LAF Fabric Documentation

Release 4.5.3

Dirk Roorda

June 25, 2015

Contents

1 Welcome **3**
- 1.1 Author . . . 3
- 1.2 Who is using LAF-Fabric? . . . 4

2 Release Notes **5**
- 2.1 4.5.3 . . . 5
- 2.2 4.5.2 . . . 5
- 2.3 4.5.1 . . . 5
- 2.4 4.5 . . . 5
- 2.5 4.4.7 . . . 5
- 2.6 4.4.6 . . . 6
- 2.7 Past . . . 6

3 About **17**
- 3.1 Description . . . 17
- 3.2 Workflow . . . 17
- 3.3 License . . . 18
- 3.4 Designed for Performance . . . 18
- 3.5 LAF feature coverage . . . 18
- 3.6 Future directions . . . 19

4 Getting Started **21**
- 4.1 About . . . 21
- 4.2 Platforms . . . 21
- 4.3 On a VM . . . 21
- 4.4 Your python setup . . . 21
- 4.5 Get LAF-Fabric . . . 22
- 4.6 Install LAF-Fabric . . . 22
- 4.7 Get the data . . . 22
- 4.8 Test and run LAF-Fabric . . . 23
- 4.9 More configuration for LAF-Fabric . . . 23
- 4.10 Writing notebooks . . . 24

5 Background **27**
- 5.1 What is LAF/GrAF . . . 27
- 5.2 Data . . . 27
- 5.3 Existing tools for LAF/GrAF resources . . . 27
- 5.4 LAF-Fabric . . . 28
- 5.5 Interactive notebooks . . . 28
- 5.6 Rationale . . . 28
- 5.7 Implementation highlights . . . 29
- 5.8 Author . . . 30
- 5.9 History . . . 30

6 API Reference **31**
6.1 Parts of the API . . . 31
6.2 Where is the API? . . . 31
6.3 Calling the API . . . 31
6.4 Node order . . . 34
6.5 LAF API . . . 34
6.6 Extra data preparation . . . 43

7 ETCBC Reference **45**
7.1 What is ETCBC . . . 45
7.2 Layers . . . 45
7.3 Node order . . . 46
7.4 Transcription . . . 47
7.5 Trees . . . 47
7.6 Annotating . . . 48
7.7 Extra Data . . . 49
7.8 Feature documentation . . . 49
7.9 MQL . . . 49

8 EMDROS2LAF reference **51**
8.1 Description . . . 51
8.2 Usage . . . 51
8.3 Input . . . 52
8.4 Output . . . 52
8.5 Definitions . . . 52
8.6 Project . . . 52
8.7 See also . . . 52

9 laf 4.5.3 **53**
9.1 Submodules . . . 53
9.2 laf.fabric module . . . 53
9.3 laf.elements module . . . 53
9.4 laf.data module . . . 54
9.5 laf.model module . . . 55
9.6 laf.parse module . . . 55
9.7 laf.names module . . . 55
9.8 laf.settings module . . . 57
9.9 laf.timestamp module . . . 57
9.10 laf.lib module . . . 58

10 etcbc 4.5.3 **59**
10.1 Submodules . . . 59
10.2 etcbc.preprocess module . . . 59
10.3 etcbc.annotating module . . . 59
10.4 etcbc.featuredoc module . . . 60

11 emdros2laf 4.5.3 **61**
11.1 Submodules . . . 61
11.2 emdros2laf.settings module . . . 61
11.3 emdros2laf.etcbc module . . . 61
11.4 emdros2laf.laf module . . . 63
11.5 emdros2laf.transform module . . . 65
11.6 emdros2laf.validate module . . . 65
11.7 emdros2laf.run module . . . 66
11.8 emdros2laf.mylib module . . . 66

12 Indices and tables **67**

Python Module Index **69**

(for provenance of this image, see [1])

Contents:

[1] Image found by an internet search on fabric and some other term that I forgot. By a google search on the image itself, I managed to find the original context at `http://www.hobbycraft.co.uk/hobbycraft-textured-fabric-reel-cream-2-metre/584337-1000`.

Today, 2014-01-16, that link is not accessible anymore. A google search on the image itself yields no hits except the present page and some related pages.

So, LAF-Fabric has given this image a second life!

CHAPTER 1

Welcome

The word **fabric** denotes a texture, and a LAF resource can be seen as a texture of annotations to a primary data source.

In other languages than English, and possibly in English as well, fabric also denotes a place were stuff is made. For etymology, see faber. The location of industry, a factory (but that word derives from the slightly different facere).

What if you want to study the data that is in the fabric of a LAF resource? You need tools. And what if you want to add your own tapestry to the fabric?

You need an interactive environment where tools can be developed and data can be combined.

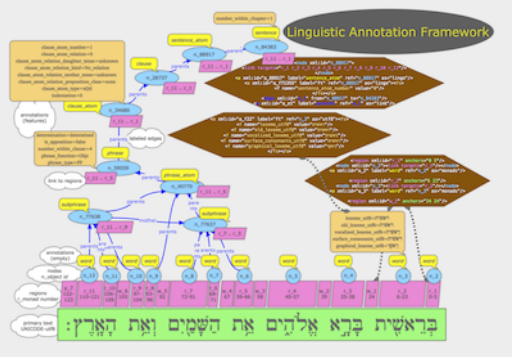
This is the LAF Fabric, and here is a simple example of what you can do with it:

- gender notebook

1.1 Author

LAF-Fabric has been developed by Dirk Roorda, working at DANS and TLA. The need for it arose while executing a CLARIN-NL project SHEBANQ, by which the contents of the Hebrew Text Database of the ETCBC was converted from EMDROS (that conversion is also part of LAF-Fabric). into LAF.

1.2 Who is using LAF-Fabric?



People at the Theology Department of the VU University Amsterdam are producing notebooks for their research into the text of the Hebrew Bible, see

- The ETCBC's github repository (highlight: trees for Data Oriented Parsing)
- Gino Kalkman's analysis of the verbal forms in the Psalms, accompanying his Ph.D. thesis
- other contributions

LAF-Fabric has been used in several ways to construct the SHEBANQ demonstrator query saver.

The development of LAF-Fabric continues, but its progress now takes place mostly inside the module *etcbc*, a specialized module to support working with the Hebrew Text Database of the ETCBC.

CHAPTER 2

Release Notes

2.1 4.5.3

The API element `L` has a new method `L.p` which enables you to drill down quickly to a book, chapter, verse, sentence, clause and phrase of your choice.

Under the hood: the `L` API element was coded in the `laf` package, although it used ETCBC-specific concepts. Now it has been moved to the `etcbc` package entirely.

In order to find the documentation of `L` you should consult the *ETCBC reference*.

Fixes: preparation of data still failed in some cases.

2.2 4.5.2

Fix: preparation of data failed in some cases.

2.3 4.5.1

Fix: prepared data is only loaded when needed, like all other data.

2.4 4.5

New API element `L` (with methods `L.d` and `L.u`) based on new preprocessed data. These methods take you from a node up to container nodes or down to contained nodes. This is a big improvement in the interplay between MQL queries and LAF-Fabric. The better practice is to write a clean MQL query to get the targeted patterns, and use `L` to retrieve information from the context of the hits.

Warning: when your LAF-Fabric needs the data for `L` for the first time, it will compute it and store it as binary data on disk. This computation takes several minutes. In subsequent cases, LAF-Fabric can load the data from disk in a matter of seconds.

2.5 4.4.7

Bug fixes and documentation.

2.6 4.4.6

The *etcbc.px* module has been replaced by *etcbc.extra*. This is a generalized module to transform extra data to annotations. It can be used to process data from *px* files, but also data from *lexicon* files. New lexicon data is underway.

2.6.1 4.4.5

The *etcbc.px* module has been generalized to *etcbc.extra*. It is a module to turn extra data into a valid annotation set.

2.6.2 4.4.4

Minor fixes.

2.6.3 4.4.3

The welcome string now contains a reference to the feature documentation.

2.7 Past

2.7.1 4.4.2

etcbc.featuredoc now produces sphinx output that can be put on a readthedocs website.

2.7.2 4.4.1

Documentation update. Links to the original data as archived in DANS-EASY.

2.7.3 4.4

Adaption to the new ETCBC4 version of the data: in documentation and in the *etcbc* and *emdros2laf* packages. Bugfixes.

2.7.4 4.3.5

Documentation update. The data source BHS4 has been rebaptized to ETCBC4, and the documentation, which was geared towards the BHS3 data source, is now adjusted to ETCBC4.

2.7.5 4.3.4

Fine tuning of the Hebrew transliteration. The new plain text looks exceedingly well now. All changes w.r.t. the previous version of the ETCBC database have been reviewed, which has resulted in new code to generate the fine points of Hebrew text and type, e.g. multiple accents and vowel pointings, and inversed nuns.

2.7.6 4.3.3

The transliteration in *etcbc.lib* which converts between Hebrew characters and transliterated latin characters, has been extended to deal with vowel pointings and accents too.

2.7.7 4.3.1

The module *etcbc.px* retrieves one more field, called *instruction* from the *px* files.

2.7.8 4.3

Changes in the annotation space, a new *etcbc.px* which can read certain types of *px* data and transform it into an extra LAF annotation package.

Incompatible changes

Due to the new names for edge features, the data for BHS3 and BHS4 has been recompiled, and all tasks that use the old names have to be updated.

2.7.9 4.2.15

A few changes in etcbc.emdros2laf: edge annotations are no longer empty annotations, but have a feature structure.

2.7.10 4.2.14

A few changes in etcbc.emdros2laf, which facilitates generating feature declaration documents.

2.7.11 4.2.13

In the API you can ask for the locations of the data directory and the output directory.

2.7.12 4.2.12

LAF-Fabric reports the date and time when it has loaded data for a task. So in every notebook you can see the version of LAF-Fabric, the datetime when the loaded data has been compiled, and the datetime when this data has been loaded for this task. This is handy when you share tasks via nbviewer.

2.7.13 4.2.11

New API element *EE*, which yield all edges in unspecified order. The module *featuredoc* can now document all features, also edge features.

2.7.14 4.2.10

Separated the data directory *laf-fabric-data* into an input directory (*laf-fabric-data*) and an output directory (*laf-fabric-output*). In this way, it is easier to download new versions of the data without overwriting your own task results.

2.7.15 4.2.9

Minor improvements in the emdros2laf conversion, discovered when converting the new BHS4 version of the Hebrew Text database. If you want to use the BHS4 data (beta), download the data again.

2.7.16 4.2.8

Minor improvements in the laf-api.

2.7.17 4.2.7

API

Added *NK*, which can be passed as a sort key for node sets. It corresponds with the "natural order" on nodes. If an additional module, such as *etcbc.preprocess* has modified the natural order, this sort key will reflect the modified order. If you let NN() yield nodes, they appear in this same order.

Also added *MK*, which can be passed as a sort key for sets of anchors. It corresponds with the "natural order" on anchor sets.

ETCBC

Improvements in *etcbc.trees*, the module that generates trees from the ETCBC database.

2.7.18 4.2.6

Developed the *etcbc.trees* module further. Trees based on the implicit embedding relationship do not exhibit all embedding structure: clauses can be further embedded by means of an explicit *mother* relationship. The rules are a bit intricate, but it has been implemented (BHS3 only, no CALAP). See the updates trees notebook.

2.7.19 4.2.5

Added tree defining functionality to the etcbc package: *etcbc.trees*. You can make the implicit embedding relationship between objects explicit by means of parent and children relationships.

Adapted the node order as customized by *etcbc.preprocess*: the order is now a total ordering. Main idea: try to order monad sets by the subset relation, where embedder comes before embedded. If the sets are equal, use the object type to force a decision. If two monad sets cannot be ordered by the subset relation, look at the elements that they do *not* share. The monad set that contains the smallest of these elements, is considered to come before the other.

2.7.20 4.2.4

Added Syriac transcription conversions.

2.7.21 4.2.3

In *emdros2laf* every source can now have its own metadata. In *etcbc* there is a workable definition between consonantal Hebrew characters and their ETCBC latin transcriptions.

2.7.22 4.2.2

More fixes in *emdros2laf*, a new source, the *CALAP* has been converted to LAF. LAF-Fabric has compiled it, and it is ready for exploration. See the example notebook plain-calap. The CALAP is included in the data download (see *Getting Started*).

2.7.23 4.2.1

Small fixes in *emdros2laf*.

2.7.24 4.2

LAF Usability

The conversion program from EMDROS to LAF (now the package *emdros2laf*) has been integrated in LAF-Fabric. Because of this a small reorganization of subdirectories was necessary (again). The EMDROS source of the LAF has a place in *laf-fabric-data* as well. So: again: a new download of the data is required.

2.7.25 4.1.4

LAF Usability

Small reorganization of subdirectories. The structure is now better adapted to work with completely different data sources. Update your configuration files. The trailing directory names must be removed. So:

```
work_dir = ~/laf-fabric-data/etcbc-bhs
```

should change into:

```
work_dir = ~/laf-fabric-data
```

Same for `laf-dir`.

Because of this reorganization you have to download the data again.

2.7.26 4.1.3

Small fixes.

2.7.27 4.1.2

LAF Usability

Small usability improvements in `etcbc` and in `laf`.

2.7.28 4.1.1

LAF Usability

After loading LAF-Fabric display the compilation data and time of the data used.

2.7.29 4.1

ETCBC Emdros integration

In the *etcbc* package there is a module *mql* that enables the user to run emdros queries, capture the results as a node set, and use that for further processing in LAF-Fabric. See notebook MQL

2.7.30 4.0.6

API

In specifying what features to load, you may omit namespaces and labels. You can specify the features to load in a much less verbose way.

The functions `load()` and `load_again()` have a new optional parameter `add`, which instructs laf fabric to do an incremental loading, without discarding anything that has already been loaded.

ETCBC

The order defined by `etcbc.preprocess` has been refined, so that it can also deal with empty words.

Under the hood

More unit tests, especially w.r.t. node order and empty words. The example data on which the unit tests act, has been enlarged: it now contains also Isaiah 41:19 in which two empty words occur.

2.7.31 4.0.5

Usability

Better error handling, especially when the load dictionary does not conform to the specs of the API reference.

Under the hood

More unit tests, especially w.r.t. error checking, and node order, and the `BF` API element.

2.7.32 4.0.4

API

The special edge features for all annotated edges and unannotated edges are now called `laf:.y` and `laf:.x`, because otherwise their names become private method names in Python.

2.7.33 Under the hood

More unit tests.

2.7.34 Incompatible changes

Because of the renaming of special edge features, a new copy of the data is needed. Download the latest version.

2.7.35 4.0.3

API

The methods of the connectivity objects (except `e()` yield all iterators and have an optional parameter `sort=False`. The API elements now can be added very easily to your local namespace by saying: `exec(Fabric.localnames.format(var='Fabric'))`.

2.7.36 4.0.2

API

For connectivity there is a new API method: `C.feature.e(n)`. This returns `True` if and only if `n` is connected to a node by means of an edge annotated with `feature`. This function can also be obtained by using `C.feature.v(n)`, but the direct `e(n)` is much more efficient.

