc 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 (2014). FoLiA: Format for Linguistic Annotation. Documentation. Language and Speech Technology Technical Report Series 14-01. Radboud University Nijmegen."*. The latest version of the documentation is always available from `http://proycon.github.io/folia`. 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, label sets are user-defined.
- Agnostic with regard to metadata. External metadata schemes such as CMDI [5] 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
- Named Entities / Multi-word units
- Syntactic Parses
- Dependency Relations
- Chunking

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

FoLiA support is incorporated directly into the following software:

- 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 various 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, each contain a number of annotation types.

- **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 structural element, most often a word token (`w`) (hence the name). These annotations appear within of the element they apply to. 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 level that also contains all the element that are being reference. This is often the sentence level, but possibly also higher levels (paragraph/division/text) for certain annotation types. Examples in this category are: the labelling of syntactic constituent structure, syntactic dependencies, chunks, co-reference, 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 marked 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 [8]. The CGN set defines main tag classes such as *WW*, *BW*, *ADJ*, *VZ*. FoLiA itself *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 usually also carries a `class` attribute, which selects a particular class from the set.

In the metadata section of the FoLiA document, sets are *declared*. This means that for each annotation type you specify the set you are going to use, this is again done by means of a URL pointing to a set definition file.

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 Figure 2.1. 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: Final since v0.12 · **Implementations:** pynlpl,libfolia

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.

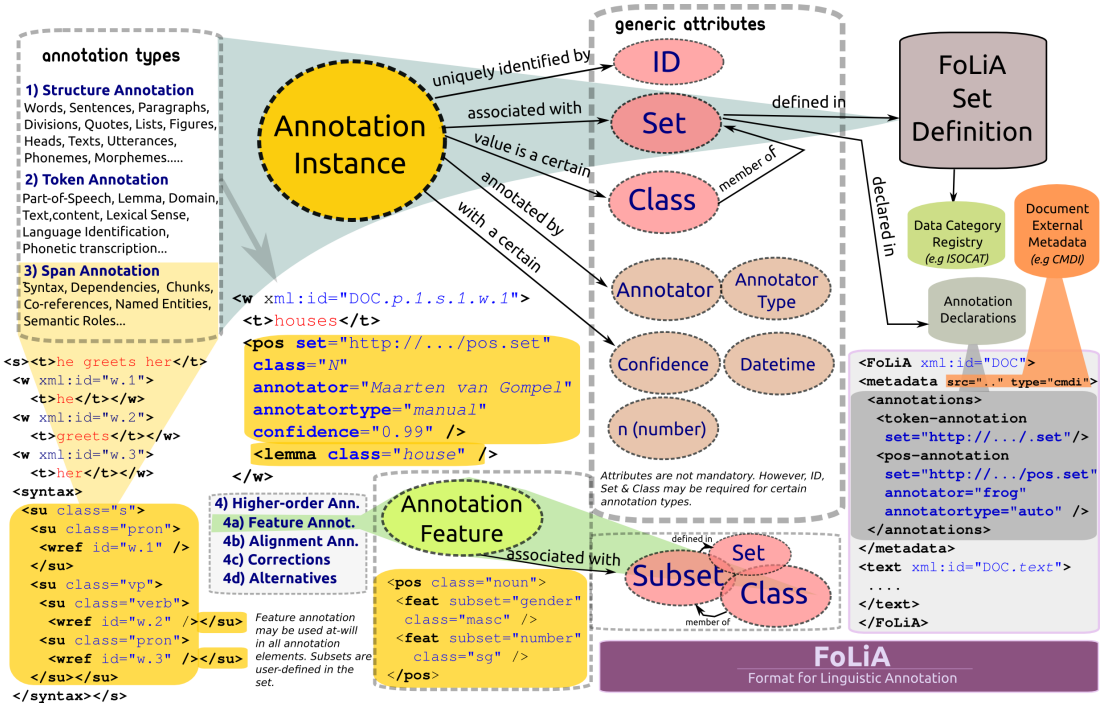

Figure 2.1: The FoLiA Paradigm

- `begintime` – **begin time** – A timestamp in `HH:MM:SS.MMM` format, indicating the begin time of the speech. If a sound clip is specified (`src`); the timestamp refers to a location in the soundclip.
- `endtime` – **end time** – A timestamp in `HH:MM:SS.MMM` format, indicating the end time of the speech. If a sound clip is specified (`src`); the timestamp refers to a location in the soundclip.
- `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 you are unable to link the speech to a specific (single) speaker.

Read 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 types of annotation and the sets that are present in the document. Annotations are declared in the `annotations` block.

The follow example declares four annotation levels with fictitious sets and several *default attributes*:

```
<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 mostly fictitious. Throughout the documentation, we will either be using the dummy value `http://url/to/your/set` for sets, or we will point to an actual set definition, in which case you need to be aware this is merely and example or suggestion which you are never obliged to use. You can always point to your own sets.

A Set Definition specifies exactly what classes are allowed in the set. It for example specifies exactly what Part-of-speech tags exist. This information is necessary to validate the document completely 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.

Though we recommend using and creating actual sets. FoLiA itself is rather agnostic about their existence for most purposes. For deep validation, proper formalisation, and for certain applications they may be required; but as long as they serve as proper *unique identifiers* you can work with non-existing sets. In

[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!

this case, simply do not use a URL but another arbitrary identification string.

If multiple sets are used for the same annotation type, they each need a separate declaration, as illustrated with the following fictitious sets:

```
<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. This is common practice as usually there is only one set per annotation type.

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 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 where 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 2.10.11 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 2.10.11.

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.

2.5.2 Paragraphs, Sentences and Words

Paragraphs, sentences and words (or tokens) are amongst the most elementary structure elements. As we saw in a previous section, word elements (`w`) can take a class, pertaining to a certain set, at which point a declaration must be present in the metadata:

Declaration

```
<annotations>
 <token-annotation set="http://url/to/your/set" />
</annotations>
```

(Note: The given sets are just examples, you are free to create and use any set of your own)

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.

Declaration

```
<annotations>
 <paragraph-annotation set="http://url/to/your/set" />
 <sentence-annotation set="http://url/to/your/set" />
</annotations>
```

(Note: The given sets are just examples, you are free to create and use any set of your own)

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 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>
          <division-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 supports 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 we saw earlier, divisions should never be used for marking these, only larger structures can be divisions.

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

Declaration

```
<annotations>
 <division-annotation set="https://raw.githubusercontent.
     com/proycon/folia/master/setdefinitions/divisions.
     foliaset.xml"
</annotations>
```

(Note: The given sets are just examples, you are free to create and use any set of your own)

2.5.4 Quotes

Status: final since v0.3 (older versions are equal but lack declarations), larger quotes allowing paragraphs and division since v0.11.3 · **Implementations:** pynlpl, libfolia

FoLiA supports quotes, using the quote element, to indicate that the structural elements within are what another person said or wrote:

```
<quote xml:id="example.quote.1">
  <p xml:id="example.quote.1.p.1">
   <t>I have a dream that one day this nation will rise up and
       live out the
true meaning of its creed: "We␣hold␣these␣truths␣to␣be␣self-
    evident,␣that␣all
men␣are␣created␣equal."</t>
  </p>
  <p xml:id="example.quote.1.p.1">
   <t>I have a dream that one day on the red hills of Georgia,
       the sons of
former slaves and the sons of former slave owners will be able
    to sit down
together at the table of brotherhood.</t>
  </p>
</quote>
```

Quotes can be embedded in multiple levels, it may be a large block containing itself divisions, paragraph or sentences, such as in the example above, or it may be embedded in a sentence, in which case no divisions or paragraphs may occur in the quote anymore.

One special case is the fact that sentences in sentences are allowed if they are in a quote, this is demonstrated in the next example:

```
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>
```

Declaration
Quotes are undeclarable elements

2.5.5 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-defines 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:

Declaration
<annotations> <gap-annotation set="https://raw.githubusercontent.com/proycon/folia/master/setdefinitions/gaps.foliaset.xml"> </annotations> (Note: The given sets are just examples, you are free to create and use any set of your own)

2.5.6 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, headers, and sentences.

Declaration
Whitespace and linebreaks are undeclarable elements.

2.5.7 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 2.10.4)

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.

Declaration

```
<annotations>
 <event-annotation
 set="https://raw.githubusercontent.com/proycon/folia/
    master/setdefinitions/events.foliaset.xml"
</annotations>
```

(Note: The given sets are just examples, you are free to create and use any set of your own)

2.5.8 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 has a `label` element, this is deprecated in favour of the `n` attribute in the item itself.

Declaration

Lists are undeclarable elements.

2.5.9 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/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>
```

The caption element may hold sentences (s) and words (w).

Declaration
Figures are undeclarable elements.

2.5.10 Tables

Status: since FoLiA 0.9.2 · **Implementations:** pynlpl

Support for simple tables is provided in a fashion similar to HTML and TEI. The element table introduces a table, within its scope row elements mark the various rows, tablehead marks the header of the table and contains one or more rows. The rows themselves consist of cell elements, which in turn may contain other structural elements such as words, sentences or even entire paragraphs.

Consider the example below (not all elements have been assigned IDs for brevity):

```
<table xml:id="example.table.1">
  <tablehead>
    <row>
      <cell>
        <w xml:id="example.table.1.w.1"><t>Name</t></w>
      </cell>
      <cell>
        <w xml:id="example.table.1.w.2"><t>Affiliation</t></w>
      </cell>
    </row>
  </tablehead>
  <row>
    <cell>
      <w xml:id="example.table.1.w.3"><t>Maarten van Gompel</t><
          /w>
    </cell>
    <cell>
      <w xml:id="example.table.1.w.4">
        <t>Radboud University Nijmegen</t>
      </w>
```

```
    </cell>
  </row>
  <row>
    <cell>
      <w xml:id="example.table.1.w.5"><t>Ko van der Sloot</t></w
          >
    </cell>
    <cell>
      <w xml:id="example.table.1.w.6"><t>Tilburg University</t><
          /w>
    </cell>
  </row>
</table>
```

Tables, rows and cells can all be assigned classes, the declaration is as follows:

Declaration

```
<annotations>
 <table-annotation set="http://url/to/your/set" />
</annotations>
```

(Note: The given sets are just examples, you are free to create and use any set of your own)

2.5.11 Notes

Status: since FoLiA 0.11.0 · **Implementations:** pynlpl, libfolia

The structure element `note` allows for notes to be included in FoLiA documents. A footnote as well as a bibliographical reference is an example of a note. The notes form an integral part of the text. For notes that are merely descriptive comments on the texts or its annotations, rather than a part of it, consider using `desc` instead. Notes themselves can contain all the usual forms of annotations.

The place of a note in the text is where it will appear. References to the note are made using a specific tag, `ref`, discussed in the next section.

