FoLiA: Format for Linguistic Annotation

v0.5

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

July 4, 2011

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 DCOI corpus, as well as by its successor; the SoNaR corpus [6]. Though being rooted in the D-Coi format, the FoLiA format goes a lot further and introduces a rich generalised framework for linguistic annotation. FoLiA is developed at the ILK research group, Tilburg University, and proposed as a CLARIN standard in the TTNWW project.

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 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
- Morphological Analysis

In later stages, the following may be added:

- Semantic Role Labelling
- Co-reference
- Topic Segmentation
- Authorship Attribution
- Subjectivity Annotation (to be used in VU-DNC)

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

FoLiA will be used in the DutchSemCor project, the DU-VNC project, and will also be proposed for consideration in the SoNaR project. It is also proposed as a CLARIN standard in the TTNWW project.

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.

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. 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">
  <metadata>
      <annotations>
          ...
      </annotations>
      <!-- (Here CMDI or IMDI metadata can be inserted) -->
  </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. 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 Basic Structural Elements

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

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

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

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

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

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

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

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 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 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>
```

It 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>
```

It gets somewhat more complicated if corrections are involved, consult section for a more in-depth discussion.

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

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

A quote may consist of one or more sentences, but may also consist of mere tokens. 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>
```

2.4 Paradigm & Terminology

The FoLiA format has a very uniform setup and its XML notation for annotation follows a generalised paradigm. We distinguish three different categories of annotation, of which the latter two apply to actual linguistic annotation:

- **Structural annotation** - Annotations marking global structure, such as chapters, sections, subsections, figures, list items, paragraphs, etc...
- **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. Some token elements may be used on higher levels (e.g. sentence/paragraph) as well and may then be referred to as **Extended Token Annotation**

- **Span annotation** - Annotations spanning over multiple tokens. Each type of annotation will be in a separate **annotation layer** with offset notation. These layers are typically embedded on the sentence level, or possibly also on higher levels (paragraph/division/text) for certain annotation types. Examples in this category are: syntactic parses, chunking, semantic roles and named entities.

- **Subtoken annotation** - This is a type of annotation that has aspects of both token annotation as well as span annotation. The former because is included inline within a token element (`w`) and describes the token. The latter because is defines a small span *within* the token. Examples in this category are: morphological analysis, named subentities

Almost all annotations are associated with what we shall call a **set**. The set determines the vocabulary, the tags or types, 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 (a tagset for Dutch part-of-speech tags). The CGN set defines main tag classes such as *WW*, *BW*, *ADJ*, *VZ*. FoLiA itself thus never commits to any tagset but leaves you to 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`.

More advanced aspects of the paradigm will introduced later in section 2.10.

2.5 Annotation Declarations

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

```
<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>
```

The set attribute is mandatory and refers to a URL of a FoLiA Set Definition file (see chapter ??). The Set Definition specifies exactly what classes are allowed in the set. It for examples specifies exactly what Part-of-Speech tags exists. This information is necessary to completely validate the document at its deepest level. If the sets point to URLs that do not exist, warnings will be issued. Validation can still proceed but with the notable exception of deep validation of these sets.

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

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

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

The `annotator` and `annotatortype` attributes acts 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!

Annotator defaults can always be overriden 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 in 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.

2.6 Structure Annotation

2.6.1 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:

```
<?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">
  <metadata>
      <annotations>
          <div-annotation set="http://ilk.uvt.nl/folia/sets/divisions" />
      </annotations>
      <!-- (Here CMDI or IMDI metadata can be inserted) -->
  </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>
```

If divisions are present, they need to be declared, as can be seen in the metadata. Divisions themselves are never mandatory, you can have a document without any divisions.

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

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

2.6.2 Gaps

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

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

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

Due to the flexible nature of FoLiA, we don't 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:

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

2.6.3 Whitespace and Linebreaks

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

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

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

2.6.4 Lists

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

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

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

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

2.6.5 Figures

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

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

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

2.6.6 Tables

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

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 2.4, 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>
```

However, for some tagsets simple part-of-speech annotation is not enough; there may for example be features associated with the part of speech tag. We will 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 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 Phonetic Annotation

Phonetic annotations can be included as follows. Similarly to lemmas, they may often refer to a set with possible open classes.

```
<w xml:id="example.p.1.s.1.w.2">
    <t>boot</t>
    <phon-annotation set="ipa" class="bu:t" />
</w>
```

This is an extended token annotation element that can also be used directly on a sentence or paragraph level.