Usability

When calling up features as in `F_shebanq_ft_part_of_speech`, you may now leave out the namespace and also the label. So `F.part_of_speech` also works.

2.7.37 4.0.1

Small bug fixes.

2.7.38 4.0

API

The API has changed for initializing the processor and for working with connectivity (`C` and `Ci`). Please consult *API Reference*.

Usability

- There is an example dataset included: Genesis 1:1 according to the ETCBC database.
- Configuration is easier: a global config file in your home directory.
- There is a *laf-fabric-test.py* script for a basic test.

Incompatible changes

More data has been precompiled. This reduces the load time when working with LAF-Fabric. The data organization has changed. Please download a new version of the data.

Configuration is easier now. A single config file in your home directory is sufficient. There are also other ways, including a config file next to your notebook.

Changes under the hood

- The mechanism to store and load LAF data now has a hook by which auxiliary modules can register new data with LAF Fabric. Currently, this mechanism is used by the `etcbc` module to inject a better ordering of the nodes than LAF Fabric can generate on its own. In future versions we will use this mechanism to load compute and load extra indices needed for working with the EMDROS database.
- Unit tests. In the file *lf-unittest.py* there are now several unit tests. If they pass most things in LAF-Fabric are working as expected. However, the set needs to be enlarged before new changes are undertaken.

2.7.39 3.7

API

- You can make additional sorting persistent now, so that it becomes part of the compiled data. See the `prep` function in the API reference.

Usability

- It is possible to set a verbosity level for messages.
- There were chunks of time consuming data that were either completely or often unnecessary. This data has been removed, or is loadable on demand respectively. Overall performance during load time is a bit better now.

Extra's

The *etcbc* module has a method to compute a better ordering on the nodes. This module works together with the new API method to store computed results.

2.7.40 3.6

API

There is a significant addition for dealing with the order of nodes:

- New function `BF(nodea, nodeb)` for node comparison. Handy to find the nodes that cannot be ordered because they have the same start points and end points in the primary data.
- New argument to `NN()` for additionally sorting those enumerated nodes that have the same start points and end points in the primary data.

Incompatible changes

- The representation of node anchors has changed. **Existing LAF resources should be recompiled**.

Usability

When LAF-Fabric starts it shows a banner indicating its version.

2.7.41 3.5.1

Bugfixes

Opening and closing of files was done without specifying explicitly the `utf-8` encoding. Python then takes the result of `locale.getprefferredencoding()` which may not be `utf-8` on some systems, notably Windows ones.

Remedy: every `open()` call for a text file is now passed the `encoding='utf-8'` parameter. `open()` calls for binary files do not get an encoding parameter of course.

2.7.42 3.5

Usability

Code supporting ETCBC notebooks has moved into separate package *etcbc*, included in the laf distribution.

2.7.43 3.4.1

Usability

When loading data in a notebook, the progress messages are far less verbose.

API

Added an introspection facility: you can ask the *F* object which features are loadable.

2.7.44 3.4

API

Changes in the way you refer to input and output files. You had to call them as methods on the `processor` object, now they are given with the `API()` call, like the `msg()` method.

Bugfixes

Under some conditions XML identifiers got mistakenly unloaded. Fixed by modifying the big table with conditions in `check_load_status` in `laf.laf`.

2.7.45 3.3.7

Usability

Configuration fix: the LAF source directory can be anywhere on the system, specified by an *optional* config setting. If this setting is not specified, LAF-Fabric works with a binary source only.

A download link to the data is provided, it is a dropbox link to a zipped file with a password. You can ask me for a password.

Focus on working with notebooks. Command line usage only supported for testing and debugging, not on Windows.

Documentation

Thoroughly reorganized and adapted to latest changes.

Notebooks

This distribution only contains example tasks and notebooks. The real stuff can be found in the ETCBC repository and in a study repo maintained by Judith Gottschalk.

2.7.46 3.3.6

Usability

The configuration file, *laf-fabric.cfg* will no longer be distributed. Instead, a file *laf-fabric-sample.cfg* will be distributed. You have to copy it to *laf-fabric.cfg* which you can adapt to your local situation. Subsequent updates will not affect your local settings.

2.7.47 3.3.5

API

New methods to find top most and bottom most nodes when traveling from a node set along annotated edges. See *C, Ci (Connectivity)*.

2.7.48 3.3.4

Notebook additions only.

The notebook clause_constituent_relation is an example how you can investigate a LAF data source and document your findings.

We intend to create a separate github dedicated to notebooks that specifically analyse the Hebrew Text Database.

2.7.49 3.3.3

Other

Bugfixes: Data loading, unloading, keeping data better adapted to circumstances.

2.7.50 3.3.2

API

- **New API element `Ci` for connectivity.** There is a new object `Ci` analogous to `C` by which you can traverse from nodes via annotated edges to other nodes. The difference is that `Ci` uses the edges in the opposite direction. See *C, Ci (Connectivity)*.

Incompatible changes

Bugfix. The order of node events turned out wrong in the case of nodes that are linked to point regions, i.e. regions with zero width (e.g. `(n, n)`, being the point between characters `n-1` and `n`). This caused weird behaviour in the tree generating notebook trees (rough path).

Yet it is impossible to guarantee natural behaviour in all cases. If there are nodes linked to empty regions in your LAF resource, you should sort the node events per anchor yourself, in your custom task. **Existing LAF resources should be recompiled**.

Other

The trees (smooth path) notebook is evolving to get nice syntax trees from the Hebrew database.

2.7.51 3.3.1

Bugfix. Thanks to Grietje Commelin for spotting the bug so quickly. My apologies for any tension it might have created in the meantime. Better code under the hood: the identifiers for nodes, edges and regions now start at 0 instead of 1. This reduces the need for many `+ 1` and `- 1` operations, including the need to figure out which one is appropriate.

3.3

2.7.52 API

- Node events are added to the API, see *NE (Next Event)*. With `NE()` you traverse the anchor positions in the primary data, and at each anchor position there is a list of which nodes start, end, resume or suspend there. This helps greatly if your task needs the embedding structure of nodes. There are facilities to suppress certain sets of node events.

Incompatible changes

- Node events make use of new data structures that are created when the LAF resource is being compiled. **Existing LAF resources should be recompiled**.

2.7.53 3.2.1

API

- **API elements are now returned as named entries in a dictionary, instead of a list.** In this way, the task code that calls the API and gives names to the elements remains more stable when elements are added to the API.
- Documentation: added release notes.
- New Example Notebook: participle.

Incompatible changes

- **`API()` in `laf.task` now returns a keyed dictionary instead of a 6-tuple.** The statement where you define API is now

  API = processor.API() F = API['F'] NN = API['NN'] ...

  (was:

  ```
  (msg, NN, F, C, X, P) = processor.API()
  ```

)

2.7.54 3.2.0

API

- **Connectivity added to the API, see *C, Ci (Connectivity)*.** There is an object C by which you can traverse from nodes via annotated edges to other nodes.
- **Documentation organization:** separate section for API reference.

Incompatible changes

- **`API()` in `laf.task` now returns a 6-tuple instead of a 5-tuple:** C has been added.
- **nodes or edges annotated by an empty annotation will get a feature based on the annotation label.** This feature yields value `''` (empty string) for all nodes or edges for which it is defined. Was `1`. **Existing LAF resources should be recompiled**.

CHAPTER 3

About

3.1 Description

LAF-fabric is a Python tool for running Python notebooks with access to the information in a LAF resource. It has these major components:

- **the *laf* package**
  - a LAF compiler for transforming a LAF resource into binary data that can be loaded nearly instantly into Python data structures;
  - an execution environment that gives Python notebooks access to LAF data and is optimized for feature lookup.
- **the *etcbc* package**
  - an extension toolkit geared to a specific LAF resource: the ETCBC Hebrew Text Database.
- **the *emdros2laf* package**
  - conversion from EMDROS to LAF. The ETCBC Hebrew is originally available as an EMDROS database. This package performs the conversion to LAF.

The selling point of LAF-fabric is performance, both in terms of speed and memory usage. The second goal is to make it really easy for you to write analytic notebooks straightforwardly in terms of LAF concepts without bothering about performance.

Both points go hand in hand, because if LAF-fabric needs too much time to execute your notebooks, it becomes very tedious to experiment with them. I wrote LAF-fabric to make the cycle of trial and error with your notebooks as smooth as possible.

3.2 Workflow

The typical workflow is:

1. download a LAF resource [1] to your computer (or work with a compiled version [2]).
2. install LAF-fabric on your computer.
3. adapt a config file to change the location of the work directory.
4. write your own iPython notebook or script

[1] A LAF resource is a directory with a primary data file, annotation files and header files. This program has been tested with *LAF version of the Hebrew Bible*.

[2] It is perfectly possible to run the workflow without the original LAF resource. If somebody has compiled a LAF resource for you, you only need to obtain you the compiled data, and let the LAF source in the configuration point to something non-existent. In that case LAF-fabric will not complain, and never attempt to recompile the original resource. You can still add extra annotation packages, which can be compiled against the original LAF source, since the original LAF XML identifiers are part of the compiled data. In case of the Hebrew Bible LAF resource: the original resource is 1.64 GB on disk, while the compiled binary data is 268 MB.

5. run the code cells in an iPython notebook or your script

You can run your cells, modify them, run them again, ad libitum. While the notebook is alive, loading and unloading of data will be done only when it is really needed.

So if you have to debug a notebook, you can do so without repeatedly waiting for the loading of the data.

The first time a source or annox [3] is used, the LAF resource will be compiled. Compiling of the full Hebrew Bible source may take considerable time, say 10 minutes for a 2 GB XML annotations on a Macbook Air (2012). The compiled source will be saved to disk across runs of LAF-fabric. Loading the compiled data takes, in the same setting with the Hebrew Bible, less than a second, but then the feature data is not yet loaded, only the regions, nodes and edges. If you need the original XML identifiers for your notebook, there will be 2 to 5 seconds of extra load time.

You must declare the LAF-features that you use in your notebook, and LAF-fabric will load data for them. Loading a feature typically adds 0.1 to 1 second to the load time. Edge features may take some seconds, because of the connectivity data that will be built on the basis of edge information.

LAF-Fabric will also unload the left-over data from previous runs for features that the current run has not declared. In this way we can give each run the maximal amount of RAM.

3.3 License

This work is freely available, without any restrictions. It is free for commerical use and non-commercial use. The only limitation is that parties that include this work may not in anyway restrict the freedom of others to use it.

3.4 Designed for Performance

Since there is a generic LAF tool for smaller resources, (POIO, Graf-python) this tool has been designed with performance in mind. In fact, performance has been the most important design criterion of all. There isa price for that: we use a simplified feature concept. See the section of LAF *LAF feature coverage* below.

3.5 LAF feature coverage

This tool cannot deal with LAF resources in their full generality.

In LAF, annotations have labels, and annotations are organized in annotation spaces. In a previous version, LAF-fabric ignored annotation spaces altogether. Now annotation spaces are fully functional.

primary data LAF-fabric deals with primary data in the form of text. It is not designed for other media such as audio and video. Further, it is assumed that the text is represented in UNICODE, in an encoding supported by python, such as utf-8. LAF-fabric assumes that the basic unit is the UNICODE character. It does not deal with alternative units such as bytes or words.

feature structures The content of an annotation can be a feature structure. A feature structure is a set of features and sub features, ordered again as a graph. LAF-fabric can deal with feature structures that are merely sets of key-value pairs. The graph-like model of features and subfeatures is not supported.

annotations Even annotations get lost. LAF-fabric is primarily interested in features and values. It forgets the annotations in which they have been packaged except for:

- the annotation space,
- the annotation label,
- the target kind of the annotation (node or edge)

[3] Shorthand for *extra annotation package*. You can add an extra package of annotations in LAF format to your data. When needed, this annox will be compiled into binary data and combined with the compiled data of the main LAF resource. So you can integrate your own annotation work with the annotations that have been done before. **You cannot add new regions, nodes, edges in this way**.

dependencies In LAF one can specify the dependencies of the files containing regions, nodes, edges and/or annotations. LAF-fabric assumes that all dependent files are present in the resource. Hence LAF-fabric reads all files mentioned in the GrAF header, in the order stated in the GrAF header file. This should be an order in which regions appear before the nodes that link to them, nodes before the edges that connect them, and nodes and edges before the annotations that target them.

3.6 Future directions

LAF-Fabric has proven to function well for in increasing number of tasks. This proves that the methodology works and we are trying more challenging things. The direction of the future work should be determined by your research needs.

3.6.1 Adding new annotations

LAF-Fabric supports adding an extra annotation package to the existing LAF resource, and contains an example workflow to create such packages. We have used it to add an extra annotation package to the ETCBC Hebrew Text Database containing data that has not made it yet to the published set of features, but it relevant to researchers. See the notebook extra px data

3.6.2 Visualization

You can invoke additional packages for data analysis and visualization right after your task has been completed in the notebook.

The division of labour is that LAF-Fabric helps you to extract the relevant data from the resource, and outside LAF-Fabric, but still inside your notebook, you continue to play with that data.

When we get more experience with visualization, we might need new ways of data extraction, which would drive a new wave of changes in LAF-Fabric.

3.6.3 Graph methodology and full feature structures

LAF-Fabric has not been implemented as a graph database. We might adopt more techniques from graph databases to make it more compatible with current graph technology. We could use the python networkx module for that. That would also help to implement feature structures in full generality.

3.6.4 API completion

The API offers functionality that covers the following aspects of a LAF resource:

node iterator iterator that produces nodes in the order by which they are anchored to the primary data (which are linearly ordered).

feature lookup a class that gives easy access to feature data and has methods for feature value lookup and mapping of feature values.

connectivity adjacency information for nodes, by which you can travel via (annotated) edges to neighbouring nodes; there are also methods to generate sets of end-points when traveling from a nodeset along featured edges until there are no outgoing edges. You can also travel in the opposite direction.

xml identifier mapping a two-way mapping from orginal identifiers in the LAF XML resource to integers that denote the corresponding nodes in LAF-Fabric.

primary data access The primary data can be accessed through nodes that are linked to regions of primary data.

hooks for custom pre-computed data Third party modules geared to a particular LAF resource may perform additional computations and store the result alongside the complied data.

CHAPTER 4

Getting Started

4.1 About

LAF-Fabric is a github project in which there are Python packages called *laf* and *etcbc* and *emdros2laf*. You must install them as packages in your current python installation. This can be done in the standard pythonic way, and the precise instructions will be spelled out below.

4.2 Platforms

LAF-Fabric is being developed on **Mac OSX** Mavericks on a Macbook Air with 8 GB RAM. It is being used on a **Linux** virtual machine running on a laptop of respectable age, and it runs straight under **Windows** as well, except for some testing/debugging functionality.

4.3 On a VM

The most hassle-free way to get started with LAF-Fabric is on a VM on your computer. You can get that nearly automatically from a Vagrant definition in the llshebanq project. Then you can skip the rest until **Writing Notebooks** below.

