GraphAnno

An annotation and query tool
for graph-based linguistic annotations

Lennart Bierkandt
Friedrich-Schiller-Universität Jena
post@lennartbierkandt.de

Version of October 21, 2019

Contents

1 Introduction

GraphAnno started off as an elementary internal tool of the Linktype project[1], but over time, it grew, got more and more features and became more user-friendly.

It was developed out of the need for an annotation tool that allows for structural annotations not restricted to a specific framework, linguistic theory or tagset. First of all, we needed the possibility to build unrestricted graph structures – not only trees – and to have multiple and arbitrarily overlapping layers.

As interface for the tool's main task, the manual annotation, we chose the command line, which enables a fast annotation flow once you have become acquainted with the commands (an effort that quickly pays off).

When it comes to working with the data, GraphAnno offers a powerful query language for finding graph fragments and exporting data to CSV files. The latter still requires a small amount of programming skills but relieves you of writing export scripts from scratch.

We hope that you will find GraphAnno as useful as we do and enjoy working with it.

2 Requirements and installation

GraphAnno is a Ruby program and thus needs an installation of Ruby (version 2.0 or higher; you may install it with your package manager or from http://www.ruby-lang.

[1] *Towards a corpus-based typology of clause linkage* (http://linktype.iaa.uni-jena.de), funded by the *Deutsche Forschungsgemeinschaft* (http://dfg.de)

org/); you don't need Ruby on Windows as there is a precompiled version of GraphAnno for Windows. For GraphAnno's user interface you need a decent browser (GraphAnno has been developed with Firefox, but Chrome should work, too).

Installation:

1. Download the ZIP file https://github.com/LBierkandt/graph-anno/archive/master.zip and extract it to a directory of your choice (or clone the GraphAnno repository if you use Git).
2. Depending on your system:
   - On Windows: simply use the binary file `main.exe` located in GraphAnno's main directory, or follow the following instructions if you want to use the Ruby version.
   - On Linux or OS X: install the needed libraries (Rubygems):
     a) navigate to the GraphAnno main directory,
     b) run `gem install bundler` if you haven't installed Bundler already,
     c) run `bundle install` if you have installed compilation tools (this is usually the case on Linux systems) or `bundle install --without=compile` if you haven't. (In the latter case you won't be able to use the media playback feature.)

Starting GraphAnno:

1. start `main.rb` in the GraphAnno main directory by running `bundle exec ruby main.rb`; on Windows without Ruby, run `main.exe`,
2. navigate to this address in your browser: `http://localhost:4567/`

To stop the program, press ctrl + C in the terminal where it is running.

3 GraphAnno's data model

All data in GraphAnno is represented as a graph, i.e., a set of nodes and a set of directed edges that connect the nodes. Nodes and edges bear attributes in the form of key-value pairs, that serve mainly the linguistic annotation. Furthermore, there are multiple types of nodes and edges that fulfil different functions.

The structure of the GraphAnno data model is explained in Section 3.1, Section 3.2 describes the details of the serialization format that is used to store GraphAnno data. The former section is probably helpful for every user in order to grasp the concept of GraphAnno data representation and get to know some terminology, whereas the latter one is intended for users who want to work with the serialized data, e.g., create or process GraphAnno data with their own scripts.

3.1 Graph model

As already said, GraphAnno represents all data in a graph, consisting of different types of nodes and edges. The most important types – those that are also displayed as a graph to the user – are *token nodes*, *annotation nodes* and *annotation edges*. These elements are directly created by the user and represent the linguistic data and annotation. They may be annoted with *attributes* (see below).

The data in GraphAnno is partitioned into *sentences* (without any theoretical implication, i.e., just user-defined chunks of language data), which may be, in turn, recursively grouped into sections. Sentences are represented by *sentence nodes*. These are connected to their token nodes and annotation nodes via *sentence edges* (there are, however, also sentence-independent nodes). Sections are represented by *section nodes* and connected to the sections or sentences they contain via *section edges*. As sentences and sections are subject to certain constraints – e.g., they must be arranged hierarchically – they are created via special commands, that also take care of creating the required edges. Sentence and section nodes may also be annotated with attributes.

Finally, we have to mention the *order edges*, that are used program-internally for representing the linear order of sentence nodes and token nodes. The structure of a full GraphAnno graph with all types of elements is shown by means of a mini example corpus in Figure 1.

