FoLiA: Format for Linguistic Annotation

v0.1.2

Maarten van Gompel
ILK Research Group
Tilburg center for Cognition and Communication
Tilburg University

February 3, 2011

Contents

1 Introduction 3

2 Format 7

2.1 Global Structure 7

2.2 Identifiers 8

2.3 Structure Elements 8

2.4 Content Elements 8

2.5 Paradigm & Terminology 12

2.6 Annotation Declarations 13

2.7 Token Annotation 14

2.7.1 Part of Speech Annotation 15

2.7.2 Lemma Annotation 16

2.7.3 Lexical Semantic Sense Annotation 16

2.7.4 Domain Tags 17

2.7.5 Lexical Annotation 17

2.7.6 Corrections 18

2.7.7 Morphological Analysis 23

2.8 Alternative Token Annotations 23

2.9 Span Annotation 24

2.9.1 Entities 24

2.9.2 Syntax 25

2.9.3 Dependency Relations 26

2.9.4 Chunking 28

2.9.5 Semantic roles 29

2.9.6 Alterative Span Annotations 29

3 Parsing & Querying 32

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., and extended for richer annotation at the ILK research group, Tilburg University. The D-Coi format is designed for used by the DCOI corpus, as well as by its successor; the SoNaR corpus [6].

FoLiA is an XML-based[4] annotation format, suitable for representing written language resources such as corpora. Its goal is to unify a variety of linguistic annotations in one single rich format, without committing to any particular standard annotation set. Instead, it seeks to accommodate any desired system or tagset, and so to offer maximum flexibility. This makes FoLiA language independent. Due to its generalised set up, it is easy to extent the FoLiA format to suit your custom needs for linguistic annotation.

XML is an inherently hierarchic format. FoLiA does justice to this by maximally utilising a hierarchic, inline, setup. We inherit from the D-Coi format, which posits to be loosely based on a minimal subset of TEI[5]. Because of the introduction of a new and 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.

In contrast to the D-Coi format, the FoLiA format introduces annotation layers separate from the token-based skeleton structure, to capture structured linguistic annotations such as syntactic parses. This is to provide FoLiA with the necessary flexibility and extensibility. Inspiration for this was in part obtained from the Kyoto Annotation Format [1].

The FoLiA format features the following:

- Open-source
- XML-based, validation against XML schema.
- Full Unicode support; UTF-8 encoded.
- Document structure consists of divisions, paragraphs, sentences and words/tokens.
- Can encode both tokenised as well as untokenised text + partial reconstructability of untokenised form even after tokenisation.
- Support for crude token categories (word, punctuation, number, etc)
- Explicit support for encoding quotations
- Provenance support for all linguistic annotations: annotator, type (automatic or manual), time.
- Support for alternative annotations, optionally with associated confidence values.
- Adaptable to different tag-sets.
- Agnostic with regard to metadata. CMDI is recommended, but alternatives like IMDI can also be used.

It supports the following linguistic annotations:

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

- Syntactic Parses
- Dependency Relations
- Chunking

In later stages, the following may be added:

- Morphological Analysis
- Semantic Role Labelling
- Co-reference
- Topic Segmentation
- Authorship Attribution

FoLiA support will be incorporated directly into the following ILK sofware:

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

And it may be used in the following corpora:

- SoNaR (yet to be confirmed)
- DutchSemCor (based primarily on SoNaR)

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.

Development Notes
This is all still subject to debate and change and may be a bit pretentious at this stage.

Chapter 2

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. An elaborate XSLT stylesheet will be provided in order to be able to instantly view FoLiA documents in any modern web browser.

```
<?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" xml:id="example">
  <!-- (Here IMDI or CMDI metadata can be inserted) -->
  <annotations>
      ...
  </annotations>
  <text xml:id="example.text">
   <gap></gap>
   <body>
     ...
   </body>
   <gap></gap>
  </text>
</FoLiA>
```

The body contains the to-be-annotated material; more about this later. The gap tags are optional and contain unannotated front matter or back matter. [2]

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. The identifiers are constructed in the same way as in the D-Coi format, thus retaining full compatibility if a D-Coi document is converted to FoLiA, any external references to any entity in these documents will remain intact.