4.4 Your python setup

First of all, make sure that you have the right Python installation. You need a python3 installation with numerous scientific packages. Below is the easiest way to get up and running with python. You can also use it if you have already a python, but in the wrong versions and without some necessary modules. The following setup ensures that it will not interfere with existing python installations and it will get you all modules in one go.

4.4.1 Getting to know interactive python

The following step may take a while, so in the meantime you can familiarize yourself with ipython, if you like. The website is a good entry point.

4.4.2 Download Anaconda

Anaconda is our distribution of choice. Choose a python3 based installer from this download page.

1. Pick the one that fits your operating system. Install it. If asked to install for single user or all users, choose single user.

2. On Windows you could get into trouble if you have another Python. If you have environment variables with the name of PYTHONPATH or PYTHONHOME, you should disable them. For diagnosis and remedy, see [1]

This will install all anaconda packages in your fresh python3 installation. Now you have *ipython*, *networkx*, *matplotlib*, *numpy* to name but a few popular python packages for scientific computing.

4.5 Get LAF-Fabric

If you have git you can just clone it from github on the command line:

```
cd «directory of your choice»
git clone https://github.com/ETCBC/laf-fabric
```

If you do not have git, consider getting it from github. It makes updating your LAF-Fabric easier later on.

Nevertheless, you can also download the latest version from github/laf-fabric. Unpack this somewhere on your file system. Change the name from *laf-fabric-master* to *laf-fabric*. In a command prompt, navigate to this directory.

4.6 Install LAF-Fabric

Preparation: you have to unpack a `tar.gz` file. On Windows you may have to install a tool for that, such as 7-zip.

Here are the steps, assuming you are in the command line, at the top level directory in *laf-fabric*:

```
cd dist
tar xvf laf-*
cd laf-*
python setup.py install
```

This installs the generic laf processor *laf* and the more specific ETCBC tools to work with the Hebrew Text Database: *etcbc*. It also installs *emdros2laf*, a conversion package from the source format of the ETCBC database (EMDROS) to LAF. This package has been used to obtain the actual LAF version of the ETCBC database.

Note: In order to use *emdros2laf* and parts of *etcbc*, you need to install EMDROS software, which is freely available. Tip: it works nicely with an sqlite3 backend.

4.7 Get the data

If you are interested in working with the Hebrew Bible, go to the DANS-EASY archive and download *laf-fabric-data.zip* and unpack it in your home directory. If all goes well you have a directory *laf-fabric-data* in your home directory.

Note: If you have already a *laf-fabric-data* directory, delete it, unless you have added stuff yourself (possibly extra annotations). In that case, move your existing directory out of the way. *laf-fabric-data* is supposed to be input data, i.e. the data you download plus the data that laf-fabric itself adds to it while converting from emdros to laf or from laf to binary.

---

[1] To check whether you have environment variables called PYTHONPATH or PYTHONHOME, go to a command prompt and say

```
echo %PYTHONPATH%
echo %PYTHONHOME%
```

If the system responds with the exact text you typed, there is nothing to worry about. Otherwise, you should rename these variables to something like `NO_PYTHONPATH` or `NO_PYTHONHOME`.

You can do that through: Configuration (Classical View) => System => Advanced Settings => button Environment Variables.

If you have a reference to an other python in your `PATH` (check by `echo %PATH%`) then you should remove it.

After these operations, quit all your command prompts, start a new one, and say `python --version`. You should see something with 3.3 and Anaconda in the answer.

4.8 Test and run LAF-Fabric

In the top-level directory of LAF-Fabric there is a gallery script. If you run it, you will also configure your LAF-Fabric:

```
python lf-gallery.py tinys
```

This points laf-fabric to the example data that comes with the distribution, which is just Genesis 1:1. If you have downloaded the binary data for the full Hebrew Text Database, then make sure the data is in *~/laf-fabric-data/etcbc4* and run:

```
python lf-gallery.py fulls
```

After this you have a default config file *~/laf-fabric-data/laf-fabric.cfg* and you can use laf-fabric scripts from anywhere on your system, also in notebooks.

On all platforms (Windows users: use Firefox or Chrome as your browser, not Internet Explorer), you can also run notebooks with LAF-Fabric:

```
cd examples
ipython notebook
```

This starts a python process that communicates with a browser tab, which will pop up in front of you. This is your dashboard of notebooks. You can pick an existing notebook to work with, or create a new one. It is recommended that you write your own notebooks in a separate directory, not under the LAF-Fabric installation. In that way you can apply updates easily without overwriting your work.

1. Create a notebook directory somewhere in your system and navigate there in a command prompt.
2. Then:

   ```
   ipython notebook
   ```

Note: If you create a notebook that you are proud of, it would be nice to include it in the example notebooks of LAF-Fabric or in the ETCBC notebooks. If you want to share your notebook this way, mail it to me.

4.9 More configuration for LAF-Fabric

If you need the data to be at another location, you must modify the *laf-fabric.cfg*. This configuration file *laf-fabric.cfg* is searched for in the directory of your script, or in a standard directory, which is *laf-fabric-data* in your home directory.

There are just a few settings:

```
[locations]
data_dir  = ~/laf-data-dir
laf_dir  = ~/laf-data-dir
output_dir  = ~/output-data-dir
```

data_dir is folder where all the input data is.

output_dir is folder where all the output data is, the stuff that your tasks create.

laf_dir is the folder where the original laf-xml data is. It is *optional*. LAF-Fabric can work without it.

Alternatively, you can override the config files by specifying the locations in your scripts. Those scripts are not very portable, of course.

4.10 Writing notebooks

4.10.1 Tutorial

Here is a quick tutorial/example how to write LAF analytic tasks in an IPython notebook.

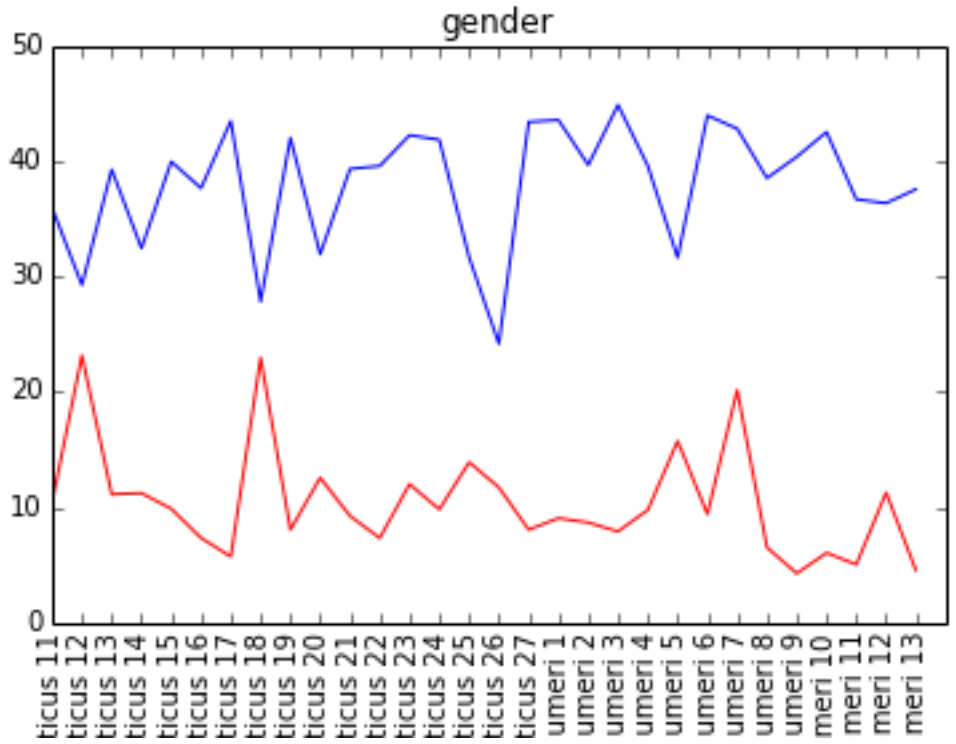
Our target LAF resource is the Hebrew text data base (see *Data*). Some nodes are annotated as words, and some nodes as chapters. Words in Hebrew are either masculine, or feminine, or unknown. The names of chapters and the genders of words are coded as features inside annotations to the nodes that represent words and chapters.

We want to plot the percentage of masculine and feminine words per chapter.

With the example notebook gender we can count all words in the Hebrew bible and produce a table, where each row consists of the bible book plus chapter, followed by the percentage masculine words, followed by the percentage of feminine words in that chapter:

```
Genesis 1,42.34769687964339,5.794947994056463
Genesis 2,38.663967611336034,7.6923076923076925
Genesis 3,37.4749498997996,10.02004008016032
Genesis 4,43.04635761589404,11.920529801324504
Genesis 5,40.74844074844075,18.91891891891892
Genesis 6,36.61327231121282,9.610983981693364
Genesis 7,33.59683794466403,11.462450592885375
Genesis 8,31.30081300813008,9.959349593495935
Genesis 9,37.97216699801193,9.74155069582505
Genesis 10,30.679156908665107,4.68384074941452
```

From this table we can easily make a chart, within the same notebook!



Note: If you click on the notebook link above, you are taken to the public notebook viewer website, which shows static versions of notebooks without storing them. In order to run them, you need to download them to your computer.

The gender notebook is self documenting, it contains general information on how to do data analysis with LAF-Fabric.

4.10.2 Next steps

Study the many ETCBC4 features.

Then have a look at the notebooks in the laf-fabric-nbs and study and contributions repositories. You find notebooks by which you can study the rich feature set in the ETCBC data and notebooks that help you to add your own annotations to the data. These notebooks require the additional *etcbc* package, which comes with LAF-Fabric.

CHAPTER 5

Background

5.1 What is LAF/GrAF

LAF/GrAF is a framework for representing linguistic source material plus associated annotations. LAF, Linguistic Annotation Framework is an ISO standard (24612:2012) that describes the organization of the data. GrAF, Graph Annotation Framework, is a set of schemas for the XML-annotations in a LAF resource.

Despite the L of linguistics, there is nothing particularly linguistic to LAF. LAF describes data that comes as a linearly ordered *primary data* stream (audio, video, text, or anything that has a one dimensional order), in which *regions* can be defined. *Annotations* are key=value pairs or *feature structures* in general, which conform to the joint definition of TEI Feature Structures and ISO 24610). Between the primary data and the annotations is a *graph* of *nodes* and *edges*. Some nodes are linked to regions of primary data. Some nodes are linked to other nodes by means of edges. An annotation may refer to a node or to an edge, but not both.

So, features target the primary data through annotations. Annotations can be labeled and they can be organized in *annotation spaces*.

5.2 Data

Although this tool is written to deal with LAF resources in general, it has been developed with a particular LAF resource in mind: the ETCBC4 Text database of the Hebrew Bible, now a dataset archived at DANS-EASY. Here is a quick link to the ETCBC4 features

The SHEBANQ project has converted this database from a special text database format into LAF (the conversion code is in the package *emdros2laf*, which is included in LAF-Fabric), and the resulting LAF resource is a file set of 1.64 GB, being predominantly linguistic annotations. It is this LAF resource that is the reference context for LAF-Fabric.

A compiled version of this LAF resource, suitable for working with LAF-Fabric, is included in the dataset. Also the original data has been included, so you can also run EMDROS tools on the data in conjunction with LAF-Fabric. LAF-Fabric even contains a notebook that integrates the use of EMDROS MQL with the proper LAF processing.

5.3 Existing tools for LAF/GrAF resources

There is an interesting Python module (POIO, Graf-python) that can read generic GrAF resources. It exposes an API to work with the graph and annotations of such resources. However, when feeding it a resource with 430 k words and 2 GB of annotation material, the performance is such that the graph does not fit into memory of a laptop. Clearly, the tool has been developed for bunches of smaller GrAF documents, and not for single documents with a half million words words and gigabytes of annotation material.

5.4 LAF-Fabric

This program seeks to remedy that situation. Its aim is to provide a framework on top of which you can write Python notebooks that perform analytic tasks on big GrAF resources. It achieves this goal by compiling xml into compact binary data, both on disk and in RAM and by selective loading of features. The binary data loads very fast. Only selected features will be loaded, and after loading they will be blown up into data structures that facilitate fast lookup of values.

With LAF-Fabric you can add an additional annotation package to the basic resource. You can also switch easily between additional packages without any need for recompiling the basic resource. The annotations in the extra package may define new annotation spaces, but they can also declare themselves in the spaces that exist in the basic source. Features in the extra annotation package that coincide with existent features, override the existing ones, in the sense that for targets where they define a different value, the one of the added annotation package is taken. Where the additional package does not provide values, the original values are used.

With this device it becomes possible for you to include a set of corrections to the original features. Or alternatively, you can include the results of your own work, whether manual or algorithmic or both, with the original data. You can then do *what-if* research on the combination.

The notebook annox_workflow demonstrates this workflow.

5.5 Interactive notebooks

LAF-Fabric is designed to work within iPython notebooks. That is a great environment to run tasks interactively, exploring the data as you go, and visualizing your intermediate results at the moment they become available. Last but not least, you can add documentation to notebooks and share them with your colleagues. As an example, look at the gender notebook notebook by which you can draw a graph of the percentage of masculine and feminine words in each chapter of the Hebrew Bible. More involved notebooks can be found at the laf-fabric-nbs repository and the study repo.

5.6 Rationale

The paradigms for biblical research are becoming *data-driven*. If you work in that field, you need increasingly sophisticated ways to get qualitative and quantitative data out of your texts. You, as a researcher, are in the best position to define what you need. You can even fulfill those needs if you or someone else in your group has basic programming experience.

LAF-Fabric is a stepping stone for teams in digital humanities to the wonderful world of computing. With it you extract data from your resources of interest and feed it into your other tools.

See for example the notebook cooccurrences, which codes in less than a page an extraction of **data tables** relevant to the study of linguistic variation in the Hebrew Bible. These tables are suitable for subsequent data analysis by means of the open source statistics toolkit R.

An other example is the notebook proper, which outputs a **visualization** of the text of the Hebrew Bible in which the syntactic structure of the text is visible plus the the genders of all the proper nouns. With this visualization it becomes possible to discern genealogies from other genres with the unaided eye, even without being able to read a letter of Hebrew.

Digging deeper into syntax, the notebook trees_etcbc4 produces syntax trees for all sentences in the Hebrew Bible.

The code of LAF-Fabric is on github, including example notebooks and extra annotation packages. You are invited to develop your own notebooks and share them, either through data archives or directly through github, or the notebook viewer. In doing so, you (together) will create a truly state-of-the-art research tool, adapted to your scholarly needs of analysis, review and publication.

5.7 Implementation highlights

There are several ideas involved in compiling a LAF resource into something that is compact, fast loadable, and amenable to efficient computing.

1. Replace nodes and edges and regions by integers.
2. Store relationships between integers in *arrays*, that is, Python arrays.
3. Store relationships between integers and sets of integers also in *arrays*.
4. Keep individual features separate.
5. Compress data when writing it to disk.