```
  <s><t>blah blah blah</t></s>
  <note xml:id="mynote" class="footnote">
    <s xml:id="mynote.s.1"><t>See our website!</t></s>
  </note>
</text>
```

Notes are also suited for building a bibliography:

```
<note xml:id="bib" class="bibref">
  <t>Maarten van Gompel (2014). FoLiA: Format for Linguistic
     Annotation.
Documentation. Language and Speech Technology Technical Report
   Series 14-01.
Radboud University Nijmegen</t>
</note>
```

Whereas this section just presented the notes themselve, the next section will discuss out to point to notes from within the text.

Declaration

```
<annotations>
 <note-annotation
 set="https://raw.githubusercontent.com/proycon/folia/
     master/setdefinitions/notes.foliaset.xml"
</annotations>
```

(Note: The given sets are just examples, you are free to create and use any set of your own)

2.5.12 Structure References

Status: since FoLiA 0.11.0 · **Implementations:** pynlpl, libfolia

In the previous section we discussed notes, in this section we show that you can make references to these notes using the `ref` element, this is a structure element with an extra higher-order annotation function:

```
<s>
  <t>We demonstrated this earlier.</t>
  <ref id="mynote" />
</s>
```

Another example in tokenised data, and now we add the *optional* `type` attribute, which holds the type of the FoLiA element that is referred to:

```
<s>
  <w><t>We</t></w>
  <w><t>demonstrated</t></w>
  <w><t>this</t></w>
  <w><t>earlier</t></w>
```

```
  <w><t>.</t></w>
  <ref id="mynote" type="note" />
</s>
```

You can optionally make explicit the symbol used for the reference:

```
<s>
  <t>We demonstrated this earlier.</t>
  <ref id="mynote" type="note"><t>1</t></ref>
</s>
```

This is often needed for bibliographical references:

```
<s>
  <t>We demonstrated this earlier.</t>
  <ref id="bib.1" type="note"><t>(van Gompel et al, 2014)</t></
      ref>
</s>
```

Although we framed this in the context of notes, the `ref` element is more general and can be used whereever you need to explicitly refer to other *structure elements*. Common targets are figures, tables, divisions (sections, chapters, etc).

Being a structure element, the note reference itself may carry an ID as well. Note that the ID attribute without the xml namespace always indicates a reference in FoLiA:

```
<s><t>We demonstrated this earlier.</t></s>
<ref xml:id="myreference" id="mynote" />
```

The difference between the reference element and the higher-order alignments (Section 2.10.8) needs to be clearly understood. Alignments lay relations between annotations of any kind and thus pertain strongly to linguistic annotation, whereas this reference element is a structural element that is explicitly shown in the text and ddraws a reference that is explicitly reflected in the text.

Declaration
The reference element itself it not declarable

2.5.13 Parts

Status: since FoLiA 0.11.2 · **Implementations:** pynlpl, libfolia

The structure element `part` is a fairly abstract structure element that should only be used when a more specific structure element is not available. Most notably, the part element should never be used for representation of morphemes or phonemes!

Part can be used to divide a larger structure element, such as a division, or a paragraph into arbitrary subparts.

```
<p>
  <part xml:id="p.1.part.1">
    <t>First part of the paragraph.</t>
  </part>
  <part xml:id="p.2.part.2">
    <t>Last part of the paragraph.</t>
  </part>
</p>
```

The part element may seem alike to the division element, but divisions are used for text blocks larger than a paragraph, typically correspondings to chapters, sections or subsections and often carrying a `head` element. Do not use parts for these structures.

The part element, on the other hand, is more abstract and plays a role on a deeper level. It can be embedded within paragraphs, sentences, and most other structure elements, even words, though we have to again emphasize it should not be used for morphology, there are other solutions for that!

Contact the FoLiA authors if you find yourself using part and you feel a more specific FoLiA element is missing.

Declaration

```
<annotations>
 <part-annotation set="http://url/to/your/set" />
</annotations>
```

(Note: The given sets are just examples, you are free to create and use any set of your own)

2.5.14 Entries, definitions & examples

Status: FoLiA 0.12 · **Implementations:** pynlpl

FoLiA has a set of structure element that can be used to represent collections such as glossaries, dictionaries, thesauri, and wordnets.

These have in common that they consist of a set of entries, represented in FoLiA by the `entry` element, and each entry is identified by one or more terms, represented by the `term` element within an entry.

Terms need not be words, but a wide variety of structural elements can be used as the term. Within the entry, these terms can subsequently be associated with one or more definitions, using the `def` element, or with examples, using the `ex` element.

The `term`, `def` and `ex` elements can all take sets and classes, and thus need to be declared. The `entry` elements themselves are simple containers and need no declaration. Entries can contain multiple terms if they are deemed dependent or related, such as in case of morphological variants such as verb conjugations and declensions. The elements `term` and `def` can only be used within an `entry`. The `ex` element, however, can also be used standalone in different contexts.

In FoLiA, linguistic annotations are associated with the structure element within the term itself. This is where a glossary can for instance obtain part-of-speech or lexical semantic sense information, to name just a few examples.

Below you see an example of a glossary entry, the sense set used comes from WordNet. The other sets are fictitious.

```
<entry xml:id="entry.1">
 <term xml:id="entry.1.term.1">
  <w xml:id="entry.1.term.1.w.1">
    <t>house</t>
    <pos class="n">
      <feat subset="number" class="sing" />
    </pos>
    <lemma class="house" />
    <sense class="house\%1:06:00::">
  </w>
 </term>
 <term xml:id="entry.1.term.2">
  <w xml:id="entry.1.term.2.w.1">
    <t>houses/t>
    <pos class="n">
      <feat subset="number" class="plural" />
    </pos>
    <lemma class="house" />
    <sense class="house\%1:06:00::">
  </w>
```

```
 </term>
 <def xml:id="entry.1.def.1" class="sensedescription">
  <p xml:id="entry.1.def.1.p.1">
    <t>A dwelling, place of residence</t>
  </p>
 </def>
 <ex>
  <s xml:id="entry.1.ex.1.s.1>
␣␣␣␣<t>My␣house␣was␣constructed␣ten␣years␣ago.</t>
␣␣</s>
␣</ex>
</entry>
```

Other semantic senses would be represented as separate entries.

The definitions (`def`) are a generic element that can be used for multiple types of definition. As always, the set is not predefined and purely fictitious in our examples, giving the user flexibility. Definitions are for instance suited for dictionaries:

```
<entry xml:id="entry.1">
 <term xml:id="entry.1.term.1">
  <w xml:id="entry.1.term.1.w.1">
    <t>house</t>
    <pos set="englishpos" class="n">
      <feat subset="number" class="sing" />
    </pos>
    <lemma set="englishlemma" class="house" />
    <sense set="englishsense" class="house\%1:06:00::">
  </w>
 </term>
 <def xml:id="entry.1.def.1" class="translation-es">
  <w xml:id="entry.1.def.1.w.1">
    <t>casa</t>
    <pos set="spanishpos"  class="n">
      <feat subset="number" class="sing" />
    </pos>
    <lemma set="spanishlemma" class="casa" />
  </w>
 </def>
</entry>
```

Or for etymological definitions:

```
 <def xml:id="entry.1.def.2" class="etymology">
  <p xml:id="entry.1.def.2.p.1">
   <t>Old English hus "dwelling,␣shelter,␣house," from Proto-
      Germanic *husan
```

```
 (cognates: Old Norse, Old Frisian hus, Dutch huis, German Haus)
    , of unknown
 origin, perhaps connected to the root of hide (v.) [OED]. In
    Gothic only in
 gudhus "temple," literally "god-house;" the usual word for "
    house" in Gothic
 being razn.  </t>
  </p>
 </def>
```

To draw relations between entries and the terms therein, such as for example for a wordnet, use FoLiA's *alignments* (see section 2.10.8).

The following two samples illustrate a dictionary distributed over multiple FoLiA files, using alignments to link the two:

English part, `doc-english.xml`:

```
<entry xml:id="en-entry.1">
 <term xml:id="en-entry.1.term.1">
  <w xml:id="en-entry.1.term.1.w.1">
    <t>house</t>
    <pos set="englishpos" class="n">
      <feat subset="number" class="sing" />
    </pos>
    <lemma set="englishlemma" class="house" />
    <sense set="englishsense" class="house\%1:06:00::">
  </w>
 </term>
 <alignment class="translation-es" xlink:href="doc-spanish.xml"
    xlink:type="simple">
      <aref id="es-entry.1" type="entry" />
 </alignment>
</entry>
```

Spanish part, `doc-spanish.xml`:

```
<entry xml:id="es-entry.1">
 <term xml:id="es-entry.1.def.1" class="translation-es">
  <w xml:id="entry.1.def.1.w.1">
    <t>casa</t>
    <pos set="spanishpos"  class="n">
      <feat subset="number" class="sing" />
    </pos>
    <lemma set="spanishlemma" class="casa" />
  </w>
 </term>
 <alignment class="translation-en" xlink:href="doc-english.xml"
```

```
    xlink:type="simple">
      <aref id="en-entry.1" type="entry" />
 </alignment>
</entry>
```

For simple multilingual documents, explicit alignments may be too much hassle, see Section 2.10.10 for a simpler methods based on convention.

Declaration

```
<annotations>
 <term-annotation set="http://url/to/your/set" />
 <definition-annotation set="http://url/to/your/set" />
 <example-annotation set="http://url/to/your/set" />
</annotations>
```

(Note: The given sets are just examples, you are free to create and use any set of your own)

2.6 Token Annotation

Token annotations are annotations that are placed within the structural elements, often words/tokens (`w`) (hence the name), but also other structure elements, in which case we speak of *extended token annotation*.

All token annotation elements may take all of the generic attributes described 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 a 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.

Declaration

```
<annotations>
 <pos-annotation set="http://url/to/your/set" />
</annotations>
```

(Note: The given sets are just examples, you are free to create and use any set of your own)

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>
```

Declaration

```
<annotations>
 <lemma-annotation set="http://url/to/your/set" />
</annotations>
```

(Note: The given sets are just examples, you are free to create and use any set of your own)

2.6.3 Language Identification Annotation

Status: final since v0.8.1 · **Implementations:** pynlpl, libfolia

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

Declaration

```
<annotations>
 <lang-annotation set="http://url/to/your/set" />
</annotations>
```

(Note: The given sets are just examples, you are free to create and use any set of your own)

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:

Declaration

```
<annotations>
 <sense-annotation set="http://url/to/your/set" />
</annotations>
```

(Note: The given sets are just examples, you are free to create and use any set of your own)

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):

Declaration

```
<annotations>
 <domain-annotation set="http://url/to/your/set" />
</annotations>
```

(Note: The given sets are just examples, you are free to create and use any set of your own)

2.6.6 Subjectivity/Sentiment Analysis

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

Subjectivity annotation is used to associate a certain subjective quality with a structural element. It is used for sentiment analysis and opinion analysis. Example:

```
<w xml:id="example.p.1.s.1.w.2">
    <t>hate</t>
    <subjectivity class="negative" />
</w>
```

The declaration (the actual set is fictitious):

Declaration

```
<annotations>
 <subjectivity-annotation set="http://url/to/your/set" />
</annotations>
```

(Note: The given sets are just examples, you are free to create and use any set of your own)

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*. Within these layers, references are made to all of 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. The annotation layers are generally embedded within the structure element that also contains all the words that are referenced. Often this corresponds to the sentence level. Any other higher level is fine too, though. Layers are always embedded *after* the word tokens or other structural elements.

Depending on the type of span annotation, it is possible that the element may be nested. This is for example the case for syntactic annotation, where the nesting of syntactic units allows the building of syntax trees. Other span annotation

elements of a more complex nature may require other span annotation elements within them, these latter span annotation elements are then known as *span roles*. Span roles can only be used in the scope of a certain span annotation element, not standalone, and therefore do not have their own dedicated layer.

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="per">
        <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.

Declaration

```
<annotations>
 <entity-annotation
 set="https://raw.githubusercontent.com/proycon/folia/
    master/setdefinitions/namedentities.foliaset.xml"
</annotations>
```

(Note: The given sets are just examples, you are free to create and use any set of your own)

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 illustrates a fairly simple syntactic parse. Dependency parses are possible too. Dependencies are listed separate from the syntax in an extra annotation layer, see Section 2.7.3.