Identifiers in D-Coi and FoLiA are cumulative and are usually formed by appending the elements name, a period, and a sequence number, to the identifier of a parent element higher in the hierarchy.

The base of all identifiers is that of the document itself, as encoded in `xml:id` attribute of the root `FoLiA` element. This is a unique ID by which the document is identifiable. We choose the identifier *example* for all of the examples in this manual. By convention, the XML file should then ideally be named: `example.xml`.

Identifiers are very important and used throughout the FoLiA format. They enable external resources and database to easily point to a specific part of the document or its annotation. FoLiA has been set up in such a way that *identifiers should never ever 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 Structure Elements

Within the document body, the structure elements div0, div1, div2, etc.. can be used to create divisions and subdivisions. They stem from D-Coi and are unmodified in FoLiA. These divisions are not mandatory, but may be used to mark extra structure.

2.4 Content Elements

Content elements occur within the body. The following exist:

- `head` - Header
- `p` - Paragraph
- `s` - Sentence
- `w` - Word (token)
- `quote` - Quote

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

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

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

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

```
<p xml:id="example.p.1">
   <s xml:id="example.p.1.s.1">
       <w xml:id="example.p.1.s.1.w.1"><t>This</t></w>
       <w xml:id="example.p.1.s.1.w.2"><t>is</t></w>
       ...
       <w xml:id="example.p.1.s.1.w.8" space="no"><t>sentence</t></w>
       <w xml:id="example.p.1.s.1.w.9"><t>.</t></w>
   </s>
</p>
<p xml:id="example.p.2">
   <s xml:id="example.p.2.s.1">
       <w xml:id="example.p.2.s.1.w.1"><t>This</t></w>
       <w xml:id="example.p.2.s.1.w.2"><t>is</t></w>
       ..
       <w xml:id="example.p.2.s.1.w.5" space="no"><t>paragraph</t></w>
       <w xml:id="example.p.2.s.1.w.6"><t>.</t></w>
   </s>
   <s xml:id="example.p.2.s.2">
       <w xml:id="example.p.2.s.2.w.1"><t>This</t></w>
       <w xml:id="example.p.2.s.2.w.2"><t>one</t></w>
       ..
       <w xml:id="example.p.2.s.2.w.5" space="no"><t>sentences</t></w>
       <w xml:id="example.p.2.s.2.w.6"><t>.</t></w>
   </s>
</p>
```

The deepest content element should always contain a text element (t) which holds the actual textual content. The necessity of having a text element shall become apparent as you progress through this documentation; there can be many different token annotations under a word element (w).

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 a 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 element does too. It is 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 reconstructability of the sentence in its untokenised form. Reconstructing sentences is generally preferred to grabbing them from the text element at the paragraph or sentence level, as there may be corrections on the token level.

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.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">
```

```
    <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>
```

The paragraph elements may be omitted if a document is described that does not distinguish paragraphs but only sentences. The identifiers of course change accordingly then. Sentences however should never be omitted; 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 may hold sentences.

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

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

A quote may consist of one or more sentences, but may also consist of mere tokens. The token identifiers in all cases simply follow the sequential numbering of the root sentence, not the embedded sentence.

```
<s xml:id="example.p.1.s.1">
  <w xml:id="example.p.1.s.1.w.1" class="WORD"><t>He</t></w>
  <w xml:id="example.p.1.s.1.w.2" class="WORD"><t>said</t></w>
  <w xml:id="example.p.1.s.1.w.3" class="PUNCTUATION" space="no"><t>:</t></w>
  <w xml:id="example.p.1.s.1.w.4" class="PUNCTUATION" space="no"><t>’’</t></w>
  <quote xml:id="example.p.1.s.1.quote.1">
    <s xml:id="example.p.1.s.1.quote.1.s.1">
       <w xml:id="example.p.1.s.1.w.5" class="WORD"><t>I</t></w>
       <w xml:id="example.p.1.s.1.w.6" class="WORD"><t>do</t></w>
```

