AnnoLab

User Manual

Richard Eckart de Castilho
Technische Universität Darmstadt
Department of Linguistic and Literary Studies
English Linguistics

AnnoLab: User Manual

Richard Eckart de Castilho
Technische Universität Darmstadt
Department of Linguistic and Literary Studies
English Linguistics

Table of Contents

1. Introduction 1
    1. History of AnnoLab 1
        1.1. PACE-Ling era (2004-2005) 1
        1.2. Diploma Thesis era (2005-2006) 1
        1.3. LingPro era (2006-2008) 1
        1.4. AnnoLab and UIMA 2
2. Getting started 3
    1. Installation 3
    2. Install PEARs 5
    3. Create a pipeline 6
    4. Run the pipeline 7
3. Analysis pipelines 8
    1. Pipeline descriptors 8
    2. Mapping 8
    3. Generalised Annotation Markup 11
        3.1. GAM in data stores 12
        3.2. GAM - Integrated representation 12
I. Command Reference 14
    add-layer 15
    ae 16
    copy 20
    delete 21
    exist 22
    export 23
    filter 25
    help 27
    ims2uam 28
    info 30
    lemmatize 31
    list 34
    matchlist 35
    pear 37
    query 38
    uam2annolab 41
    webserver 42
II. XQuery Extensions Reference 43
    ds:layer 44
    ds:meta 45
    ds:signal 46
    manage:delete 47
    manage:import 48
    seq:containing 49
    seq:grow 50
    tree:following-sibling 51
    txt:find 52
    txt:get-text 53
Glossary 54
Bibliography 55

List of Figures

2.1. Set the ANNOLAB_HOME environment variable 3
2.2. Setting the PATH variable 4
2.3. Access the command line 4
2.4. Testing AnnoLab 4
2.5. Installing PEARs 5
2.6. Listing the installed PEARs 6
2.7. Example pipeline definition 6
2.8. Running the pipeline 7
3.1. AnnoLab pipeline descriptor outline 8
3.2. Outline of a mapping file 8
3.3. Step 1: Define the layer 9
3.4. Step 2: Define the elements 9
3.5. Step 3: Define the dominance relation of the tree 9
3.6. Step 4: Define where the segments are located 10
3.7. Step 5: Define how the segments are encoded 10
3.8. XML for an abstract GAM segment 12
3.9. Linking between text and annotations 13
18. eXist login settings 22
19. Example substitutions file 33
20. Query template descriptor file: query-example.template 39
21. AnnoLab asking for the value of an unset query variable 39

List of Tables

3.1. Examples of <relation> and <feature> sections 10
3.2. Valid mapAs values 11
3. Available fields 35

Chapter 1. Introduction

AnnoLab is an XML-based framework for working with annotated text. In the area of linguistic research text is annotated for linguistic features as a preparation step for linguistic analysis. XML is widely being used to encode these annotations. AnnoLab was born from the question *Can XML and associated technology be easily extended to offer comprehensive support for linguistic annotations?*

1. History of AnnoLab

1.1. PACE-Ling era (2004-2005)

Work on the foundations of AnnoLab started in 2004 in the context of the PACE-Ling[1] project at Technische Universität Darmstadt, Germany. The annotations created manually, semi-automatically and automatically using various tools had to be unified and made available as an integrated resource to allow linguistic analysis of relations between all annotations regardless of their source [Teich05].

The Generalised Annotation Markup (GAM) was developed. This markup format allows to integrate XML annotations from various sources into a single XML file to allow processing using an XSLT file. At the time a naïve Java implementation was created to merge multiple XML annotation files into one GAM file. Research-relevant data was then extracted from GAM files using XSLT style sheets.

1.2. Diploma Thesis era (2005-2006)

In 2005 AnnoLab became the topic of my diploma thesis [Eckart06a]. The research question underlying AnnoLab development at the time was *Can the XML data model be extended to support linguistic annotations while remaining substantially compatible with existing XML tools?* The XML tools in question were XML parsers, XSLT engines and XQuery engines.

At this time AnnoLab evolved into a web-application implemented in Java and based on Apache Cocoon for the web front-end and using Apache Avalon in the back-end. The front-end allowed to upload texts and annotations in XML format as well as to perform queries using XQuery with several GAM-specific extensions. GAM was extended to support two modes: the *integrated mode* known already from the PACE-Ling era as well as a *non-integrated* mode. The latter mode facilitated uploading and managing XML annotation files and annotated text as separate objects. Relations between these objects were derived dynamically at the time of querying. The web front-end allowed users to easily interact with AnnoLab, to browse text and annotations, to show display annotations side-by-side for comparison and to issue queries using predefined templates, e.g. a Keyword-in-Context query template [Eckart06b] [Teich06].

1.3. LingPro era (2006-2008)

Following the diploma thesis I joined the project Linguistic profiles of interdisciplinary registers[2] (LingPro). The requirements in this project were similar to the requirements of the PACE-Ling project but more ambitious with respect to quality, detail and volume of the analysed data.

While it had already been shown that it is in principle possible to extend XML to support linguistic annotations and still remaining substantially compatible with existing XML tools [Eckart07], there were issues regarding performance and handling. In addition, the LingPro project required the integration of additional tools for automatic annotation and support for processing PDF files. Since a large corpus of texts had to be repeatedly processed the focus shifted from interactive operation to unsupervised batch processing.

During this time AnnoLab underwent a major refactoring. The web front-end was dropped due to time constraints and shift of focus. It was replaced by a command-line interface. Much of the Avalon-related code that facilitated integration with Cocoon was also dropped to simplify the code base. The OSGi framework

[1] `http://www.linglit.tu-darmstadt.de/index.php?id=pace_ling`
[2] `http://www.linglit.tu-darmstadt.de/index.php?id=lingpro_projekt`

was adopted as a component framework. Data storage mechanisms and file format support were refactored into plug-ins. Apache UIMA was adopted as a framework for linguistic processing. Linguistic analysis components were turned into UIMA components. The following plug-ins were developed (some are currently not released due to licensing issues):

Import	Text, XML, XHTML, FLOB/FROWN corpus format, PDF
Storage	BibTeX-based read-only storage with meta data support (unreleased), eXist-based storage with query support
Processing (non-UIMA)	TreeTagger integration (unreleased), Partial annotation support with IMS Corpus Workbench and UAM Corpus Tool
Processing (UIMA)	Tokeniser, sentence splitter, TreeTagger wrapper, Stanford parser wrapper (unreleased), Tree Rule Processor with rules for Theme/Rheme annotation based on the Stanford parser output

In the context of a cooperation with the project C2: Sustainability of Linguistic Data[3] a plug-ins to run AnnoLab in a server mode and XQuery extensions for managing data store contents were developed [Rehm07a] [Rehm07b].

1.4. AnnoLab and UIMA

In the course of the development of AnnoLab it became increasingly obvious that using XML as the predominant data model required more compromises than the development effort saved by using XML databases or transformation engines could compensate. In particular the work with the UIMA data model, the Common Analysis System, and with UIMA itself showed that a sound model and framework for the processing of annotated data had emerged here. Still the AnnoLab framework provides extended functionality in comparison with the UIMA framework.

UIMA provides an abstraction for accessing all kinds of data sources, so called Collection Readers. AnnoLab provides an abstraction for managing data stores and for transparently importing all kinds of data formats into an UIMA pipeline. A Collection Reader is provided by AnnoLab which interfaces with its import handler infrastructure to transparently load data from a data store or import it from text files, XML files, PDF files, etc.

The CAS Consumer abstraction provided by UIMA is designed to persist or use CAS data for use after the processing is done. AnnoLab provides a CAS Consumer to transparently store data from the CAS either in the file system or in a data store from which it may be queried or read again for further processing.

Thus AnnoLab adds transparent data import, a data repository and a query mechanism.

[3] `http://www.sfb441.uni-tuebingen.de/c2/`

Chapter 2. Getting started

This section will take you headlong into AnnoLab. It will take you on a short tour from the installation of AnnoLab itself, over the installation analysis components in the form of PEARs (Processing Engine ARchive), towards the creating of a pipeline involving these analysis components and finally running the pipeline.

All screenshots for this section were taken on Windows XP SP3. For other versions of Windows you may get a different visual experience. AnnoLab also runs on Mac OS X and on Linux - for those platforms you should choose paths appropriate for your platform, e.g. on Mac OS X use `/Applications/annolab-cli` instead of `C:\Programme\annolab-cli`.

1. Installation

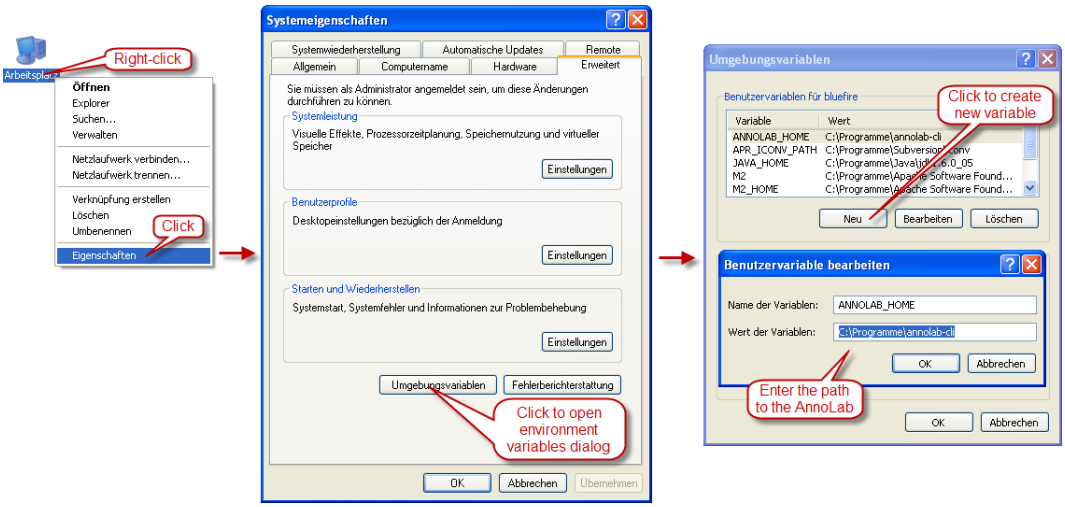
Extract the contents of the AnnoLab archive to the folder `C:\Program Files` (or to the equivalent location depending for localised versions of Windows, e.g. `C:\Programme` for a German version). This creates a folder named `annolab-cli`.

Note

Setting the `ANNOLAB_HOME` environment variable is not necessary on Linux or Mac OS X. It is still necessary to add the AnnoLab home directory to the `PATH`.

Now configure the AnnoLab home directory. Do a right-click on your Desktop icon and select the Properties item. In the dialog that opens use the button labelled Environment. This opens a dialog allowing to create a new user environment variable. Create a new environment variable called `ANNOLAB_HOME` with the value `C:\Program Files\annolab-cli` (or the equivalent on your localised version). Remember this directory - it is the home directory of your AnnoLab installation. When I later refer to the *AnnoLab home*, I mean this directory.