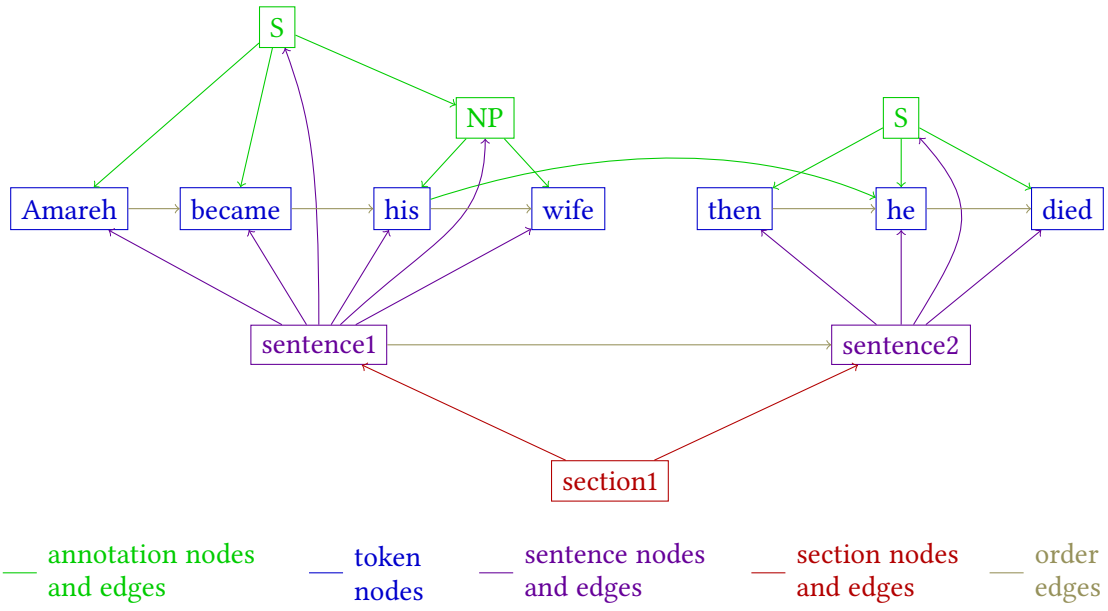

Figure 1: The GraphAnno data model – an example

As mentioned above, token nodes, annotation nodes and edges as well as sentence and section nodes may be annotated with *attributes* (I will also call them *annotations*). Attributes are key-value pairs, i.e., each element may bear a set of keys each of which has an associated value. The keys forming a set implies that each key may only appear

once on an element. Both keys and values are always strings. There are some privileged keys that bear a special meaning in GraphAnno: The text of token nodes is stored as value of the key `token`; the key `cat` on annotation nodes and edges is privileged insofar as its value is displayed on top of the element's annotations without the key; on sentence and section nodes the value of the key `name` is used as label for the elements in the navigation window and for referencing.

GraphAnno also provides the possibility to assign annotation nodes and edges to different layers (token nodes cannot belong to layers). Each node or edge may belong to one or more layers, which is represented by the element carrying a list of layers to which it belongs. This representation allows for layers to overlap in an arbitrary manner. Nodes and edges of different layers may be displayed in different colors and aligned in different ways (see 5.1 for how to configure the layers and their display).

3.2 Serialization format

For persisting, the GraphAnno data is serialized as multiple JSON[2] files: A GraphAnno corpus consists of one master file and at least one part file, each containing a JSON object. The details of the file contents are laid out in Section 3.2.2. How the graph itself (i.e., its nodes and edges) is serialized is explained in Section 3.2.1.

3.2.1 Data structure

The JSON objects represented by the GraphAnno corpus files all contain the keys `nodes` and `edges` with an array of nodes and an array of edges, respectively. This is shown in Figure 2. The other attributes they (may) contain are dealt with in Section 3.2.2.

```
{
  "nodes": <array>,
  "edges": <array>
}
```

Figure 2: A basic GraphAnno graph JSON object

Each node has the attributes `id` with a unique identifier (integer) and `type` for representing the type of the node (string). For the strings representing the different node and edge types see Table 1. Additionally, a node bears the attribute `attr`, an object that holds information on the node's layers and annotations. The keys of the `attr` object represent the node's layers (where the keys have to be taken from the layer shortcuts as defined in the `conf` attribute of the graph). The values are objects that represent the annotations on the respective layer. Nodes that don't belong to any layer (like token nodes, e.g.) bear their annotations under the key `""`.

[2] See `http://json.org/` for a description.

Edges have the same attributes as nodes plus the additional attributes `start` and `end` with the IDs of the edges's start and end node, respectively. Only annotation edges, however, may bear `attr`.

The structure of nodes and edges in JSON format can be seen in Figure 3 and 4, respectively.

Type	String
token node	`t`
annotation node	`a`
sentence node	`s`
section node	`p`
annotation edge	`a`
sentence edge	`s`
section edge	`p`
order edge	`o`

Table 1: Node and edge type strings used in JSON format

```
{
  "id": <integer>,
  "type": "t" | "a" | "s" | "p",
  "attr": {
    <layer shortcut>: {
      <annotation key (string)>: <annotation value (string)>,
      ...
    },
    ...
  }
}
```

Figure 3: A GraphAnno node JSON object

This structure could describe arbitrary directed graphs, but in order to work with GraphAnno, the graph has to meet a number well-formedness conditions. These include the permitted start and end node types for the different edge types listed in Table 2 and the following conditions:

- All sentence nodes must be linearily ordered via order edges.
- Every token node must be directly dominated by exactly one sentence node via a sentence edge.
- The token nodes of a sentence must be linearily ordered via order edges.

```
{
  "id":<integer>,
  "start":<integer>,
  "end":<integer>,
  "type": "a" | "s" | "p" | "o",
  "attr": {
    <layer shortcut>: {
      <annotation key (string)>: <annotation value (string)>,
      ...
    },
    ...
  }
}
```

Figure 4: A GraphAnno edge JSON object

Edge type	start/end	permitted node types
annotation	start	annotation, token
	end	annotation, token
sentence	start	sentence
	end	token, annotation
section	start	section
	end	section, sentence
order[a]	start	token, sentence
	end	token, sentence

[a] An order edge may only connect nodes of the same type.

Table 2: Permitted node types for start and end nodes of different edge types

- A sentence node or section node may be dominated by at most one section node.
- The sentence nodes directly dominated by a section node must be contiguous.
- The section nodes directly dominated by another section node must be contiguous in the sense that no other section node intervene in terms of the order defined by the dominated sentence nodes.