Use of integers In LAF the pieces of data are heavily connected, and the connections are expressed by means of XML identifiers. In the compiled version we get rid of all XML identifiers. Instead, we will represent everything that comes in great quantities by integers: regions, nodes, edges. But feature names and values, annotation labels and spaces will be kept as is.

Relationships between integers as Python arrays In Python, an array is a C-like structure of memory slots of fixed size. You do not have arrays of arrays, nor arrays with mixed types. This makes array handling very efficient, especially loading data from disk and saving it to disk. Moreover, the amount of space in memory needed is like in C, without the overhead a scripting language usually adds to its data types.

There is an other advantage: a mapping normally consists of two columns of numbers, and numbers in the left column map to numbers in the right column. In the case of arrays of integers, we can leave out the left column: it is the array index, and does not have to be stored.

Relationships between integers as Python arrays If we want to map numbers to sets of numbers, we need to be more tricky, because we cannot store sets of numbers directly in the slots of an array. What we do instead is: we build two arrays, the first array points to data records in the second array. A data record in the second array consists of a number giving the length of the record, followed by that number of integers. The function `arrayify()` in `laf.lib` takes a list of items and turns it in a double array.

Keep individual features separate A feature is a mapping from either nodes or edges to string values. Features are organized by the annotations they occur in, since these annotations have a *label* and occur in an *annotation space*. We let features inherit the label and the space of their annotations. Within space and label, features are distinguished by name. And the part of a feature that addresses edges is kept separate from the part that addresses nodes.

So an individual feature is identified by *annotation space*, *annotation label*, *feature name*, and *kind* (node or edge). For example, in the Hebrew Bible data, we have the feature:

```
etcbc4:ft.sp (node)
```

with annotation space `etcbc4`, annotation label `ft`, feature name `sp` (part of speech), and kind `node`. The data of this feature is a mapping that assigns a string value to each of the 426,555 word nodes. So this individual feature represents a significant chunk of data.

The individual features together take up the bulk of the compiled data. Here is a break down of the compiled data:

```
features                          150 MB
graph (nodes, edges, regions)      17 MB
primary data linking               33 MB
LAF XML identifiers mappings       59 MB
precomputed data for node order     8 MB
extra annotation package            1 MB
--------------------------------+------
total                           269 MB
```

Most notebooks require only a limited set of individual features. So when we run tasks and switch between them, we swap feature data in and out. The design of LAF-fabric is such that feature data is neatly chunked per individual feature.

Note: Here is the reason that we do not have an overall table for feature values, identified by integers. We miss some compression here, but with a global feature value mapping, we would burden every task with a significant amount of memory. Moreover, the functionality of extra annotation packages is easier to implement when individual features are cleanly separable.

Note: Features coming from the source and features coming from the extra annotation package will be combined before the you can touch them in tasks. This merging occurs late in the process, even after the loading of features by LAF-fabric. Only at the point in time when a task declares the names of the API methods (see `API()` in `laf.fabric`) the feature data coming from main source and annox will be assembled into objects. Yet the underlying tables will not mixed, so that features do not have to be unloaded and reloaded when you change your annox. The price is a small overhead for each feature lookup: it will be looked up first in the annox data, and only if it is not found there, in the main data.

5.8 Author

This work has been undertaken first in November 2013 by Dirk Roorda, working for Data Archiving and Networked Services (DANS) and The Language Archive (TLA). The work has been triggered by the execution of the SHEBANQ project together with the researchers Wido van Peursen, Oliver Glanz and Janet Dyk at the Eep Talstra Centre for Bible and Computing (ETCBC), VU University.

Thanks to Martijn Naaijer and Gino Kalkman for first and on-going experiments with LAF-Fabric.

5.9 History

2014-07-31 Publication of the ETCBC4 dataset in DANS-EASY.

2014-02-16 A new github repository, study, has been created by our associate programmer Judith Gottschalk. This repository will host the actual notebooks written for and by the ETCBC people. The LAF-Fabric repository will only host example/tutorial notebooks.

2014-01-17 Joint presentation with Martijn Naaijer at CLIN (Computational Linguistics In the Netherlands).

2013-12-18 Demonstration on the StandOff Markup and GrAF workshop (CLARIN-D) in Köln.

2013-12-12 Demonstration for the ETCBC team Amsterdam. Updated the slides.

2013-12-09 Abstract sent to CLIN (Computational Linguistics In the Netherlands) accepted. To be delivered 2014-01-17.

2013-11-26 Vitamin Talk to the TLA team Nijmegen. Here are the slides.

CHAPTER 6

API Reference

6.1 Parts of the API

The API deals with several aspects of task processing. First of all, getting information out of the LAF resource. But there are also methods for writing to and reading from task-related files and for progress messages.

Finally, there is information about aspects of the organization of the LAF information, e.g. the sort order of nodes.

6.2 Where is the API?

The API is a method of the task processor: `API()` in `laf.fabric`. This method returns you a set of *API elements*: objects and/or methods that you can use to retrieve information from the LAF resource: its features, nodes, edges, primary data and even its original XML identifiers.

By calling this method you can insert the API elements in your local namespace. This has the advantage of efficiency, because API elements might easily be called millions of times in a loop, so no time should be wasted by things as method lookup. Local names are faster.

6.3 Calling the API

First you have to get a *processor* object. This is how you get it:

```
from laf.fabric import LafFabric
fabric = LafFabric(
    work_dir='/Users/you/laf-fabric-data',
    laf_dir='/Users/you/laf-fabric_data/laf',
    output_dir='/Users/you/laf-fabric_output',
    save=True,
    verbose='NORMAL',
)
```

All arguments to `LafFabric()` are optional. If you have a config file in your home directory, you can leave out `work_dir=...` and `laf_dir=...` and `save=....` If you have not, or if you want to modify that file, you can pass the desired values to `work_dir` and `laf_dir` and say `save=True`. You have to do that once, after that you can leave out this stuff again.

The `verbose` argument tells LAF-Fabric how much feedback it should give you. Possible values in increasing level of verbosity:

```
SILENT      after initialization absolutely no messages
ERROR       only error messages
WARNING     only error and warning messages
INFO        important information, warnings and errors
```

```
NORMAL      normal progress messages and everything above (default)
DETAIL      detailed messages and above
DEBUG       absolutely everything
```

Once you have the processor, you can load data, according to the source you choose:

```
fabric.load('etcbc4', '--', 'cooccurrences',
    {
        'xmlids': {
            'node': False,
            'edge': False,
        },
        'features': {
            'etcbc4': {
                'node': [
                    'db.otype',
                    'ft.sp,lex_utf8',
                    'sft.book',
                ],
                'edge': [
                ],
            },
        },
        'primary': False,
    },
    compile_main=False, compile_annox=False,
    verbose='NORMAL',
)
exec(fabric.localnames.format(var='fabric'))
```

LAF-Fabric will figure out which data can be kept in memory, which data has to be cleared, and which data needs to be loaded. You can access the LAF data by means of local variables that correspond to various elements of the API, see below.

If you want to change what is loaded in your program, you can simply call the loader as follows:

```
fabric.load_again(
    {
        'xmlids': {
            'node': True,
            'edge': False,
        },
        'features': {
            'etcbc4': {
                'node': [
                    'db.otype,oid',
                    'ft.sp,lex_utf8',
                    'sft.book',
                ],
                'edge': [
                ],
            },
        },
        'primary': False,
    },
    compile_main=False, compile_annox=False,
    verbose='NORMAL',
)
exec(fabric.localnames.format(var='fabric'))
```

Caution: If you want to call the load function inside another function, this trick with `exec` does not work. Then you have to use the other method to get to the API:

```
API = fabric.load( ...)
F = API['F']
...
```

If you only want to add a bit of data, you can simply call:

```
fabric.load_again(
    {
        'features': {
            'etcbc4': {
                'node': [
                    'db.oid',
                ],
            },
        },
    }, add=True
)
exec(fabric.localnames.format(var='fabric'))
```

You can also leave specify the features as a tuple, containing node feature specs and edge feature specs:

```
{
    'features': (
    ''' etcbc4:db.oid
        etcbc4:ft.sp
    ''',
    ''' etcbc4:ft.functional_parent
        etcbc4:ft.mother
    '''
    )
}
```

The features for nodes and edges are specfied as a whitespace separated list of feature names.

Finally, you may omit the namespace (`etcbc4:`) and the labels (`db`, `ft`, `sft`). If this causes ambiguity, LAF-Fabric will choose an arbitrary variant, and inform you about the choice it has made. If that choice does not suit you, you can always disambiguate yourself by supplying label and possibly namespace yourself. So the shortest way is:

```
{'features': ('oid sp', 'functional_parent mother')}
```

compile-source and compile-annox: If you have changed the LAF resource or the selected annotation package, LAF-fabric will detect it and recompile it. The detection is based on the modified dates of the GrAF header file and the compiled files. In cases where LAF-fabric did not detect a change, but you need to recompile, use this flag.

After loading, the individual API methods can be accessed by means of local variables. These variables exist only if they correspond with things that you have called for. Here is an overview.

F: Features (of nodes), only if you have declared node features.

FE: Features (of edges), only if you have declared edge features.

C, **Ci**: Connectivity, only if you have declared *edge* features.

P: Primary data, only if you have specified `'primary': True`.

X: XML identifiers, only in sofar as declared under `'xmlids'`.

NN: The "next node" iterator.

EE: The "next edge" iterator.

NE: The "next event" iterator, only if you have specified `'primary': True`.

msg: The function to issue messages with

infile, **outfile**, **close**, **my_file**: File handling (opening for input, output, , closing, getting full path)

fabric: the laf processor itself

6.4 Node order

There is an implicit partial order on nodes, derived from their attachment to *regions* which are stretches of primary data, and the primary data is totally ordered. The order we use in LAF-Fabric is defined as follows.

Suppose we compare node *A* and node *B*. Look up all regions for *A* and for *B* and determine the first point of the first region and the last point of the last region for *A* and *B*, and call those points *Amin, Amax*, *Bmin, Bmax* respectively.

Then node *A* comes before node *B* if and only if *Amin* < *Bmin* or *Amin* = *Bmin* and *Amax* > *Bmax*.

In other words: if *A* starts before *B*, then *A* becomes before *B*. If *A* and *B* start at the same point, the one that ends last, counts as the earlier of the two.

If neither *A* < *B* nor *B* < *A* then the order is not specified. LAF-Fabric will select an arbitrary but consistent order between thoses nodes. The only way this can happen is when *A* and *B* start and end at the same point. Between those points they might be very different.

This order, while not perfect, is the standard order that LAF-Fabric applies to the nodes. The nice property of this ordering is that if a set of nodes consists of a proper hierarchy with respect to embedding, the order specifies a walk through the nodes were enclosing nodes come first, and embedded children come in the order dictated by the primary data. If two nodes *start and end at the same place* in the primary data, only extra knowledge can decide which embeds which.

A particularly nasty case are nodes that link to a zero-width region in the primary data. How should they be ordered with respect to neighbouring nodes? Is the empty one embedded in its right neighbour, or its left one, or in both, or in neither? All possibilities make sense without further knowledge. LAF-Fabric's default ordering places empty nodes *after* all nodes that start at the same place.

So, LAF-Fabric may not able to order the nodes according to all of your intuitions, because the explicit information in a LAF resource may not completely model those intuitions.

Yet, if you have a particular LAF resource and a method to order the nodes in a more satisfying manner, you can supply a module in which you implement that order. You can then tell LAF-Fabric to override its default order by the custom one. See *Extra data preparation*.

6.5 LAF API

Here is a description of the API elements as returned by the API() call.

6.5.1 F, FE, F_all, FE_all (Features)

Examples:

```
F.otype.v(node)

FE.mother.v(edge)

F.gn.s()

F.gn.s(value='feminine')
```

```
all_node_features = API['F_all']
all_edge_features = API['FE_all']
```

All that you want to know about features and are not afraid to ask.

F is an object, and for each *node* feature that you have declared, it has a member with a handy name. Likewise for *FE*, but now for *edge* features.

`F.etcbc4_db_otype` is a feature object that corresponds with the LAF feature given in an annotation in the annotation space `etcbc4`, with label `db` and name `otype`.

`FE.etcbc4_ft_mother` is also a feature object, but now on an edge, and corresponding with an empty annotation.

You can also leave out the namespace and the label, so the following are also valid:

`F.db_otype` or even `F.otype`. And also: `FE.mother`. However, if the feature name is empty, you cannot leave out the label: `FE.` is not valid.

When there is ambiguity, you will get a warning when the features are requested, from which it will be clear to what features the ambiguous abbreviated forms refer. In other to use the other possibilities, use the more expanded names.

If a node or edge is annotated by an empty annotation, we do not have real features, but still there is an annotation label and an annotation space. In such cases we leave the feature name empty. The values of such annotations are always the empty string.

You can look up feature values by calling the method `v(«node/edge»)` on feature objects. Here `«node/edge»` is an integer denoting the node or edge you want the feature value of.

Note: In LAF-Fabric, nodes and edges are not data structures, they are integers. So they are their own IDs. All data about nodes exists in other global tables: how nodes are attached to regions, how nodes are connected to each other by edges, and the values nodes and edges carry for each of the features.

Alternatively, you can use the slightly more verbose alternative forms:

```
F.item['otype'].v(node)
FE.item['mother'].v(edge)
```

They give exactly the same result: `F.otype` is the same thing as `F.item['otype']` provided the feature has been loaded.

The advantage of the alternative form is that the feature is specified by a *string* instead of a *method name*. That means that you can work with dynamically computed feature names. All abbrevitions that are valid as method name, are also valid as key in the `F.item` dictionary.

You can use features to define sets in an easy manner. The `s()` method yields an iterator that iterates over all nodes for which the feature in question has a defined value. For the order of nodes, see *Node order*.

If a value is passed to `s()`, only those nodes are visited that have that value for the feature in question.

The `F_all` and `FE_all` list all features that are loadable. These are the features found in the compiled current source or in the compiled current annox.

Main source and annox

If you have loaded an extra annotation package (*annox*), each feature value is looked up first according to the data of the *annox*, and only if that fails, according to the main source. The `s()` method combines all relevant information.

6.5.2 C, Ci (Connectivity)

Examples:

A. Normal edge features:

```
target_node in C.feature.v(source_node)
(target_node, value) in C.feature.vv(source_node)
target_nodes in C.feature.endnodes(source_nodes, value='val')

source_node in Ci.feature.v(target_node)
(source_node, value) in Ci.feature.vv(target_node)
source_nodes in Ci.feature.endnodes(target_nodes, value='val')
```

B. Special edge features:

```
target_node in C.laf__x.v(source_node)
target_node in C.laf__y.v(source_node)

source_node in Ci.laf__x.v(target_node)
source_node in Ci.laf__y.v(target_node)
```

C. Sorting the results:

```
target_node in C.feature.v(source_node, sort=True)
(target_node, value) in C.feature.vvs(source_node, sort=True)
target_nodes in C.feature.endnodes(source_nodes, value='val', sort=True)
```

D. Existence of edges:

```
if C.feature.e(node): has_outgoing = True # there is an outgoing edge from node carrying feature
if Ci.feature.e(node): has_incoming = True # there is an incoming edge to node carrying feature
```

(the methods `vv` and `endnodes` are also valid for the special features.