Declaration

```
<annotations>
 <syntax-annotation set="http://path/to/your/set"
</annotations>
```

(Note: The given sets are just examples, you are free to create and use any set of your own)

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 example below, 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 two span roles: one

head element (hd) and one dependent element (dep). The words they cover are reiterated in the usual fashion, using wref. For a better understanding, Figure 2.2 illustrates the syntactic parse with the dependency relations. Both span roles hd and dep can optionally make extra reference to a syntactic unit (or anything else for that matter) by means of the aref element.

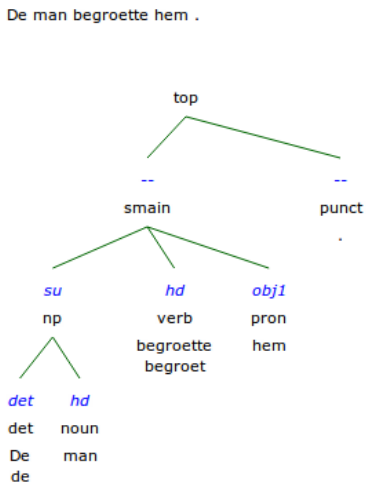

Figure 2.2: Alpino dependency parse for the Dutch sentence "De man begroette hem."

```
<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 dependant is just "man" rather than "de man" . That is, we point only to the head of dependants, the full scope follows automatically when building the dependency tree.

Declaration

```
<annotations>
 <syntax-annotation set="http://url/to/your/set" />
 <dependency-annotation set="http://url/to/your/set" />
</annotations>
```

(Note: The given sets are just examples, you are free to create and use any set of your own)

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):

Declaration

```
<annotations>
 <chunking-annotation set="http://url/to/your/set" />
</annotations>
```

(Note: The given sets are just examples, you are free to create and use any set of your own)

2.7.5 Time Segmentation

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

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 2.5.7, 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:

```
<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" begindatetime="2011-12-15T19:03"
   enddatetime="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 predefined 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.

If you are only interested in an annotation of events, and a coarser level of annotation suffices, then use the structure annotation element `event` instead. See Section 2.5.7.

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`.

Declaration

```
<annotations>
 <timesegment-annotation set="http://url/to/your/set" />
</annotations>
```

(Note: The given sets are just examples, you are free to create and use any set of your own)

Time segmentation in a speech context

Status: final since v0.10 · **Implementations:** pynlpl,libfolia

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.MMM` 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="00:00:00.000"
   endtime="00:00:00.250">
    <wref id="example.p.1.s.1.w.1" t="I" />
  </timesegment>
  <timesegment begintime="00:00:00.250"
   endtime="00:00:00.500">
    <wref id="example.p.1.s.1.w.2" t="think" />
  </timesegment>
  <timesegment begintime="00:00:00.500"
   endtime="00:00:00.750">
    <wref id="example.p.1.s.1.w.3" t="I" />
  </timesegment>
  <timesegment begintime="00:00:00.750"
   endtime="00:00:01.000">
    <wref id="example.p.1.s.1.w.4" t="have" />
  </timesegment>
  <timesegment begintime="00:00:01.000"
   endtime="00:00:01.250">
    <wref id="example.p.1.s.1.w.5" t="to" />
  </timesegment>
  <timesegment begintime="00:00:01.250"
   endtime="00:00:01.500">
    <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="00:00:00.000" endtime="00:00:00.250"><t>I</t></w>
 <w xml:id="example.p.1.s.1.w.2"
  begintime="00:00:00.250" endtime="00:00:00.500"><t>think</t></
     w>
 <w xml:id="example.p.1.s.1.w.3"
```

```
  begintime="00:00:00.500" endtime="00:00:00.750"><t>I</t></w>
 <w xml:id="example.p.1.s.1.w.4"
  begintime="00:00:00.750" endtime="00:00:01.000"><t>have</t></w
     >
 <w xml:id="example.p.1.s.1.w.5"
  begintime="00:00:01.000" endtime="00:00:01.250"><t>to</t></w>
 <w xml:id="example.p.1.s.1.w.6"
  begintime="00:00:01.250" endtime="00:00:01.500"><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>
```

Declaration

```
<annotations>
 <semrole-annotation set="http://url/to/your/set" />
</annotations>
```

(Note: The given sets are just examples, you are free to create and use any set of your own)

2.7.7 Coreference Relations

Status: new in 0.9 (example improved in revision 4.1) · **Implementations:** pynlpl

Relations between words that refer to the same referent are expressed in FoLiA using the coreferencechain span annotation element and the coreferencelink span role within it. The annotation layer is coreferences. The co-reference relations are expressed by specifying the entire chain in which all links are coreferent. The head of a coreferent may optionally be marked with the hd element, another span role. 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 smiled.</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="dalailama">
      <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>
    <coreferencechain class="dalailama">
      <coreferencelink>
        <wref id="example.p.1.s.2.w.5" t="him" />
      </coreferencelink>
      <coreferencelink>
        <wref id="example.p.1.s.2.w.6" t="him" />
      </coreferencelink>
      <coreferencelink>
        <wref id="example.p.1.s.3.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.

Declaration

```
<annotations>
 <coreference-annotation set="http://url/to/your/set" />
</annotations>
```

(Note: The given sets are just examples, you are free to create and use any set of your own)

2.8 Morphological Annotation

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

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.

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 their 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.

Declaration

```
<annotations>
 <morphology-annotation set="http://url/to/your/set" />
</annotations>
```

(Note: The given sets are just examples, you are free to create and use any set of your own)

2.9 Speech Annotation

2.9.1 Speech Structure Annotation

Status: proposed in v0.9, most parts final in v0.12 · **Implementations:** pynlpl,libfolia

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: Final in v0.12 · **Implementations:** pynlpl,libfolia

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

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

Declaration

```
<annotations>
 <utterance-annotation set="http://url/to/your/set" />
</annotations>
```

(Note: The given sets are just examples, you are free to create and use any set of your own)

Non-speech events

Non-speech events are simply covered by `event` annotation as seen in Section 2.5.7. Consider the following small example, with speech-context attributes associated:

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

2.9.2 Phonetic Content

Status: final in v0.12 · **Implementations:** pynlpl,libfolia

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

```
<utt src="helloworld.mp3"  begintime="..." endtime="...">
        <ph>hel'oʊ wɔːld</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>wɔːld</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 Phonological Annotation

Status: final in v0.12 · **Implementations:** pynlpl,libfolia

The smallest unit of annotatable speech in FoLiA 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 phonology 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>
    <phonology>
      <phoneme begintime="..."  endtime="...">
          <ph>b</ph>
      </phoneme>
      <phoneme begintime="..." endtime="...">
          <ph>ʊ</ph>
      </phoneme>
      <phoneme begintime="..." endtime="...">
          <ph>k</ph>
      </phoneme>
    </phonology>
  </w>
</utt>
```

Declaration

```
<annotations>
 <phonological-annotation set="http://url/to/your/set" />
</annotations>
```

(Note: The given sets are just examples, you are free to create and use any set of your own)

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):

Declaration

```
<annotations>
 <distortion-annotation set="http://url/to/your/set" />
</annotations>
```

(Note: The given sets are just examples, you are free to create and use any set of your own)

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 the 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 annotation 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 within `altlayers` need to be marked as non-autoritative using `auth="no"`.

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

```
<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. However, it 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 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. Two scenarios can be envisioned, 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. The type of measurement is defined by the class, which in turn is defined by the set as always. The metric element has a value attribute that stores the actual measurement,

the value is often numeric but this needs not be the case. It is a higher-level annotation element that 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>
```

Declaration

```
<annotations>
 <metric-annotation set="http://url/to/your/set" />
</annotations>
```

(Note: The given sets are just examples, you are free to create and use any set of your own)

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`). However, it is 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>
```

Corrections need to be be declared:

Declaration

```
<annotations>
 <correction-annotation set="http://url/to/your/set" />
</annotations>
```

(Note: The given sets are just examples, you are free to create and use any set of your own)

Error detection

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.

Error detection has to be declared seperately from corrections, as they can be used independently. However, nothing stops you from pointing them both to the same set.

Declaration

```
<annotations>
 <errordetection-annotation set="http://url/to/your/set" /
     >
</annotations>
```

(Note: The given sets are just examples, you are free to create and use any set of your own)

Suggestions for 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 postpone this actual selection to another module or human annotator. The output of a speller check is typically a suggestion for correction. 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.

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 within the `original` element. 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 class, as always, is just a fictitious example and users can assign their own according to their own sets.

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="swap">
    <new>
        <w xml:id="example.p.1.s.1.w.2_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>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 swapped tokens tokens are new. They are essentially copies of the originals. Likewise, any token annotations you want to preserve explicitly need to be copies.

Insertions and Deletions

Insertions are words that are omitted in the original and have to be inserted in correction, while deletions are words that are erroneously inserted in the original and have to be removed in correction. FoLiA deals with these in a similar way to merges, splits and swaps. For deletions, 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/>
    <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 insertions, 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"/>
 </correction>
 <w xml:id="example.p.1.s.1.w.2">
    <t>man</t>
 </w>
</s>
```

Although we limited our discussion to merges, splits, insertions and deletions applied to words/tokens, they may be applied to any other structural element just as well.

Suggestions for correction: structural changes

The earlier described suggestions for correction can be extended to merges, splits, insertions and deletions as well. This is done by embedding the newly suggested structure in `suggestion` elements. The current version of the structure is moved to within the scope of a `current` element.

We illustrate the splitting of online to on line as a suggestion for correction:

```
<s xml:id="example.p.1.s.1">
 <correction xml:id="example.p.1.s.1.c.1" class="split">
    <current>
        <w xml:id="example.p.1.s.1.w.1">
            <t>online</t>
        </w>
    </current>
```

```
    <suggestion auth="no">
        <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>
    </suggestion>
 </correction>
</s>
```

Special cases are insertions and deletions. In case of suggested insertions, the current element is empty (but always present!), in case of deletions, the suggestion element is empty (but always present!).

For non-structural suggestions for correction, we simply have multiple correction elements if there are suggestions for correction of different classes. When structural changes are proposed, however, this is not possible, as there can be only one `current` element. The remedy here is to nest corrections, a current element may hold a correction with its own current element, and so on.

We can use suggestions for correction on any structural level; so we can for instance embed entire sentences or paragraphs within a suggestion. However, this quickly becomes very verbose and redundant as all the lower levels are copied for each suggestion. Common structural changes, as we have seen, are splits and merges. The `suggestion` element has a special additional facility to signal splits and merges, using the `split` and `merge` attribute, the value of which points to the ID (or IDs, space delimited) of the elements to split or merge with. When applied to sentences, splits and merges often coincide with an insertion of punctuation (for a sentence split), or deletion of redundant punctuation (for a sentence merge). The following two examples illustrate both these cases:

```
<p xml:id="correctionexample.p.2">
    <s xml:id="correctionexample.p.2.s.1">
        <w xml:id="correctionexample.p.2.s.1.w.1"><t>I</t></w>
        <w xml:id="correctionexample.p.2.s.1.w.2"><t>think</t>
            </w>
        <correction xml:id="correctionexample.p.2.correction.1
            " class="redundantpunctuation">
            <suggestion auth="no" merge="correctionexample.p
                .2.s.2" />
            <current>
                <w xml:id="correctionexample.p.2.s.1.w.3"><t>.
                    </t></w>
            </current>
```

```
        </correction>
    </s>
    <s xml:id="correctionexample.p.2.s.2">
        <w xml:id="correctionexample.p.2.s.2.w.1"><t>and</t></
            w>
        <w xml:id="correctionexample.p.2.s.2.w.2"><t>therefore
            </t></w>
        <w xml:id="correctionexample.p.2.s.2.w.3"><t>I</t></w>
        <w xml:id="correctionexample.p.2.s.2.w.4"><t>am</t></w
            >
        <w xml:id="correctionexample.p.2.s.2.w.5"><t>.</t></w>
    </s>
</p>