3.2.2 Corpus files

A GraphAnno corpus consists of a master file and one or several part files. This is handy when working with larger corpora, as it allows you to load only parts (i.e., single files) of a corpus. Usually, the master and part files will be located in one directory (other structures are possible but not useful in most cases).

The master file contains the configuration of the corpus and, optionally, independent nodes or edges, and the part files contain (consecutive) sentence nodes with their dominating section nodes and their associated token and annotation nodes, plus all connecting edges. Additionally, bot types of files contain references to each other.

The master file contains the version attribute, all configuration attributes of the corpus, attributes related to the corpus structure and nodes and edges as shown in Figure 5. The `version` attribute serves the correct handling of different formats that were produced by the different historic versions of GraphAnno. The subsequent (optional) attributes hold the different types of configuration, tagset, makros etc. `files` is an array of the relative paths of the part files (in the order their content has in the corpus; if the part files reside in the same directory, it will be the bare file names). The value for `max_node_id` and `max_edge_id` is the highest node ID or edge ID, respectively, of the whole corpus graph (to avoid mutliple assignment of IDs when not all corpus files are loaded). The master file also contains the corpus' speaker nodes and sentence-independent nodes, but no sentence or section nodes (i.e., a section cannot cross the boundaries of part files) or annotation and token nodes. As for the edges, it contains those ones that connect nodes of the master file itself; nodes that connect nodes of the master file and nodes of a part file belong to the part file.

```
{
  "version": <integer>,

  "info": <object>,
  "conf": <object>,
  "tagset": <array>,
  "anno_makros": <object>,
  "search_makros": <array>,
  "annotators": <array>,
  "file_settings": <object>,

  "files": <array>,
  "max_node_id": <integer>,
  "max_edge_id": <integer>,

  "nodes": <array>,
  "edges": <array>
}
```

Figure 5: Structure of the master file

A part file contains the version attribute, the attribute `master`, and nodes and edges as shown in Figure 6. The `version` attribute, as in the master file, serves the correct handling of the format. The value of `master` is the relative path of the master file (i.e.,

the bare file name if the master file is located in the same directory). The nodes are a consecutive number of sentence nodes, with their dominating section nodes as well as their associated tokens and annotation nodes. The edges comprise all edges that connect the nodes of the part file, plus those edges that connect nodes of the part file to nodes of the master file. It is not possible to have edges between nodes of different part files.

```
{
  "version": <integer>,
  "master": <string>,
  "nodes": <array>,
  "edges": <array>
}
```

Figure 6: Structure of a part file

See Section 6.1 for how to handle loading and saving of multi-file corpora.

4 User interface

The main elements of GraphAnno's user interface are the area where the annotation graph is displayed and, below, the command line, that is used for most operations (esp. for annotation; see Section 4.1). Further elements on the bottom of the screen are a dropdown for annotation layer choice and the display of, i.a., the loaded file. For some operations like, e.g., searching your annotations there are dedicated windows; these are discussed in detail in Section 4.2. Finally, you can use keyboard shortcuts for some operations like, e.g., toggling windows, navigating or zooming. These shortcuts are listed in Section 4.3.

4.1 Command line

The annotation work in GraphAnno (as well as workspace-related operations like loading and saving files) is operated from the command line. Most commands are executed immediately; some (like, e.g., the configuration commands) open a dialogue window. For issuing a command, you just type it, followed by its arguments, into the command line field and press the enter key. For a complete list of the available commands see Section 6.

There is also an autocomplete feature for the command line, that may speed up typing your commands. This feature can be activated and customized in the preferences (see 5.7). It suggests commands and, depending on the current command, file names, sentence names, annotations, makros etc. A list of suggestions appears as soon as you have typed the first letter; then you choose the desired item with the up/down keys and

select it with the tab key (attention: the enter key does not work for this, but submits the command line); you can close the suggestion list with the esc key.

4.2 Windows

For some purposes GraphAnno has dedicated windows. These windows can be toggled via keys (see the list in 4.3 or the following sections) or the buttons in the top left corner of the screen (this button bar may also be hidden, see 5.7). The windows can also be dragged around and resized with the mouse. Their position, size and display status (shown or hidden) is saved in the browser (in a cookie) or, optionally, in the corpus file (see 5.6) so you have the windows positioned as needed when you open your corpus. The different windows are described in the following.

4.2.1 Navigation

The navigation window shows the sentences and sections of your corpus and is toggled with the F9 key. The sentences are listed from top to bottom and labelled with their `name` attribute and their text. To the left of the sentences you can see the sections (again, labelled with their `name` attribute) as boxes whose height indicates which sentences/sections they span. The active sentence(s) or section(s) are highlighted.

In order to navigate to a sentence or section, you can doubleclick the corresponding element. For choosing multiple sentences or sections, you can use the ctrl and shift keys and select the elements with single clicks. With the enter key you confirm your selection (or cancel with esc). You may also use the up/down arrow keys for navigating (and selecting) inside the navigation window. Note that you may only choose elements of the same level (i.e., the same column in the window).

Instead of clicking into the navigation window, you can jump into it using alt + the up/down arrow keys (the window will be shown if it has been hidden before). The currently chosen sentences/sections will be selected and you may proceed as described in the previous paragraph.

For a quicker navigation from sentence to sentence (without explicitly using the navigation window), you may want to use the keyboard shortcuts alt + [left, right, pos1, end] as listed in 4.3; moreover, you can use the command `s` from the command line (see 6.1.6).