Figure 2.1. Set the `ANNOLAB_HOME` environment variable



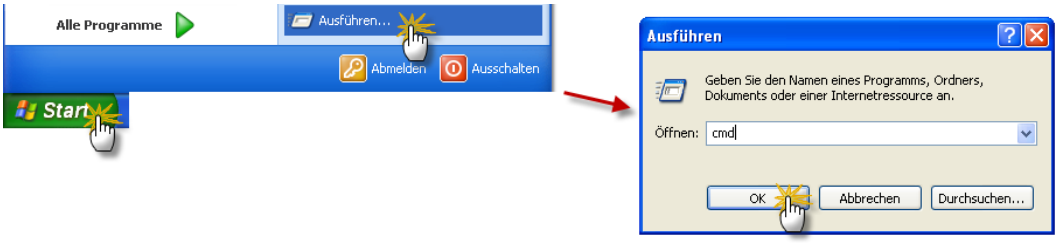
Next we need to add the AnnoLab installation directory to the Path environment variable.

Figure 2.2. Setting the PATH variable

If the variable does not yet exist, you need to create it with the value `%ANNOLAB_HOME%;%Path%`. Otherwise add `%ANNOLAB_HOME%`to the beginning of the semi-colon separated list of path entries.

Now try running AnnoLab. Go to the Start menu and select Run. Enter the command **`cmd`** in the dialog that opens and click OK.

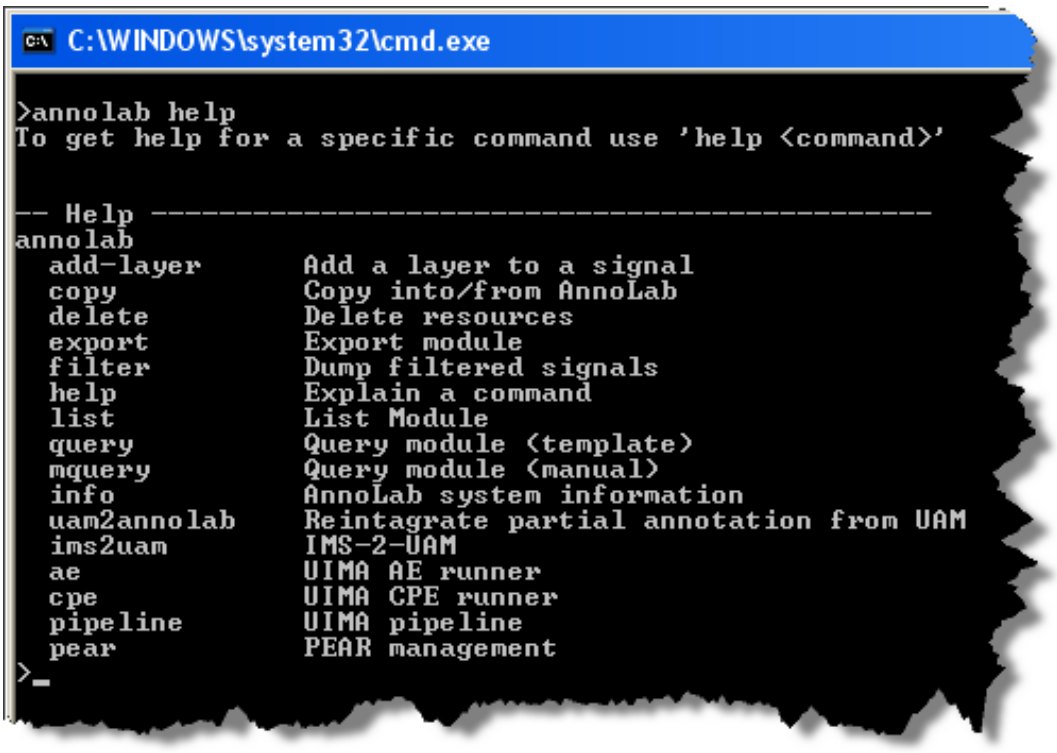
Figure 2.3. Access the command line



At the command line prompt type **`annolab help`** and press the **Enter** key to run AnnoLab. This should cause AnnoLab to display a list of all available commands.

Figure 2.4. Testing AnnoLab

```
$ annolab help
```



2. Install PEARs

Now it is time to install the PEARs. These are the components we will later use to create a processing pipeline. Open the Windows Explorer, navigate to the AnnoLab home a and create a new directory called `pears` there. Now go back to the AnnoLab home page and download the following PEAR packages using *right-click + save link* as to this `pears` directory you have just created:

- Sentence Boundary Annotator
- Token Boundary Annotator
- Manual Language Setter
- Stanford Parser Annotator
- Tree Rule Processing Engine Annotator

Now go back to the command line prompt. Change to the AnnoLab home using the command **`cd %ANNOLAB_HOME%`**. Now we will use AnnoLab's **`pear install`** command to make the PEARs available in AnnoLab pipelines.

Figure 2.5. Installing PEARs

```
$ annolab pear install pears/*
```

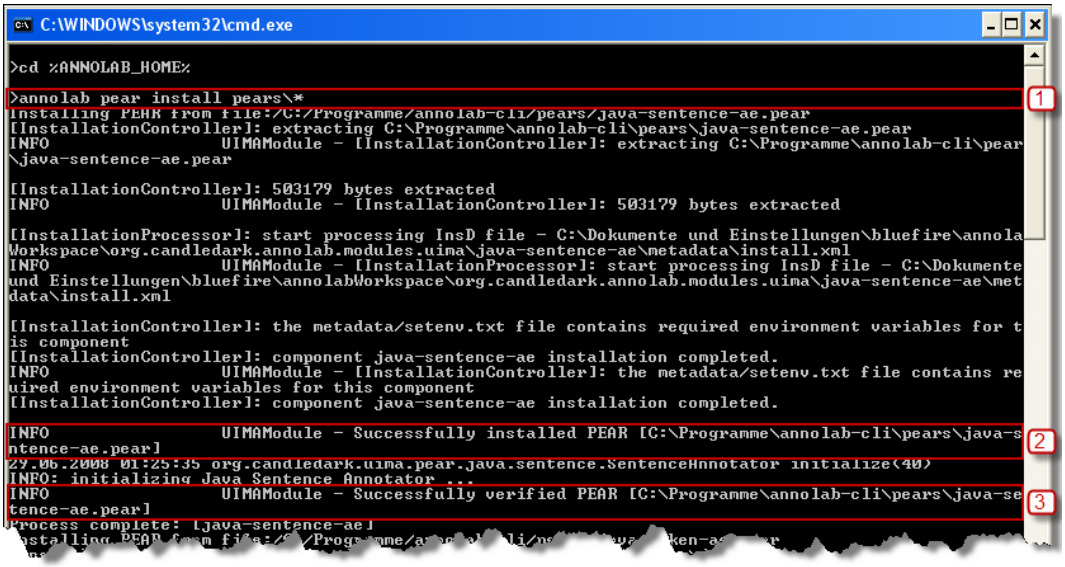


Figure 2.5 shows the command to install the PEARs and its output:

1. The command **`annolab pear install pears/*`** installs all PEARs in the directory `pears`.
2. This message indicates that the PEAR was successfully extracted and registered.
3. This message indicates that the PEAR could be successfully initialised. If this message is not present, the PEAR will most probably not work.

You can use the pear list command to get a list of the installed PEARs (see Figure 2.6).

Figure 2.6. Listing the installed PEARs

```
$ annolab pear list
```

3. Create a pipeline

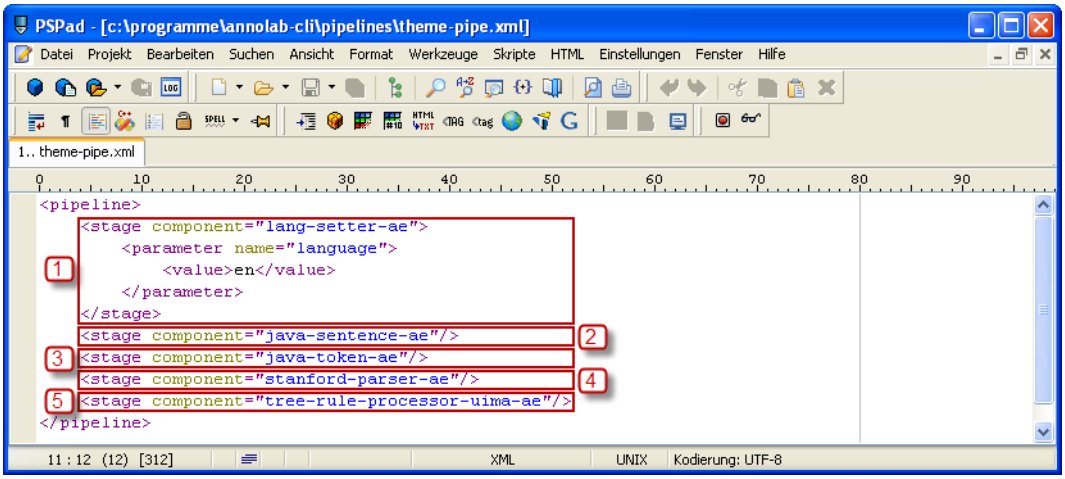
Now we will create a simple pipeline. The pipeline will do the following things:

- Annotate sentence boundaries
- Annotate token boundaries
- Create a syntactic parse annotation
- Annotate the Theme of the sentences

Create new directory called `pipelines` in your AnnoLab home and within create a new text file called `theme-pipe.xml` . Figure 2.7 shows this file. The example uses the editor PSPad, but you can also use the windows Notepad application or any other text editor.

The pipeline definition contains five stages:

Figure 2.7. Example pipeline definition



1. The first stage configures the language for the documents. This is a very simple component allowing to manually define as which language input documents should be treated. A more sophisticated component might try to detect the language of a document automatically. The following stages use this information to determine which model they should use for boundary detection and parsing. In this example the language is set to English (en). Check the documentation of the components to see what languages are supported.
2. This stage invokes the sentence boundary detector.
3. This stage invokes the token boundary detector. It depends on the output of the previous stage.
4. This stage invokes the Stanford Parser. It depends on the two previous stages.

5. The final stage invokes the Tree Rule Processor. This component per default uses a rule set that annotates the Theme of a sentence based on the syntactic parse annotation created by the Stanford Parser.