<p xml:id="correctionexample.p.2">
    <s xml:id="correctionexample.p.2.s.1">
        <w xml:id="correctionexample.p.2.s.1.w.1"><t>I</t></w>
        <w xml:id="correctionexample.p.2.s.1.w.2"><t>go</t></w
            >
        <w xml:id="correctionexample.p.2.s.1.w.3"><t>home</t><
            /w>
        <correction xml:id="correctionexample.p.2.correction.1
            " class="missingpunctuation">
            <suggestion auth="no" split="correctionexample.p
                .2.s.1">
                <w xml:id="correctionexample.p.2.s.1.w.3a"><t>
                    .</t></w>
            </suggestion>
            <current />
        </correction>
        <w xml:id="correctionexample.p.2.s.1.w.4">
          <t>you</t>
          <correction xml:id="correctionexample.p.2.correction
              .2" class="capitalizationerror">
            <suggestion auth="no">
              <t>You</t>
            </suggestion>
          </correction>
        </w>
        <w xml:id="correctionexample.p.2.s.1.w.5"><t>welcome</
            t></w>
        <w xml:id="correctionexample.p.2.s.1.w.6"><t>me</t></w
            >
        <w xml:id="correctionexample.p.2.s.1.w.7"><t>.</t></w>
    </s>
</p>
```

In the second example, we also add an additional non-structural suggestion for

correction, suggesting to capitalize the first word of what is suggested to become a new sentence.

Corrections on span annotation

Status: added in v0.11.1 · **Implementations:** pynlpl,libfolia

All the previous sections focussed on corrections on token annotation, text content, or structure elements such as words. The correction element, however, is one of the most ubiquitous elements in FoLiA and can also be used for correcting span annotation elements, such as named entities. Recall that span annotation elements are embedded in an annotation layer. The correction element may be used within such annotation layers, as well as within span elements themselves. Corrections on span annotation elements can be corrections on either the class or on the tokens over which the annotation spans, the following two examples illustrate each of these cases:

```
<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>
    <correction class="wrongclass">
      <new>
        <entity xml:id="example.p.1.s.1.entity.1.corrected"
            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>
      </new>
      <original auth="no">
        <entity xml:id="example.p.1.s.1.entity.1" class="
            organisation">
            <wref id="example.p.1.s.1.w.2" t="Dalai" />
            <wref id="example.p.1.s.1.w.3" t="Lama" />
        </entity>
      </original>
    </correction>
  </entities>
</s>
```

```
<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>
    <correction class="wrongclass">
      <new>
        <entity xml:id="example.p.1.s.1.entity.1.corrected"
            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>
      </new>
      <original auth="no">
        <entity xml:id="example.p.1.s.1.entity.1" class="person"
            >
            <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" />
        </entity>
      </original>
    </correction>
  </entities>
</s>
```

When correcting span annotation elements that are nested (such as syntax), the child elements are an inherent part of the correction, and will often need to be duplicated if the correction is on an element higher up in the tree.

2.10.8 Alignments

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

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, 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 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 Translations

In Section 2.10.8 shows that alignments are an excellent tool for specifying translations. Section 2.5.14 shows how to use this in combination with the `entry` element to form dictionaries.

For situations in which alignments seem overkill, a simple multi-document mechanism is available. This mechanism is based purely on convention: It assumes that structural elements that are translations simply share the same ID. This approach is quite feasible when used on higher-level structural elements, such as divisions, paragraphs, events or entries.

2.10.11 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. Text-content elements may never be empty nor contain only whitespace or non-printable characters, in such circumstances you simply omit the text-content element alltogether.

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:

[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.

Declaration

```
<annotations>
 <text-annotation set="http://url/to/your/set" />
</annotations>
```

(Note: The given sets are just examples, you are free to create and use any set of your own)

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>
    <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 a 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.

2.10.12 Substrings

Status: final since v0.9.1 · **Implementations:** pynlpl,libfolia

A `str` element is available in FoLiA to allow annotations on untokenised sub-

strings. The `str` element refers to a substring of the text-content (`t`) element on the same level and allows the assigning of identifiers to substrings. Consider the following example:

```
<p xml:id="example.p.1">
    <t>Hello. This is a sentence. Bye!</t>
    <str xml:id="example.p.1.str.1">
        <t offset="0">Hello</t>
    </str>
</p>
```

In substrings, using an offset attribute on the text-content element enables substrings to be properly positioned with respect to their parent text.

The class *current* is assigned when no explicit class is mentioned. In case of multiple `t` elements the class tells to which `t` element the substring refers: Both are covered by the `text-annotation` declaration. Both the substring element, as well as the text content element should always carry 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>
    <t class="ocroutput">Hell0 Th1s iz a sentence, Bye1</t>

    <str xml:id="example.p.1.s.1">
        <t class="ocroutput" offset="0">Hell0</t>
    </str>

    <str xml:id="example.p.1.s.2">
        <t class="normalised" offset="0">Hello.</t>
    </str>

    <str xml:id="example.p.1.s.3">
        <t offset="0">Hello.</t>
    </str>
</p>
```

Relations between the various substrings can be explicitly represented using alignments:

```
<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>

    <str xml:id="example.p.1.s.1">
        <t class="ocroutput" offset="0">Hell0</t>
        <alignment>
```

```
            <aref id="example.p.1.s.2" type="str" />
        </alignment>
    </str>

    <str xml:id="example.p.1.s.2">
        <t offset="0">Hello.</t>
        <alignment>
            <aref id="example.p.1.s.1" type="str" />
        </alignment>
    </str>
  </p>
```

Or one string can refer back to two or more different text content elements:

```
 <p xml:id="example.p.1">
    <t>Hello. This is a sentence. Bye!</t>
    <t class="normalised">Hello. This iz a sentence. Bye!</t>
    <t class="ocroutput">Hell0 Th1s iz a sentence, Bye1</t>

    <str xml:id="example.p.1.s.1">
        <t class="ocroutput" offset="0">Hell0</t>
        <t class="normalised" offset="0">Hello</t>
                <t offset="0">Hello.</t>
    </str>

</p>
```

Substring elements are a form of higher-order annotation, they are similar to structure annotation but carry several different properties. Unlike structure elements, substring order does not matter and substrings may overlap. The difference between `w` and `str` has to be clearly understood, the former refers to actual tokens and supports further token annotation, the latter to untokenised or differently tokenised substrings.The `str` element is especially powerful when combined with alignments, as this allows the user to relate multiple alternative tokenisations. This is also the limit as to what you can do with differing tokenisations in FoLiA, as FoLiA only supports one authoritative tokenisation.

The `str` element does not allow all (extended) token annotations. However, certain extended token annotations are allowed; such as language identification, domain identification, metrics and corrections.

The `str` element has a counterpart in the group of text markup elements: `t-str`. See Section 2.10.13. Whilst not as powerful as the `str` element, it is more intuitive to use. The following example illustrates both `t-str` and `str`, with proper references making their relation explicit:

```
<p xml:id="example.p.1">
  <t><t-str id="example.p.1.str.1">Hello</t-str>. This
   is a sentence. Bye!</t>
  <str xml:id="example.p.1.str.1">
      <t offset="0">Hello</t>
  </str>
</p>
```

The declaration:

Declaration

```
<annotations>
 <string-annotation set="http://url/to/your/set" />
</annotations>
```

(Note: The given sets are just examples, you are free to create and use any set of your own)

2.10.13 Text Markup

Status: new in FoLiA v0.10 · **Implementations:** pynlpl

The text content element `t` allows within its scope elements of a certain group; these are *Text Markup* elements. The value of any of these elements is always textual content. Likewise, the elements have in common that in some way they modify or otherwise apply markup to textual content. Text markup elements mark certain spans within the text. Elements of this category may be nested.

Text markup elements may carry an optional identifier. However, it may happen that textual content is repeated on multiple levels, which implies the textual markup elements are also repeated. However, only one of them may carry the customary `xml:id` attribute; duplicates may carry the `id` *reference* attribute (in the FoLiA namespace instead of the XML namespace!) which is interpreted as a reference. Such an element should be identical to the one it refers to, and explicitly include the value (if applicable) to facilitate the job of XML parsers. Certain elements may also use this `id` reference attribute to refer to structural elements that cover the very same data. A markup element may thus take either `xml:id` or `id` (a reference to another element); they may never occur together.

Text markup elements may take sets and classes as most other elements, and

any of the remaining common FoLiA attributes may be freely associated with any of the text-markup elements.

As text markup operates in the scope of the text content element, it is ideally suited for untokenised text. You should, however, limit your usage of text markup elements and only use them when other existing annotation elements do not suffice, or for extra verbosity in addition to existing elements.

Each text-markup element, save for one exception, starts with `t-` and demands a declaration. The following subsections will discuss the various text markup elements available.

Style

The text markup element `t-style` marks a specific portion of textual concent to be rendered in a specific style. Styles in turn are simply classes in your set, closely following the FoLiA paradigm. FoLiA does not predefine any actual styles.

```
<s>
  <t>to <t-style class="bold">be</t-style> or not to be</t>
</s>
```

Text-markup elements may always be nested, as the following example shows:

```
<s>
  <t>to <t-style class="italic"><t-style class="bold">be
  </t-style></t-style> or not to be</t>
</s>
```

Declaration

```
<annotations>
 <style-annotation set="https://raw.githubusercontent.com/
     proycon/folia/master/setdefinitions/style.foliaset.xml
     "
</annotations>
```

(Note: The given sets are just examples, you are free to create and use any set of your own)

Gaps

The text markup element t-gap indicates a gap in the textual content. Either text is not available or there is a deliberate blank for, for example, fill-in exercises. It is recommended to provide a textual value when possible, but this is not required. The following examples show both possibilities:

```
<s>
  <t>to <t-gap xml:id="gap.1" class="fillin">be</t-gap> or not
      to be</t>
</s>

<s>
  <t>to <t-gap xml:id="gap.1" class="unknown" /> or not to be</t
      >
</s>
```

The t-gap element is related to the structural element gap, but offers a more fine-grained variant. If you find that you want to mark your whole text content as being a t-gap, then this is a sure sign you should use the structural element gap instead. The t-gap element may take an ID reference attribute that refers to a gap element, as shown in the following example:

```
<s>
  <t>to <t-gap id="gap.1" class="fillin">be</t-gap> or not to be
      </t>
  <w><t>to</t></w>
  <gap xml:id="gap.1"><content>be</content></gap>
  <w><t>or</t></w>
  <w><t>not</t></w>
  <w><t>to</t></w>
  <w><t>be</t></w>
</s>
```

Being related, the t-gap uses the same declaration as the gap element.

Declaration

```
<annotations>
 <gap-annotation
 set="https://raw.githubusercontent.com/proycon/folia/
     master/setdefinitions/gaps.foliaset.xml"
</annotations>
```

(Note: The given sets are just examples, you are free to create and use any set of your own)

Substrings

The t-str element relates to the higher-order annotation element str introduced in Section 2.10.12 and shares the same declaration. The id reference attribute of t-str may therefore refer to the id of a str element. The element is used to mark arbitrary substrings:

```
<s>
  <t>to <t-str xml:id="str.1">be</t-str> or not to be</t>
</s>
```

Note that t-str, unlike the str element, does not support overlap due to the nature of text markup elements.