4.2.2 Filter

GraphAnno's filter function allows you to hide or filter a set of nodes and edges of the current sentence in order to obtain a better overview of what is relevant to your current annotation task. You may, e.g., decide to display only those elements that belong to a certain annotation layer.

In the filter window, toggled with the F6 key, you choose a set of elements via an attribute description of GraphAnno's query language as described in Section 7.1.2. By

clicking one of the respective buttons, you may then choose to either hide/filter the described set or hide/filter the rest. *Hide* here means that hidden elements are displayed in a non-prominent color (you may customize this color in the respective settings, see 5.1); *filter* means that the filtered elements are not displayed at all. In order to show all elements in the usual way again, click the *display all* button.

4.2.3 Search

GraphAnno's search feature is controlled from the corresponding window, that is toggled with the F7 key. The search window has a single input area for search, annotation and export queries. Query parts relating to all of these three functions can be mixed in the input area. The queries are formulated in GraphAnno's query language, which is explained in Section 7. As soon as you have entered your query, you can start the desired action (search, annotation or export) by clicking the corresponding button.

When you want to search for a graph fragment, enter your search query in the input area and click the *search* button. The search will then be executed and, as soon as it is finished, the number of matches will be displayed on the bottom of the search window. The matching fragments are highlighted in the graph view (default is red, but the color can be customized, see 5.1), and in the navigation window all sentences and sections that contain matches are highlighted as well. If your query is not correct or if another error occurs during the search, an error message is displayed on the bottom of the search window. If you don't want the matches to be highlighted anymore, just click the *clear search* button (this will also internally clear the search results, so you have to search again before annotation – see below).

In order to jump to the next or previous sentence/section with a match, you may use the shortcuts alt + n or alt + p, respectively. If your sentences/sections are long and contain multiple matches, the shortcuts alt + shift + n and alt + shift + p may come in handy for centering the next/previous match in the current view.

When you want to annotate using an annotation query, you enter a search query and an annotation query. The search query finds a set of matches and saves the relevant elements of the match in variables (or *IDs*). In your annotation query you write commands that refer to these variables. In order to perform the annotation, you must first execute the search and afterwards the annotation. If you make changes in your search query and want them to take effect for your annotation, you must, again, first execute the modified search.

For exporting a data table, you proceed analogously to the annotation procedure described in the preceding paragraph: Enter a search query and export commands that refer to the IDs from the search query; then execute the search first and the export afterwards.

4.2.4 Log

GraphAnno records the change history of an annotation session (or of the whole corpus; you can configure this in the file settings, see 5.6). The commands recorded include those commands that create or delete nodes or edges, as well as creating and deleting of sentences and sections, but not work space-related commands such as `append` or `clear`.

You can see this change history in the log window (toggled with F8), where the command executed, the time it was issued and, if you use the multi-annotator feature (see 5.5), the annotator are displayed. You can jump to points in history by clicking the corresponding line in the log window. But beware! the change history is linear, i.e., if you issue an annotation command while being in a point in history, the commands following this point will be lost.

You can also navigate the history using the `undo` and `redo` commands – see Section 6.3.12.

4.2.5 Independent nodes

Sentence-independent nodes are only displayed in the graph view if directly connected to a sentence-bound node (otherwise, the view could become crowded very quickly). For viewing the existing independent nodes and their element references (needed for referencing them in annotation commands) you have the independent nodes window, that is toggled with the F10 key. This window shows a list of all sentence-independent nodes with their element reference (composed of the letter `i` and a number) and their annotations.

Neither the independent nodes window nor the regular sentence view shows connections between independent nodes; in order to see these you may have a look at the independent nodes graph view (command `s i`, cf. also 6.1.6).

For performance reasons, you may set the maximum number of nodes that are displayed in this window in the program preferences – see 5.7.

4.2.6 Media

For corpora with an associated media file, the media window offers an audio/video player. The window is toggled with Ctrl + F10. You can operate the player with the mouse, but you can also play specific chunks of the media (i.e. corresponding to a given span of tokens) using the `play` command (see 6.1.7).

4.3 Keyboard shortcuts

Most of the functions related to navigation and display can be controlled by keybord shortcuts. The following is a table of the available shortcuts:

Shortcut	Function
Navigation	
Alt + ←/→	previous/next sentence or section
Alt + Home/End	first/last sentence or section
Alt + p/n	previous/next sentence or section with match
Alt + ↑/↓	jump into navigation window
Graph	
Ctrl + Shift + −/+	scale down/up graph
Ctrl + Shift + 0	zoom to original size (fitting height)
Ctrl + Shift + arrows	move graph
Ctrl + Shift + Home/End	go to left/right edge of graph
Ctrl + Shift + Page up/Page down	go to upper/lower edge of graph
Alt + Shift + p/n	previous/next match in current view
F4	toggle element references
Window	
F1	show/hide help window
F2	show/hide text and sentence annotations
F6	show/hide filter window
F7	show/hide search window
F8	show/hide change log window
F9	show/hide navigation window
F10	show/hide independent nodes window
Ctrl + F10	show/hide media window

5 Configuration

There are several aspects of your corpus you may configure. The configuration is done in windows you open with a command from the command line. The available configurations are presented in the following sections.

5.1 Layers and visualization

The window for configuration of the layers of the currently loaded graph and its visualization is opened with the commmand line command `config`.