Ad A. Normal edge features

This is the connectivity of nodes by edges. `C` and `Ci` are objects that specify completely how you can walk from one node to another by means of edges.

For each *edge*-feature that you have declared, it has a member with a handy name, exactly as in the `FE` object.

`C.feature` is a connection table based on the edge-feature named `feature`.

Such a table yields for each node `node1` a list of pairs `(node2, val)` for which there is an edge going from `node1` to `node2`, annotated by this feature with value `val`.

This is what the `vv()` methods yields as a generator.

If you are not interested in the actual values, there is a simpler generator `v()`, yielding the list of only the nodes. If there are multiple edges with several values going from `node1` to `node2`, `node2` will be yielded only once.

If you want to travel onwards until there are no outgoing edges left that qualify, use the method `endnodes()`.

For all this functionality there is also a version that uses the opposite edge direction. Use `Ci` instead of `C`.

If you have loaded an extra annotation package (*annox*), lookups are first performed with the data from the *annox*, and only if that fails, from the main source. All relevant data will be combined.

Ad B. Special edge features

There may be edges that are completely unannotated. These edges are made available through the special `C` and `Ci` members called `laf__x`. (Annotation namespace `laf`, no annotation label, name `'x'`.)

If you have loaded an *annox*, it may have annotated formerly unannotated edges. However, this will not influence the `laf__x` feature.

`laf__x` always corresponds to the unannotated edges in the main source, irrespective of any *annox* whatsoever.

But loading an annox introduces an other special edge feature: `laf__y`: all edges that have been annotated by the annox.

In your script you can compute what the unannotated edges are according to the combination of main source and annox. It is all the edges that you get with `laf__x`, minus those yielded by `laf__y`.

Think of `x` as *excluded* from annotations, and `y` as *yes annotations*.

Ad C. Sorting the results

The results of the `v` and `vv` methods are unordered, unless `sort=True` is passed. In that case, the results are ordered in the standard ordering or in the custom ordering if you have loaded a prepared ordering.

See the example notebook trees for working code with connectivity.

Ad D. Existence of edges

If you want to merely check whether a node has outgoing edges with a certain annotated feature, you can use the direct method `e(node)`. This is much faster than using the `v(node)` mode, since the `e()` method builds less data structures.

General remark All methods of `C` and `Ci` objects that deliver multiple results, yield them one by one as iterators.

6.5.3 BF (Before)

Examples:

```
if BF(nodea, nodeb) == None:
    # code for the case that nodea and nodeb do not have a mutual order
elif BF(nodea, nodeb):
    # code for the case that nodea comes before nodeb
else:
    # code for the case that nodea comes after nodeb
```

With this function you can do an easy check on the order of nodes. The *BF()* ordering orders the nodes as *NN()* does, but it indicates when two nodes cannot be ordered. There is no mutual order between two nodes if at least one of the following holds:

- at least one of them is not linked to the primary data
- both start and end at the same point in the primary data (what happens in between is immaterial).

BF(n,m) yields `True` if *n* comes before *m*, `False` if *m* comes before *n*, and `None` if none of these is the case.

Note: The *BF()* ordering is **not** influenced by an additional ordering that you might have added to LAF-Fabric by data preparation. So even if you have loaded a more complete ordering, you still can analyse for which pairs of nodes the extra ordering introduces extra order.

6.5.4 EE (Next Edge)

Examples:

```
(a0) for edge in EE():
         pass
```

EE() walks through edges, in unspecified order. It yields for every edge a tuple *(id, from, to)*, where *id* is the identifier of the edge (an integer), and *from* and *to* are the nodes from which and to which the edge goes. These nodes are specified by their node identifiers (integers).

6.5.5 NN (Next Node)

Examples:

```
(a0) for node in NN():
         pass

(a1) for node in NN(nodes=nodeset):
         pass
```

```
(a2) for node in NN(nodes=nodeset, extrakey=your_order):
         pass

(b)  for node in NN(test=F.otype.v, value='book'):
         pass

(c)  for node in NN(test=F.book.v, values=['Isaiah', 'Psalms']):
         pass

(d)  for node in NN(
         test=F.otype.v,
         values=['phrase', 'word'],
         extrakey=lambda x: F.otype.v(x) == 'phrase',
     ):
         pass
```

NN() walks through nodes, not by edges, but through a predefined set, in the natural order given by the primary data (see *Node order*). Only nodes that are linked to a region (one or more) of the primary data are being walked. You can walk all nodes, or just a given set.

It is an *iterator* that yields a new node everytime it is called.

All arguments are optional. They mean the following, if present.

- `test`: A filter that tests whether nodes are passed through or inhibited. It should be a *callable* with one argument and return some value;
- `value`: string
- `values`: an iterable of strings.

`test` will be called for each passing node, and if the value returned is not in the set given by `value` and/or `values`, the node will be skipped. If neither `value` or `values` are provided, the node will be passed if and only if `test` returns a true value.

- `nodes`: this will limit the set of nodes that are visited to the given value, which must be an iterable of nodes. Before yielding nodes, `NN(nodes=nodeset)` will order the nodes according to the standard ordering, and if you have provided an extra, prepared ordering, this ordering will be taken instead.

The `nodes` argument is compatible with all other arguments.

Note: `nodelist = NN(nodes=nodeset)` is a practical way to get the nodeset in the right order. If your program works a lot with nodeset, and then needs to produce orderly output, this is your method. If you have a custom ordering defined in your task, you can apply it to arbitrary node sets via `NN(nodes=nodeset, extrakey=your_order)`.

Alternatively, you can say: `nodelist = sorted(nodeset, key=NK)`. See the API element NK.

Example (a) iterates through all nodes, (a1) only through the nodes in nodeset, (a2) idem, but applies an extra ordering beforehand, (b) only through the book nodes, because *test* is the feature value lookup function associated with the `otype` function, which gives for each node its type.

Note: The type of a node is not a LAF concept, but a concept in this particular LAF resource. There are annotations which give the feature `otype` to nodes, stating that nodes are books, chapters, words, phrases, and so on.

In example (c) you can give multiple values for which you want the corresponding nodes.

Example (d) passes an extra sort key. The set of nodes is sorted on the basis of how they are anchored to the primary data. Left comes before right, embedding comes before embedded. But there are many cases where this order is not defined, namely between nodes that start at the same point and end at the same point.

If you have extra information to order these cases, you can do so by passing `extrakey`. In this case the extrakey is `False` for nodes with carry a certain feature with value `phrase`, and `True` for the other nodes, which carry

value `word` for that feature. Because `False` comes before `True`, the phrases come before the words they contain.

Note: Without extrakey, all nodes that have not identical start and end points have already the property that they are yielded in the proper mutual order. The difficulty is where the `BF` method above yields `None`. It is exactly these cases that are remedied with `extrakey`. The rest of the order remains untouched.

Caution: Ordering the nodes with `extrakey` is costly, it may take several seconds. The etcbc module comes with a method to compute this ordering once and for all. This supplementary data can easilyand quickly be loaded, and then you do not have to bother about `extrakey` anymore. See *Extra data preparation*.

Note: You can invoke a supplementary module of your choice to make the ordering more complete. See the section on extra data preparation below.

See `next_node()` in `laf.fabric`.

6.5.6 NK (node sort key)

Example:

```
nodelist = sorted(nodeset, key=NK)
```

This can be passed as a sort key for node sets. It corresponds with the "natural order" on nodes. If an additional module, such as *etcbc.preprocess* has modified the natural order, this sort key will reflect the modified order. If you let NN() yield nodes, they appear in this same order.

6.5.7 MK (anchor set sort key)

Example:

```
anchorsets = sorted(anchorsets, key=MK)
```

This can be passed as a sort key for anchor sets. It corresponds with the "natural order" on anchor sets, which is: Let *sa* and *sb* are two anchor sets.

If *sa* is a proper subset of *sb* then *sb* comes before *sa* and vice versa.

Otherwise, if *sa* and *sb* are not equal, the one that has the smallest element not occurring in the other comes first.

6.5.8 NE (Next Event)

Examples:

```
for (anchor, events) in NE():
    for (node, kind) in events:
        if kind == 3:
            '''close node event'''
        elif kind == 2:
            '''suspend node event'''
        elif kind == 1:
            '''resume node event'''
        elif kind == 0:
            '''start node event'''

for (anchor, events) in NE(key=filter):
for (anchor, events) in NE(simplify=filter):
for (anchor, events) in NE(key=filter1, simplify=filter2):
```

‘‘NE()‘‘ is only available if you have specified in the *load* directives: ‘‘primary: True‘‘.

NE() walks through the primary data, or, more precisely, through the anchor positions where something happens with the nodes.

It is an *iterator* that yields the set of events for the next anchor that has events everytime it is called. It will return a pair, consisting of the anchor position and a list of events.

See `next_event()` in `laf.fabric`.

What can happen is that a node *starts*, *resumes*, *suspends* or *ends* at a certain anchor position. This things are called *node_events*.

start The start anchor of the first range that the node is linked to

resume The start anchor of any non-first range that the node is linked to

suspend The end anchor of any non-last range that the node is linked to

end The end anchor of the last range that the node is linked to

The events for each anchored are are ordered according to the primary data order of nodes, see *Node order*, where for events of the kind *suspend* and *end* the order is reversed.

Caution: While the notion of node event is quite natural and intuitive, there are subtle difficulties. It all has to do with embedding, gaps and empty nodes. If your nodes link to portions of primary data with gaps, and if some nodes link to points in de primary data (rather than stretches), then the node events generated by NE() will in general not completely ordered as desired. You should consider using more explicit information in your data about embedding, such as edges between nodes. If not, you have to code intricate event reordering in your notebook.

Note: For non-empty nodes (i.e. nodes linked to at least one region with a distinct start and end anchor), this works out nicely. At any anchor the closing events are before the opening events. However, an empty node would close before all other closing events at that node, and open after all other opening events at that node. It would close before it would open. That is why we treat empty nodes differently: their open-close events are placed between the list of close events of other nodes and the list of open events of other nodes.

Note: The embedding of empty nodes is hard to define without further knowledge. Are two empty nodes at the same anchor position embedded in each other or not? Is an empty node embedded in a node that opens or close at the same anchor? We choose a minimalistic interpretation: multiple embedded nodes at the same anchor are not embedded in each other, and are not embedded in nodes that open or close at the same anchor.

The consequence of this ordering is that if the nodes correspond to a tree structure, the node events correspond precisely with the tree structure. You can use the events to generate start and end tags for each node and you get a properly nested representation.

Note however, that if two nodes have the same set of ranges, it is impossible to say which embeds which.

You can, however, pass a *key=filter* argument to NE(). Before a node event is generated for a node, *filter* will be applied to it. If the outcome is `None`, the events for this node will be skipped, the consumer of events will not see them. If the outcome is not `None`, the value will be used as a sort key for additional sorting.

The events are already sorted fairly good, but only those node events that have the same kind and corresponds to nodes with the same start and end point, may occur in an undesirable order. By assigning a key, you can remedy that. The key will be used in inversed order for opening/resume events, and in normal order for close/suspend events.

For example, if you pass a filter as *key* that assigns to nodes that correspond to *sentences* the number 5, and to nodes that correspond to *clauses* the number 4, then the following happens.

Whenever there is a sentence that coincides with a clause, then the sentence-open event will occur before the clause-open event, and the clause-close before the sentence-close.

Note: The ordering induced by *key=filter* is also applied to multiple empty nodes at the same anchor. Without the

ordering, they are not embedded in each other, but the ordering may embed some empty nodes in other ones. This additional ordering will not reorder events for empty nodes with those of enclosing non-empty nodes, because it is impossible to tell whether an empty node is embedded in a node that is closing at this point or at a node that is opening at this point.

If there are many regions in the primary data that are not inside regions or in regions that are not linked to nodes, or in regions not linked to relevant nodes, it may bethe case that many relevant nodes get interrupted around these gaps. That will cause many spurious suspend-resume pairs of events. It is possible to suppress those.

Example: suppose that all white space is not linked to nodes, and suppose that sentences and clauses are linked to their individual words. Then they become interrupted at each word.

If you pass the *simplify=filter* argument to NE() the following will happen. First of all: a gap is now a stretch of primary data that does not occur between the start and end position of any node for which the filter is not None.

In our example of sentences and clauses: suppose that a verse is linked to the continuous regions of all its material, including white space. Suppose that by our *key=filter1* argument we are interested in sentences, clauses and verses. With respect to this set, the white spaces are no gaps, because they occur in the verses.

But if we give a simplify=filter2 that only admits sentences and clauses, then the white spaces become true gaps. And NE(simplify=filter2) will actively weed out all node-suspend, node-resume pairs around true gaps.

Even if the nodes do not correspond with a tree, the order of the node events correspond to an intuitive way to mark the embedding of nodes.

Note that we do not say *region* but *range*. LAF-Fabric has converted the region-linking of nodes by range-linking. The range list of a node is a sequence of maximal, non-overlapping pieces of primary data in primary data order.

Consequently, if a node suspends at an anchor, it will not resume at that anchor, so the node has a real gap at that anchor.

Formally, a node event is a tuple `(node, kind)` where `kind` is 0, 1, ,2, or 3, meaning *start*, *resume*, *suspend*, *end* respectively.

6.5.9 X, XE (XML Identifiers)

Examples:

```
X.r(i)
X.i(x)
XE.r(i)
XE.i(x)
```

If you need to convert the integers that identify nodes and edges in the compiled data back to their original XML identifiers, you can do that with the *X* object for nodes and the *XE* object for edges.

Both have two methods, corresponding to the direction of the translation: with `i(«xml id»)` you get the corresponding number of a node/edge, and with `r(«number»)` you get the original XML id by which the node/edge was identified in the LAF resource.

6.5.10 P (Primary Data)

Examples:

```
P.data(node)
```

The primary data is only available if you have specified in the *load* directives: ‘‘primary: True‘‘.

Your gateway to the primary data. For nodes `node` that are linked to the primary data by one or more regions, `P.data(node)` yields a set of chunks of primary data, corresponding with those regions.

The chunks are *maximal*, *non-overlapping*, *ordered* according to the primary data.

Every chunk is given as a tuple (*pos*, *text*), where *pos* is the position in the primary data where the start of *text* can be found, and *text* is the chunk of actual text that is specified by the region.

Caution: Note that *text* may be empty. This happens in cases where the region is not a true interval but merely a point between two characters.

6.5.11 Input and Output

Examples:

```
data_dir
output_dir
out_handle = outfile('output.txt')
in_handle  = infile('input.txt')
file_path = my_file('thefile.txt')
close()

msg(text)
msg(text, verbose='ERROR')
msg(text, newline=False)
msg(text, withtime=False)
```

data_dir is the top-level directory where all input data (laf resources, extra annotation files) reside.

output_dir is the top-level directory where all task output data is collected.

You can create an output filehandle, open for writing, by calling the `outfile()` method and assigning the result to a variable, say *out_handle*.