The following example illustrates both t-str and str, with proper references:

```
 <p xml:id="example.p.1">
    <t><t-str id="example.p.1.str.1">Hello</t-str>. This
     is a sentence. Bye!</t>
    <str xml:id="example.p.1.str.1">
     <t offset="0">Hello</t>
    </str>
</p>
```

The declaration:

```
<annotations>
    <string-annotation set="http://path/to/your/set" />
</annotations>
```

Error Detection

The text-markup counterpart to the token annotation element errordetection is t-error. This element may be used to signal errors in text content. Note that the use of errordetection is preferred when possible.

```
<s>
  <t>to <t-error xml:id="error.1" class="spelling">bee</t-error>
  or not to be</t>
</s>

<annotations>
    <errordetection-annotation set="http://path/to/your/set" />
</annotations>
```

Correction

Corrections may be expressed using the `correction` element discussed in Section 2.10.7. A text markup counterpart is available in the form of the `t-correction` element, which marks specific text as being a correction. The definition is shared with the `correction` element and id references may be made to correction elements, which offer far more expressive power. The example below shows both:

```
<s>
  <t>to <t-correction class="confusible" id="correction.1"
   original="bee">be</t-correction> or not to be</t>
  <w><t>to</t></w>
  <correction class="confusible" xml:id="correction.1">
    <new>
      <w><t>be</t></w>
    </new>
    <original>
      <w><t>bee</t></w>
    </original>
  </correction>
  <w><t>or</t></w>
  <w><t>not</t></w>
  <w><t>to</t></w>
  <w><t>be</t></w>
</s>
```

The `t-correction` element allows an additional attribute `original`, which may be used to record the version prior to correction.

The declaration:

Declaration

```
<annotations>
 <correction-annotation
 set="https://raw.githubusercontent.com/proycon/folia/
     master/setdefinitions/spellingcorrection.foliaset.xml"
</annotations>
```

(Note: The given sets are just examples, you are free to create and use any set of your own)

Linebreaks

The element
 may be used both as a structural element as well as a text markup element. It is not declared.

2.10.14 Hyperlinks

Status: new in FoLiA v0.11.3 · **Implementations:** pynlpl

Hyperlinks are ubiquitous in documents from the web and are therefore supported in FoLiA as well. A hyperlink is defined as a pointer from a span of text to an external resource. In FoLiA, it is therefore implemented as a simple property of text itself. Text content elements (t) as well as any Text Markup elements that may be contained there in, may act as a hyperlink. The link itself is implemented through XLink semantics:

```
<s>
 <w><t>The</t></w>
 <w><t>FoLiA</t></w>
 <w><t>website</t></w>
 <w><t>is</t></w>
 <w><t xlink:type="simple" xlink:href="http://proycon.github.io/
     folia">here</t></w>
 <w><t>.</t></w>
</s>
```

Or on a substring in untokenised text:

```
<s>
 <t>The FoLiA website is <t-str xlink:type="simple"
  xlink:href="http://proycon.github.io/folia">here</t-str>.</t>
</s>
```

Note that hyperlinks offer a simpler and less extensive notion of linking than FoLiA's *alignments*. Always use alignments when linking between FoLiA elements, in-document or cross-document, as alignments are actual annotations, unlike these hyperlinks.

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 through the `meta` element. The following example shows this:

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

The native metadata format just offers a simple key-value store. You can define fields with custom IDs, but the following fields are pre-defined 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*)

The native metadata format is deliberately limited, as various other formats already tackle the metadata issue. FoLiA is able to operate with any other metadata format, such as for example CMDI [5], or Dublin Core. The `type` attribute specifies what metadata format is used. We see it was set to `native` for FoLiA's native metadata format, for foreign formats it can be set to any other string.

Foreign metadata can be stored in two ways:

1. Externally in a different file
2. Inside the FoLiA document itself

When the metadata is stored externally in a different file, a reference is made from the `src` attribute. As shown in the following example:

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

If you want to store the metadata in the FoLiA document itself, then the metadata must be places inside a `foreign-data` element. All elements under foreign-data *must* be in another XML namespace, that is, not the default FoLiA namespace.

```
<metadata type="dc">
  <annotations>
  ..
  </annotations>
  <foreign-data xmlns:dc="http://purl.org/dc/elements/1.1/">
    <dc:identifier>mydoc</dc:identifier>
    <dc:format>text/xml</dc:format>
    <dc:type>Example</dc:type>
    <dc:contributor>proycon</dc:contributor>
    <dc:creator>proycon</dc:creator>
    <dc:language>en</dc:language>
    <dc:publisher>Radboud University</dc:publisher>
    <dc:rights>public Domain</dc:rights>
  </foreign-data>
</metadata>
```

2.12 External documents and full stand-off annotation

Status: v0.11 · **Implementations:** only partially implemented in pynlpl.format.folia and libfolia

Although FoLiA is explicitly constructed around a document-centric paradigm that stores everything in a single file, it does offer mechanisms to employ a setup

that includes subdocuments or splits annotations accross multiple files. These are two distinct but comparable mechanisms, each with their own uses:

1. **Subdocuments** - At any point in the FoLiA structure, a reference can be made to a subdocument, this subdocument may optionally be flagged for inclusion, which makes available all subdocument data as if they were contained in the main file. Subdocuments not flagged as such are simple references.
2. **Stand-off documents** - Specific annotation types may be split into external documents. These external documents must duplicate a minimal part of the structure of the main document and may never introduce new structure.

Working with a single file, as opposed to the mechanisms offered in this section, is usually recommended. Nevertheless, FoLiA wants to offer the necessary flexibility for those who need it. Such advanced users must always be aware of the caveats inherent in working with multiple files:

- Make sure to distribute all applicable files if annotations are split. Always distribute the main file. only
- Make sure all external files describe the same version of the document
- Annotation declarations must be consistent between main file and external files, and may never clash.
- Not all FoLiA implementations will support this: simple FoLiA libraries and XPath or XSL based processors may not know how to deal with external documents.

FoLiA forces a certain level of duplicity/redundancy to make violating these constraints harder.

The first mechanism, subdocuments, is implemented using the `external` element. This element refers to another FoLiA document and may mark it for inclusion using the `include="yes"` attribute. The `src` attribute may refer to either a local file or a web-based URL. Consider the following excerpt of a parent document:

```
<text>
    <div>
        <head><t>chapter 1</t></head>
    </div>

    <external src="chapter2.xml" include="yes" />
</text>
```

The subdocument itself is just like any other normal FoLiA document, including all the mandatory declarations! Moreover, declarations must not conflict with any declarations in the parent document, or inclusion will be impossible. The subdocument may have an *optional* attribute `external="yes"`, which states that it is never to be loaded independently, but only when included in another document. It is possible for multiple parent documents to include the same subdocument.

```
<FoLiA id="chapter2" external="yes">
    <metadata>
      <annotations>
      </annotations>
    </metadata>
    <text>
      <div>
          <head><t>chapter 2</t></head>
      </div>
    </text>
</FoLiA>
```

When inclusion is enabled through `include="yes"`, FoLiA libraries will make the contents of the `text` element available at the exact point where the `external` reference is made.This implies that the elements contained under `text` must be valid in the context in which they are placed. If `include="no"` or unset, the external element acts as a mere reference.

Note that a structure element is required in subdocuments, the following main and subdocument are **invalid**! Invalid main document excerpt:

```
<w xml:id="example.w.1">
    <t>house</t>
    <external src="chapter2.xml" include="yes" />
</w>
```

Invalid subdocument:

```
<FoLiA id="chapter2">
    <metadata>
```

```
      <annotations>
        <pos-annotation set=".." />
      </annotations>
    </metadata>
    <text>
      <pos class="N" />
    </text>
</FoLiA>
```

Similarly, using the external element for including annotation layers is invalid too. This is the case because such cases fall in the second category, **Stand-off documents** instead of **Subdocuments**, which follow a slightly different syntax:

Stand-off documents are not included by means of the `external` element, but are associated directly in the declaration section of the main document through an `external` *attribute*. Stand-off documents must replicate at least a minimal part of the structure of the main document and must always have `external="yes"` set. A stand-off document may not introduce new structure elements as was the case with subdocuments. Main document excerpt:

```
<FoLiA id="main">
  <metadata>
    <annotations>
        <pos-annotation set="http://../adhoc-pos.xml" external="
            pos.xml" />
    </annotations>
  </metadata>
  <text>
    <p id="example.p.1">
      <s id="example.p.1.s.1">
        <t>The big house</t>
        <w xml:id="example.p.1.w.1">
          <t>The</t>
        </w>
        <w xml:id="example.p.1.w.2">
          <t>big</t>
        </w>
        <w xml:id="example.p.1.w.3">
          <t>house</t>
        </w>
      </s>
    </p>
  </text>
</FoLiA>
```

The stand-off document would then for example look like this, replicating only a minimal part, the deepest relevant layer, of the structure:

```
<FoLiA id="main" external="yes">
  <metadata>
    <annotations>
        <pos-annotation set="http://../adhoc-pos.xml" />
    </annotations>
  </metadata>
  <text>
    <w xml:id="example.p.1.w.1">
      <pos class="det">
    </w>
    <w xml:id="example.p.1.w.2">
      <pos class="adj">
    </w>
    <w xml:id="example.p.1.w.3">
      <pos class="n">
    </w>
  </text>
</FoLiA>
```

Any other type of annotation present in this document will be ignored. Only the type for which it is expliclitly referred to from the parent document is loaded. This does open up the interesting option of using the same stand-off document for multiple annotation types. Consider the main document declaration were as follows:

```
  <metadata>
    <annotations>
        <pos-annotation set="http://../adhoc-pos.xml" external="
            poslemma.xml" />
        <lemma-annotation set="http://../adhoc-lemma.xml"
            external="poslemma.xml" />
    </annotations>
  </metadata>
</FoLiA>
```

The stand-off document could then include both:

```
<FoLiA id="main" external="yes">
  <metadata>
    <annotations>
        <pos-annotation set="http://../adhoc-pos.xml" />
        <lemma-annotation set="http://../adhoc-lemma.xml" />
    </annotations>
  </metadata>
  <text>
    <w xml:id="example.p.1.w.1">
      <pos class="det">
      <lemma class="the" />
```

```
    </w>
    <w xml:id="example.p.1.w.2">
      <pos class="adj">
      <lemma class="big" />
    </w>
    <w xml:id="example.p.1.w.3">
      <pos class="n">
      <lemma class="house" />
    </w>
  </text>
</FoLiA>
```

Of course, stand-off documents work for span annotation as well. In this case it is always required to specify a full annotation layer. The items referred to with `wref` (or `aref` for that matter) need not exist in the stand-off document as long as they exist in the parent. This is a general principle that applies to all documents marked with `external="yes"`. In the library implementation they share an ID index with their parent.

Consider the following excerpt of the same main document as in the previous examples:

```
  <metadata>
    <annotations>
        <syntax-annotation set="http://../adhoc-syntax.xml"
            external="syntax.xml" />
    </annotations>
  </metadata>
</FoLiA>
```

And here the stand-off document:

```
<FoLiA id="main" external="yes">
  <metadata>
    <annotations>
        <syntax-annotation set="http://../adhoc-syntax.xml"
            external="syntax.xml" />
    </annotations>
  </metadata>
  <text>
    <s xml:id="example.p.1.s.1">
     <syntax>
      <su class="np">
        <su class="det">
          <wref id="example.p.1.w.1" />
        </su>
        <su class="np">
```

```
            <su class="adj">
              <wref id="example.p.1.w.2" />
            </su>
            <su class="n">
              <wref id="example.p.1.w.3" />
            </su>
          </su>
        </su>
      </syntax>
    </s>
  </text>
</FoLiA>
```

Note that in the above example, we did minimal duplication of the structure by copying the sentence element. This allows us to still adhere to the convention of *locality* FoLiA follows, the notion of embedding layers in-line in the deepest elements possible according to the scope. This is, however, not a requirement and the document would still be valid and describe the same situation if the `s` element were omitted alltogether.