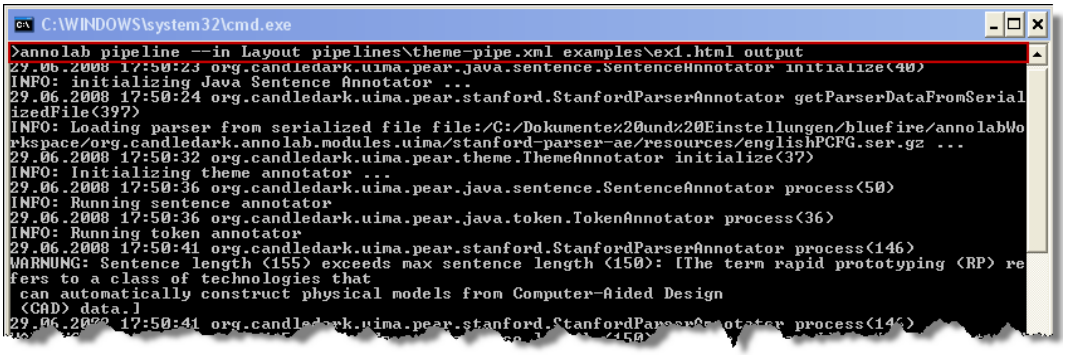
4. Run the pipeline

Now create another directory called `output` in your AnnoLab home.

Finally we can run the pipeline as shown in Figure 2.8. The **`pipeline`** command requires three arguments: the name of the pipeline definition file, a file or directory containing the input texts (here `examples\texts\ex1.html`) and an output directory name.

Figure 2.8. Running the pipeline

```
$ annolab pipeline --in Layout pipelines\theme-pipe.xml
    examples\texts\ex1.html output
```



After the command completes, you will find the following files in the output directory:

- `ex1.html.txt` - the text extracted from the input file.
- `ex1.html_Layout.xml` - the layout layer.
- `ex1.html_Sentence.xml` - the sentence annotation layer.
- `ex1.html_Token.xml` - the token annotation layer.
- `ex1.html_Parse.xml` - the parse annotation layer containing the syntactic parse and the theme annotations.

The *`--in Layout`* parameter causes the layout information from the HTML to be imported before running the pipeline. Thus header and paragraph boundaries are available to the sentence splitter.

Tip

Importing annotations into and exporting annotations from the pipeline is imperfect. This is mainly relevant for the Layout layer. While HTML <h1> and <p> tags will be imported as header and paragraph boundaries, nothing else will be imported. Thus when the Layout layer is exported afterwards it will only contain these two tags. Usually you will want to use the *`--out parameter`* to control which layers should be exported from the pipeline, e.g. *`--out Token,Sentence,Parse`*.

Chapter 3. Analysis pipelines

For doing corpus analysis, AnnoLab integrates the Apache UIMA framework. This framework allows to compose so-called Analysis Engines (AE) into a pipeline. Each AE performs a particular analysis task, e.g. tokenisation, sentence boundary detection, part-of-speech analysis, deep parsing, etc. AnnoLab offers a simplified XML syntax for configuring analysis pipelines, but it can also use native UIMA aggregate AE or Collection Processor Engine (CPE) descriptors.

AEs that can be used by AnnoLab must have been packaged as PEARs (Processing Engine ARchive). This is a UIMA standard for packaging and automatically deploying components. These PEARs have to be installed into AnnoLab using the **pear install** command.

1. Pipeline descriptors

In addition to the standard UIMA Analysis Engine descriptors and CPE descriptors, AnnoLab supports a simplified pipeline descriptor format. The descriptor uses four sections: `<pipeline>`, `<stage>`, `<parameter>` and `<value>`. A `pipeline` can contain any number of stages and a stage can contain any number of parameters. A parameter may contain multiple values if the parameter allows that. Use the **pear explain** command to find out more about the parameters and values supported by a particular PEAR.

Figure 3.1. AnnoLab pipeline descriptor outline

```
<pipeline>
  <stage component="pear-id">
    <parameter name="param">
      <value>param-value</value>
    </parameter>
  </stage>
</pipeline>
```

For an example of a pipeline, please refer to the *Getting Started* chapter.

2. Mapping

The data models used by AnnoLab and UIMA are different. While AnnoLab is largely XML-oriented, UIMA employs the Common Analysis System (CAS) as its internal data structure. The mapping between the UIMA CAS and XML can be configured through an XML file. The basic structure of the file is as follows:

Figure 3.2. Outline of a mapping file

```
<mapping>
  <layer ...>
    <element ...>
      <relation .../>
      <feature .../>
    </element>
  </layer>
  <segment .../>
</mapping>
```

The `<layer>` sections defines an AnnoLab layer with annotations elements and features and declares which and how CAS types are mapped to these. The `<segment>` sections define which CAS types bear stand-off anchors and how they should be mapped to AnnoLab segments. The output of the `StandfordParserAnnotator` consists of a tree made up of constituents and words (and a lot of relations between them we will not talk about here). We want to map this tree to a tree layer. So we start defining a layer:

Figure 3.3. Step 1: Define the layer

```
<mapping>
  <layer type="tree" name="parse" segType="sequential">
  </layer>
</mapping>
```

The `type` attribute set to `tree` indicates that we want to map to a tree layer. The other possible value here is `positional` in order to create a positional layer. The attribute `name` defines the name the layer should have and can be chosen freely. The attribute `segType` indicates that the layer will be anchored on a sequential medium, as the parser works on text. Currently the only valid value here is `sequential`. Next we define that two CAS types Token and Constituent should be included into the layer. To do this we add `<element>` tags. The `casType` attribute of the tags bear the qualified name of the CAS types.

Figure 3.4. Step 2: Define the elements

```
<mapping>
  <layer type="tree" name="parse" segType="sequential">
    <element casType="de.julielab.jules.types.Constituent">
    </element>
    <element casType="org.annolab.uima.pear.stanford.Token">
    </element>
  </layer>
</mapping>
```

Now we define how these CAS types are mapped to annotation elements. That is we define how to interpret the relations Token and Constituent have to each other and how to interpret features they bear. If you know about the CAS you may wonder what the terms *relation* and *feature* mean here, because in UIMA-lingua there are only *feature structures*. What we call *feature* here is a feature structure of a primitive type: integers, strings, etc. When a feature of a feature structure is another feature structure, we say there is a *relation* between the two feature structures. For the moment we are only interested in the parse tree produced by the `StandfordParserAnnotator`. This tree is encoded in the `children` relation that is present in Constituent. To treat this relation as the dominance relation of the tree layer, we add a `<relation>` section:

Figure 3.5. Step 3: Define the dominance relation of the tree

```
<mapping>
  <layer type="tree" name="parse" segType="sequential">
    <element casType="de.julielab.jules.types.Constituent">
      <relation casName="children" mapAs="dominance"/>
    </element>
    <element casType="org.annolab.uima.pear.stanford.Token">
    </element>
  </layer>
</mapping>
```

Now we need to define that the `segments` relation of Token points to the stand-off anchors. So we a `<relation>` section to the respective `<element>` section.

Figure 3.6. Step 4: Define where the segments are located

```
<mapping>
  <layer type="tree" name="parse" segType="sequential">
    <element casType="de.julielab.jules.types.Constituent">
      <relation casName="children" mapAs="dominance"/>
    </element>
    <element casType="org.annolab.uima.pear.stanford.Token">
      <relation casName="segments" mapAs="segments"/>
    </element>
  </layer>
</mapping>
```

Finally we need to define the segments. Thus we add a `<segment>` section under `<mapping>`. The `StandfordParserAnnotator` uses the type GAMSequentialSegment to encode segments. This type bears two features `start` and `end` which are mapped to the start and end anchors of a sequential segment.

Figure 3.7. Step 5: Define how the segments are encoded

```
<mapping>
  <layer type="tree" name="parse" segType="sequential">
    <element casType="de.julielab.jules.types.Constituent">
      <relation casName="children" mapAs="dominance"/>
    </element>
    <element casType="org.annolab.uima.pear.stanford.Token">
      <relation casName="segments" mapAs="segments"/>
    </element>
  </layer>
  <segment type="sequential" casType="uima.tcas.Annotation">
    <anchor gamName="start" casName="start"/>
    <anchor gamName="end" casName="end"/>
  </segment>
</mapping>
```

For the `casName` there are two special values, namely `*` which matches the name of any relation or feature that is not match by any other `<relation>` or `<feature>` section and `this` which matches the current element. Well, and that's essentially it. We could add further `<relation>` and `<feature>` sections to fine-tune the mapping.

Table 3.1. Examples of `<relation>` and `<feature>` sections

Section	Description
`<relation casName="parent" mapAs="ignore"/>`	Completely ignore the *parent* relation.
`<relation casName="segments" mapAs="error"/>`	Treat the presence of the *segments* relation as an error. For example the Constituent type should not exhibit any segments, only the Token type should.
`<feature casName="value" mapAs="ignore"/>`	Completely ignore the feature *value*.
`<relation casName="this" mapAs="segments"/>`	Indicates that the element has a double-function as element bearing features and relations as well as stand-off anchors.
`<feature casName="*" mapAs="error"/>`	If the element bears any features that are not explicitly defined, trigger an error.

The following values are valid for `mapAs`:

Table 3.2. Valid `mapAs` values

Value	Description	`<relation>`	`<feature>`
`segments`	This contains the stand-off anchors	yes	no
`ignore`	Don't do anything with this	yes	yes
`error`	Trigger an error if this is present	yes	yes
`reference`	Map this as a reference	yes	no
`feature`	Map this as a feature	yes	yes
`default`	Automatically determine whether something should be mapped as reference or feature	yes	yes
`dominance`	In a tree layer this indicates the relation that makes up the tree	yes	no

3. Generalised Annotation Markup

Generalised Annotation Markup (GAM) is a set of XML tags and attributes that can be used to extend XML formats so they can be used in a multi-layer environment such as AnnoLab. The format has been inspired by existing XML annotation formats such as the CD3 format used by Systemic Coder [1], the MAS-XML format used by GuiTAR [2], the TEI XML [3] format as well as HTML. All use different tag-sets and encode different semantics by the XML document structure. However, they also have some similarities:

- they are used to annotate text;
- the complete text exists in the XML document;
- the text is not contained in attributes but in text nodes;
- iterating over all text nodes from the beginning to the end of the XML document yields the full text in the correct order.

Any XML format conforming to these four points may be called *document-centric XML* as the text being marked up by the XML tags provides the dominant structure. Document-centric XML formats can easily be converted for use within AnnoLab by adding *stand-off anchors* allowing the XML annotations and the underlying text to exist independently of each other. The idea is to leave the original XML format of as much as possible untouched and the conversion process from or to AnnoLab as simple as possible. During the conversion process two changes are applied:

- the text nodes are replaced by `gam:seg` tags representing segments that anchor an annotation layer to a signal;
- they are wrapped in a `gam:layer` tag that carries attributes such as `gam:id` and `name` that are necessary to address and handle of a layer within the framework.

All GAM tags traditionally reside in the XML namespace `http://www.linglit.tu-darmstadt.de/PACE/GAM` and use the namespace prefix `gam`.

GAM is used in the context of AnnoLab data stores and when exporting data from AnnoLab. Depending on the context, different elements are defined.

[1] `http://www.wagsoft.com/Coder/`
[2] `http://cswww.essex.ac.uk/Research/nle/GuiTAR/`
[3] `http://www.tei-c.org`

3.1. GAM in data stores

This section explains the GAM format used in data stores.