```
      <w xml:id="example.p.1.s.1.w.7" class="WORD"><t>not</t></w>
      <w xml:id="example.p.1.s.1.w.8" class="WORD"><t>know</t></w>
      <w xml:id="example.p.1.s.1.w.9" class="PUNCTUATION" space="no"><t>.</t></w>
   </s>
   <s xml:id="example.p.1.s.1.quote.1.s.2">
      <w xml:id="example.p.1.s.1.w.10" class="WORD"><t>I</t></w>
      <w xml:id="example.p.1.s.1.w.11" class="WORD"><t>think</t></w>
      <w xml:id="example.p.1.s.1.w.12" class="WORD"><t>you</t></w>
      <w xml:id="example.p.1.s.1.w.13" class="WORD"><t>are</t></w>
      <w xml:id="example.p.1.s.1.w.14" class="WORD"><t>right</t></w>
   </s>
 </quote>
 <w xml:id="example.p.1.s.1.w.15" class="PUNCTUATION" space="no"><t>’’</t></w>
 <w xml:id="example.p.1.s.1.w.16" class="PUNCTUATION"><t>,</t></w>
 <w xml:id="example.p.1.s.1.w.17" class="WORD"><t>and</t></w>
 <w xml:id="example.p.1.s.1.w.18" class="WORD"><t>left</t></w>
 <w xml:id="example.p.1.s.1.w.19" class="PUNCTUATION" space="no"><t>.</t></w>
</s>
```

2.5 Paradigm & Terminology

The FoLiA format has a very uniform setup and its XML notation for annotation follows a generalised paradigm. We distinguish two different categories of annotation:

- **Token annotation** - Annotations pertaining to one specific token. These will be elements of the token element (w) in inline notation. Linguistic annotations in this category are for example: part-of-speech annotation, lemma annotation, sense annotation, morphological analysis, spelling correction.
- **Span annotation** - Annotations spanning over multiple tokens. Each type of annotation will be in a seperate **annotation layer** with offset notation. These layers are embedded on the sentence level. Examples in this category are: syntactic parses, chunking, semantic roles and named entities.

Almost all linguistic annotations are associated with what we shall call a **set**. The set determines the vocabulary (the tags) of the annotation. An element of such a set is referred to as a **class** from the FoLiA perspective. For example, we may have a document with Part-of-Speech annotation according to the CGN set. The CGN set defines main tag classes such as *WW*, *BW*, *ADJ*, *VZ*. FoLiA itself thus never commits to any tagset but leaves you to explicitly 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 hosting the file that defines the set, and a `class` attribute, which selects a class from the set.

The following example shows a simple Part-of-Speech annotation without features, but with all common attributes according to the FoLiA paradigm:

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

The example demonstrates that any annotation element can take an `annotator` attribute and an `annotatortype`. The latter is either "manual" for human annotators, or "auto" for automated systems. The value for `annotator` is open and should be set to the name or ID of the system or human annotator that made the annotation. Last, there is a `confidence` attribute which is set to a floating point value between zero and one, the value expresses the confidence the annotator places in his annotation. None of these options are mandatory, only `class` may be mandatory for some types of annotation, such as `pos`.

Development Notes
In this stage, the sets are not actually defined yet, i.e. the URLs they point to don't exist yet. But the idea is that a set always points to a URL that defines all its classes. The format for this is still to be specified however. Links to the ISOCAT Data Category Registry can later be included at that level. For now, ad-hoc sets that will later be defined will do.

2.6 Annotation Declarations

Explicitly referring to a set and annotator for each annotation element can be cumbersome, especially in a document with a single set and a single annotator for that type particular of annotation. This problem can be solved by declaring defaults in the annotation declaration.

The annotation declaration is a mandatory part of the metadata that declares all the types of annotation that are present in the document. In addition it may define defaults such as the tagset used, a default annotator, and the type of annotator. These defaults can always be overriden at the annotation level itself, using the XML attributes `set`, `annotator` and `annotatortype`, as discussed in the previous section. None of the attributes are mandatory in the declaration, though the declarations themselves are; they declare what annotations are to be expected in the document. Having a type of annotation that is not declared is invalid. Do note that if you do not specify a set, annotator or annotator-type in either the declaration or in the annotation elements themselves, they will be left undefined. Not declaring sets is generally a bad idea.

Annotations are declared in the `annotations` block, as shown in the following example. We here define four annotation levels.