In the section *general settings*, you can configure the settings for nodes and edges that do not belong to any layer.[3] *Default color* applies to all nodes and edges that are

[3]Depending on the browser you use, the color fields are displayed as color picker or as simple text field. In the latter case, you have to provide the color as hexadecimal RGB value: A hash (`#`) followed by three two-digit hexadecimal numbers for red, green and blue, respectively. `#000000`,

not tokens and that do not belong to any layer, *token color* applies to tokens, *found color* is used for the highlighting of nodes and edges found in a search, and *filtered color* for elements that are filtered out by the filter function. The setting for *edge weight* affects the layout of the displayed graph and shows its effect only when edges with different weight are present (details will follow in the next paragraph). Finally, the setting *edge label compatibility mode* enables a different way of edge label placing. In most cases, this leads to a poorer graph layout (more collisions), but it can improve the layout in some cases (some styles of dependency annotations).

In the section *layers*, you can configure the layers of the graph. The *name* is an arbitrary label for the layer, that will be shown in the dropdown field for the layer selection. The *shortcut* is an identifier that is used for representing the layers of elements internally (cf. Section 3.2), for setting the layers of elements (cf. Section 6.3.1 or 6.3.4) and for matching elements of specific layers in searches; this shortcut may only consist of alphanumeric characters and underscores and must not have the same structure as element references (i.e., `t`, `n`, `i`, `e` or `s` followed by a number, or `m`; cf. Section 6.3.3). *Color* means the color that is used for displaying the elements of the layer; *edge weight* is the weight of the layer's edges. The higher this value (integer values), the shorter the rendering algorithm will try to make the edges. When you create two layers, one with a high edge weight and one with a low edge weight, the graph will be rendered such that the graph of the first layer is as compact as possible; the elements of the second layer will be placed in a way that they distort the first layer only to a low degree. If you enter *0* as edge weight, the edges will not enforce a hierachy between the nodes of that layer (otherwise, the start node of an edge is always placed higher than the end node). If a negative value is entered, all nodes of that layer are displayed horizontally on one tier.

The section *layer combinations* contains the settings for elements that belong to multiple layers. Via the checkboxes under *layers* you determine, to which layers an element has to belong in order to be subject to the definition of the combination in question. The other settings work like those described for the *layers*, and for those elements that belong to the layer combination, they override the values given for the single layers. Note that elements can only be assigned to layer combinations that are defined here.

Note that changes in the layers window do not affect existing nodes and edges! That means, e.g., that if you change the shortcut of a layer, the elements formerly belonging to this layer don't belong to it anymore (but now have a non-meaningful shortcut in their layers list).

e.g., stands for black, `#ffffff` for white, `#ff0000` for light red etc.

5.2 Tagset (permitted annotations)

In the window opened by the command `tagset`, you can specify which annotations are permitted for the nodes and edges of the graph, i.e., you can define the graph-specific tagset.

If there are no tagset rules defined, all annotations are possible. But if you have defined at least one tagset rule, only those annotations are allowed that are explicitly permitted by one of your rules, and illicit annotation attempts will be answered with an error message. Already existing annotations, however, are not affected by changing the rules. As a special case, the `token` annotation on token nodes – i.e., changing the token text – is always possible.

Tagset rules consist of four components: context, layer, key and values.

The *context* restricts the scope of a rule; i.e., the rule only applies to those elements that match its context definition. The context definition is basically an *element description* of the query language (cf. Section 7.1.1), but only a subset may be used for context definitions: You may only use the keywords `token`, `node`, `edge`, `i` and the layer shortcuts defined in the layer configuration (cf. Section 5.1), and combine them with logical operators (`!`, `&`, `|` and parentheses). If you leave the context input field empty, the rule applies to all elements.

In the *layer* field you may add the shortcut for a layer or layer combination. This will restrict the annotations that are permitted by the rule to the given layer(s). To reduce confusion: layers you mention in the context field refer to the element's layers, the layer(s) in the layer field refer to the layer(s) of the annotation.

The *key* is the annotation key that will be permitted by the rule. Write the key without quotes, even if it contains special characters. If you leave this input field empty, all annotations are allowed for the given context.

The *values* restrict the permitted annotation values for the given key. Enter the values separated by spaces. The notation of the single values is subject to the same rules as in the query language (cf. Section 7.1.1, p. 31): Simple values are entered without further markup; if a value contains special characters (see p. 31) it has to be enclosed in double quotes (`"..."`). Additionally – just like in the query language (p. 31) – you may use regular expressions, which are enclosed in slashes (`/.../`). In this case the regular expressions are anchored, i.e. an annotation value has to match the whole regular expression in order to be permitted. If you leave this input field empty, all values are allowed for the given key (and context).

5.3 Annotation and search makros

With the command `makros` you open a window where you may define makros. You can define annotation makros, that facilitate the annotation with frequently needed attribute combinations, as well as search makros that you can use in graph searches for frequently used attribute combinations or connection descriptions (cf. Section 7.1.9).

For the annotation makros, you enter the shortcut for your makro (it has to consist of alphanumeric characters including the underscore, and it should not have the form of an element reference or be identical to a layer shortcut, cf. Section 5.1) in the fields on the left; in the corresponding field on the right you enter the desired annotations for your makro. For these annotations you must use the same syntax as in the annotation commands, i.e. a set of attributes in the form `key:value`, separated by spaces (see Section 6.3.1 for details).

For the seach makros, you enter the name (again, a string of alphanumeric characters including the underscore) in the left field and a predefined graph-specific search makros as described in Section 7.1.9 in the right field. These makros are then available for search queries in the graph.

5.4 Metadata

Additionally to the configuration of layers, visualization, makros and permitted annotations, you may save metadata for a graph as key-value pairs. The command `metadata` opens the window, where you can enter an arbitrary number of keys with a text as corresponding value.