3.1.1. Segment

A segment identifies an area of a signal using a number of anchors. The GAM tag representing a segment is `gam:seg`. It carries two mandatory attributes:

`gam:sig` — the ID of the data store and of the signal the segment anchors to separated by a colon (`:`)

`gam:type` — determines the type of segment

Depending on the type of segment, additional attributes or child tags representing anchors are required. The following figure shows the XML serialisation of an abstract untyped segment.

Figure 3.8. XML for an abstract GAM segment

```
<gam:seg gam:type="..." gam:sig="default:03faa92e" />
```

AnnoLab implements only the segment type `seq` which stands for *sequential*. Sequential segments bear two additional attributes:

`gam:s` — the offset of the first character addressed by the segment.

`gam:e` — the offset of the character following the last character addressed by the segment. If the both attributes are equal the segment has a width of zero.

3.2. GAM - Integrated representation

This section explains the GAM format used in the integrated representation XML files produced by the **export**.

3.2.1. Overall structure

The following figure shows the outline of the overall structure of the integrated representation.

```
<gam:root>
  <gam:headers>
    <gam:header>
      ?
  <gam:annotations>
    <gam:layer>
      ?
        <gam:a>
  <gam:layout>
    <gam:root>
      <gam:seg>
        <gam:content>
        <gam:ref>
```

The integrated representation has three principal sections below the root element:

`gam:headers` — This section may contain one `gam:header` child for each layer or signal contained in the integrated representation. Its `gam:id` and `gam:name` attributes correspond with the respective attributes of a signal or layer. Headers are used to store arbitrary meta data about a resource, e.g. about authors, origin, licenses, etc. A `gam:header` el-

ement has a single child which is the root of the meta data XML document.

`gam:annotations` — This section contains one `gam:layer` child for each included annotation layer. Each of these children must bear `gam:id` and `gam:name` attributes. A `gam:layer` can have a single child element which is the root of the XML annotation. All text nodes originally contained in the annotation are replaced with `gam:a` elements each bearing a `gam:id` attribute. The original text is moved to the `gam:layout` section which references the annotation layers via `gam:ref` elements.

`gam:layout` — This section contains the common segmentation induced by all annotation layers. Each segment is represented by a `gam:seg` element which contains the annotated text in its `gam:content` child. `gam:ref` elements link to the `gam:a` elements in the annotation layers. Each segment contains one referent to each annotation layer annotation the portion of text represented by the segment.

Figure 3.9. Linking between text and annotations

Command Reference

Name

add-layer — add layer to existing signal

Synopsis

`add-layer` *`source`* *`destination`*

Description

This command can be used to add an annotation layer to an existing signal. The annotation has to be available as an XML file containing the same text (whitespace may vary) as the target signal. The layer will be added with the name `layout`. An existing layer with the same name will be overwritten if present.

Arguments

`source`	An XML annotation file that can be anchored on the specified signal.
`destination`	The signal to anchor the annotation on.

Examples

Add a HTML annotation from the file `manual.html` to the signal at `annolab://default/manual`.

```
$ annolab add-layer manual.html annolab://default/manual
```

Name

ae, cpe, pipeline — analyse data

Synopsis

`ae` [switches and options] *`descriptor`* [*`source`* ...] *`destination`*

`cpe` [switches and options] *`descriptor`* [*`source`* ...] *`destination`*

`pipeline` [switches and options] *`descriptor`* [*`source`* ...] *`destination`*

Description

These commands offer different ways of employing UIMA Analysis Engines to create annotation layers. Signals are recursively loaded signal from the given AnnoLab URI(s), analysed and the analysis results are saved as new layers.

If the **cpe** command is invoked any collection reader specified in the CPE descriptor will be ignored and the AnnoLab collection reader will be used instead. An AnnoLab-internal CAS consumer is automatically added to the CPE and used in addition to any CAS consumers already present.

If the destination of this command is a location on the file system, the command runs in *export mode*. In this mode it accepts all parameters and switches also accepted by the **export** command. Those are marked with *export mode* below.

Arguments

`descriptor`
: For each command the *`descriptor`* has a slightly different meaning. With **ae** it has to specify an Analysis Engine (AE) descriptor file, with **cpe** it has to specify a Collection Processing Engine (CPE) descriptor file and with **pipeline** it has to specify an AnnoLab pipeline descriptor file.

`source`
: Zero or more AnnoLab URIs from which the signals are loaded. If no sources are specified, the destination will be used as source and destination. This makes it easy to add new annotations to existing signals. If the source and destination locations differ, the signals will be copied to the new destination and the generated annotations will be anchored on those new signals.

`destination`
: An AnnoLab URI to which the signals and analysis results are saved. If no sources have been specified, the signals are loaded from the destination URI and analysis results are added to these signals.

Options

`--filter` *`layer`*
: Name of an annotation layer that contains filtering information. Filtered data will be hidden from the pipeline. Analysis components will not be able to see this data. This is useful for excluding for hiding parts of document, e.g. tables, figures or other parts of the document that are likely to be analysed incorrectly.

`--in` *`layer`*
: A comma-separated list of layer names. These layers are loaded into the CAS prior to running the pipelines. In this way a pipeline can make use of annotations present in the data store. If this parameter is not specified, no layers are loaded into the CAS.

Caution

Translating annotations from existing layers into the CAS it less well implemented and tested than translating annotations from the CAS to XML.

`--map` *`file`*
: Specify a mapping file that is to be used instead of the default built-in mapping file. This mapping controls how the annotations are mapped from XML to the UIMA CAS model and vice versa.

`--out` *`layer`*
: A comma-separated list of layer names. These layers are extracted from the CAS, translated to XML and added to the processed signals. If this parameter is not specified, the all layers specified by the mapping are extracted if the CAS type system contains the required types.

`--rules` *`file`*
: Specifies the rules to be used for filtering. The parameter accepts the name of a text file which specifies the attribute/value combinations that cause parts of the signal to be filtered out. See the examples below for more information.

`--suffix` *`suffix`*
: Per default exported files will be created with the suffix `.xml`. However, if you transform the integrated representation using the parameter *`--xslt`* you may want to specify another suffix. (export mode only)

`--xslt` *`file`*
: An XSLT file used to transform the integrated representation before writing it to a file. (export mode only)

Switches

`(+|-)drop-empty`
: When enabled all segments containing only white-space are excluded from the output. This results in a smaller integrated representation, but the original signal may not be extractable from it. Default: disabled. (export mode only)

`(+|-)split`
: When enabled one output file is generated for each signal. This is turned on per default if the *`destination`* is a directory. (export mode only)

`(+|-)report`
: Enable/disable display of a performance report for each processed resource.

`(+|-)trim`
: When enabled all trailing and leading whitespace in segments is trimmed. I.e. the segment boundaries are changed to the first and last non-whitespace characters. This switch does not affect the signal - the original signal can still be extracted from the integrated representation. Default: disabled. (export mode only)

Mapping specification

The mapping controls how annotations are translated from XML to the UIMA CAS model and vice versa. While the CAS is based on an object-graph formalism to encode annotations, AnnoLab uses either a list or a tree. Thus the CAS has to be decomposed into a set of list and tree layers for AnnoLab to make use of it. The decomposition is controlled by the mapping file.

The mapping file is an XML file and its root is the `<mapping>` element. Children of is are any number of `<layer>` and `<segment>` sections. The `<layer>` section specifies how types from the CAS are converted to and from XML. The `<segment>` sections specify how stand-off anchors are extracted from the CAS.

A layer has a *name*, a *type* and a *segment type* (`segType`). The name specified the name of the layer extracted from the CAS and can be anything. The type has to be either `tree` or `positional`. A positional layer can hold a list of annotations on non-overlapping segments - it can be used e.g. for simple part-of-speech annotations. A tree layer is more flexible. Annotated segments may overlap and annotation elements may form a hierarchy. If in doubt use the type `tree`. AnnoLab currently only supports the segment type `sequential` (segments with integer *start* and *end* offsets).

A `<layer>` contains any number of `<element>` sections. Each of these defines how one UIMA type, determined by the `casType` attribute, is mapped to an XML element. The last component of the UIMA type name is used as the XML element name - for `de.julielab.jules.types.Constituent` it is `Constituent`. A feature declared in an UIMA type is either of a primitive type (integer, string, etc.) or of another UIMA type. Features of primitive types are mapped by a `<feature>` section. The attribute `casName` specifies the feature to be mapped while the `mapAs` attribute controls how it is mapped to XML:

`ignore`	The feature is not mapped.
`error`	If the feature is present on this type an error is generated.
`feature`	The feature is mapped to an XML attribute. This is also the default for all primitive features that are not explicitly mapped.

Non-primitive features are mapped using <relation> section. Per default non-primitive features are not mapped to XML. They can be explicitly mapped as

`ignore`	The feature is not mapped.
`error`	If the feature is present on this type an error is generated.
`feature`	The feature is mapped to an XML attribute. If this mapping type is used the attribute select has to be present and specify the name of a primitive feature of the mapped type whose value will be used as the attribute value.
`leaf-segments`	If the annotation being mapped is a leaf, segments are extracted from the specified feature. An annotation is a leaf if it dominates no other annotation. (tree layer only)
`segments`	Segments are extracted from the specified feature.
`dominance`	The annotated specified by the feature is dominated by the annotation being mapped. This is usually used with the attribute `inverted` being set to `true`. In the example below the annotations dominated by a `Constituent` annotation are determined by finding all `Constituent` annotations referencing it in their `parent` attribute. (tree layer only)

If attribute `inverted` is set to true the specified feature does not have to be present in the mapped type. Instead it has to be present in some other type and reference the mapped type. In the example below the feature with the name `constituent` is present in the UIMA type `Theme` and it references the `Constituent` type. Whenever a `Theme` annotation references a `Constituent` annotation, the value of the primitive feature `label` (as indicated by the `select` attribute) is mapped to the XML attribute `theme` (as per the `xmlName` attribute).

```
<mapping>
 <layer type="tree" name="Parse-Theme" segType="sequential">
  <element casType="de.julielab.jules.types.Constituent">
   <feature casName="begin" mapAs="ignore"/>
   <feature casName="end" mapAs="ignore"/>
   <relation casName="parent" mapAs="dominance" inverted="true"/>
   <relation casName="cat" mapAs="feature"/>
```

```
   <relation casName="constituent" xmlName="theme" mapAs="feature"
             inverted="true" select="label"/>
  </element>
  <element casType="org.annolab.uima.pear.stanford.Token">
   <feature casName="begin" mapAs="ignore"/>
   <feature casName="end" mapAs="ignore"/>
   <feature casName="componentId" mapAs="ignore"/>
   <relation casName="parent" mapAs="dominance" inverted="true"/>
   <relation casName="this" mapAs="segments"/>
   <relation casName="posTag" mapAs="feature" select="value"/>
   <relation casName="lemma" mapAs="feature" select="value"/>
  </element>
 </layer>
</mapping>
```

Filter rules specification

Each line of the filter specification file corresponds to one rule. The first part of the line (before the colon) is the name of a feature (XML attribute). After the colon follows a regular expression. The rule matches if an annotation element bears the given feature and the feature value matches the regular expression.