From then on you can write output simply by saying:

```
out_handle.write(text)
```

You can create as many output handles as you like in this way. All these files end up in the task specific working directory.

Likewise, you can place additional input files in that directory, and read them by saying:

```
text = in_handle.read()
```

You can have LAF-Fabric close them all by means of `close()` without arguments.

If you want to refer in your notebook, outside the LAF-Fabric context, to files in the task-specific working directory, you can do so by saying:

```
full_path = my_file('thefile.txt')
```

The method `my_file` prepends the full directory path in front of the file name. It does not check whether the file exists.

You can issue progress messages while executing your task. These messages go to the output of a code cell.

You can adjust the verbosity level of messages, see above for possible values.

These messages get the elapsed time prepended, unless you say `withtime=False`.

A newline will be appended, unless you say `newline=False`.

The elapsed time is reckoned from the start of the task, but after all the task-specific loading of features.

6.5.12 fabric

You also have access to the laf processor itself, by means of the `fabric` key in the `API`.

Here are some useful methods.

resolve_feature

Example:

```
fabric.resolve_feature('node', 'otype')
fabric.resolve_feature('node', 'db.otype')
fabric.resolve_feature('node', 'etcbc4:db.otype')
```

Resolves incomplete and complete feature names. Raises FabricError if there is no resolution in the current resource. If there are resolutions, delivers the last one found, in the form of a tuple (*namespace*, *label*, *feature name*). If there aremultiple resolutions, lists all the candidates and tells which one has been chosen.

6.6 Extra data preparation

Caution: This section is meant for developers of extra modules on top of LAF-Fabric

LAF-Fabric admits other modules to precompute data to which it should be pointed. See :doc:`etcbc-reference` for an example.

CHAPTER 7

ETCBC Reference

7.1 What is ETCBC

The *etcbc* package has modules that go beyond *laf*. They utilize extra knowledge of the specific LAF resource which is the ETCBC Hebrew Text Database. They make available a better ordering of nodes, add more ways of querying the data, and ways of creating new annotations. There is also a solution for the problem of getting relevant context around a node. For example, if you do a walk through phrases, you want to be able to the clauses that contain the phrases that you iterate over, or to siblings of it.

Most of the functionality is demonstrated in dedicated notebooks. This text is only a rough overview.

7.2 Layers

The `L` (*layer*) part of the API enables you to find objects that are embedded in other objects and vice versa. It makes use of the ETCBC object types `book`, `chapter`, `verse`, `half_verse`, `sentence`, `sentence_atom`, `clause`, `clause_atom`, `phrase`, `phrase_atom`, `subphrase`, `word`. An object of a certain type may contain objects of types following it, and is contained by objects of type preceding it.

By means of `L` you can go from an object to any object that contains it, and you can get lists of objects contained in it. This is how it works. You have to import the `prepare` module:

```
from etcbc.preprocess import prepare
```

and say in your load instructions:

```
``'prepare': prepare``
```

Then you can use the following functions:

```
L.u(otype, node)
L.d(otype, node)
L.p(otype, book='Genesis', chapter=21, verse=3, sentence=1, clause=1, phrase=1)
```

`L.u` (up in the hierarchy) gives you the object of type `otype` that contains `node` (in the ETCBC data there is at most one such an object). If there is no such object, it returns `None`.

`L.d` (down in the hierarchy) gives you all objects of type `otype` that are contained in `node` as a list in the natural order. If there are no such objects you get `None`.

`L.p` (passage nodes) give you all objects of type `otype` that are contained in the nodes selected by the other arguments. All other arguments are optional. So if you leave out the `sentence clause phrase` arguments, you get all nodes in a specific verse. If you leave out the `book chapter verse` arguments, and leave the others at `1`, you get the nodes in all first phrases of first clauses of first sentences of all verses of all chapters of all books.

Examples (if `phr` is a node with object type `phrase`):

```
b = L.u('book', phr)                    # the book node in which the node occurs
F.book.v(b)                             # the name of that book

b = F.code.v(L.u('clause_atom', phr)) # the *clause_atom_relationship* of the clause_atom of whic
```

It is now easy to get the full text contained in any object, e.g. the phrase `phr`:

```
''.join('{}{}'.format(F.g_word_utf8.v(w), F.trailer_utf8.v(w)) for w in L.d(phr))
```

Conversely, it is easy to get all subphrases in a given verse:

```
subphrases = L.p('subphrase', book='Exodus', chapter=5, verse=10)
```

or get all clause_atoms of all first sentences of all second verses of all chapters in Genesis:

```
clause_atoms = L.p('clause_atom', book='Genesis', verse=2, sentence=1)
```

7.3 Node order

The module `etcbc.preprocess` takes care of preparing a table that codes the optimal node order for working with ETCBC data.

It orders the nodes in a way that combines the left-right ordering with the embedding ordering. Left comes before right, and the embedder comes before the embedded.

More precisely: if we want to order node *a* and *b*, consider their monad sets *ma* and *mb*, and their object types *ta* and *tb*. The object types have ranks, going from a low rank for books, to higher ranks for chapters, verses, half_verses, sentences, sentence_atoms, clauses, clause_atoms, phrases, phrase_atoms, subphrases and words.

In the ETCBC data every node has a non-empty set of monads.

If *ma* is equal to *mb* and *ta* is equal to *tb*, then *a* and *b* have the same object type, and cover the same monads, and in the etcbc that implies that *a* and *b* are the same node.

If *ma* is equal to *mb*, then if *ta* is less than *tb*, *a* comes before *b* and vice versa.

If *ma* is a proper subset of *mb*, then *a* comes *after b*, and vice versa.

If none of the previous conditions hold, then *ma* has monads not belonging to *mb* and vice versa. Consider the smallest monads of both difference sets: *mma = min(ma-mb)* and *mmb = min(mb-ma)*. If *mma* < *mmb* then *a* comes before *b* and vice versa. Note that *mma* cannot be equal to *mmb*.

Back to your notebook. Say:

```
from etcbc.preprocess import prepare

processor.load('your source', '--', 'your task',
    {
        "xmlids": {"node": False, "edge": False},
        "features": { ... your features ...},
        "prepare": prepare,
    }
)
```

then the following will happen:

- LAF-Fabric checks whether certain data files that define the order between nodes exist next to the binary compiled data, and whether these files are newer than your module *preprocess.py*.
- If so, it loads these data files quickly from disk.
- If not, it will compute the node order and write them to disk. This may take some time! Then it replaces the *dumb* standard ordering by the *smart* ETCBC ordering.

- Likewise, it looks for computed files with the embedding relationship, and computes them if necessary. This takes even more time!

This data is only loaded if you have done an import like this:

```
from etcbc.preprocess import prepare
```

and if you have:

```
'prepare': prepare
```

in your load instructions,

7.4 Transcription

7.4.1 Hebrew

The ETCBC has a special way to transcribe Hebrew characters into latin characters. Sometimes it is handier to work with transcriptions, because some applications do not render texts with mixed writing directions well.

In *etcbc.lib* there is a conversion tool. This is how it works:

```
from etcbc.lib import Transcription

tr = Transcription()

t = 'DAF DAC'
h = tr.to_hebrew(t)
tb = tr.from_hebrew(h)

print("{}\n{}\n{}".format(t, h, tb))
```

`to_hebrew(word)` maps from transcription to Hebrew characters, `from_hebrew(word)` does the opposite.

There are some points to note:

- if characters to be mapped are not in the domain of the mapping, they will be left unchanged.
- there are two versions of the shin, each consists of two combined unicode characters. Before applying the mappings, these characters will be combined into a single character. After applying the mapping `hebrew()`, these characters will be *always* decomposed.
- up till now we have only transcription conversions for *consonantal Hebrew*.

7.4.2 Syriac

We have a transcription for consonantal Syriac. The interface is nearly the same as for Hebrew, but now use:

```
to_syriac(word)
from_syriac(word)
```

7.5 Trees

The module *etcbc.trees* gives you several relationships between nodes: *parent*, *children*, *sisters*, and *elder_sister*.:

```
from etcbc.trees import Tree

tree = Tree(API, otypes=('sentence', 'clause', 'phrase', 'subphrase', 'word'),
    clause_type='clause',
    ccr_feature='rela',
```

```
    pt_feature='typ',
    pos_feature='sp',
    mother_feature = 'mother',
)
ccr_class = {
    'Adju': 'r',
    'Attr': 'r',
    'Cmpl': 'r',
    'CoVo': 'n',
    'Coor': 'x',
    'Objc': 'r',
    'PrAd': 'r',
    'PreC': 'r',
    'Resu': 'n',
    'RgRc': 'r',
    'Spec': 'r',
    'Subj': 'r',
    'NA':   'n',
}

tree.restructure_clauses(ccr_class)

results = tree.relations()
parent = results['rparent']
sisters = results['sisters']
children = results['rchildren']
elder_sister = results['elder_sister']
```

When the `Tree` object is constructed, the monadset-embedding relations that exist between the relevant objects, will be used to construct a tree. A node is a parent of another node, which is then a child of that parent, if the monad set of the child is contained in the monad set of the parent, and if there are not intermediate nodes (with respect to embedding) between the parent and the child. So this *parent* relationship defines a *tree*, and the *children* relationship is just the inverse of the *parent* relationship. Every node has at most 1 parent, but nodes may have multiple children. If two nodes have the same monad set, then the object type of the nodes determines if one is a parent and which one that is. A sentence can be parent of a phrase, but not vice versa.

It can not be the case that two nodes have the same monad set and the same object type.

You can customize your trees a little bit, by declaring a list of object types that you want to consider. Only nodes of thos object types will enter in the parent and children relationships. You should specify the types corresponding to the ranking of object types that you want to use. If you do not specify anything, all available nodes will be used and the ranking is the default ranking, given in *etcbc.lib.object_rank*.

There is something curious going on with the *mother* relationship, i.e. the relationship that links on object to another on which it is linguistically dependent. In the trees just constructed, the mother relationship is not honoured, and so we miss several kinds of linguistic embeddings.

The function `restructure_clauses()` remedies this. If you want to see what it going on, consult the trees_etcbc4 notebook.

7.6 Annotating

The module `etcbc.annotating` helps you to generate data entry forms and translate filled in forms into new annotations in LAF format, that actually refer to nodes and edges in the main ETCBC data source.

There is an example notebook that uses this module for incorporating extra data (coming from so-called *px* files) into the LAF resource. See *Extra Data* below.

7.7 Extra Data

The ETCBC data exists in so-called *px* files, from which the EMDROS databases are generated. Some *px* data did not made it too EMDROS, hence this data does not show up in LAF. Yet there might be useful data in the *px*. The module **etcbc.extra** helps to pull that data in, and delivers it in the form of an extra annotation package.

You can also use this module to add other kinds of data. You only need to write a function that delivers the data in the right form, and then *extra* turns it into a valid annotation set.

Usage:

```
from etcbc.extra import ExtraData
```

More info: notebook para from px

7.8 Feature documentation

The module `etcbc.featuredoc` generates overviews of all available features in the main source, including information of their values, how frequently they occur, how many times they are filled in with (un)defined values. It can also look up examples in the main data source for you.

Usage:

```
from etcbc.featuredoc import FeatureDoc
```

More info: notebook feature-doc

7.9 MQL

The module `etcbc.mql` lets you fire mql queries to the corresponding Emdros database, and process the results with LAF-Fabric. More info over what MQL, EMDROS are, and how to use it, is in notebook mql.

On the Mac and in Linux it runs out of the box, assuming Emdros is installed in such a way that the command to run MQL is `/usr/local/bin/mql`. If that is not the case, or if you work on windows, you should manually change the first line of *mql.py*. Its default value is:

```
MQL_PROC = '/usr/local/bin/mql'
```

and on windows is should become something like:

```
MQL_PROC = 'c:\\Program Files (x86)\\Emdros\\Emdros 3.4.0\\bin\\mql'
```

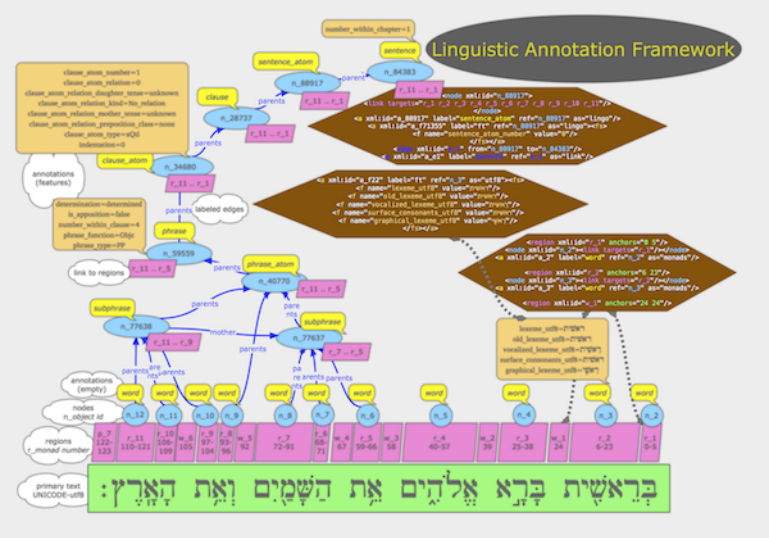
(check your system). After modifying this file, you should go to your *laf-fabric* directory and run again:

```
python setup.py install
```

Regrattably, this must be repeated when you update laf-fabric from Github.

CHAPTER 8

EMDROS2LAF reference



8.1 Description

EMDROS2LAF is a package that can convert EMDROS databases into LAF resources. It is assumed that the EMDROS database is organized in the ETCBC way.

8.2 Usage

There is an example script in the top level *laf-fabric* directory, called *lf-convert.py*. It can be called as follows:

```
python lf-convert.py [--raw] [--validate] [--fdecls-only] --source source --parts [Part]*
```

where * *source* is the name of a data source, corresponding to a subdirectory of the *laf-fabric-data* directory, * and part is `monad`, `section`, `lingo`, `all`, `none`.

Transforms the ETCBC database into a LAF resource. Because of file sizes, not all annotations are stored in one file. There are several parts of annotations: monads (words), sections (books, chapters, verses, etc), lingo (sentence, phrase, etc)

If –raw is given , a fesh export from the EMDROS database is made. For each part there is a separate export.

If –validate is given, generated xml files will be validated against their schemas.

If –fdecls-only is given, only the feature declaration file is generated.

The conversion is driven by a feature specification file. This file contains all information about objects, features and values that the program needs. The division into parts, but also the mapping to ISOcat is given in this file.

8.3 Input

The main input for the program is an EMDROS database, from which data will be exported by means of MQL queries. For every part (monad, section, lingo) an mql query file is generated, and this query is run against the database. The result is a plain text file (unicode utf8) per part.

8.4 Output

This is what the program generates:

The main output are annotation files plus a primary data file. And there are descriptive headers. The primary data file is a plain text file (unicode utf8) containing the complete underlying text of the database in question. There is some chunking into books, chapters and verses, only by means of newlines. No section indications occur in the primary text. This file is obtained from a few text-carrying features present in the database.