5.5 Annotators

For the case of multiple annotators working with the same corpus, GraphAnno offers the possibility of creating annotator-specific annotations. With this feature, the same elements can be annotated by differend annotators, which enables you, e.g., to assess the inter-annotator agreement.

In order to use the multi-annotator feature, you first have to create annotators; the command `annotators` opens the window for this task. In this window you can create annotators, bearing a unique name (prefer short names without spaces, as these will be used for loggin in) and information about the annotator (free text). If, later on, you delete annotators, all their annotations will be deleted as well.

If you annotate without having logged in as an annotator, your annotations will be stored as public annotations (as is the case when working without multiple annotators). For annotating as a specific annotator, you first have to log in using the command `annotator` (or the synonymous command `user`), followed by the annotator's unique name. The current annotator's name will now be displayed in the corresponding field (next to the input line on the bottom of the user interface), and you will see only your own annotations; the annotations of other annotators or public annotations will be hidden. Also for searching only your own annotations will be evaluated. If you want to log out the current annotator and work on the public annotations again, use the `annotator` command without argument.

5.6 File settings

The command `file` opens a window, in which you can specify some settings for the files in which the corpus will be saved:

- *compact file format*: Should the file be stored in a compact JSON format? This has the advantage of reducing the file size, but the file will be less readable for potential inspection or editing in a text editor.
- *save editing history*: Should the the change log (cf. 6.3.12) be saved, so that it stays available accross settings? The change log will be saved in a file called `log.json` in the corpus directory. (Attention, log files can become very big and lead to high saving/loading times.)
- *save window positions*: Should the position of the windows (filter, search etc.) be saved in the graph file (i.e., project-specific)? (Otherwise, the positions are only saved in the browser.)

Additionally, the window displays a list of the corpus' part files, each with a (collapsible) list of the sentences that are (or will be) saved in it. Here you may also rename the part files. Keep in mind that the file names must be unique and cannot be `master.json` or `log.json`. The real renaming won't take place until the corpus is saved.

5.7 Program preferences

For settings that aren't related to a corpus but to the program, there is the preferences window, opened by the command `pref`. These preferences are saved in a local file. There are the following settings:

- *Menu*: You can enable or disable the window button bar (cf. 4.2)
- *Autocomplete*: You may activate the autocomplete feature (cf. 4.1) and then choose which kinds of entities (commands, file names, sentence names, annotations, makros etc.) you want to get suggestions for. Note that the autocomplete feature is useless if you don't choose things that should be suggested.
- *Autosave*: If you enable this option, your current corpus file is saved at intervals of a length you may specify (in minutes). This option has no effect when your current workspace is not related to a file (you see this from the indication next to the input line).
- *Independent nodes window*: Here you may set a maximum number up to which sentence-independent nodes are displayed in the independent nodes window (cf. 4.2.5). This setting serves to preserve performance in big corpora: the list of independent nodes is rebuilt on each command, this may slow down the reactivity of the program when your copurs contains many sentence-independent nodes. To prevent this, adjust the number to your needs (or set it to 0 if you don't need the list at all).

6 Command line commands

6.1 Data and navigation

6.1.1 Load file: `load`

With the command `load` you load a corpus (or a corpus' part file) into the work space. By default, the corpus will be loaded from the `data` directory located in the GraphAnno main directory; but you can also provide a path (directories are separated by the normal slash `/` on all operating systems; the path has to be enclosed in double quotes (`"..."`) if it contains spaces). If the path does not start with a slash, it will be interpreted relative to the `data` directory; if it starts with a slash, it will be interpreted as an absolute path (in the partition GraphAnno is installed in if you are working on a Windows systems).

If you want to load a whole corpus, you may either provide the path to the master file (with or without the extension `.json`) or the path to the corpus directory (ending in `/`), then GraphAnno will look for the file `master.json` in that directory. If you want to load only one file of a multi-file corpus, then provide the path to that file (you can add other parts of the corpus using the command `add`).

Before loading, the work space is cleared from all data. So, changes that were not saved explicitly (using the command `save`) are lost. The name of the master file of the corpus is shown next to the input line on the bottom of the user interface.

6.1.2 Add part file of multi-file corpus: `add`

The command `add` adds to the work space a part file of a multi-file corpus that is already partially loaded. Provide the path of the part file as described for `load`. You will see an error message if you try to load a file that is not part of the loaded corpus or if the file has been loaded already.

6.1.3 Append file: `append`

The command `append` appends the contents of the given corpus or file (specify the file as described for `load`) to the work space. When you save the work space, the appended files will be appended as files to your corpus (if there are duplicate file names, they will be suffixed).

6.1.4 Save file: `save`

The command `save` saves the work space to a GraphAnno corpus. If there is a corpus name indicated next to the input line, the work space can be saved to this corpus without specifying a path. Note that only the loaded parts and the master file will be rewritten (the other parts remain unchanged).

If you specify a path (see the command `load` for details concerning the path), `save` will act like the *save as* function of other programs and save a new corpus. The path

you provide will be turned into a directory, and in this directory GraphAnno will save the corpus. If you had created a corpus from scratch, GraphAnno will create the master file `master.json` and one part file `0001.json`. If you save a loaded corpus, GraphAnno will save only the loaded parts (preserving their file names) and the master file.

Attention: no warning is issued when an existing file will be overwritten.

6.1.5 Clear work space: `clear`

The command `clear` clears the work space from all data. Changes that were not saved are lost. Next to the input line no corpus will be indicated anymore.