There is no declaration necessary for the use of the `external` element or attribute. However, stand-off documents and subdocuments with `external="yes"` may not themselves make use of neither the `external` element in their structure, nor attribute on declarations. That privelege is reserved for the main document. The sole exception are subdocuments not marked with `external="yes"`, as they can act as fully independent. Needless to say, circular references are forbidden in such cases.

Chapter 3

Set Definition Format

Status: Final since v0.10.1 · **Implementations:** pynlpl

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 sets used 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 cannot 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 Class Hierarchy

In FoLiA Set Definitions, classes can be nested to create more complex hierarchies or taxonomy trees, in which both nodes and leaves act as valid classes. Consider the following set definition for named entities, in which the *location* class has been extended into more fine-grained subclasses.

```
<set xml:id="namedentities" type="closed">
  <class xml:id="per" label="Person" />
  <class xml:id="org" label="Organisation" />
  <class xml:id="loc" label="Location">
    <class xml:id="loc.country" label="Country" />
    <class xml:id="loc.street" label="Street" />
    <class xml:id="loc.building" label="Building">
      <class xml:id="loc.building.hospital" label="Hospital" />
      <class xml:id="loc.building.church" label="Church" />
      <class xml:id="loc.building.station" label="Station" />
    </class>
  </class>
</set>
```

It is recommended, but not mandatory, to set the class ID of any nested classes to represent a full path, as a full path makes substring queries possible. FoLiA, however, does not dictate this and neither does it prescribe a delimiter for such paths, so the period in the above example is merely a convention. Each ID, however, does have to be unique in the entire set.

3.5 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`.

3.6 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 with 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 the 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.

```
    <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.

```
    <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>
```

Chapter 4

Querying

4.1 XPath

Considering the fact that FoLiA is an XML-based format, XPath and its derivatives are available as 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 the sake of 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 cannot 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(ancestor-or-self::*/str) and not(@class)]//text()`
  Explanation: The `not(@class)` predicate is important here and makes sure to select only the "current" text content element in case there are multiple text content elements in different classes. (See also Section 2.10.11). The `not(ancestor-or-self::*/morpheme` makes sure morphemes are excluded, `not(ancestor-or-self::*/str` makes sure strings 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-self::*/@auth) 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 cannot use the @set attribute to select, and in the latter case, you must.

4.2 FoLiA Query Language

Whereas XPath is a very generic query language, the FoLiA Query Language (FQL) is a very specific language, designed purely for FoLiA. It allows advanced querying and document editing.

FQL statements are separated by newlines and encoded in UTF-8. The expressions are case sensitive, all keywords are in upper case, all element names and attributes in lower case.

FQL is also strict about parentheses, they are generally either required or forbidden for an expression. Parentheses usually indicate a sub-expression, and it is also used in boolean logic.

As a general rule, it is more efficient to do a single big query than multiple standalone queries.

Note that for readability, queries may have been split on multiple lines in the presentation here, whereas in reality they should be on one.

4.2.1 Global variables

- `SET variable=value` - Sets global variables that apply to all statements that follow. String values need to be in double quotes. Available variables are:
- `annotator` - The name of the annotator
- `annotatortype` - The type of the annotator, can be *auto* or *manual*

Usually your queries on a particular annotation type are limited to one specific set. To prevent having to enter the set explicitly in your queries, you can set defaults. The annotation type corresponds to a FoLiA element::

```
DEFAULTSET entity https://raw.githubusercontent.com/proycon/
    folia/master/setdefinitions/namedentitycorrection.foliaset.
    xml
```

If the FoLiA document only has one set of that type anyway, then this is not even necessary and the default will be automatically set.

4.2.2 Declarations

All annotation types in FoLiA need to be declared. FQL does this for you automatically. If you make an edit of a previously undeclared set, it will be declared for you. These default declarations will never assign default annotators or annotator types.

Explicit declarations are possible using the **DECLARE** keyword followed by the annotation type you want to declare, this represented the tag of the respective FoLiA annotation element::

```
DECLARE entity OF "https://github.com/proycon/folia/blob/master/
    setdefinitions/namedentities.foliaset.xml"
WITH annotator = "me" annotatortype = "manual"
```

Note that the statement must be on one single line, it is split here only for ease of presentation.

The **WITH** clause is optional, the set following the **OF** keyword is mandatory.

Declarations may be chained, i.e. multiple **DECLARE** statements may be issued on one line, as well as prepended to action statements (see next section).

4.2.3 Actions

The core part of an FQL statement consists of an action verb, the following are available

- `SELECT <focus expression> [<target expression>]` - Selects an annotation
- `DELETE <focus expression> [<target expression>]` - Deletes an annotation
- `EDIT <focus expression> [<assignment expression>] [<target expression>]` - Edits an existing annotation
- `ADD <focus expression> <assignment expression> <target expression>` - Adds an annotation (to the target expression)
- `APPEND <focus expression> <assignment expression> <target expression>` - Inserts an annotation after the target expression
- `PREPEND <focus expression> <assignment expression> <target expression>` - Inserts an annotation before the target expression

Following the action verb is the focus expression, this starts with an annotation type, which is equal to the FoLiA XML element tag. The set is specified using `OF <set>` and/or the ID with `ID <id>`. An example:

```
pos OF "http://some.domain/some.folia.set.xml"
```

If an annotation type is already declared and there is only one in document, or if the **DEFAULTSET** statement was used earlier, then the **OF** statement can be omitted and will be implied and detected automatically. If it is ambiguous, an error will be raised (rather than applying the query regardless of set).

To further filter a the focus, the expression may consist of a **WHERE** clause that filters on one or more FoLiA attributes:

- class
- annotator
- annotatortype

- n
- confidence
- src
- speaker
- begintime
- endtime

The following attributes are also available on when the elements contains text and/or phonetic/phonological content:

- text
- phon

The **WHERE** statement requires an operator (=,! =,>,<,<=,>=,CONTAINS,MATCHES), the **AND**, **OR** and **NOT** operators are available (along with parentheses) for grouping and boolean logic. The operators must never be glued to the attribute name or the value, but have spaces left and right.

We can now show some examples of full queries with some operators:

- `SELECT pos OF "http://some.domain/some.folia.set.xml"`
- `SELECT pos WHERE class = "n" AND annotator = "johndoe"`
- `DELETE pos WHERE class = "n" AND annotator != "johndoe"`
- `DELETE pos WHERE class = "n" AND annotator CONTAINS "john"`
- `DELETE pos WHERE class = "n" AND annotator MATCHES "^john$"`

The **ADD** and **EDIT** change actual attributes, this is done in the *assignment expression* that starts with the **WITH** keyword. It applies to all the common FoLiA attributes like the **WHERE** keyword, but has no operator or boolean logic, as it is a pure assignment function.

SELECT and DELETE only support WHERE, EDIT supports both WHERE and WITH, if both are use they than WHERE is always before WITH. the ADD action

supports only WITH. If an EDIT is done on an annotation that can not be found, and there is no WHERE clause, then it will fall back to ADD.

Here is an **EDIT** query that changes all nouns in the document to verbs:

```
EDIT pos WHERE class = "n" WITH class "v" AND annotator = "
    johndoe"
```

The query is fairly crude as it still lacks a *target expression*: A *target expression* determines what elements the focus is applied to, rather than to the document as a whole, it starts with the keyword **FOR** and is followed by either an annotation type (i.e. a FoLiA XML element tag) *or* the ID of an element. The target expression also determines what elements will be returned. More on this in a later section.

The following FQL query shows how to get the part of speech tag for a word:

```
SELECT pos FOR ID mydocument.word.3
```

Or for all words:

```
SELECT pos FOR w
```

The **ADD** action almost always requires a target expression:

```
ADD pos WITH class "n" FOR ID mydocument.word.3
```

Multiple targets may be specified, comma delimited:

```
ADD pos WITH class "n" FOR ID mydocument.word.3 , ID myword.
    document.word.25
```

The target expression can again contain a **WHERE** filter:

```
SELECT pos FOR w WHERE class != "PUNCT"
```

Target expressions, starting with the **FOR** keyword, can be nested:

```
SELECT pos FOR w WHERE class != "PUNCT" FOR event WHERE class =
    "tweet"
```

You may also use the SELECT keyword without focus expression, but only with a target expression. This is particularly useful when you want to return multiple distinct elements, for instance by ID:

```
SELECT FOR ID mydocument.word.3 , ID myword.document.word.25
```

The **SELECT** keyword can also be used with the special **ALL** selector that selects all elemens in the scope, the following two statement are identical and will return all elements in the document:

```
SELECT ALL
SELECT FOR ALL
```

It can be used at deeper levels too, the following will return everything under all words:

```
SELECT ALL FOR w
```

Target expressions are vital for span annotation, the keyword **SPAN** indicates that the target is a span (to do multiple spans at once, repeat the SPAN keyword again), the operator & is used for consecutive spans, whereas , is used for disjoint spans:

```
ADD entity WITH class "person" FOR SPAN ID mydocument.word.3 &
    ID myword.document.word.25
```

This works with filters too, the & operator enforced a single consecutive span:

```
ADD entity WITH class "person" FOR SPAN w WHERE text = "John" &
    w WHERE text = "Doe"
```

Remember we can do multiple at once:

```
ADD entity WITH class "person" FOR SPAN w WHERE text = "John" &
    w WHERE text = "Doe"
SPAN w WHERE text = "Jane" & w WHERE text = "Doe"
```

The **HAS** keyword enables you to descend down in the document tree to siblings. Consider the following example that changes the part of speech tag to "verb", for all occurrences of words that have lemma "fly". The parentheses are mandatory for a **HAS** statement:

```
EDIT pos OF "someposset" WITH class = "v" FOR w WHERE (lemma OF
    "somelemmaset" HAS class "fly")
```

Target expressions can be former with either **FOR** or with **IN**, the difference is that **IN** is much stricter, the element has to be a direct child of the element in the **IN** statement, whereas **FOR** may skip intermediate elements. In analogy with XPath, **FOR** corresponds to // and **IN** corresponds to /. **FOR** and **IN** may be nested and mixed at will. The following query would most likely not yield any results because there are likely to be paragraphs and/or sentences between the wod and event structures:

```
SELECT pos FOR w WHERE class != "PUNCT" IN event WHERE class = "
    tweet"
```

Multiple actions can be combined, all share the same target expressions:

```
ADD pos WITH class "n" ADD lemma WITH class "house" FOR w WHERE
    text = "house" OR text = "houses"
```

It is also possible to nest actions, use parentheses for this, the nesting occurs after any WHERE and WITH statements:

```
ADD w ID mydoc.sentence.1.word.1 (ADD t WITH text "house" ADD
    pos WITH class "n") FOR ID mydoc.sentence.1
```

Though explicitly specified here, IDs will be automatically generated when necessary and not specified.

The **ADD** action has two cousins: **APPEND** and **PREPEND**. Instead of adding something in the scope of the target expression, they either append or prepend an element, so the inserted element will be a sibling:

```
APPEND w (ADD t WITH text "house") FOR w WHERE text = "the"
```

This above query appends/inserts the word "house" after every definite article.

4.2.4 Text

Our previous examples mostly focussed on part of speech annotation. In this section we look at text content, which in FoLiA is an annotation element too (t).

Here we change the text of a word:

```
EDIT t WITH text = "house" FOR ID mydoc.word.45
```

Here we edit or add (recall that EDIT falls back to ADD when not found and there is no further selector) a lemma and check on text content:

```
EDIT lemma WITH class "house" FOR w WHERE text = "house" OR text
    = "houses"
```

You can use WHERE text on all elements, it will cover both explicit text content as well as implicit text content, i.e. inferred from child elements. If you want to be really explicit you can do:

```
EDIT lemma WITH class "house" FOR w WHERE (t HAS text = "house")
```

Advanced:

Such syntax is required when covering texts with custom classes, such as OCRed or otherwise pre-normalised text. Consider the following OCR correction:

```
ADD t WITH text = "spell" FOR w WHERE (t HAS text = "5pe11" AND
    class = "OCR" )
```

4.2.5 Query Response

We have shown how to do queries but not yet said anything on how the response is returned. This is regulated using the **RETURN** keyword:

- **RETURN focus** (default)
- **RETURN parent** - Returns the parent of the focus
- **RETURN target** or **RETURN inner-target**
- **RETURN outer-target**
- **RETURN ancestor-target**

The default focus mode just returns the focus. Sometimes, however, you may want more context and may want to return the target expression instead. In the following example returning only the pos-tag would not be so interesting, you are most likely interested in the word to which it applies:

```
SELECT pos WHERE class = "n" FOR w RETURN target
```

When there are nested FOR/IN loops, you can specify whether you want to return the inner one (highest granularity, default) or the outer one (widest scope). You can also decide to return the first common structural ancestor of the (outer) targets, which may be specially useful in combination with the **SPAN** keyword.

The return type can be set using the **FORMAT** statement:

- **FORMAT xml** - Returns FoLiA XML, the response is contained in a simple `<results><result/></results>` structure.

- **FORMAT single-xml** - Like above, but returns pure unwrapped FoLiA XML and therefore only works if the response only contains one element. An error will be raised otherwise.
- **FORMAT json** - Returns JSON list
- **FORMAT single-json** - Like above, but returns a single element rather than a list. An error will be raised if the response contains multiple.
- **FORMAT python** - Returns a Python object, can only be used when directly querying the FQL library without the document server
- **FORMAT flat** - Returns a parsed format optimised for FLAT. This is a JSON reply containing an HTML skeleton of structure elements (key html), parsed annotations (key annotations). If the query returns a full FoLiA document, then the JSON object will include parsed set definitions, (key setdefinitions), and declarations.

The **RETURN** statement may be used standalone or appended to a query, in which case it applies to all subsequent queries. The same applies to the **FORMAT** statement, though an error will be raised if distinct formats are requested in the same HTTP request.

When context is returned in *target* mode, this can get quite big, you may constrain the type of elements returned by using the **REQUEST** keyword, it takes the names of FoLiA XML elements. It can be used standalone so it applies to all subsequent queries:

```
REQUEST w, t , pos , lemma
```

..or after a query:

```
SELECT pos FOR w WHERE class!="PUNCT" FOR event WHERE class="
    tweet" REQUEST w, pos , lemma
```

Two special uses of request are `REQUEST ALL` (default) and `REQUEST NOTHING`, the latter may be useful in combination with **ADD**, **EDIT** and **DELETE**, by default it will return the updated state of the document.

Note that if you set REQUEST wrong you may quickly end up with empty results.

4.2.6 Span Annotation

Selecting span annotations is identical to token annotation. You may be aware that in FoLiA span annotation elements are technically stored in a separate stand-off layers, but you can forget this fact when composing FQL queries and can access them right from the elements they apply to.

The following query selects all named entities (of an actual rather than a fictitious set for a change) of people that have the name John:

```
SELECT entity OF "https://github.com/proycon/folia/blob/master/
    setdefinitions/namedentities.foliaset.xml"
WHERE class = "person" FOR w WHERE text = "John"
```

Or consider the selection of noun-phrase syntactic units (su) that contain the word house:

```
SELECT su WHERE class = "np" FOR w WHERE text CONTAINS "house"
```

Note that if the **SPAN** keyword were used here, the selection would be exclusively constrained to single words "John":

```
SELECT entity WHERE class = "person" FOR SPAN w WHERE text = "
    John"
```

We can use that construct to select all people named John Doe for instance:

```
SELECT entity WHERE class = "person" FOR SPAN w WHERE text = "
    John" & w WHERE text = "Doe"
```

Span annotations like syntactic units are typically nested trees, a tree query such as "//pp/np/adj" can be represented as follows. Recall that the **IN** statement starts a target expression like **FOR**, but is stricter on the hierarchy, which is what we would want here:

```
SELECT su WHERE class = "adj" IN su WHERE class = "np" IN su
    WHERE class = "pp"
```

In such instances we may be most interested in obtaining the full PP:

```
SELECT su WHERE class = "adj" IN su WHERE class = "np" IN su
    WHERE class = "pp" RETURN outer-target
```

The **EDIT** action is not limited to editing attributes, sometimes you want to alter the element of a span. A separate **RESPAN** keyword (without FOR/IN/WITH)

accomplishes this. It takes the keyword **RESPAN** which behaves the same as a **FOR SPAN** target expression and represents the new scope of the span, the normal target expression represents the old scope:

```
EDIT entity WHERE class= "person" RESPAN ID word.1 & ID word.2
    FOR SPAN ID word.1 & ID word.2 & ID word.3
```

WITH statements can be used still too, they always preceed **RESPAN**:

```
EDIT entity WHERE class= "person" WITH class="location" RESPAN
    ID word.1 & ID word.2 FOR SPAN ID word.1 & ID word.2 & ID
    word.3
```

4.2.7 Corrections and Alternatives

Both FoLiA and FQL have explicit support for corrections and alternatives on annotations. A correction is not a blunt substitute of an annotation of any type, but the original is preserved as well. Similarly, an alternative annotation is one that exists alongside the actual annotation of the same type and set, and is not authoritative.

The following example is a correction but not in the FoLiA sense, it bluntly changes part-of-speech annotation of all occurrences of the word "fly" from "n" to "v", for example to correct erroneous tagger output:

```
EDIT pos WITH class "v" WHERE class = "n" FOR w WHERE text = "
    fly"
```

Now we do the same but as an explicit correction:

```
EDIT pos WITH class "v" WHERE class = "n" (AS CORRECTION OF "
    some/correctionset" WITH class "wrongpos")
FOR w WHERE text = "fly"
```

Another example in a spelling correction context, we correct the misspelling *concous* to *conscious**:

```
EDIT t WITH text "conscious" (AS CORRECTION OF "some/
    correctionset" WITH class "spellingerror")
FOR w WHERE text = "concous"
```

The **AS CORRECTION** keyword (always in a separate block within parentheses) is used to initiate a correction. The correction is itself part of a set with a class that indicates the type of correction.

Alternatives are simpler, but follow the same principle:

```
EDIT pos WITH class "v" WHERE class = "n" (AS ALTERNATIVE) FOR w
    WHERE text = "fly"
```

Confidence scores are often associationed with alternatives:

```
EDIT pos WITH class "v" WHERE class = "n" (AS ALTERNATIVE WITH
    confidence 0.6)
FOR w WHERE text = "fly"
```

The **AS** clause is also used to select alternatives rather than the authoritative form, this will get all alternative pos tags for words with the text "fly":

```
SELECT pos (AS ALTERNATIVE) FOR w WHERE text = "fly"
```

If you want the authoritative tag as well, you can chain the actions. The same target expression (FOR..) always applies to all chained actions, but the AS clause applies only to the action in the scope of which it appears:

```
SELECT pos SELECT pos (AS ALTERNATIVE) FOR w WHERE text = "fly"
```

Filters on the alternative themselves may be applied as expected using the WHERE clause:

```
SELECT pos (AS ALTERNATIVE WHERE confidence > 0.6) FOR w WHERE
    text = "fly"
```

Note that filtering on the attributes of the annotation itself is outside of the scope of the AS clause:

```
SELECT pos WHERE class = "n" (AS ALTERNATIVE WHERE confidence >
    0.6) FOR w WHERE text = "fly"
```

Corrections by definition are authoritative, so no special syntax is needed to obtain them. Assuming the part of speech tag is corrected, this will correctly obtain it, no AS clause is necessary:

```
SELECT pos FOR w WHERE text = "fly"
```

Adding **AS CORRECTION** will only enforce to return those that were actually corrected:

```
SELECT pos (AS CORRECTION) FOR w WHERE text = "fly"
```

However, if you want to obtain the original prior to correction, you can do so using **AS CORRECTION ORIGINAL**:

```
SELECT pos (AS CORRECTION ORIGINAL) FOR w WHERE text = "fly"
```

FoLiA does not just distinguish corrections, but also supports suggestions for correction. Envision a spelling checker suggesting output for misspelled words, but leaving it up to the user which of the suggestions to accept. Suggestions are not authoritative and can be obtained in a similar fashion by using the **SUGGESTION** keyword:

```
SELECT pos (AS CORRECTION SUGGESTION) FOR w WHERE text = "fly"
```

Note that **AS CORRECTION** may take the **OF** keyword to specify the correction set, they may also take a **WHERE** clause to filter:

```
SELECT t (AS CORRECTION OF "some/correctionset" WHERE class = "
    confusible") FOR w
```

The **SUGGESTION** keyword can take a WHERE filter too:

```
SELECT t (AS CORRECTION OF "some/correctionset" WHERE class = "
    confusible" SUGGESTION WHERE confidence > 0.5) FOR w
```

To add a suggestion for correction rather than an actual authoritative correction, you can do:

```
EDIT pos (AS CORRECTION OF "some/correctionset" WITH class "
    poscorrection" SUGGESTION class "n") FOR w ID some.word.1
```

The absence of a WITH statement in the action clause indicates that this is purely a suggestion. The actual suggestion follows the **SUGGESTION** keyword.

Any attributes associated with the suggestion can be set with a **WITH** statement after the suggestion:

```
EDIT pos (AS CORRECTION OF "some/correctionset" WITH class "
    poscorrection" SUGGESTION class "n" WITH confidence 0.8) FOR
    w ID some.word.1
```

Even if a **WITH** statement is present for the action, making it an actual correction, you can still add suggestions:

```
EDIT pos WITH class "v" (AS CORRECTION OF "some/correctionset"
    WITH class "poscorrection" SUGGESTION class "n" WITH
    confidence 0.8) FOR w ID some.word.1
```

The **SUGGESTION** keyword can be chaineed to add multiple suggestions at once:

```
EDIT pos (AS CORRECTION OF "some/correctionset" WITH class "
    poscorrection"
SUGGESTION class "n" WITH confidence 0.8
SUGGESTION class "v" wITH confidence 0.2) FOR w ID some.word.1
```

Another example in a spelling correction context:

```
EDIT t (AS CORRECTION OF "some/correctionset" WITH class "
    spellingerror"
SUGGESTION text "conscious" WITH confidence 0.8 SUGGESTION text
    "couscous" WITH confidence 0.2)
FOR w WHERE text = "concous"
```

A similar construction is available for alternatives as well. First we establish that the following two statements are identical:

```
EDIT pos WHERE class = "n" WITH class "v" (AS ALTERNATIVE WITH
    confidence 0.6) FOR w WHERE text = "fly"
EDIT pos WHERE class = "n" (AS ALTERNATIVE class "v" WITH
    confidence 0.6) FOR w WHERE text = "fly"
```

Specifying multiple alternatives is then done by simply adding enother **ALTERNATIVE** clause:

```
EDIT pos (AS ALTERNATIVE class "v" WITH confidence 0.6
    ALTERNATIVE class "n" WITH confidence 0.4 ) FOR w WHERE text
    = "fly"
```

When a correction is made on an element, all annotations below it (recursively) are left intact, i.e. they are copied from the original element to the new correct element. The same applies to suggestions. Moreover, all references to the original element, from for instance span annotation elements, will be made into references to the new corrected elements.

This is not always what you want, if you want the correction not to have any annotations inherited from the original, simply use **AS BARE CORRECTION** instead of **AS CORRECTION**.

You can also use **AS CORRECTION** with **ADD** and **DELETE**.

The most complex kind of corrections are splits and merges. A split separates a structure element such as a word into multiple, a merge unifies multiple structure elements into one.

In FQL, this is achieved through substitution, using the action **SUBSTITUTE**:

```
SUBSTITUTE w WITH text "together" FOR SPAN w WHERE text="to" \
& w WHERE text="gether"
```

Subactions are common with SUBSTITUTE, the following is equivalent to the above:

```
SUBSTITUTE w (ADD t WITH text "together") FOR SPAN w WHERE \
text="to" & w WHERE text="gether"
```

To perform a split into multiple substitutes, simply chain the SUBSTITUTE clause:

```
SUBSTITUTE w WITH text "each" SUBSTITUTE w WITH \
TEXT "other" FOR w WHERE text="eachother"
```

Like **ADD**, both **SUBSTITUTE** may take assignments (**WITH**), but no filters (**WHERE**).

You may have noticed that the merge and split examples were not corrections in the FoLiA-sense; the originals are removed and not preserved. Let's make it into proper corrections:

```
SUBSTITUTE w WITH text "together"
(AS CORRECTION OF "some/correctionset" WITH class "spliterror")
FOR SPAN w WHERE text="to" & w WHERE text="gether"
```

And a split:

```
SUBSTITUTE w WITH text "each" SUBSTITUTE w WITH text "other"
(AS CORRECTION OF "some/correctionset␣WITH␣class␣"runonerror")
FOR␣w␣WHERE␣text="eachother"
```

To make this into a suggestion for correction instead, use the **SUGGESTION** folloed by **SUBSTITUTE**, inside the **AS** clause, where the chain of substitute statements has to be enclosed in parentheses:

```
SUBSTITUTE (AS CORRECTION OF "some/correctionset" WITH class "
    runonerror" SUGGESTION (SUBTITUTE w WITH text "each"
    SUBSTITUTE w WITH text "other") )
FOR w WHERE text="eachother"
```

4.2.8 Dealing with context

We have seen that with the **FOR** keyword we can move to bigger elements in the FoLiA document, and with the **HAS** keyword we can move to siblings. There

are several *context keywords* that give us all the tools we need to peek at the context. Like **HAS** expressions, these need always be enclosed in parentheses.

For instance, consider a part-of-speech tagging scenario. If we have a word where the left neighbour is a determiner, and the right neighbour a noun, we can be pretty sure the word under our consideration (our target expression) is an adjective. Let's add the pos tag:

```
EDIT pos WITH class = "adj" FOR w WHERE (PREVIOUS w WHERE (pos
    HAS class == "det")) AND (NEXT w WHERE (pos HAS class == "n")
    )
```

You may append a number directly to the **PREVIOUS**/**NEXT** modifier if you're interested in further context, or you may use **LEFTCONTEXT**/**RIGHTCONTEXT**/**CONTEXT** if you don't care at what position something occurs:

```
EDIT pos WITH class = "adj" FOR w WHERE (PREVIOUS2 w WHERE (pos
    HAS class == "det")) AND (PREVIOUS w WHERE (pos HAS class ==
    "adj")) AND (RIGHTCONTEXT w WHERE (pos HAS class == "n"))
```

Instead of the **NEXT** and **PREVIOUS** keywords, a target expression can be used with the **SPAN** keyword and the **&** operator:

```
SELECT FOR SPAN w WHERE text = "the" & w WHERE (pos HAS class ==
     "adj") & w WHERE text = "house"
```

Within a **SPAN** keyword, an **expansion expression** can be used to select any number, or a certain number, of elements. You can do this by appending curly braces after the element name (but not attached to it) and specifying the minimum and maximum number of elements. The following expression selects from zero up to three adjectives between the words "the" and "house":

```
SELECT FOR SPAN w WHERE text = "the" & w {0,3} WHERE (pos HAS
    class == "adj") & w WHERE text = "house"
```

If you specify only a single number in the curly braces, it will require that exact number of elements. To match at least one word up to an unlimited number, use an expansion expression such as `{1,}`.

If you are now perhaps tempted to use the FoLiA document server and FQL for searching through large corpora in real-time, then be advised that this is not a good idea. It will be prohibitively slow on large datasets as this requires smart indexing, which this document server does not provide. You can therefore not do this real-time, but perhaps only as a first step to build an actual search index.

Other modifiers are PARENT and and ANCESTOR. PARENT will at most go one element up, whereas ANCESTOR will go on to the largest element:

```
SELECT lemma FOR w WHERE (PARENT s WHERE  text CONTAINS "wine")
```

Instead of **PARENT**, the use of a nested **FOR** is preferred and more efficient:

```
SELECT lemma FOR w FOR s WHERE text CONTAINS "wine"
```

Let's revisit syntax trees for a bit now we know how to obtain context. Imagine we want an NP to the left of a PP:

```
SELECT su WHERE class = "np" AND (NEXT su WHERE class = "pp")
```

... and where the whole thing is part of a VP:

```
SELECT su WHERE class = "np" AND (NEXT su WHERE class = "pp") IN
    su WHERE class = "vp"
```

... and return that whole tree rather than just the NP we were looking for:

```
SELECT su WHERE class = "np" AND (NEXT su WHERE class = "pp") IN
    su WHERE class = "vp" RETURN target
```

4.2.9 Shortcuts

Classes are prevalent all throughout FoLiA, it is very common to want to select on classes. To select words with pos tag "n" for example you can do:

```
SELECT w WHERE (pos HAS class = "n")
```

Because this is so common, there is a shortcut. Specify the annotation type directly preceeded by a colon, and a HAS statement that matches on class will automatically be constructed:

```
SELECT w WHERE :pos = "n"
```

The two statements are completely equivalent.

Another third alternative to obtain the same result set is to use a target expression:

```
SELECT pos WHERE class = "n" FOR w RETURN target
```

This illustrates that there are often multiple ways of obtaining the same result set. Due to lazy evaluation in the FQL library, there is not much difference performance-wise.

Another kind of shortcut exists for setting text on structural elements. The explicit procedure to add a word goes as follows:

```
ADD w (ADD t WITH text "hello") IN ID some.sentence
```

The shortcut is:

```
ADD w WITH text "hello" IN ID some.sentence
```

Appendix A

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, the latest version of which can always be obtained from the following URL:
`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.

A.1 Extending FoLiA

If you add your own custom tags, always make sure you use a different XML namespace rather than the default FoLiA namespace! FoLiA libraries are designed to ignore and discard other namespaces.

In case you cannot find a suitable way to encode your particular annotation in the latest version of the FoLiA specificaton, we urge you to write to `proycon@anaproy.nl` so that we can discuss whether FoLiA needs to be expanded to accommodate your particular annotation.

Appendix B

Implementations

Currently, the following FoLiA implementations exist. 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. See `https://pypi.python.org/pypi/PyNLPl/` for sources and documentation.
2. `libfolia` - A FoLiA library in $C++$. Obtain from `https://proycon.github.io/folia`

Both libraries are shipped as part of our **LaMachine** software distribution[1].

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

[1] `ttps://proycon.github.io/LaMachine`

	PyNLPl	libfolia
Programming Language	python	C++
From official specification[2]	yes	yes
Annotation Types		
Text content	yes	yes
Phonetic/phonological content (v0.12)	yes	yes
Structure annotation	yes	yes
Token annotation	yes	yes
Span annotation	yes	yes
Morphology	yes	yes
Phonology (v0.12)	yes	yes
Alternatives	yes	yes
Corrections	yes	yes
Text Markup	yes	yes
Hyperlinks on markup (v0.11.3)	yes	yes
Alignments	yes	yes
Strings	yes	yes
Search & Query		
FoLiA Query Language (FQL)	yes	no
Document.findwords() method	yes	yes
Validation		
Shallow validation	yes	yes
RelaxNG schema generation	yes	no
Set definition support	yes	no
Deep validation	not yet	no
Quality Control		
Unit/integration tests	yes	yes
Legacy		
IMDI interpretation	partial[3]	no
D-Coi read compatibility	partial[4]	no

[2]These libraries draw a large part of their implementation from the common external FoLiA specification to keep them in sync better

[3]Only for in-document IMDI

[4]Only basic elements, no List, Figure, etc..

Appendix C

FoLiA Tools

C.1 Introduction

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

- `foliavalidator` – Tests if documents are valid FoLiA XML. **Always use this to test your documents if you produce your own FoLiA documents!**
- `foliaquery` – Advanced query tool that searches FoLiA documents for a specified pattern, or modifies a document according to the query. Supports FQL (FoLiA Query Language) and CQL (Corpus Query Language).
- `folia2txt` – Convert FoLiA XML to plain text (pure text, without any annotations)
- `folia2annotatedtxt` – Like above, but produces output simple token annotations inline, by appending them directly to the word using a specific delimiter.
- `folia2columns` – This conversion tool reads a FoLiA XML document and produces a simple columned output format (including CSV) in which each token appears on one line. Note that only simple token annotations are supported and a lot of FoLiA data can not be intuitively expressed in a simple columned format!

- `folia2html` – Converts a FoLiA document to a semi-interactive HTML document, with limited support for certain token annotations.
- `folia2dcoi` – Convert FoLiA XML to D-Coi XML (only for annotations supported by D-Coi)
- `dcoi2folia` – Convert D-Coi XML to FoLiA XML
- `rst2folia` – Convert ReStructuredText, a lightweight non-intrusive text markup language, to FoLiA, using docutils[1].

All of these tools are written in Python, and thus require a Python (2.7, 3 or higher) installation to run. More tools are added as time progresses.

C.2 Installation

The FoLiA tools are published to the Python Package Index and can be installed effortlessly using `pip`, from the command-line, type:

```
$ pip install folia-tools
```

Add `sudo` to install it globally on your system, if you install locally, we strongly recommend you use virtualenv to make a self-contained Python environment.

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

On Debian/Ubuntu-based systems:

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

On RedHat-based systems:

```
$ yum install python-pip
```

[1] http://docutils.sourceforge.net/

On Arch Linux systems:

```
$ pacman -Syu python-pip
```

On Mac OS X and Windows we recommend you install Anaconda[2] or another Python distribution.

The FoLiA tools are also included as part of our **LaMachine** software distribution[3].

The source code is hosted on github (`https://github.com/proycon/folia`), once downloaded and extracted, it can also be installed using `python setup.py install`.

C.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.

[2] See http://continuum.io/

[3] `ttps://proycon.github.io/LaMachine`

Bibliography

[1] Eneko Agirre, Xabier Artola, Arantza Diaz de Ilarraza, German Rigau, Aitor Soroa, 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, and Michael Sperberg-McQueen. Extensible Markup Language (XML) 1.0. Technical report, W3C, 2 1998.

[5] Daan Broeder, Oliver Schonefeld, Thorsten Trippel, Dieter Van Uytvanck, and Andreas Witt. A pragmatic approach to XML interoperability the Component Metadata Infrastructure (CMDI). In *Balisage: The Markup Conference 2011*, volume 7, 2011.

[6] Lou Burnard and Syd Bauman, editors. *TEI P5: Guidelines for Electronic Text Encoding and Interchange*. Text Encoding Initiative Consortium, 2007.

[7] 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.

[8] Frank van Eynde. Part of speech tagging en lemmatisering van het Corpus Gesproken Nederlands. Technical report, Centrum voor Computerlinguïstiek, K.U. Leuven, February 2004.