```
<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 annotator="Frog" annotatortype="auto" />
        <sense-annotation set="http://ilk.uvt.nl/folia/sets/Cornetto"
         annotator="SupWSD1" annotatortype="auto" />
</annotations>
```

2.7 Token Annotation

Token annotations are annotations that are placed within the word (`w`) element. They all can take any of the attributes described in section 2.5, 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 a `token-annotation` declaration and `w` elements. The declaration can be as in the following example:

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

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

```
<s xml:id="example.p.1.s.1">
```

```
    <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>
```

2.7.1 Part of Speech Annotation

The following example illustrates a simple Part-of-Speech annotation for the Dutch word “boot”:

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

Part-of-Speech annotations may also include extra features, which are explicitly listed and are defined by the set:

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

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 attributes are optional. In the declaration example here it is as if the annotations were made by the software *Frog*. Do note the requirement of a `token-annotation` as well.

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

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

2.7.2 Lemma Annotation

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

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

And the example declaration:

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

2.7.3 Lexical Semantic Sense Annotation

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.

The actual value of the *sense* element, “beeldhouwwerk”, can be set to a human-readable description, 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">beeldhouwwerk</sense>
</w>
```

The example declaration is as follows:

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

2.7.4 Domain Tags

This is a bit of a peculiar token annotation element, in the sense that it is more than just that. 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 body element itself. This annotation defines the domain of the token/content. Example:

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

The value of the element may optionally be set to a human-readable label for the domain.

The declaration:

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

2.7.5 Lexical Annotation

Lexical annotation allows for token to be looked up in a lexicon. The annotation states the presence of this token in a particular lexicon (a particular set), and may offer possible suggestions.

The following example shows a misspelled word, that was not found in the lexicon, and where the lexicon comes with two better suggestions:

```
<w xml:id="example.p.1.s.1.w.1">
    <t>treee</t>
    <lexicon set="http://ilk.uvt.nl/folia/sets/lexicon-nl" known="no">
        <suggestion confidence="0.6">tree</suggestion>
        <suggestion confidence="0.4">three</suggestion>
    <lexicon>
</w>
```

As always, the confidence attribute is optional. The class *may* be set to the word-form/lemma as it occurs in the lexicon, this too is optional.

Similar to `lexicon`, there is also an element for error lists, called `errorlist`. Whereas a lexicon holds correct word forms, an error list holds common errors and has suggestions for correcting those, and thus is in a certain sense the opposite of a lexicon. Note in particular the fact that the `known` attribute in particular is inverted:

```
<w xml:id="example.p.1.s.1.w.1">
    <t>treee</t>
    <errorlist set="http://ilk.uvt.nl/folia/sets/errorlist-nl" known="yes">
        <suggestion confidence="0.6">tree</suggestion>
        <suggestion confidence="0.4">three</suggestion>
    <errorlist>
</w>
```

The declaration:

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

If multiple lexicons, i.e. multiple sets, are used, simply declare without a default set attribute, but explicitly set it for each `lexicon` element. This is a general principle that applies to all kind of annotations:

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

2.7.6 Corrections

Corrections, including but not limited to spelling corrections, can be annotated using the `correction` element. There is a significant difference between for example lexical annotations and these corrections; lexical annotation offers merely suggestions, it does not indicate actual correction of the token. The `correction` element does. It can be applied as a token annotation element as in the following example, which shows a spelling correction of the misspelled word “treee” to its corrected form “tree”.

```
<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" class="spelling">
        <t>tree</t>
        <original>
            <t>treee</t>
        </original>
    </correction>
</w>
```

Note that all corrections must carry an identifier, consisting of the ID of the token with a *c*, a period and a sequence number appended. Within the `correction` element there should be a text element (`t`) containing the token content after correction. The token prior to correction is recorded in the `original` element, which in turn also has a `t` element. The class indicates the kind of correction, according to the set used.