And the example declaration:

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

2.7.4 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.

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

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

The example declaration is as follows:

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

2.7.5 Domain Tags

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. This annotation defines the domain of the token of content element. Example:

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

2.7.6 Corrections

Corrections, including but not limited to spelling corrections, can be annotated using the `correction` element. It can be applied as an extended 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">
    <correction xml:id="TEST-000000001.p.1.s.1.w.1.c.1" class="spelling">
        <new>
            <t>tree</t>
        </new>
        <original>
            <t>treee</t>
        </original>
    </correction>
</w>
```

The class indicates the kind of correction, according to the set used. The `new` elements 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`).

```
<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>
            <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>
                    <t>treee</t>
                </original>
            </correction>
        </original>
    </correction>
</w>
```

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

```
<w xml:id="example.p.1.s.1.w.1">
    <t>tree</t>
    <correction xml:id="TEST-000000001.p.1.s.1.w.1.c.1">
        <new>
            <pos class="n" />
        </new>
        <original>
            <pos class="v" />
        </original>
    </correction>

</w>
```

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

Error detection and correction with suggestions

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 existance of errors, or absence thereof:

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

The `error` attribute is set to "yes" (which is the default value), and thus marks this as an error of class "spelling". We can also imagine it specifically marking something as *not* being an error (in which case class is always redundant), for example due to the occurence of the word according to a lexicon:

```
<w xml:id="example.p.1.s.1.w.1">
    <t>tree</t>
    <errordetection annotator="lexiconX" error="no" />
</w>
```

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

The correction tag can also be used in such situations in which you want to list suggestions for correction, but not yet commit to any single one. You may for example want to postponed this actual selection to another module or human annotator. Recall that the actual correction is always included in the "new" tag, non-comitting 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">
            <t>tree</t>
        </suggestion>
        <suggestion confidence="0.2">
            <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.

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"
          annotatortype="auto" confidence="0.8">
            <t>tree</t>
        </suggestion>
        <suggestion annotator="errorlistX"
          annotatortype="auto" confidence="0.2">
            <t>three</t>
        </suggestion>
        <original>
            <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>
            <t>treee</t>
        </original>
        <errordetection class="spelling" annotator="errorlist" annotatortype="auto
    </correction>
</w>
```

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

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

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

Merges, Splits and Swaps

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

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

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

And after merging:

```
<s xml:id="example.p.1.s.1">
 <correction xml:id="example.p.1.s.1.c.1" class="merge">
    <new>
        <w xml:id="example.p.1.s.1.w.1-2">
            <t>online</t>
        </w>
    </new>
    <original>
        <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, being a kind of extended token annotation, is here a member of the sentence (`s`), rather than the word token (`w`) as in all previous examples. The new identifier denotes the span of the merge, 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:

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

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...

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

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

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

Omissions and Insertions

Omissions, words that are removed in correction, and insertions, words inserted during correction, are dealt with in a way similar to merges, splits and swaps. For omissions, the `new` element is simply empty. In the following example the word "the" was duplicated and removed in correction:

```
<s xml:id="example.p.1.s.1">
 <w xml:id="example.p.1.s.1.w.1">
    <t>the</t>
 </w>
 <correction xml:id="example.p.1.s.1.c.1" class="duplicate">
    <new>
    </new>
    <original>
        <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>
    </original>
 </correction>
 <w xml:id="example.p.1.s.1.w.2">
    <t>man</t>
```

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

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">
    <t>Look at thi.s untokenised sentence.</t>
</s>
```