The following example of a filter specification defines two rules. The first rule matches all XML elements bearing an attribute `class` with either the value `table` or `abstract`. The second rule matches all XML elements bearing an attribute `speaker` ending in `Smith`. Data covered by XML elements matching either of these rules is not included in the output.

```
class: table|abstract
speaker: .*Smith
```

Examples

To add layers to documents already in a data store simply do not specify any source. The following command will run the pipeline `pos-pipe.xml` and on each signal in the data store `default` and add the resulting annotations as layers to those signals.

```
$ annolab pipeline pos-pipe.xml annolab://default
```

Name

copy — copy data

Synopsis

`copy` [(+|-)fanout] *`source`* ... *`destination`*

Description

This command copies data from the file system into a data store, from a data store into another data store, or from a data store to the file system. Importers are used to convert data originating from the file system. AnnoLab ships with importers for plain text, HTML, XML, PDF and FLOB/FROWN. The **info** command shows all installed importers.

If the destination is on the file system, AnnoLab will dump signals as raw data (e.g. as text files) and layers as XML files. If the source is a file system directory or a collection within a data store, it is copied recursively.

Arguments

`source`	One or more locations from which to copy data to the destination. The sources can be files or directories on the local file system or data store locations.
`destination`	One location to which the data is copied. The location can be a directory on the local file system or a collection in a data store. If the destination is a directory or collection, it should always end in a `/`.

Switches

`(+|-)fanout`	Enable to create a sub-directory for each signal. This directory will contain the signal data and all layers. Default behaviour is to dump all signals and layers into one directory. This should only be used when copying to the file system. Default: off.

Examples

Import the PDF file `test.pdf` into the store default to the collection test.

```
$ annolab copy test.pdf annolab://default/test
```

Convert the PDF file `test.pdf` to text and HTML using AnnoLab's PDF importer and save the results on the file system in the directory *`some/directory`*.

```
$ annolab copy test.pdf some/directory
```

Convert the XHTML file `file.html` to text and HTML using AnnoLab's XML importer and save the results on the file system in the directory *`some/directory`*.

```
$ annolab copy file.html some/directory
```

Name

delete — delete from data store

Synopsis

`delete` [--layer *`name`*] *`location`* ...

Description

This command recursively deletes resources. Per default all resources are deleted. The `--layer` option can be used to delete only particular layers.

Arguments

`location`	One or more data store locations to delete. For safety reasons AnnoLab will not delete locations on the file system.

Options

`--layer` *`name`*	A comma-separated list of layers to be deleted. If this option is specified, all specified layers are deleted recursively on any signal within the given location. No signals are deleted if this option is present.

Examples

Delete all contents within the data store `default`.

```
$ annolab delete annolab://default
```

Delete only the layer `Token` from all signal in the data store `default`.

```
$ annolab delete --layer Token annolab://default
```

Name

exist — eXist client

Synopsis

`exist client` *`location`*

Description

AnnoLab comes with an embedded eXist XML database. This command starts the eXist GUI client that ships with the embedded eXist and configures it to access the data store underlying location given. The given location has to point to an eXist-based data store.

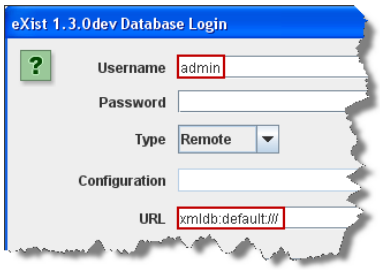
Arguments

`location`	An AnnoLab URI pointing to an eXist-based data store.

Login Problems

It may happen that AnnoLab fails to pre-configure the database login dialog. If the log in fails, make sure the *username* is `admin`, the *type* is `Remote` and the *URL* is `xmldb:`*`store`*`:///`. Replace store in the URL with the name of the data store addressed by the *`location`* argument given on the command line when starting the client. All other fields should remain empty.

Figure 18. eXist login settings



Examples

This command starts the eXist client on the data store `default`.

```
$ annolab exist client annolab://default
```

Name

export — export data

Synopsis

`export` [(+|-)drop-empty] [(+|-)split] [(+|-)trim] [--suffix *`suffix`*] [--xslt *`file`*] *`source`* *`destination`*

Description

This command recursively exports all signals and their layers to the given destination. A so-called *integrated representation* is created for each signal. This is an XML document containing all layers on that signal plus the signal itself. Such a document is suitable for transformation to arbitrary target formats using XSLT style sheets.

Arguments

`source`	A data store location.
`destination`	A location on the file system. This can be a file name or the path to an existing directory.

Options

`--suffix` *`suffix`*	Per default exported files will be created with the suffix `.xml`. However, if you transform the integrated representation using the parameter *`--xslt`* you may want to specify another suffix.
`--xslt` *`file`*	An XSLT file used to transform the *integrated representation* before writing it to a file.

> **Tip**
>
> It is a good idea to write the integrated representation to files without using the *`--xslt`* parameter and then transform them afterwards using a fast XSLT processor such as xsltproc. This approach especially saves time when you want export to multiple target formats.

Switches

`(+|-)drop-empty`	When enabled all segments containing only white-space are excluded from the output. This results in a smaller integrated representation, but the original signal may not be extractable from it. Default: disabled.
`(+|-)split`	When enabled one output file is generated for each signal. This is turned on per default if the *`destination`* is a directory.
`(+|-)trim`	When enabled all trailing and leading whitespace in segments is trimmed. I.e. the segment boundaries are changed to the first and last non-whitespace characters. This switch does not affect the signal - the original signal can still be extracted from the integrated representation. Default: disabled.

Examples

In the following example we create a directory called `integrated` and then export all data in the data store `default` to that directory.

```
$ mkdir integrated
$ annolab export annolab://default integrated
```

In the next example we use an XSLT style sheet called `plaintext.xslt` to extract only the signal (text) from the integrated representation. The suffix of the exported files is explicitly set to `.txt`.

```
$ annolab export --xslt plaintext.xslt --suffix .txt
          annolab://default text
```

Name

filter — extract filtered signals

Synopsis

`filter` [--filter *`layer`*] [--rules *`file`*] *`source`* ... *`destination`*

Description

This command copies signals (without layers), optionally filtering the signals by removing parts of the signal annotated for a particular features in the specified layer. The features causing a part of the signal to be removed are either a built-in default set or the set of features given in the optional rule file. If the source is a collection, all signals are filtered recursively.

Arguments

`source`	One or more locations from which to copy data to the destination. The sources can be files or directories on the file system or data store locations.
`destination`	One location to which the data is copied. The location can be a directory on the file system or a collection in a data store. The location should always terminate in a `/` to indicate that it is a collection.

Options

`--filter` *`layer`*	Specifies a layer to be used for filtering. If a filter layer is specified, per default it will remove any parts of the signal that have been annotated with the feature `class` with the values `keywords`, `bibliography`, `figure`, `footnote` , `formula`, `ignore`, `pagebreak`, `table` or `affiliation`. To change this use the `--rules` option.
`--rules` *`file`*	Specifies the rules to used. The parameter accepts the name of a text file in which you can specify which attribute/value combinations that cause parts of the signal to be filtered out. See the examples below for more information.

Filter rules specification

Each line of the filter specification file corresponds to one rule. The first part of the line (before the colon) is the name of a feature (XML attribute). After the colon follows a regular expression. The rule matches if an annotation element bears the given feature and the feature value matches the regular expression.

The following example of a filter specification defines two rules. The first rule matches all XML elements bearing an attribute `class` with either the value `table` or `abstract`. The second rule matches all XML elements bearing an attribute `speaker` ending in `Smith`. Data covered by XML elements matching either of these rules is not included in the output.

```
class: table|abstract
speaker: .*Smith
```

Examples

The following command recursively copies a filtered version of the complete data store `annolab://default/` to the directory `filteredOutputDirectory`.

```
$ annolab filter --filter Layout annolab://default/
    filteredOutputDirectory/
```

Assuming the above is the content of a file named `filterRules.properties`, the following command uses this file to control the filter.

```
$ annolab filter --rules filterRules.properties
    --filter Layout annolab://default/ filteredOutputDirectory/
```

Name

help — self-documentation

Synopsis

`help` [*`command`*]

Description

Invoking this command without any arguments prints a list of the available commands. Optionally a command can be given as the only argument. In this case detail help for the command will be printed.

Arguments

`command`	A command for which detail information should be printed.

Examples

Get a list of all available commands.

```
$ annolab help
```

Get help for the **copy** command.

```
$ annolab help copy
```

Name

ims2uam — selective annotation

Synopsis

`ims2uam` [(+|-)offsets] [--remap *`from to`*] *`source destination`*

Description

If a corpus is large the resources to annotate it completely and exhaustively may not be available. A query can be used to extract particularly interesting sections of the corpus for further annotation.

Arguments

`source`	The file containing the results exported from the IMS CWB.
`destination`	The directory in which the new UAM Corpus Tool project should be created. The directory has to exist and should be empty.

Options

`--remap` *`from to`*	Defines a re-mapping of AnnoLab URIs during the integration. This can be used if the project was generated from a different data store than it will be re-integrated into.

Switches

`(+|-)offsets`	Turn on when offsets are present in the IMS CWB results. If they are present, they can later be used to get the original signal. If this is turned off, it is unlikely that the manual annotations made in the UAM Corpus Tool can be re-integrated into the corpus. Default: on.

IMS CWB database requirements

This command can be used to convert a search result from the IMS CWB to a UAM Corpus Tool project. It retains information about the original location of the extracted data in a special layer of the UAM Corpus Tool project, which can be used later to integrate the annotation with the complete corpus.

The IMS CWB corpus database must have been created with at least the following positional attributes:

`uri`	AnnoLab URI of the source signal
`s`	start-offset of the segment within the signal
`e`	end-offset of the segment within the signal

These must appear in exactly the given relative order: `uri`, `s`, `e`. There can be other positional attributes present in the database.

To integrate the partial annotations made in the generated UAM Corpus Tool project back onto the full texts, use the command **uam2annolab**.

Tolerance to changes

The data store from which the IMS CWB database has been created has to be available when this command is used. For best results the signals not have changed between the time the IMS CWB database has been created

and the time the command is used. If the data has changed, AnnoLab tries its best to fix the offsets so that the generated UAM Corpus Tool project is aligned to the changed signals.

Examples

After you have prepared an IMS CWB database fulfilling the above requirements, you can log into it with **cqp** and perform queries as usual.