6.1.6 Go to sentence/section: `s`

In order to navigate to a sentence or section (or multiple of them), you may use (alternatively to keyboard shortcuts or the navigation window) the command `s`, followed by the name(s) of the sentence(s) or section(s) you want to change to.

There is one special use of this command: when typing `i` instead of a sentence name, you are directed to the graph view of sentence-independent nodes. This view shows all sentence-independent nodes of your corpus, together with the edges existing between them. In order to go back to the regular sentence view, just use the navigation key commands (cf. 4.3), the navigation window (cf. 4.2.1) or the command `s` with a sentence name.

6.1.7 Play associated media: `play`

If you are working with a corpus that has associated audio or video, the command `play` serves for playing specific chunks of the media. The command without any parameter plays the whole current view; when you provide one token reference, it play from the given token to the end of the current view; if you provide two tokens, it plays from the first to the second one. The media window where you can see the video and operate the player manually opens with Ctrl + F10.

6.1.8 Save image: `image`

With the command `image`, you can export the graphic that GraphAnno is currently displaying. As argument you must specify the desired format (`svg` or `png`). The file will then be downloaded.

6.1.9 Import text: `import text`

You can import texts with the command `import text`. After issuing the command, a window opens where you can enter the text and set the preferences for its processing. The work space will be cleared before the text is imported (but not as soon as the

window opens). Changes that hadn't been saved are lost and no file is indicated next to the input line anymore.

In the import window you can choose between two methods of entering your text: You can upload a text file or you can paste it in the text area. For the processing of the text there are two methods available, too. For unedited text you may use the method *punkt segmenter*. This method uses an automatic segmenter to split the text into sentences and tokens. In order to process abbreviation etc. correctly, you need to specify the language of the text.

The second processing method is *regular expressions*; this method is made for pre-formatted texts. First, you have to enter a string that will be used for segmenting the sentences. The preset is `\n` [4] for a file in which every sentence starts on a new line. The second string you have to enter a a regular expression that matches the tokens. The preset here is `(\S+)`. This stands for a sequence of non-spaces, so all words that are separated by spaces are matched as tokens. The purpose of the parentheses is to save the matched string in the variable `$1`, so it can be used in the next field. The next field is for an annotation command (see 6.3.3) for the tokens, that uses the string matched by the regular expression in the preceding field. The preset here is `token:$1`. That means that the string matched as token is used for the annotation of the token text. Another example would be a text tagged for parts of speech, where the part of speech is appended to every word with an underscore. In this case you would enter the regular expression `(\S+)_(\S+)` and the annotation command `token:$1 pos:$2`. The regular expression in this case finds two strings of non-spaces that are joined by an underscore; the strings are saved to the variables `$1` (the word) and `$2` (the POS tag). In the annotation command these variables are used to annotate the token text and the `pos` attribute.

6.1.10 Import Toolbox data: `import toolbox`

Toolbox files can be imported using the command `import toolbox`. This command opens a window in which you can choose the file to be imported and enter the format description. The format description has to be in JSON format and consists of a list of a list of markers. The lists are sorted according to their levels – the highest (*record* level) first – and contain the markers that belong to the respective level (markers are entered without backslash). The first marker of the first level (`ref` in the example below) will be used as record ID. The marker whose line is to be used as token text is preceded by an asterisk. Elements that lie below the token level will be joined and integrated into their respective tokens.

A format description for a toolbox file with three levels (record, word, morpheme) could look like this, e.g.:

```
[["ref", "eng"], ["*gw"], ["mph", "ge", "ps"]]
```

[4] `\n` stands for a line break, `\t` for a tab.

Like with the command `load`, the work space will be cleared when importing. Changes that haven't been saved will be lost; next to the input line no file will be indicated anymore.

6.1.11 Export and import configuration: `export` and `import`

The command `export` serves also for exporting graph configurations, that can be imported in other graphs using the command `import`. As first argument you enter the type of the configuration to be exported or imported: `config` for layers and visualization configuration (see 5.1), `tagset` for permitted annotations / tagset (see 5.2). The second argument is the filename for the configuration file to be saved or to be imported (without file extension). The exported file will be saved in a subdirectory – named like the configuration type – of the `exports` directory. Attention: when importing, the existing configuration will be replaced completely.

6.1.12 Edit configurations: `config`, `tagset`, `makros`, `metadata`, `annotators` and `file`

These commands open the windows for the settings that were described in Section 5. `config` for layers and visualization, `tagset` for permitted annotations / tagset, `makros` for annotation makros, `metadata` for metadata, `annotators` for annotators and `file` for file settings.

6.2 Sections and sentences

The following commands create, delete or modify sentence and section nodes. All of these command may take names as parameters. These names may be given as bare strings or quoted strings (for details see Section 7.1.1).

Where you are referencing existing sections/sentences, the names may match more than one section/sentence as the names are not unique by design (it is strongly recommended to keep them unique, though). In such cases all sections/sentences with the given names are considered. Keep in mind that the referenced sections/sentences must always be one the same level.

You may also reference multiple sections/sentences at once, either by using regular expressions or sequences. Regular expressions are given in slashes (`/.../`) and find all sections/sentences whose names match the expression (for further information regarding regular expressions see 7.1.1).

For referencing sequences of sections/sentences, provide the names of the first and the last section/sentence, joined with to dots, e.g., `sent-01..sent-13` (this may not work as expected if your names are not unique). The different methods of referencing section/sentence references may be combined in one command.