Whilst it may seem redundant to specify the token content both under the word element (`w`), and the `correction` element, and to list the original so verbosely rather than in a mere attribute, there is a good reason for this: 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">
    <t>tree</t>
    <correction xml:id="TEST-000000001.p.1.s.1.w.1.c.2" class="spelling"
      annotator="Jane␣Doe" annotatortype="manual" confidence="1.0">
        <t>tree</t>
        <original>
            <correction xml:id="TEST-000000001.p.1.s.1.w.1.c.1" class="spelling"
               annotator="John␣Doe" annotatortype="manual" confidence="0.6">
                <t>three</t>
                <original>
                    <t>treee</t>
                </original>
            </correction>
        </original>
    </correction>
</w>
```

Corrections are fairly complex token annotations, they may be corrections *over* other token annotations rather than simply the token content as in the prior two examples. 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>
    <pos class="n" />
    <correction xml:id="TEST-000000001.p.1.s.1.w.1.c.1">
        <pos class="n" />
        <original>
            <pos class="v" />
        </original>
    </correction>

</w>
```

Again, there is a small level of necessary redundancy; the corrected element is within the `correction` element as well as the `w` element. Furthermore, if these two `pos` elements would differ, the FoLiA notation would be invalid.

The `lexicon` and `errorlist` elements behave in a special manner when combined with `correction`. On any of the `suggestion` elements of these, the attribute used=yes can be set to indicate that the lexical suggestion was indeed used as the basis of the correction.

```
<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" class="spelling">
        <t>tree</t>
        <original>
            <t>treee</t>
            <lexicon set="http://ilk.uvt.nl/folia/sets/lexicon-nl" known="no">
                <suggestion confidence="0.6" used="yes">tree</suggestion>
                <suggestion confidence="0.4">three</suggestion>
            </lexicon>
        </original>
    </correction>
</w>
```

Like everything, corrections have to be declared:

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

Merges and Splits

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

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

```
<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>
```

And after merging:

```
<w xml:id="example.p.1.s.1.w.1-2">
    <t>online</t>
    <correction xml:id="example.p.1.s.1.w.1-2.c.1" class="merge">
        <original>
            <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>
</w>
```

The new identifier denotes the span of the merge, separated by a hyphen, so we get `.w.1-2` if we merge from `.w.1` to `.w.2`.

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

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

After splitting:

```
<w xml:id="example.p.1.s.1.w.1_1">
    <t>on</t>
    <correction xml:id="example.p.1.s.1.w.1_1.c.1" class="merge">
       <original>
        <w xml:id="example.p.1.s.1.w.1">
         <t>online</t>
        </w>
       </original>
    <correction>
</w>
<w xml:id="example.p.1.s.1.w.1_2">
    <t>line</t>
    <correction xml:id="example.p.1.s.1.w.1_2.c.2"
      ref="example.p.1.s.1.w.1_1.c.1" />
</w>
```

Note the usage of a little bit of stand-off notation here, the second split token has a correction that refers to the same correction as the first one, which contains the original.

The new identifiers represent the index of the new tokens, separated by a underscore, so given `.w.1` we get `.w.1_1` for the first split result, `.w.1_2` for the second, and so on...

Correction prior to tokenisation

There is another special use of the correction element. Sometimes corrections or normalisations occur prior to tokenisation, think for example about correcting OCR-errors. To accommodate this, the `correction` element can be used inline within the text content element (`t`) of a paragraph or sentence, which is by definition untokenised.

Without correction:

```
<s xml:id="example.p.1.s.1.w.1">
    <s>Look at thi.s untokenised sentence.</s>
</s>
```

With correction:

```
<s xml:id="example.p.1.s.1.w.1">
    <t corrections="yes">Look at <correction xml:id="example.p.1.s.1.c.1"
      class="ocrcorrection">
      <t>this</t>
      <original>
        <t>thi.s</t>
      </original></correction> untokenised sentence.
    </t>
</s>
```

Although correction is used inline here, rather than as a normal token annotation, its usage is still identical. For clarity's sake, the class of course depends on the set and is as always never predefined in FoLiA itself.