Once you wish to export a query result to UAM Corpus Tool, you need to CONFIGURE IMS CWB to produce the output format needed by **ims2uam**. Turn on only the display of the three positional parameters mentioned above (`+uri +s +e`) and turn off any other parameters:

```
MYCORPUS> show +uri +s +e;
```

Run the query again storing the results in a variable. The use the cat command to save to results to a file. Here we did a query extracting all sentences containing the word *algorithm* within all texts in `annolab://default/A`.

```
MYCORPUS> results = "algorithm" ::
          match.document_uri="annolab://default/A/.*" within s;
MYCORPUS> cat results > "cqpresults.txt";
```

Now leave **cqp** and run **ims2uam** on the saved results. First create an empty directory to take up the project. For this example the directory is called `target`.

```
$ mkdir target
$ annolab ims2uam queryresults.txt target
```

After this you will find a UAM Corpus Tool project named `project` in the target directory.

Tip

The original location information taken from the positional parameters `uri`, `s` and `e` is stored in the comment field of the segments in the `annolabSync` layer. They are encoded as a JSON string.

Name

info — information about the installation

Synopsis

```
info
```

Description

This command shows some pieces of information about the AnnoLab installation.

Examples

Get some basic information about the AnnoLab installation.

```
$ annolab info
```

Name

lemmatize — TreeTagger wrapper

Synopsis

`lemmatize list`

`lemmatize process` [(+|-)split] [--filter *`layer`*] [--suffix*`suffix`*=*`value`*] [--xslt *`file`*] *`source`* ... *`destination`*

Description

This command provides a small processing pipeline using a simple tokeniser and TreeTagger for part-of-speech tagging and lemmatisation.

The sub-command **list** prints a list of the available tagging models. One of these models has to be specified when using the **process**

The sub-command **process** runs the pipeline. It can optionally filter the texts (hide certain parts from the analysis components as not to confuse them) and transform the results from the *integrated representation format* using an XSLT style sheet.

This command is deprecated. It requires that the module `org.annolab.module.treetagger.TreeTaggerModule` has been configured in the `annolab.xconf` file. Instead the **pipeline** command should be used in conjunction with the `treetagger-ae` PEAR. This combination provides more flexibility and control and does not require a local installation of TreeTagger or modifications to the `annolab.xconf` file.

Arguments

`model`	One of the models found by the **list** command.
`source`	One or more locations from which to read data to be annotated. The sources can be files or directories on the local file system or data store locations.
`destination`	A location on the file system. This can be a file name or the path to an existing directory.

Options

`--filter` *`layer`*	Specifies a layer to be used for filtering. If a filter layer is specified, per default it will remove any parts of the signal that have been annotated with the feature `class` with the values `keywords`, `bibliography`, `figure`, `footnote` , `formula`, `ignore`, `pagebreak`, `table` or `affiliation`. This command does not allow to change the filter rules.
`--suffix` *`suffix`*	Per default exported files will be created with the suffix `.xml`. However, if you transform the integrated representation using the parameter *`--xslt`* you may want to specify another suffix.
`--xslt` *`file`*	An XSLT file used to transform the *integrated representation* before writing it to a file.

Switches

`(+|-)split`	When enabled one output file is generated for each signal. This is turned on per default if the *`destination`* is a directory.

Built-in XSLT style sheets

The command comes with a few built-in XSLT style sheets.

`builtin:bnc`	Formats output in BNC SGML format suitable for example for the WordSmith Tools. The output encoding is UTF-16LE, which is the encoding required by WordSmith.
`builtin:imscwb`	Formats output in a tab separated suitable for importing into the IMS Corpus Workbench. The included positional fields are in the following order: signal, part-of-speech tag, lemma, signal uri, start-offset, end-offset. The output encoding is UTF-8.
`builtin:wconcord`	Formats suitable for WConcord. The output format is *`signal_postag`* . The encoding is UTF-8.

Module configuration (`annolab.xconf`)

This command requires a local installation of TreeTagger. After you have installed TreeTagger on your system, AnnoLab has to be instructed where to find it and which tagging models are available. In order the following section has to be added to the `modules` section of the `annolab.xconf` file. You have to adapt the following example to your installation. Put the absolute path to your TreeTagger executable into the `executable` section. Maintain a `model` section for each tagging model you have. For each model specify a name (`name` section), the absolute path to the model file (`file` section), the model encoding (`encoding` section) and a simple full sentence in the language on which the model was trained (`flushSequence` section). Note that there has to be a space between each token of that sentence including before the final full-stop (`.`). Optionally a substitution file can be specified (`substitutions` section).

```
<module class="org.annolab.module.treetagger.TreeTaggerModule">
 <executable>/Applications/treetagger/bin/tree-tagger</executable>
 <models>
  <model>
   <name>english</name>
   <file>/Applications/treetagger/lib/english.par</file>
   <encoding>Latin1</encoding>
   <flushSequence>This is a dummy sentence .</flushSequence>
   <substitutions>/Applications/treetagger/lib/en.xml</substitutions>
  </model>
 </models>
</module>
```

Substitution file

A substitution file can be used to substitute characters or sequences of characters that are known to be broken or to be misinterpreted by an analysis component. They can be replaced by the what is known to be the correct character or sequence or by some sensible substitute the analysis component can deal with.

For example a tagging model may not know about the Unicode quotes “ and ” and consequently tag them wrong. The example below substitutes such quotes with a regular quote " (written as the XML entity `&quot;` here because in an XML file literal quotes have to be written as XML entities). The example also substitutes some greek letters with the letter `x`. The model does not know about greek letter, but knows that `x` is usually a mathematical symbol and thus tags it as `SYM`.

Figure 19. Example substitutions file

```
<?xml version="1.0" encoding="UTF-8"?>
<substitutionTable>
 <substitution orig="“" subst="&quot;"/>
 <substitution orig="”" subst="&quot;"/>

 <!-- x is tagged as SYM -->
 <substitution orig="#" subst="x"/>
 <substitution orig="#" subst="x"/>
 <substitution orig="#" subst="x"/>
 <substitution orig="#" subst="x"/>
 <substitution orig="#" subst="x"/>
</substitutionTable>
```

Examples

Print a list of available tagging models.

```
$ annolab lemmatize list
```

Create a new directory called `results`. Then process all files in the directory `texts` using the model `english` and write one result file per text. Transform the results using the XSLT file `bnc.xslt`.

```
$ mkdir results
$ annolab lemmatize process +split --format bnc.xslt english texts results
```

Name

list — list data store contents

Synopsis

`list` *`source`*

Description

This command recursively lists the content of a data store. Each signal is listed along with its annotation layers listed indented below it.

Arguments

`location`	One location to recursively list the content of.

Examples

List the contents of the data store `default`.

```
$ annolab list annolab://default
```

Name

matchlist — statistical distribution

Synopsis

`matchlist` [(+|-)lowercase] [-f *`fields`*] -m *`model`* *`source`* *`destination`*

Description

Given a collection or a directory this command generates a table with all children of the given collection or a directory listed on the X axis and tags or signal data on the Y axis. For example one can get a table showing the distribution of lemmas (Y axis) across all texts in a collection (X axis), with one row per lemma and column per one sub-collection. Or you can get a table showing the distribution of part-of-speech tags (Y axis) across a set of directories (X axis). The command uses TreeTagger to generate the part-of-speech and lemma. The output is a CSV file in UTF-8 (works well with OpenOffice Calc).

It is mandatory to specify a model using the parameter *`-m`*. To get a list of available models use the command **lemmatize list**. This command requires the same set up as the **lemmatize** command. Please refer to the documentation of that command to find out how to set up AnnoLab to work with a local installation of TreeTagger.

The source has to be a collection in a data store or a directory on the file system. For each child of the source a column will be created in the output table. Assuming the following data store structure the source `annolab://default/A` results in a table with the columns `Text1` and `Text2` while the source `annolab://default` results in a table with the columns `A` and `B`.

```
default
  +-- A
  |   +-- Text 1
  |   +-- Text 2
  +-- B
      +-- Text 3
```

Per default the Y axis of the table shows the lemma and part-of-speech. This can be changed using the option `-f`, which takes a comma-separated list of field names. Note that field names are case-sensitive.

Arguments

`source`
: One locations from which to read data to be annotated. The source can be a directory on the local file system or collection in a data store.

`destination`
: The name of a file on the file system to which to write the results. For your convenience use a file name ending in `.csv`.

Options

`-f` *`field`*
: A comma-separated list of field names to be used as row labels. Note that field names are case-sensitive. These are the available fields:

Table 3. Available fields

Field	Description
posTag	Part-of-Speech
lemma	Lemma
OFS_START	Start offset of the signal (in characters).

Field	Description
OFS_END	End offset of the signal (in characters).
SIGNAL	Signal data.
SIGNAL_ID	ID of the signal. This is unique per data store. The ID is not available when the source location is on the file system.
SIGNAL_NAME	Name of the signal.
SIGNAL_URI	AnnoLab URI of the signal.

`-m` *`model`* — One of the models found by the **lemmatize list** command.

Switches

`(+|-)lowercase` — Enable to force all data on the Y axes to be lowercase. That means that e.g. numbers for `Be` and `be` are conflated into a single row.

Examples

The following example accesses the data store `default`. TreeTagger is invoked with the model `english` to generate the part-of-speech tags. Data on the horizontal axis consists of the signal and part-of-speech tags and is lowercase. The output is written to the file `matchlist.csv`.

```
$ annolab matchlist +lowercase -f SIGNAL,posTag -m english
    annolab://default matchlist.csv
```

Name

pear — PEAR management

Synopsis

`pear install` *`file`* …

`pear uninstall` *`name`*

`pear explain` *`name`*

`pear list`

Description

This command allows to manage UIMA PEARs in AnnoLab. A PEAR is a packaged analysis component that can be used in a pipeline. Before a PEAR can be used in AnnoLab it has to be installed. When it is no longer needed or before a new version is installed, a PEAR has to be uninstalled. It is also possible get a list of the currently installed PEARs and to get an detailed information about an installed PEAR.

The **install** sub-command is used to install a PEAR into AnnoLab. When a PEAR is installed, it receives a unique name. To work with an installed PEAR in a pipeline or with any commands, this name has to be used to address the PEAR.

The **uninstall** sub-command is used to uninstall a PEAR, either because it is no longer needed or as a preparation to install a newer version.

The **explain** sub-command prints detail information about a PEAR. In particular it prints a list of configuration parameters that can be changed in a pipeline descriptor file to configure the analysis component. Also a list of input and output capabilities is printed. Within a pipeline the analysis components need to be ordered in such a way that all data a particular PEAR lists as its inputs have been produced by previous pipeline stages. The data produced by an analysis component is listed as its outputs.

The **list** sub-command prints a list of all installed PEARs.

Arguments

`file`	A PEAR file.
`name`	The name of a PEAR as shown by the **pear list** command.