Annotation files are xml files that describe regions of in the primary data, and properties of those regions. Annotations are the translation of the ETCBC objects and features. Annotation files start with header information.

There are several header files, one for the LAF resource as a whole, one for the primary data file, and one for linking the object types and features to descriptions in the ISOcat registry.

All generated XML files will be validated against their schemas by means of xmllint.

8.5 Definitions

The conversion process is defined by a substantial amount of information outside the program. This information comes in the form of a main configuration file, a feature definition file, a bunch of templates, and several XML schemas.

The main config files specifies file locations, the version of the text database, and the location of the ISOcat registry. The feature definition file is a big list of object types, their associated features with their enumerated values plus the ISOcat correspondences of it all. It also chunks the LAF materials to be generated into a monad, section and lingo part, providing even one more layer of subdivisions, in order to keep the resulting xml files manageable.

8.6 Project

SHEBANQ, funded by CLARIN-NL, 2013-05-01 till 2014-05-01

8.7 See also

There is another package in this distribution, *laf*, that gives you analytic access to LAF resources, such as the conversion result of the ETCBC database. And *etcbc* is a package by which you can integrate working on both the EMDROS version of the data and the LAF version of the data.

CHAPTER 9

laf 4.5.3

9.1 Submodules

9.2 laf.fabric module

class Bunch
Bases: builtins.object

class LafAPI(*names*)
Bases: laf.data.LafData

Makes all API methods available. API() returns a dict keyed by mnemonics and valued by API methods.

API()

APIprep()

get_all_features()

class LafFabric(*data_dir=None*, *laf_dir=None*, *output_dir=None*, *save=False*, *verbose=None*)
Bases: builtins.object

Process manager.

load(params): given the source, annox and task, loads the data, assembles the API, and returns the API.

load(*source*, *annox*, *task*, *load_spec*, *add=False*, *compile_main=False*, *compile_annox=False*, *verbose=None*)

load_again(*load_spec*, *add=False*, *compile_main=False*, *compile_annox=False*, *verbose=None*)

resolve_feature(*kind*, *feature_given*)

9.3 laf.elements module

class Connection(*lafapi*, *feature*, *inv*)
Bases: builtins.object

Connection info according to an edge feature.

Holds the mapping from nodes to a set of (node, value) pairs for which there is an edge for which this edge feature has value. Has distinct mappings for main source data and annox data.

v(node) yields the nodes (without the values). vv(node) yields the node/value pairs. endnodes(nodeset, value=None) yields the set of end nodes after traveling from ``nodeset along edges (having this feature with this value or any value).

e(*n*)

endnodes(*node_set*, *value=None*, *sort=False*)

v(*n*, *sort=False*)

vv(*n*, *sort=False*)

class Feature(*lafapi*, *feature*, *kind*)
Bases: `builtins.object`

Feature data and lookup.

Holds the mapping from nodes/edges to values corresponding to a single feature. Has distinct mappings for main source data and annox data.

`v(node_or_edge)` is the lookup method. `s(value=None)` yields the nodes/edges that have this value or any value.

V(*ne*)

s(*value=None*)

v(*ne*)

class PrimaryData(*lafapi*)
Bases: `builtins.object`

Primary data.

`data(node)` is a list of chunks of primary data attached to that node. The chunk is delivered as a pair of the position where the chunk starts and the chunk itself. Empty chunks are possible. Consecutive chunks have been merged. The chunks appear in primary data order.

data(*node*)

class XMLid(*lafapi*, *kind*)
Bases: `builtins.object`

Mappings between XML identifiers in original LAF resource and integers identifying nodes and edges in compiled data.

`r(node or edge int) = xml identifier` and `i(xml identifier) = node or edge int`.

i(*xml_id*)

r(*int_code*)

9.4 laf.data module

class LafData
Bases: `builtins.object`

Manage the compiling and loading of LAF/GraF data.

add_logfile(*compile=None*)

adjust_all(*annox*, *req_items*, *add*, *force*)
Load manager.

compile_all(*force*)

finish_task(*show=True*)
Close all open files that have been opened by the API

load_all(*req_items*, *prepare*, *add*)

prepare_all(*api*)

prepare_dirs(*annox*)

unload_all()

9.5 laf.model module

model(*origin*, *data_items*, *stamp*)
Augment the results of XML parsing by precomputing additional data structures.

normalize_ranges(*ranges*)

9.6 laf.parse module

class AnnotationHandler(*annotation_file*, *stamp*)
Bases: xml.sax.handler.ContentHandler

aid = None

characters(*ch*)

endElement(*name*)

file_name = None

nid = None

stamp = None

startElement(*name*, *attrs*)

truth = {'on': True, 'yes': True, 'true': True, 'no': False, 'off': False, 'false': False, '1': True, '0': False}

class HeaderHandler
Bases: xml.sax.handler.ContentHandler

characters(*ch*)

endElement(*name*)

startElement(*name*, *attrs*)

init()

parse(*origin*, *graf_header_file*, *stamp*, *data_items*)
Parse a LAF/GrAF resource and deliver results.

9.7 laf.names module

exception FabricError(*message*, *stamp*, *cause=None*)
Bases: builtins.Exception

class Names(*data_dir*, *laf_dir*, *output_dir*, *save*, *verbose*)
Bases: laf.settings.Settings

Manage the names of compiled LAF data items.

Data items are stored in a dictionary with keys that tell a lot about the kind of data stored under that key. Keys have the following format:

```
origin group kind direction ( item )
```

and **item** is a comma separated list of a variable number of components, possibly zero.

Group:

- `P`: primary data items,
- `G`: items for regions, nodes, edges,
- `X`: xml identifiers,
- `F`: features,
- `C`: connectivity,
- `T`: temporary during compiling.

Origin: `m` or `a` meaning *main* and *annox* resp. Indicates the source data. The value `z` indicates that this data is not prepared by Laf-Fabric but by auxiliary modules.

Kind: `n` or `e` meaning *node* and *edge* resp.

Direction: `f` or `b` meaning *forward* and *backward* resp.

The direction can mean the direction in which edges are followed, or the direction in which a mapping goes.

Components:

Features are items, with three components: (*namespace*, *label*, *name*).

In group `P`, `G`, `T` there are one-component items, such as (`edges_from`,) and (`edges_to`).

In group `X` there is only one item, and it has no components: ().

For each data item we have to know the conditions under which it has to be loaded and its data type.

The **condition** is a key in a dictionary of conditions. The loader determines the condition dictionary by filling in its slots with relevant components.

The **data type** is either array, or dict, or string.

Class methods The class methods `comp` and `decomp` and `decompfull` take care of the composition and decomposition of keys in meaningful bits.

Instance data and methods The instance data contains a list of datakeys, adapted to the present environment, which is based on the source, annox and task chosen by the user. The previous list is also remembered, so that the loader can load/unload the difference.

The instance method `request_files` determines the difference between previously and presently requested data items. It uses an instance method `dinfo` that provides all relevant information associated with a datakey, including the location and name of the corresponding data file on disk. This method is an instance method because it needs values from the current environment.

`DCOMP_SEP` = ‘,’

`E_ANNOT_NON` = (‘laf’, ‘’, ‘x’)

`E_ANNOT_YES` = (‘laf’, ‘’, ‘y’)

`apiname`(*dcomps*)

`check_load_spec`(*load_spec*, *stamp*)

`comp`(*dkeymin*, *dcomps*)

`comp_file`(*dgroup*, *dkind*, *ddir*, *dcomps*)

`decomp`(*dkey*)

`decomp_full`(*dkey*)

`deliver`(*computed_data*, *dest*, *data_items*)

`dinfo`(*dkey*)

`dmsg`(*dkey*)

`kind_types` = {False, True}

`load_spec_keys` = {‘xmlids’, ‘prepare’, ‘primary’, ‘features’}

load_spec_subkeys = {'node', 'edge'}

orig_key(*dkey*)

query(*dorigin=None*, *dgroup=None*)

request_files(*req_items*, *prepare_dict*)

request_init(*req_items*)

9.8 laf.settings module

class Settings(*data_dir*, *laf_dir*, *output_dir*, *save*, *verbose*)
Bases: builtins.object

Manage the configuration.

The directory structure is built as a set of names in an environment dictionary. The method setenv builds a new structure based on user choices. Local settings can be passed as arguments to object creation, or in a config file in the current directory, or in a config file in the user's home directory. It is possible to save these settings in the latter config file.

setenv(*source=None*, *annox=None*, *task=None*, *zspace=None*)

9.9 laf.timestamp module

class Timestamp(*log_file=None*, *verbose=None*)
Bases: builtins.object

Timed progress messages.

There are specialized methods for distinct verbosity levels: Emsg, Wmsg etc. With set_verbose you can set the verbosity of the application. Only messages with that verbosity level of lower will reach the output. You can suppress the time indication and the newline at the end.

raw_msg has complete flexibility. This method is exposed as msg in the API.

Dmsg(*msg*, *newline=True*, *withtime=True*)

Emsg(*msg*, *newline=True*, *withtime=True*)

Imsg(*msg*, *newline=True*, *withtime=True*)

Nmsg(*msg*, *newline=True*, *withtime=True*)

Wmsg(*msg*, *newline=True*, *withtime=True*)

Xmsg(*msg*, *newline=True*, *withtime=True*)

connect_log(*log_file*)

disconnect_log()

raw_msg(*msg*, *newline=True*, *withtime=True*, *verbose=None*)

reset()

set_verbose(*verbose*)

verbose_level = OrderedDict([('SILENT', -10), ('ERROR', -3), ('WARNING', -2), ('INFO', -1), ('NORMAL', 0),

9.10 laf.lib module

arrayify(*source_list*)

grouper(*iterable*, *n*, *fillvalue=None*)

Collect data into fixed-length chunks or blocks

grouper([1,2,3,4,5], 2, 0) –> [1,2] [3,4] [5,0]

make_array_inverse(*arraylist*)

make_inverse(*mapping*)

CHAPTER 10

etcbc 4.5.3

10.1 Submodules

10.2 etcbc.preprocess module

fill (*NN*, *F*, *er*, *ed*)

getmonads (*attr*)

node_down (*API*)

node_order (*API*)

node_order_inv (*API*)

node_ud (*API*)

node_up (*API*)

prep_post (*lafapi*)

10.3 etcbc.annotating module

class GenForm (*API*, *form_name*, *config*)

Bases: `builtins.object`

Generates input forms for new annotations and creates new annotations based on filled in forms.

Upon creation, takes in the config information for a new form.

Args:

API(object): the API object of the LAF processor, so that the form creator can use its methods for creating files and issuing messages and accessing features.

form_name(string): the base name of the form to be created

config(dict): which nodes and feature data to fill the form with, which new feature columns to make.

make_annots ()

Converts a filled in form into a set of new annotations.

make_form ()

Creates a form based on the information passed when creating this object.

create_annots (*API*, *data*)

Converts a list of node, value,feature entries to a string which can be saved as an annotation file.

The columns must be: nodeid, value, feature name, feature_label (optional), namespace (optional)

nonerep(*val*)

10.4 etcbc.featuredoc module

class FeatureDoc(*processor*, *study*)

Bases: builtins.object

Extracts feature information for selected features.

The information returned consists of value lists, number of occurrences, and an summary spreadsheet.

Upon creation, re-initializes the laf processor with requested features plus some needed features.

Args:

study: A dictionary directing the feature study. Contains:

- a list of features to be studied. It is a list of feature names
- a set of *absence values*, i.e. values like none or unknown that somehow count as the absence of a value.
- VALUE_THRESHOLD: a parameter that indicates how many distinct values to list in the summary.

feature_doc()

Create the feature information.

Based on the study information given at the creation of the FeatureDoc object, a set of files is created.

•A tab separated overview of statistical feature/value information.

•For each feature, a file with its values and number of occurrences.

•A file of node types and the features they carry.

CHAPTER 11

emdros2laf 4.5.3

11.1 Submodules

11.2 emdros2laf.settings module

class `Settings`

Bases: `builtins.object`

Stores configuration information from the main configuration file and the command line.

Defines an extra function in order to get the items in a section as a dictionary, without getting the DEFAULT items as wel

`annotation_skip` = {'self'}

`flag`(*name*)

`laf_switches` = {'comment_local_deps'}

11.3 emdros2laf.etcbc module

class `Etcbc`(*settings*)

Bases: `builtins.object`

Knows the ETCBC data format.

All ETCBC knowledge is stored in a file that describes objects, features and values. These are many items, and we divide them in parts and subparts. We have a parts for monads, sections and linguistic objects. When we generate LAF files, they may become unwieldy in size. That is why we also divide parts in subparts. Parts correspond to sets of objects and their features. Subparts correspond to subsets of objects and or subsets of features. N.B. It is "either or": either

- a part consists of only one object type, and the subparts divide the features of that object type

or

- a part consists of multiple object types, and the subparts divide the object types of that part. If an object type belongs to a subpart, all its features belong to that subpart too.

In our case, the part 'monad' has the single object type, and its features are divided over subparts. The part 'lingo' has object types sentence, sentence_atom, clause, clause_atom, phrase, phrase_atom, subphrase, word. Its subparts are a partition of these object types in several subsets. The part 'section' does not have subparts. Note that an object type may occur in multiple parts: consider 'word'. However, 'word' in part 'monad' has all non-relational word features, but 'word' in part 'lingo' has only relational features, i.e.features that relate words to other objects.

The Etcbc object stores the complete information found in the Etcbc config file in a bunch of data structures, and defines accessor functions for it.

The feature information is stored in the following dictionaries:

(Ia) part_info[part][subpart][object_type] = set of feature_names NB: object_types may occur in multiple parts.

(Ib) part_object[part] = set of object_types

(Ic) part_feature[part][object_type] = set of feature_names

(Id) object_subpart[part][object_type] = subpart

Stores the subpart in which each object type occurs, per part

2.object_info[object_type] = [attributes]

Stores the information on objects, except their features and values.

3.feature_info[object_type][feature_name] = [attributes]

Stores the information on features, except their values.

4.value_info[object_type][feature_name][feature_value] = [attributes]

Stores the feature value information

22.reference_feature[feature_name] = True | False

Stores the names of features that reference other object. The feature 'self' is an example. But we skip this feature. 'self' will get the value False, other features, such as mother and parents get True

6.annotation_files[part][subpart] = (ftype, medium, location, requires, annotations, is_region)

Stores information of the files that are generated as the resulting LAF resource

The files are organized by part and subpart. Header files and primary data files are in part ''. Other files may or may not contain annotations. If not, they only contain regions. Then is_region is True.

ftype the file identifier to be used in header files

medium text or xml

location the last part of the file name. All file names can be obtained by appending location after the absolute path followed by a common prefix.

requires the identifier of a file that is required by the current file

annotations the annotation labels to be declared for this file

The feature information file contains lines with tab-delimited fields (only the starred ones are used):
0* 1* 2* 3* 4* 5* 6 7* 8 9 10 11* 12* object_type, feature_name, defined_on, etcbc_type, feature_value, isocat_key, isocat_id, isocat_name, isocat_type, isocat_def, note, part, subpart 0 1 2 3 4 5 6 7 8

Initialization is: reading the excel sheet with feature information.