With correction:

```
<s xml:id="example.p.1.s.1">
    <t corrected="inline">Look at <correction xml:id="example.p.1.s.1.c.1"
      class="ocrcorrection">
      <new>
         <t>this</t>
      </new>
      <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, `corrected` with value *inline*, which signals that there are inline corrections *within* the text element. Please see section for a more elaborate discussion on 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.entity.1" class="person">
        <wref id="example.p.1.s.1.w.2" t="Dalai" />
        <wref id="example.p.1.s.1.w.3" t="Lama" />
    </entity>
  </entities>
</s>
```

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

2.9.2 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">
             <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>
```

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.

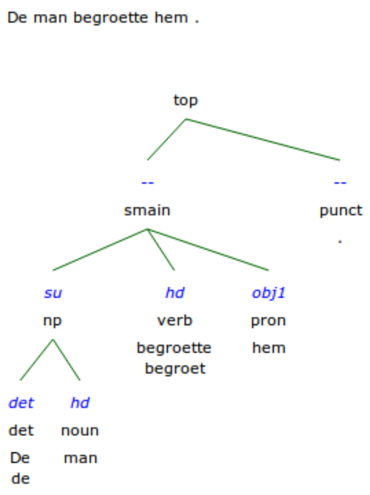

Figure 2.1: Alpino dependency parse for the Dutch sentence "De man begroette hem."

```
<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">
                    <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="det">
        <hd su="example.p.1.s.1.su.1_1_1_2">
           <wref id="example.p.1.s.1.w.2" t="man" />
        </hd>
        <dep ref="example.p.1.s.1.su.1_1_1_1">
            <wref id="example.p.1.s.1.w.1" t="De" />
        </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">
            <wref id="example.p.1.s.1.w.3" t="begroette">
        </hd>
        <dep su="example.p.1.s.1.su.1_1_3">
            <wref id="example.p.1.s.1.w.4" t="hem" />
        </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">
        <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:

```
<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">
        <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">
         <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 Advanced Paradigm

We introduced the FoLiA paradigm in section 2.4. Now we will introduce some of the more advanced aspects of the FoLiA paradigm. These are relevant especially if you want to submit suggestions for extending the FoLiA format with annotations not yet supported.

2.10.1 Human readable Descriptions

Any token annotation element or span annotation element may hold a `desc` element containing a human readable description for the annotation. An example of this has been already shown for the `sense` and `gap` elements.

2.10.2 Text content and pre-tokenised text

In section 2.3 we have seen the text content element `t` used on different levels, and the `offset` attribute 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. This does imply that there are some challenges to solve. First of all, by default, the offset refers to the direct parent of whatever 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>
```

Another challenge is corrections, as they often change the text and we want this to reflect back on the untokenised text. The text content element (`t`) may be used with the `corrected` attribute set to "yes" to state the text *after correction*, in *untokenised* form! As this may changes offsets, the `newoffset` attribute is used to refer to the offsets in this new corrected text. This text content element does not replace the already existing one, there can be *one* of each typeAdditionally, there can be one where corrected is set to "inline":

```
<p xml:id="example.p.1">
    <t>Hello. This is a sentence. Bye!</t>
    <t corrected="yes">Hello. This was a sentence. Bye!</t>
    <s xml:id="example.p.1.s.1">
        <t offset="7">This is a sentence.</t>
        <t corrected="yes" offset="7" length="19" newoffset="7">
        This was a sentence.
        </t>
        <w xml:id="example.p.1.s.1.w.1">
          <t offset="0" newoffset="0">This</t>
        </w>
        <w xml:id="example.p.1.s.1.w.2">
          <t offset="5" length="2" newoffset="5">was</t>
          <correction>...</correction>
        </w>
        <w xml:id="example.p.1.s.1.w.3">
          <t offset="8" newoffset="9">a</t>
        </w>
        <w xml:id="example.p.1.s.1.w.4" space="no">
          <t offset="10" newoffset="11">sentence</t>
        </w>
        <w xml:id="example.p.1.s.1.w.5">
          <t offset="48">.</t>
        </w>
    </s>
</p>
```

Order matters here, so parsers can always grab the last (t) element to have the most edited version, and the first to have the version closest to the original. The attribute newoffset always refers to the newest text content, whilst offset refers to the most original text content. The length attribute is used to specify the length of the original text prior to correction, as this can no longer be derived automatically, in all other cases it can be omitted as it can be derived from the textual content itself.