Examples

Uninstall a the PEAR `java-sentence-ae` so a new version can be installed..

```
$ annolab pear uninstall java-sentence-ae
```

Install the PEAR `java-sentence-ae` from the file `java-sentence-ae.pear`.

```
$ annolab pear install java-sentence-ae.pear
```

Get a list of all installed PEARs.

```
$ annolab pear list
```

Get more information about the `java-sentence-ae` PEAR.

```
$ annolab pear explain java-sentence-ae
```

Name

query, mquery — query a data store

Synopsis

`query` [(+|-)unanchor] [--query *`file`*] [--repeat *`X`*] [-V[*`var=value`*...]] [--xslt *`file`*] *`template source`* [*`destination`*]

`mquery` [(+|-)unanchor] [--query *`file`*] [--repeat *`X`*] [-V[*`var=value`*...]] [--xslt *`file`*] *`source`* [*`destination`*]

Description

These commands allow to perform queries from the command line. Queries can be run completely manually using the **mquery** command or using *query templates* with the **query** command.

In manual mode, the query is read in from the terminal after the command has been started. Alternatively you can create a text file containing the query and feed it to the command using input redirection.

In template mode a query template descriptor file has to be specified as the *`template`* argument.

Arguments

`source`	A data store location. The addressed data store has to support querying. Currently only the *RepoDatastore* and *ExistDatastore* support querying.
`destination`	The output file. If omitted the output goes to the terminal.

Options

`--query` *`file`*	File from which the query is read in manual mode.
`--repeat` *`X`*	Repeat the query *X* times. With this option no output is generated. Instead run-time statistics are printed at the end. The serialisation pipeline is completely disabled so that only the performance of the query itself (without any XSLT transformation or serialisation overhead) can be measured.
`-V` *`var`* = *`value`*	This parameter can be used to set the value of a free query variable. E.g. `-Vword=algorithm` sets the variable `word` in the query to the value `algorithm`. The parameter can be specified multiple times to set multiple variables. If a template requires a variable not set in this way, it will prompt the user for a value.
`--xslt` *`file`*	In manual mode this is the path to an XSLT file used to transform the query output. In template mode it is the name of one of the output formats available in the template. Specifying `none` turns off XSLT transformation.

Switches

`(+|-)unanchor`	Enable/disable replacement of all segments in the query results with the content they are referring to. This allows to perform queries returning segments instead of using e.g. `txt:get-text()` in the query. Unfortunately it means that the complete query output needs to be buffered in memory - thus for very large results this may not work.

Query templates

A query template consists of a template descriptor file (actually a Java property file), a query file and optionally XSLT files that can be used to transform the query results.

For our example we will create a query template file called `query-example.template` and a directory `query-example`. Into the latter we put the query file and the XSLT files.

Each line of the template descriptor file consists of a property name followed by an equals sign (=) followed by the property value. For properties pointing to files, paths are always treated as being relative to the template descriptor file. The `query-example.template` file should look like this:

Figure 20. Query template descriptor file: `query-example.template`

```
summary = This is an example query searching for a word
description = Here we would put a much longer description.
query = query-example/query.xq
xslt.default = query-example/html.xslt
xslt.html = query-example/html.xslt
xslt.csv = query-example/csv.xslt
variable.word.prompt = Search word
variable.word.description = Word to search for
```

The property `summary` specifies a short summary of what the query does. A longer description can be given using the property `description`.

The property `query` defines the name of the file containing the query. This is relative to the location of the template descriptor file. For more information on writing the query file `xquery.xq` itself, see the chapter on querying.

Following are optional properties starting with `xslt.`. These specify XSLT style sheets that can be used to transform the output of the query. If the property `xslt.default` is present, the specified XSLT file is always used unless a *`--xslt`* parameter is used to explicitly specify another. In the given case we could specify `--xslt csv` to use the XSLT style sheet specified with the property `xslt.csv` instead of the default. Since you have complete control of the query results, any discussion of how to do the XSLT style sheets is omitted here. It is suggested that to keep the query simple and the results plain and use XSLT style sheets to do aggregation and/or formatting.

Free query variables need to be declare using properties starting with `variable.`. The example above declares a variable named `word`. When the user does not specify a variable value using the *`-V`* parameter, the value is asked for. The description from `variable.word.description` is shown on the screen and the user can enter the value after the prompt specified in `variable.word.prompt`.

Figure 21. AnnoLab asking for the value of an unset query variable

```
--- Unset variable ---
Description: Word to search for
Search word: _
```

Examples

Run a query against the data store `default` using input redirection. The file `query.xq` contains the actual query.

```
$ annolab mquery annolab://default < query.xq
```

Profile the query template `word.template` by running it 10 times in a row searching for the word `be`.

```
$ annolab query --repeat 10 -Vword=be word.template annolab://default
```

Run the query template `word.template` using the *html* XSLT style sheet specified in the template descriptor and saving the results to `results.html`.

```
$ annolab query --xslt html word.template annolab://default results.html
```

Name

uam2annolab — partial annotation re-integration

Synopsis

`uam2annolab` [--remap *`from to`*] {--layer *`name`* } [--as-layer *`name`*] *`source`*

Description

This is the sister-command to **ims2uam**. It re-integrates the partial annotations made in an UAM Corpus Tool project generated with **ims2uam** back into the corpus.

Arguments

`source`	The directory containing the UAM Corpus Tool project.

Options

`--layer` *`name`*	The layer from the UAM Corpus Tool project that should be re-integrated.
`--as-layer` *`name`*	The name of the layer into which the data should be integrated. Per default this is the same that has been specified for `--layer`.
`--remap` *`from to`*	Defines a re-mapping of AnnoLab URIs during the integration. This can be used if the project was generated from a different data store than it will be re-integrated into.

Switches

`(+|-)offsets`	Disable if no offsets are present in the IMS CWB results. Without offsets, it is likely that re-integration will not be possible. Default: on.

Tolerance to changes

The data store into which the annotations should be integrated already has to contain the signals. For best results the signals not have changed between the time the IMS CWB database has been created and the time the command is used. If the data has changed, AnnoLab tries its best to find the annotated text in the changed signal and transfer the annotations accordingly.

Examples

The following command re-integrates the layer `Transitivity` from the UAM Corpus Toolproject in the directory `meaning-C1`. The `--remap` option is used to integrate the annotations on the results originally extracted from `annolab://dfg/archive/` into `annolab://default/dfg/`.

```
$ annolab uam2annolab --remap annolab://dfg/archive/ annolab://default/dfg/
    --layer Transitivity meaning-C1
```

Name

webserver — AnnoLab server

Synopsis

`webserver` [switches and options] *`descriptor`* [*`source`* ...] *`destination`*

Description

This command starts AnnoLab as a server. Two services exposed by the server are provided directly by the embedded eXist database: the eXist REST service and the XML-RPC service.

The REST service runs at `http://localhost:8080/stores/`*`store`*`/exist/rest`.

The XML-RPC service runs at `http://localhost:8080/stores/`*`store`*`/exist/xmlrpc`.

In both URLs *`store`* has to be replaced by the name of the data store that was being addressed by the AnnoLab URL given as a parameter when starting the server.

Both services offer way of modifying the content of the eXist database but neither those methods nor the built XQuery management functions of eXist should be used. Instead the XQuery extensions provided by AnnoLab for data store management should be used to add or remove content.

Queries issued through these services have access to the same AnnoLab XQuery extensions that are available for the **mquery** and **query** commands.

For more information on how to use these services, please refer to the eXist documentation at `http://exist.sourceforge.net/`.

Per default the server can only be access locally. To access the server from remote machines or to change the port the server is running on, please refer to the file `configuration/config.ini` in the AnnoLab installation directory. There you can modify the two configuration items `org.eclipse.equinox.http.jetty.http.host` and `org.osgi.service.http.port`.

Press **CTRL**+**C** to terminate the server.

Arguments

`source`
: An AnnoLab URIs pointing to a data store which will be exposed through by the server. Only one data store can be exposed at a time. The addressed data store has to be eXist-based.

Examples

To start a server for the data store `default` use:

```
$ annolab webserver annolab://default
```

XQuery Extensions Reference

Name

ds:layer — access annotation layers (http://annolab.org/module/repo/exist/xq/datastore)

Synopsis

```
ds:layer($name as xs:string) as xs:string

ds:layer($uri as xs:string*, $name as xs:string) as element()
```

Description

Find all layers of the given name. Layers can be searched for in the whole data store or only within a particular collection and its sub-collections.

Arguments

$name	The layer name.
$uri	An AnnoLab URI addressing a collection in the data store. The addressed data store has to be identical with the data store against which the query is run. The **query** and **mquery** commands define the variable `$QUERY_CONTEXT` that can be used here. The value of the variable is the *source* argument to those commands.

Examples

This example queries for all XHTML headers (`h1`) in the layer `Layout` within `annolab://default`.

```
$ annolab mquery annolab://default

Enter query. Press <CTRL-D> when done to start execution.

declare namespace xhtml="http://www.w3.org/1999/xhtml";
ds:layer($QUERY_CONTEXT, "Layout")//xhtml:h1
CTRL+D
```

Name

ds:meta — access meta data (http://annolab.org/module/repo/exist/xq/datastore)

Synopsis

```
ds:meta($uri as xs:string) as element()
```

Description

Get the meta data of the resource addressed by the given AnnoLab URI. Meta data can be stored with a layer when it is imported using the `manage:import()` function.

Arguments

`$uri`	An AnnoLab URI.

Examples

This example gets the meta data of a signal at `annolab://default/SomeText`.

```
$ annolab mquery annolab://default

Enter query. Press <CTRL-D> when done to start execution.

ds:meta("annolab://default/SomeText")
CTRL+D
```

Name

ds:signal — access signals (http://annolab.org/module/repo/exist/xq/datastore)

Synopsis

```
ds:signal($uri as xs:string) as xs:string
```

Description

Get the contents of the signal addressed by the given AnnoLab URI.

Arguments

$uri	An AnnoLab URI addressing a signal.

Examples

This example gets the contents of a signal at `annolab://default/SomeText`.

```
$ annolab mquery annolab://default

Enter query. Press <CTRL-D> when done to start execution.

ds:signal("annolab://default/SomeText")
CTRL+D
```

Name

manage:delete — delete from a data store (http://annolab.org/annolab/manage)

Synopsis

```
manage:delete($uri as xs:string) as xs:string*

manage:delete($uri as xs:string, $name as xs:string) as xs:string*
```

Description

Delete the signal at the specified location. The second argument can be the name of a layer. In that case the layer is deleted from the signal. The signal is deleted as well if the layer being deleted is the last one on that signal.

The return value of the command is a sequence of messages stating which signal and layers have been deleted.

Arguments

$name	A layer name.
$uri	An AnnoLab URI addressing a signal in the data store.

Examples

This example deletes the layer `Layout` from the signal `annolab://default/SomeText`.

```
$ annolab mquery annolab://default

Enter query. Press <CTRL-D> when done to start execution.

manage:delete("annolab://default/SomeText", "Layout")
CTRL+D
```

Name

manage:import — import into a data store (http://annolab.org/annolab/manage)

Synopsis

```
manage:import($source as xs:string, $dest as xs:string,
              $mimetype as xs:string, $name as xs:string) as xs:string*
```

Description

Import a layer from an URL or from a location within the XML database. If there is already a signal at the destination URI, the layer will be anchored on that signal. Otherwise the text is extracted from the layer and be used to create a new signal at the destination URI.