Note that the text element gains an extra mandatory attribute, `correction` with value *yes* (default if unspecified is no), which signals that there are inline corrections. This is to make the job of parsers easier.

2.7.7 Morphological Analysis

Development Notes
Still to be done.. The `morphemes` and `morpheme` elements will be reserved for this.

2.8 Alternative Token Annotations

The FoLiA format does not just allow for a single "favoured" annotation per token, in addition it allows for the recording 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 has a unique identifier, formed in the already familiar fashion. 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.

```
<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 class="finance" />
        <sense 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 class="geology" />
        <sense class="r_n-5920" synset="d_n-38257"
         annotator="Jim␣Doe" annotatortype="manual"
         confidence="0.1">zandbank</sense>
    </alt>
</w>
```

2.9 Span Annotation

Not all annotations can be realised as token annotations. Some typically span multiple tokens. For these we introduce a kind of offset notation in separate *annotation layers*. These annotation layers are embedded at the sentence level, *after* the word tokens. Within these layers, references are made to these word tokens. Each annotation layer is specific to a kind of span annotation.

The layer elements themselves may also take the `set`, `annotator`, `annotatortype`, or `confidence` attributes. Which introduces the defaults for all the span annotations under it. They in turn may of course always chose to override this.

2.9.1 Entities

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. Each entity should 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.e.1" class="person">
        <ref xml:id="example.p.1.s.1.w.2" />
        <ref xml:id="example.p.1.s.1.w.3" />
    </entity>
  </entities>
</w>
```

Note that elements that are not part of any span annotation need never be included in the layer.

2.9.2 Syntax

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. Each syntactic unit should 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">
             <ref xml:id="example.p.1.s.1.w.1" />
          </su>
          <su xml:id="example.p.1.s.1.su.1_1_2" class="pn">
             <ref xml:id="example.p.1.s.1.w.2" />
             <ref xml:id="example.p.1.s.1.w.3" />
          </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">
            <ref xml:id="example.p.1.s.1.w.4" />
        </su>
        <su xml:id="example.p.1.s.1.su.1_1_2" class="pron">
          <ref xml:id="example.p.1.s.1.w.5" />
        </su>
     </su>
    </su>
  </syntax>
```

```
</s>
```

Just to prevent any misunderstanding, the classes depend on the set used, so you can use whatever system of syntactic annotation you desire. Moreover, any of the su elements can have the common attributes annotator, annotatortype and confidence.

The above example illustrated a fairly simple syntactic parse. Dependency parses are possible too. Dependencies are listed separate from the syntax in an extra annotation layer, as shall be explained in the next section.

The declaration is as follows:

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

2.9.3 Dependency Relations

Dependency relations are relations between syntactic units (or spans of tokens). This relation is often of a particular class and consists of a head component and a 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.

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

In the below example, we show a Dutch sentence parsed with the Alpino Parser [3]. We show not only the dependency layer, but also the syntax layer. The dependency element always contains one head element (hd) and one dependant element (dep), both can refer to a syntactic unit by means of the su attribute. Additionally, the words they cover are reiterated in the usual fashion. For a better understanding, the figure below illustrates the syntactic parse with the dependency relations.