Text content (t) in word elements (w) are *by definition* a single token *and* corrected. Text content (t) on a sentence/paragraph/division level is by definition always untokenised, and can be in a state prior to correction, with inline corrections or post-correction. Multiple t elements are allowed only if their value for corrected differs.

Last, but not least; all offset and length attributes 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.

2.10.3 Subsets and multiple classes

For some annotations, associating a single class from a set is not sufficient. FoLiA provides the option to associate multiple classes, or to associate classes with specific subsets.

Take a look at the following abstract case: X here is a *fictional* token/span annotation. It is associated with two classes from the same set. The fictional element Y goes with X, and is custom to the specific annotation.

```
<X set="a">
    <Y class="p">
    <Y class="q">
</X>
```

The classes can be made members of a certain subset, rather than the main set. Subsets are assumed to in turn defined by the main set. No special declarations will be necessary.

```
<X set="a">
    <Y subset="f" class="p">
    <Y subset="g" class="q">
</X>
```

Optionally, X itself might also be associated with a class, the class in this case would either represent all sub-parts, *or* be the basis upon which redefinement is made. Which of the two is chosen depends on the set used.

FoLiA has a default element feat that can be used instead of Y in the last-mentioned example above, so that no special elements needs to be introduced. This feat element can always be used to freely to associate any additional classes of *any* designed subset with with *any* annotation element !!!

If elements such as the ficticious Y are introduced then they can be implicitly associated with a specific subset:

```
<X set="a" class="p">
    <Y class="q">
</X>
```

In such cases, if a subset is implied and there can be only one instance, a more common and concise solution would be to add Y as an attribute:

```
<X set="a" class="p" y="q" />
```

An example of this is seen in the sense element, which takes a synset attribute that is implicitly associated with a subset of synsets.

```
<sense class="p" synset="q" />
```

FoLiA implicitly relates the synset attribute here to the "synset" subset of whatever set you defined. Note that any such implicit subset relation needs to be defined by the FoLiA standard itself, you can't just make up such attributes yourself. This section mainly served to gain insight in the advanced aspects of FoLiA, and to be able to suggest additions to it that follow the same principles.

2.10.4 Part-of-Speech tags with features

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

Option one:

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

In FoLiA, this attribute head is implicitly associated with the subset "head" of whatever set you defined.

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>
```

2.11 Subtoken Annotation

Subtoken category combines aspects from both token annotation and span annotation. It introduces an annotation layer at the token level, as child of the `w` element. Within this annotation layer, subtoken annotation elements can be used to annotate parts within the token itself. Whereas span elements use `wref` to refer to words, subannotation elements use `t` to refer to part of the text of the token. 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).

2.11.1 Morphological Analysis

```
<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>

<annotations>
```

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

2.11.2 Named Entities within a token

Named entities may sometimes occur as *part* of a token. The `subentities` annotation layer and the `subentity` subtoken annotation element can be used for annotating these:

```
<w xml:id="example.p.4.s.2.w.4">
    <t>CDA-voorzitter</t>
    <subentities>
        <subentity class="org" annotator="Maarten␣van␣Gompel"
            annotatortype="manual">
            <t offset="0">CDA</t>
        </subentity>
    </subentities>
</w>
```

The declaration:

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

2.12 Alignments

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 either the `alignment` element, a form of extended token annotation (that may thus also be applied on other levels such as sentences or paragraphs), or `complexalignment`, a form of span annotation for more complex word alignments.

Consider the following two aligned sentences from two *distinct* FoLiA documents in different languages. Note that the `t` attribute to the `aref` element is merely optional and is this overhead is added merely to facilitate the job of simple/limited FoLiA parsers. The `xlink:href` argument is used to link to the target document, if any. If the alignment is within the same document then it can be simply omitted.

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

Although the above example has a single alignment reference (aref), it is not forbidden to link to specify multiple references within the alignment block. For more complex alignments however, such as word alignments that include many-to-one, one-to-many or many-to-many alignments, the span annotation element complexalignment may be more suitable. The following example illustrates a many-to-many word-alignment of the word "Dalai Lama" from English to French. This takes places 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 class="french-translation" xlink:href="doc-french.xml" x
            <wref id="example-english.p.1.s.1.w.2" t="Dalai">
            <wref id="example-english.p.1.s.1.w.3" t="Lama">
            <aref id="example-french.p.1.s.1.w.2" t="Dalai">
            <aref id="example-french.p.1.s.1.w.3" t="Lama">
        </complexalignment>
        ...
  </complexalignments>
</s>
```

2.13 Metadata

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

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

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

The procedure for IMDI is the same:

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

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

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

You can simply define fields with custom IDs, but the following fields are 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*)

Chapter 3

Set Definition Format

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 set, including any subsets.

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

3.2 Types and classes

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

Here is a simple example:

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

The `ID` of the class is the value that may be used in the `class` attribute in the FoLiA document. 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
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 different custom XML namespace.

3.4 Subsets

Section 2.10.3 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.

3.5 Constraints

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

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

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

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

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

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

Using the restrict element, you force a certain class from the main set or any subset, thus enumerating all the allowed classes. For example, the follow constraint demands masculine or feminine noun in either nominative or accusative case. All of the restrictions must be satisfied for the constraint to match.

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

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

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

Restrict and except elements can also be mixed.

Appendix A

Common Queries

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

- XPath query for all paragraphs: `/FoLiA/text//p`
- XPath query for all sentences: `/FoLiA/text//s[not(parent::s)]` *Selects only top-level sentences, not sentences embedded within sentences (e.g quotes)*
- XPath query for all words: `/FoLiA/text//w[not(ancestor::original) and not(ancestor::suggestion) and not(ancestor::alternative)]` *The conditional clause is required to correctly deal with correction syntax.*
- XPath query for all words with lemma X:
  `/FoLiA/text//w[not(ancestor::original) and not(ancestor::suggestion) and not(ancestor::alternative)]/lemma[@class == "X"]`
- XPath query for the text of all words:
  `/FoLiA/text//w[not(ancestor::original) and not(ancestor::suggestion) and not(ancestor::alternative)]/t/child::text()`

Note that in various cases, you will want to include the condition "`not(ancestor::original) and not(ancestor::suggestion) and not(ancestor::alternative) and not(ancestor::altlayers)`" to skip over elements that are not actually active but instead pertain to a mere suggestion, alternative, or the original of a suggestion.

Appendix B

Validation

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

Shallow validation is performed using a RelaxNG schema, to be found at http://github.com/proycon/folia.rst
.

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

Appendix C

Implementations

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

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

The following table shows the level of FoLiA support in these libraries:

	PyNLPl python	libfolia $C++$
Token Annotation	yes	yes
Span Annotation	yes	yes
Subtoken Annotation	yes	yes
Subset/feature support	yes	yes
Alternatives	yes	yes
Correction Annotation	yes	yes
Alignments	not yet	not yet
Text offset support	incomplete[1]	incomplete
Native metadata	yes	unknown
IMDI metadata	partial[2]	?
CMDI metadata	no	no
RelaxNG schema generation	yes[3]	no
RelaxNG validation	yes	no
Set definition support	not yet	no
Deep validation (Schematron)	not yet	no
D-Coi read compatibility	partial[4]	no

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.