The return value of the command is a sequence of messages stating which signal and layers have been deleted.

Arguments

$dest	An AnnoLab URL indicating the destination. The destination can be an existing signal to which a new layer should be added. It can also be the name of a non-existing signal which is then extracted from the layer being imported and stored at the given location.
$mimetype	A MIME-type information can be stored with each layer. If in doubt, use `application/xml`.
$name	The name with which the layer is being added to the signal.
$source	An URL or XML:DB URL from which to read the layer. It is possible to access any URL type known to Java, such as *http*, *ftp* or *file*. The source has to be a valid XML file.

Examples

This example shows how to import the file `README.xhtml` from the current directory to the data store location `annolab://default/README`. Since no signal existed at this location before, a new signal is created and the XHTML is added as the layer `Layout` to that signal with the MIME-type for XHTML data `application/xhtml+xml`.

```
$ annolab mquery annolab://default

Enter query. Press <CTRL-D> when done to start execution.

manage:import("file:SomeText.xhtml", "annolab://default/SomeText",
              "application/xhtml+xml", "Layout")
CTRL+D
```

Name

seq:containing, seq:contained-in, seq:same-extent, seq:overlapping, seq:left-overlapping, seq:right-overlapping — containment and overlap (http://annolab.org/module/exist/nativexq/sequential)

Synopsis

```
seq:containing($A as element()*, $B as element()) as element()

seq:contained-in($A as element()*, $B as element()) as element()

seq:same-extent($A as element()*, $B as element()) as element()

seq:overlapping($A as element()*, $B as element()) as element()

seq:left-overlapping($A as element()*, $B as element()) as element()

seq:right-overlapping($A as element()*, $B as element()) as element()
```

Description

These functions can be used to filter a set of elements A with respect to a set of elements B and a relation *R* between the two. These relations can be *containing*, *contained-in*, *same-extent*, *overlapping*, *left-overlapping* and *right-overlapping*.

The functions all work following the same principle: *return each a in A which is in the given relation* R *with any b in B.*

When an element a and/or b of the sequences A or B is not a segment, a segment is calculated from the left-most and the right-most positions addressed by any descendant segment of the element and used for comparing the two elements.

Arguments

$A	A sequence of elements being filtered.
$B	A sequence of elements against each of the elements in the sequence *$A* is matched.

Examples

Assume the layer `Speaker` contains `turn` elements with the speaker encoded in the attribute `speaker`. Assume also that the layer `Token` contains `segment` elements with the part-of-speech encoded in the attribute `posTag`. The following query extracts all nouns spoken by the speaker Chad.

```
$ annolab mquery annolab://default

Enter query. Press <CTRL-D> when done to start execution.

element results {
  seq:contained-in(
    ds:layer($QUERY_CONTEXT, "Token")//segment[starts-with(@posTag, "N")],
    ds:layer($QUERY_CONTEXT, "Speaker")//turn[@speaker="Chad"])
}
CTRL+D
```

Name

seq:grow — calculating covering segment (http://annolab.org/module/exist/nativexq/sequential)

Synopsis

```
seq:grow($A as element()*) as element()
```

Description

The function `seq:grow` calculates a segment from the left-most and the right-most positions addressed by any descendant segment of the sequence of elements passed to it. The result is a single segment that covers all the data addressed by the sequence. This function is commonly used in conjunction with `txt:get-text` to retrieve a meaningful and easily readable section of the signal that includes whitespace and line breaks.

Arguments

$A	The sequence of elements for which the covering segment is calculated.

Examples

See the example for the function `txt:get-text()`.

Name

tree:following-sibling, tree:preceding-sibing — sibling navigation (http://annolab.org/annolab/tree)

Synopsis

```
tree:following-sibling($A as node()*, $n as xs:integer) as node()

tree:preceding-sibling($A as node()*, $n as xs:integer) as node()
```

Description

These functions get *$n* th the following or preceding siblings of the nodes in the sequence of nodes *$A*. If the parameter *$n* is `0` the input list is returned, if it is `1` all immediately following siblings are returned, if it is `2`, all 2nd following siblings are returned, and so on. This function was implemented to be more efficient than using the *following-sibling* axis.

Arguments

$A	A sequence of nodes.
$n	The offset of the siblings to return.

Examples

Assume also that the layer `Token` contains `segment` elements with the part-of-speech encoded in the attribute `posTag`. The following query extracts all verbs and one token to the left and to the right of them.

```
$ annolab mquery annolab://default

Enter query. Press <CTRL-D> when done to start execution.


element results {
  for $v in ds:layer($QUERY_CONTEXT, "Token")//segment[
      starts-with(@posTag, "V")]
  return element result {
    tree:preceding-sibling($v, 1),
    $v,
    tree:following-sibling($v, 1)
  }
}
CTRL+D
```

Name

txt:find — pattern-based search (http://annolab.org/annolab/textual)

Synopsis

```
txt:find($elements as element()*, $pattern as xs:string) as element()
```

Description

Search for the given regular expression pattern in the areas addressed by the elements and segments given in the first parameter. Returns a set of segments indicating the matches.

Arguments

$elements	The XML elements in which to search.
$pattern	A regular expression.

Examples

Find all occurrences of the string `zero knowledge` within XHTML paragraphs (`xhtml:p`) of the layer `Layout`.

```
$ annolab mquery annolab://default

Enter query. Press <CTRL-D> when done to start execution.

declare namespace xhtml="http://www.w3.org/1999/xhtml";
txt:find(ds:layer($QUERY_CONTEXT, "Layout")//xhtml:p, "zero knowledge")
CTRL+D
```

Find all XHTML paragraphs (`xhtml:p`) in the layer `Layout` containing the string `zero knowledge`. The wildcards (`.*`) at the beginning and the end of the search pattern cause the whole paragraph to be returned as a result.

```
$ annolab mquery annolab://default

Enter query. Press <CTRL-D> when done to start execution.

declare namespace xhtml="http://www.w3.org/1999/xhtml";
txt:find(ds:layer($QUERY_CONTEXT, "Layout")//xhtml:p, ".*zero knowledge.*")
CTRL+D
```

Name

txt:get-text, txt:get-text-left, txt:get-text-right — access textual signals (http://annolab.org/annolab/textual)

Synopsis

```
txt:get-text($elements as element()*) as element()

txt:get-text-left($elements as element()*, $window as xs:string)
    as element()

txt:get-text-right($elements as element()*, $window as xs:string)
    as element()
```

Description

The function `txt:get-text` gets the contents addressed by the sequence of segments passed as the first argument. The functions txt:get-text-left and txt:get-text-right a number of characters left or right of each item in the sequence of segments passed as the first argument. The number of characters is specified as the second argument.

Arguments

`$elements`	The segments addressing the text to be retrieved.
`$window`	The number of characters left or right of the segment.

Examples

In this example the PDF file `SomeText.pdf` is copied into the data store `default`. AnnoLab automatically extracts some layout information from the PDF and stores it as XHTML annotations in the layer `Layout`. Then a query is issued against that data store looking for all page breaks sections (`xhtml:div[@class="pagebreak"]`). For each an XML element `result` is created containing the text extracted from the page break sections by the `txt:get-text` function. Because the serialiser can only serialise XML fragments with a single root element, all is wrapped in the element `results`.

The function `txt:get-text` fetches the text addressed by each segment. Often segments addresses words, but not the whitespace between them. In order to get an actual piece of text, the `seq:grow` function can be used. It calculates a segment from the left-most and the right-most positions addressed by any descendant segment of the sequence of elements passed to it. Thus this function is commonly used in conjunction with `txt:get-text` to retrieve a meaningful and easily readable section of the signal that includes whitespace and line breaks.

```
$ annolab copy SomeText.pdf annolab://default
$ annolab mquery annolab://default

Enter query. Press <CTRL-D> when done to start execution.

declare namespace x="http://www.w3.org/1999/xhtml";
element results {
  for $pb in ds:layer($QUERY_CONTEXT, "Layout")//x:div[@class="pagebreak"]
  return element result { txt:get-text(seq:grow($pb)) }
}
CTRL+D
```

Glossary

Layer	A layer contains information that is overlaid on a signal. Technically speaking, a layer is an XML file where text nodes have been replaced by stand-off anchors addressing parts of a signal.
Resource	A collective term for objects in a data store (collections, signals and layers). When objects on the file system are read, AnnoLab automatically tries to convert them to resources that can be stored in a data store. The conversion is done through importers.
Signal	A primary data object that is annotated through layers.

Bibliography

[Eckart06a] Richard Eckart. *A Framework For Storing, Managing and Querying Multi-Layer Annotated Corpora*. July 2006. Technische Universität Darmstadt Department of Linguistic and Literary Studies English Linguistics

[Eckart06b] Richard Eckart. “Systemic Functional Corpus Resources: Issues in Development and Deployment”. *Proceedings of the COLING/ACL 2006 Main Conference Poster Sessions*. Association for Computational Linguistics. 183-190. Sydney, Australia. July 2006. http://www.aclweb.org/anthology/P/P06/P06-2024 .

[Eckart07] Richard Eckart and Elke Teich. “An XML-based data model for flexible representation and query of linguistically interpreted corpora”. *Data Structures for Linguistic Resources and Applications - Proceedings of the Biennial GLDV Conference 2007*. 327-336. Georg Rehm. Andreas Witt. Lothar Lemnitzer. Gunter Narr Verlag Tübingen. Tübingen, Germany. 2007.

[Rehm07a] Georg Rehm, Richard Eckart, Christian Chiarcos, and Johannes Dellert. “Ontology-Based XQuery'ing of XML-Encoded Language Resources on Multiple Annotation Layers”. European Language Resources Association (ELRA). *Proceedings of the Sixth International Language Resources and Evaluation (LREC'08)*. 510-514. Marrakech, Morocco. May 2008.

[Rehm07b] Georg Rehm, Richard Eckart, and Christian Chiarcos. “An OWL- and XQuery-Based Mechanism for the Retrieval of Linguistic Patterns from XML-Corpora”. *Proceedings of the International Conference Recent Advances in Natural Language Processing - RANLP 2007*. 510-514. Borovets, Bulgaria. 2007.

[Teich05] Elke Teich, Peter Fankhauser, Richard Eckart, Sabine Bartsch, and Mônica Holtz. “Representing SFL-annotated corpus resources”. *Proceedings of the 1st Computational Systemic Functional Workshop*. Sydney, Australia. 2005.

[Teich06] Elke Teich, Richard Eckart, and Mônica Holtz. “Systemic Functional Corpus Resources: Issues in Development and Deployment”. *Proceedings of the Fifth Workshop on Treebanks and Linguistic Theories (TLT 2006)*. 247-258. Jan Hajic. Joakim Nivre. Institute of Formal and Applied Linguistics. Prague, Czech Republic. December 2006. 80-239-8009-2.