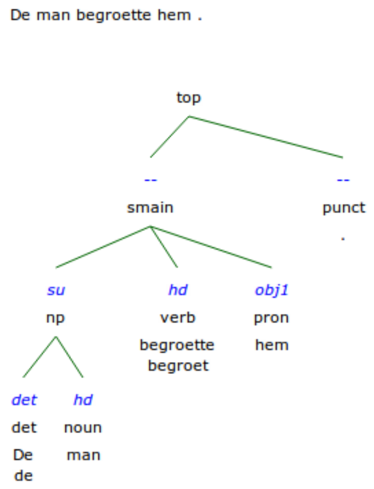

Figure 2.1: Alpino dependency parse for the Dutch sentence “De man begroette hem.”

```
<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 set="http://ilk.uvt.nl/folia/sets/alpino">
    <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">
                    <ref xml:id="example.p.1.s.1.w.1" />
                </su>
                <su xml:id="example.p.1.s.1.su.1_1_1_2" class="top">
                    <ref xml:id="example.p.1.s.1.w.2" />
                </su>
            </su>
            <su xml:id="example.p.1.s.1.su.1_1_2" class="verb">
                <ref xml:id="example.p.1.s.1.w.3" />
            </su>
            <su xml:id="example.p.1.s.1.su.1_1_3" class="pron">
                <ref xml:id="example.p.1.s.1.w.4" />
            </su>
        </su>
        <su xml:id="example.p.1.s.1.su.1_2" class="punct">
            <ref xml:id="example.p.1.s.1.w.5" />
        </su>
    </su>
  </syntax>
  <dependencies>
    <dependency xml:id="example.p.1.s.1.dependency.1" class="det">
        <hd su="example.p.1.s.1.su.1_1_1_2">
           <ref xml:id="example.p.1.s.1.w.2" />
        </hd>
        <dep ref="example.p.1.s.1.su.1_1_1_1">
            <ref xml:id="example.p.1.s.1.w.1" />
        </dep>
    </dependency>
    <dependency xml:id="example.p.1.s.1.dependency.2" class="obj1">
        <hd su="example.p.1.s.1.su.1_1_2">
            <ref xml:id="example.p.1.s.1.w.3">
        </hd>
        <dep su="example.p.1.s.1.su.1_1_3">
            <ref xml:id="example.p.1.s.1.w.4" />
        </dep>
    </dependency>
  </dependencies>
</s>
```

The declaration:

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

2.9.4 Chunking

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">
        <ref xml:id="example.p.1.s.1.w.1" />
        <ref xml:id="example.p.1.s.1.w.2" />
        <ref xml:id="example.p.1.s.1.w.3" />
    </chunk>
    <chunk xml:id="example.p.1.s.1.chunk.2">
        <ref xml:id="example.p.1.s.1.w.4" />
    </chunk>
    <chunk xml:id="example.p.1.s.1.chunk.3">
        <ref xml:id="example.p.1.s.1.w.5" />
        <ref xml:id="example.p.1.s.1.w.6" />
    </chunk>
  </chunking>
</s>
```

The declaration:

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

2.9.5 Semantic roles

Development Notes
Still to be done.. The `semroles` layer and `semrole` span annotation element will be reserved for this.

2.9.6 Alterative Span Annotations

With token annotations one could specify an unbounded number of alternative annotations. This is possible 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 should be associated with a unique identifier, which uses “alt” rather than “altlayers”.

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">
        <ref xml:id="example.p.1.s.1.w.1" />
        <ref xml:id="example.p.1.s.1.w.2" />
        <ref xml:id="example.p.1.s.1.w.3" />
    </chunk>
    <chunk xml:id="example.p.1.s.1.chunk.2">
        <ref xml:id="example.p.1.s.1.w.4" />
    </chunk>
    <chunk xml:id="example.p.1.s.1.chunk.3">
        <ref xml:id="example.p.1.s.1.w.5" />
        <ref xml:id="example.p.1.s.1.w.6" />
    </chunk>
  </chunking>
  <altlayers xml:id="example.p.1.s.1.alt.1">
       <chunking annotator="John␣Doe"
        annotatortype="manual" confidence="0.0001">
        <chunk xml:id="example.p.1.s.1.alt.1.chunk.1">
            <ref xml:id="example.p.1.s.1.w.1" />
            <ref xml:id="example.p.1.s.1.w.2" />
        </chunk>
        <chunk xml:id="example.p.1.s.1.alt.1.chunk.2">
            <ref xml:id="example.p.1.s.1.w.3" />
            <ref xml:id="example.p.1.s.1.w.4" />
        </chunk>
        <chunk xml:id="example.p.1.s.1.alt.1.chunk.3">
            <ref xml:id="example.p.1.s.1.w.5" />
            <ref xml:id="example.p.1.s.1.w.6" />
        </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.

Chapter 3

Parsing & Querying

Development Notes
To be written still...

Bibliography

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

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

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

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

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

[6] N. Oostdijk, M. Reynaert, P. Monachesi, G. Van Noord, R. Ordelm an, I. Schuurman, and V. 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.