The sheet should be in the form of a tab-delimited text file.

There are columns with:

ETCBC information: object_type, feature_name, also_defined_on, type, value.

ISOcat information key, id, name, type, definition, note

LAF sectioning part, subpart

See the list of columns above.

So the file gives essential information to map objects/features/values to ISOcat data categories. It indicates how the LAF output can be chunked in parts and subparts.

check_raw_files (*part*)

feature_atts (*object_type*, *feature_name*)

feature_info = {}

feature_list (*object_type*)

feature_list_subpart (*part*, *subpart*, *object_type*)

is_ref_skip (*feature_name*)

list_ref_noskip ()

make_mql (*name*, *query*)

make_query_file (*part*)

mql (*query*)

object_atts (*object_type*)

object_info = {}

object_list (*part*, *subpart*)

object_list_part (*part*)

object_subpart = defaultdict(<function Etcbc.<lambda> at 0x7f5b57532620>, {})

part_feature = defaultdict(<function Etcbc.<lambda> at 0x7f5b57532378>, {})

part_info = {}

part_list ()

part_object = defaultdict(<function Etcbc.<lambda> at 0x7f5b57532400>, {})

raw_file (*part*)

reference_feature = {}

run_mql (*query_file*, *result_file*)

settings = None

subpart_list (*part*)

the_subpart (*part*, *object_type*)

value_atts (*object_type*, *feature_name*, *feature_value*)

value_info = {}

value_list (*object_type*, *feature_name*)

11.4 emdros2laf.laf module

class Laf (*settings*, *et*, *val*)

Bases: builtins.object

Knows the LAF data format.

All LAF knowledge is stored in template files together with sections in the main configuration file. The LAF class finds those templates, sets up the result files, and fills them.

Note: Templates

template[key] = text where key is an entry in the *laf_templates* section of the main config file.

Note: Files and Filetypes

annotation_files[part][subpart] = (ftype, medium, location, requires, annotations, is_region)

The order is important, so we generate a list too:

file_order list of ftypes according *file_types section* in main config file, expanded, in the order encountered

where

ftype comes from the file_types section in the main config file. It has the shape of LAF file identifier, but with wild cards.

f.xxxxxx not an annotation file, but primary data or a header file

f_part.subpart annotation file for part, subpart

for each ftype there is an infostring consisting of fields

location file name of corresponding file, modulo a common prefix

medium file type (text or xml)

annotations space separated annotation labels occurring in this part, subpart

requires space separated list of ftypes of required files

is_region reveals whether the file only contains regions or not. A pure region file needs a different template.

Note: Header Generation

All header files are generated here: * the feature declaration file * the header for the resource as a whole * the header for the primary data file

The headers of the annotation files are included in those files. Those headers contain statistics: counts of the number of annotations with a given label. We know those number only after generation because these statistics will be collected during further processing.

When the annotation files are generated, we use placeholders for the statistics. In a post-generation stage we read/write the annotation files and replace the place holders by the true numbers. The files are written in situ. So we must take care that the placeholders contain enough space around them.

Note: Processing

This class provides methods to initialize and finalize the generation of primary data files and annotation files. There are methods to open/close all files that are relevant to the part that is being processed. (Part being: 'monad', 'section', 'lingo').

Note: Statistics

Counts are collected in a *stats* dictionary.

- stats[statistic_name] = statistic_value*

annotation_files = defaultdict(<function Laf.<lambda> at 0x7f5b573f7b70>, {})

et = None

file_handles = {}

file_order = []

finish_annot(*part*)

finish_primary()

gstats = defaultdict(<function Laf.<lambda> at 0x7f5b573f7a60>, {})

makefeatureheader()

makeheaders()

makeprimaryheader()

makeresourceheader()

primary_handle = None

report()

settings = None

start_annot(*part*)

start_primary()

stats = defaultdict(<function Laf.<lambda> at 0x7f5b573f7ae8>, {})

template = {}

11.5 emdros2laf.transform module

class Transform(*settings*, *et*, *lf*)
Bases: builtins.object

Transforms ETCBC data into a LAF resource

ETCBC knowledge comes from the Etcbc class LAF knowledge comes from the Laf class

read data from raw MQL export and build the annotations files For part monad there are extra things: * the primary data file will be built * one of the annotations files only contains regions, and no annotations

et = None

lf = None

process_lines(*part*)
Data transformation for part. Input: the lines of a raw emdros output file, which is processed line by line. Every line contains an object type, object identifier, monad indicator and list of features. This has to be translated to primary data and annotations.

Efficiency is very important. It will not do to call functions or follow long chains of dereferencing. Yet a lot has to happen. That is why this is a lengthy loop, and we maintain quite a lot of information from elsewhere in the program in loop-global variables. Not doing so might increase the running time 10-fold. Currently the complete programs runs within 15 minutes (inclusing generating raw data and validating) on an MacBook Air mid 2012.

settings = None

transform(*part*)

interval(*iv*)

makeuni(*match*)
Make proper unicode of a text that contains byte escape codes such as backslash xb6

primary_data(*text*, *trailer*)
Distil primary data from two features on the word objects. Apply necessary tweaks!

11.6 emdros2laf.validate module

class Validate(*settings*)
Bases: builtins.object

Validates all generated files, knows the schemas involved.

The main program generates a bunch of XML files, according to various schemas. They can be sent to this object, with or without a schema specification. All files with a schema specification will be validated.

The base locations of the schemas and of the generated files will be retrieved from the main configuration. All schemas will be copied from source to destination.

generated_files = list of [absolute_path, schema in destination, validation result]

Initialization is: get from config the schema locations and copy them all over

add (*xml*, *xsd*)
Add an item to the generated files list. If xsd is given, the file will eventually be validated.

The validation result will be stored in a member of the item, which is initially None. If validation takes place, None will be replaced by True or False, depending on whether the xml is valid wrt. the xsd.

generated_files = []

report ()
Print a list of all generated files and indicate validation outcomes

settings = None

validate ()
Validate all eligible files, but only if the validation flag is on

11.7 emdros2laf.run module

dotask (*part*)

final ()

init ()

processor ()

11.8 emdros2laf.mylib module

class Timestamp
Bases: `builtins.object`

elapsed ()

progress (*msg*)

timestamp = None

camel (*text*)

fillup (*size*, *val*, *lst*)

pretty (*data*)

run (*cmd*)

runx (*cmd*)

today ()

CHAPTER 12

Indices and tables

- *genindex*
- *modindex*
- *search*

e

emdros2laf.etcbc, 61
emdros2laf.laf, 63
emdros2laf.mylib, 66
emdros2laf.run, 66
emdros2laf.settings, 61
emdros2laf.transform, 65
emdros2laf.validate, 65
etcbc.annotating, 59
etcbc.featuredoc, 60
etcbc.preprocess, 59

l

laf.data, 54
laf.elements, 53
laf.fabric, 53
laf.lib, 58
laf.model, 55
laf.names, 55
laf.parse, 55
laf.settings, 57
laf.timestamp, 57

A

add() (Validate method), 66
add_logfile() (LafData method), 54
adjust_all() (LafData method), 54
aid (AnnotationHandler attribute), 55
annotation_files (Laf attribute), 64
annotation_skip (Settings attribute), 61
AnnotationHandler (class in laf.parse), 55
API() (LafAPI method), 53
apiname() (Names method), 56
APIprep() (LafAPI method), 53
arrayify() (in module laf.lib), 58

B

Bunch (class in laf.fabric), 53

C

camel() (in module emdros2laf.mylib), 66
characters() (AnnotationHandler method), 55
characters() (HeaderHandler method), 55
check_load_spec() (Names method), 56
check_raw_files() (Etcbc method), 63
comp() (Names method), 56
comp_file() (Names method), 56
compile_all() (LafData method), 54
connect_log() (Timestamp method), 57
Connection (class in laf.elements), 53
create_annots() (in module etcbc.annotating), 59

D

data() (PrimaryData method), 54
DCOMP_SEP (Names attribute), 56
decomp() (Names method), 56
decomp_full() (Names method), 56
deliver() (Names method), 56
dinfo() (Names method), 56
disconnect_log() (Timestamp method), 57
dmsg() (Names method), 56
Dmsg() (Timestamp method), 57
dotask() (in module emdros2laf.run), 66

E

e() (Connection method), 53
E_ANNOT_NON (Names attribute), 56
E_ANNOT_YES (Names attribute), 56
elapsed() (Timestamp method), 66
emdros2laf.etcbc (module), 61
emdros2laf.laf (module), 63
emdros2laf.mylib (module), 66
emdros2laf.run (module), 66
emdros2laf.settings (module), 61
emdros2laf.transform (module), 65
emdros2laf.validate (module), 65
Emsg() (Timestamp method), 57
endElement() (AnnotationHandler method), 55
endElement() (HeaderHandler method), 55
endnodes() (Connection method), 54
et (Laf attribute), 64
et (Transform attribute), 65
Etcbc (class in emdros2laf.etcbc), 61
etcbc.annotating (module), 59
etcbc.featuredoc (module), 60
etcbc.preprocess (module), 59

F

FabricError, 55
Feature (class in laf.elements), 54
feature_atts() (Etcbc method), 63
feature_doc() (FeatureDoc method), 60
feature_info (Etcbc attribute), 63
feature_list() (Etcbc method), 63
feature_list_subpart() (Etcbc method), 63
FeatureDoc (class in etcbc.featuredoc), 60
file_handles (Laf attribute), 64
file_name (AnnotationHandler attribute), 55
file_order (Laf attribute), 64
fill() (in module etcbc.preprocess), 59
fillup() (in module emdros2laf.mylib), 66
final() (in module emdros2laf.run), 66
finish_annot() (Laf method), 64
finish_primary() (Laf method), 64
finish_task() (LafData method), 54
flag() (Settings method), 61

G

generated_files (Validate attribute), 66
GenForm (class in etcbc.annotating), 59
get_all_features() (LafAPI method), 53
getmonads() (in module etcbc.preprocess), 59

grouper() (in module laf.lib), 58
gstats (Laf attribute), 64

H

HeaderHandler (class in laf.parse), 55

I

i() (XMLid method), 54
Imsg() (Timestamp method), 57
init() (in module emdros2laf.run), 66
init() (in module laf.parse), 55
interval() (in module emdros2laf.transform), 65
is_ref_skip() (Etcbc method), 63

K

kind_types (Names attribute), 56

L

Laf (class in emdros2laf.laf), 63
laf.data (module), 54
laf.elements (module), 53
laf.fabric (module), 53
laf.lib (module), 58
laf.model (module), 55
laf.names (module), 55
laf.parse (module), 55
laf.settings (module), 57
laf.timestamp (module), 57
laf_switches (Settings attribute), 61
LafAPI (class in laf.fabric), 53
LafData (class in laf.data), 54
LafFabric (class in laf.fabric), 53
lf (Transform attribute), 65
list_ref_noskip() (Etcbc method), 63
load() (LafFabric method), 53
load_again() (LafFabric method), 53
load_all() (LafData method), 54
load_spec_keys (Names attribute), 56
load_spec_subkeys (Names attribute), 57

M

make_annots() (GenForm method), 59
make_array_inverse() (in module laf.lib), 58
make_form() (GenForm method), 59
make_inverse() (in module laf.lib), 58
make_mql() (Etcbc method), 63
make_query_file() (Etcbc method), 63
makefeatureheader() (Laf method), 64
makeheaders() (Laf method), 64
makeprimaryheader() (Laf method), 65
makeresourceheader() (Laf method), 65
makeuni() (in module emdros2laf.transform), 65
model() (in module laf.model), 55
mql() (Etcbc method), 63

N

Names (class in laf.names), 55
nid (AnnotationHandler attribute), 55
Nmsg() (Timestamp method), 57
node_down() (in module etcbc.preprocess), 59
node_order() (in module etcbc.preprocess), 59
node_order_inv() (in module etcbc.preprocess), 59
node_ud() (in module etcbc.preprocess), 59
node_up() (in module etcbc.preprocess), 59
nonerep() (in module etcbc.annotating), 59
normalize_ranges() (in module laf.model), 55

O

object_atts() (Etcbc method), 63
object_info (Etcbc attribute), 63
object_list() (Etcbc method), 63
object_list_part() (Etcbc method), 63
object_subpart (Etcbc attribute), 63
orig_key() (Names method), 57

P

parse() (in module laf.parse), 55
part_feature (Etcbc attribute), 63
part_info (Etcbc attribute), 63
part_list() (Etcbc method), 63
part_object (Etcbc attribute), 63
prep_post() (in module etcbc.preprocess), 59
prepare_all() (LafData method), 54
prepare_dirs() (LafData method), 54
pretty() (in module emdros2laf.mylib), 66
primary_data() (in module emdros2laf.transform), 65
primary_handle (Laf attribute), 65
PrimaryData (class in laf.elements), 54
process_lines() (Transform method), 65
processor() (in module emdros2laf.run), 66
progress() (Timestamp method), 66

Q

query() (Names method), 57

R

r() (XMLid method), 54
raw_file() (Etcbc method), 63
raw_msg() (Timestamp method), 57
reference_feature (Etcbc attribute), 63
report() (Laf method), 65
report() (Validate method), 66
request_files() (Names method), 57
request_init() (Names method), 57
reset() (Timestamp method), 57
resolve_feature() (LafFabric method), 53
run() (in module emdros2laf.mylib), 66
run_mql() (Etcbc method), 63
runx() (in module emdros2laf.mylib), 66

S

s() (Feature method), 54
set_verbose() (Timestamp method), 57
setenv() (Settings method), 57

Settings (class in emdros2laf.settings), 61
Settings (class in laf.settings), 57
settings (Etcbc attribute), 63
settings (Laf attribute), 65
settings (Transform attribute), 65
settings (Validate attribute), 66
stamp (AnnotationHandler attribute), 55
start_annot() (Laf method), 65
start_primary() (Laf method), 65
startElement() (AnnotationHandler method), 55
startElement() (HeaderHandler method), 55
stats (Laf attribute), 65
subpart_list() (Etcbc method), 63

T

template (Laf attribute), 65
the_subpart() (Etcbc method), 63
Timestamp (class in emdros2laf.mylib), 66
Timestamp (class in laf.timestamp), 57
timestamp (Timestamp attribute), 66
today() (in module emdros2laf.mylib), 66
Transform (class in emdros2laf.transform), 65
transform() (Transform method), 65
truth (AnnotationHandler attribute), 55

U

unload_all() (LafData method), 55

V

v() (Connection method), 54
V() (Feature method), 54
v() (Feature method), 54
Validate (class in emdros2laf.validate), 65
validate() (Validate method), 66
value_atts() (Etcbc method), 63
value_info (Etcbc attribute), 63
value_list() (Etcbc method), 63
verbose_level (Timestamp attribute), 57
vv() (Connection method), 54

W

Wmsg() (Timestamp method), 57

X

XMLid (class in laf.elements), 54
Xmsg() (Timestamp method), 57