6.2.1 Create new sentence: `ns`

With the command `ns` – followed by one or more sentence names – you can create one or more new sentences. The command creates sentence nodes with the corresponding `name` attribute and inserts them after the current sentence (or the last of the current sentences), of which also the possible affiliation to a section is inherited. Afterwards you are directed to the first one of the new sentences.

6.2.2 Create section: `s-new`

The command `s-new` creates a new section. When you don't provide any section or sentence names with the command, the new section will comprise the current sentences or sections. When you provide section or sentence names (the sentences/sections must be contiguous and on the same level), these will be grouped instead of the current ones. You may alyso provide annotations for the new section (see 6.3.1 for how these have to look like); it is advisable to specify at least the `name` attribute.

6.2.3 Remove section: `s-rem`

The command `s-rem` is the reverse of `s-new`: it removes sections (without touching their descendants). You cannot remove all sections from their parent section, and you cannot remove sections from the middle of their parent section. Provide the names of the sections to be removed together with the command.

6.2.4 Add to section: `s-add`

With the command `s-add` you can add sections/sentences to an existing section. Provide the name of the parent section first and then the names of the sections/sentences to be added to the parent section. The sections/sentences to be added must be contiguous and one level below the parent section.

6.2.5 Detach section: `s-det`

The command `s-det` is the reverse of `s-add`: it detaches sections/sentences from their parent section. Restrictions and parameters of the command are the same as for `s-rem`.

6.2.6 Delete section: `s-del`

The command `s-del` deletes the current sentences or sections including their descendants and all text and annotation contained. If you provide section or sentence names as arguments, these sections/sentences will be deleted instead of the current ones.

6.3 Annotation

GraphAnno's annotation commands are designed to be entered quickly, so their syntax is rather compact: they consist of a short command (often one letter only) and are followed by parameters seperated by spaces.

6.3.1 New node: `n`

The command for creating a new node is `n`, followed by the attributes the new node should have and optionally the layer(s) it should belong to.

The attributes can be either simple key-value pairs in the format `key:value` – which will add the annotation to all levels of the new node – or a combination of layer shortcut, key and value in the format `layer:key:value` – which will add the annotation only for the given layer (or layer combination) of the new node.

Key and value can be given either as bare string (if it doesn't contain any of the special characters used in the annotation language: `␣:"#`)[5]) or as string in double quotes (`"..."`), that may contain any character (double quotes themselves have to be excaped with a backslash: `"...\"..."`). The layer shortcut is always a bare string.

You can also use a shortcut from previously defined annotation makros (see Section 5.3). When you additionally enter attributes with keys present in the makro, these override the annotations defined in the makro.

Additionally, you can specify the layer or layer combination to which the new node is to belong (if you don't, it will belong to the layer/combination set in the layer dropdown field). For this purpose you use the shortcuts defined in the layer configuration (cf. Section 5.1). The use of these shortcuts also has the effect of a switch, insofar as it sets the layer/combination for the following operations. If you want to create a node without layer affiliation, you have to set the current layer to *none* via the command `l` (see 6.3.4) or the layer dropdown.

Per default, the new node will belong to the first of the current sentences. You may, however, provide in your command a node to whose sentence the new node should belong instead. If you want to create a sentence-independent node, use the keyword `i` in your command.

Command `n` in modified BNF:

```
command_n            =   "n " n_parameters
n_parameters         =   n_parameters " " n_parameters
                         attributes
                         node_reference
                         layer_shortcut
                         "i"
attributes           =   attributes " " attributes
```

[5] The symbol `␣` stands for the space.

```
                        attribute
                        annotation_shortcut
attribute           =   (layer_shortcut ":")? key string
key                 =   string ":"
string              =   character_not_special+
                        """ character* """
alnum               =   letter | digit | "_"
annotation_shortcut =   alnum+
layer_shortcut      =   alnum+
node_reference      =   anno_node_reference
                        token_node_reference
token_node_reference =  "t" number
anno_node_reference =   dependent_node_ref | independent_node_ref
dependent_node_ref  =   "n" number
independent_node_ref =  "i" number
```

6.3.2 New edge: `e`

The command for creating a new edge is `e`, followed by start and end node of the edge to be created and the attributes it is to bear (you may give nodes and attributes in arbitrary order; if you accidently provide more than two nodes, the first two nodes will be considered). Like with `n`, you may also specify the layer(s) of the new edge.

Command `e` in modified BNF:

```
command_e       =   "e " e_parameters
e_parameters    =   e_parameters " " e_parameters
                    node_reference
                    attributes
                    layer_shortcut
```

6.3.3 Annotate: `a`

The command for annotating elements is `a`, followed by the elements to be annotated and the attributes with which they are to be annotated (all given elements will be annotated with all given attributes – if the tagset allows it, see below). The order of elements and attributes is free.

You can also annotate sequences of elements of the same type (i.e., `n`, `i`, `t` or `e`) by entering the first and the last element joined by two dots. E.g., when you enter `t3..t7`, all tokens from `t3` to `t7` will be annotated (you may enter the sequence also in the inverse order, i.e., `t7..t3`).

If the given annotations are not allowed for one of the given elements by the corpus' tagset (cf. 5.2) you will receive a warning, but the allowed annotations will be applied nevertheless. If the tagset restricts the permitted layer(s) for an annotation and you don't specify the layer, the annotation will only be added for the permitted layer(s).

The command `a` serves also for deleting annotations. In order to do so, enter the key (or layer key combination) to be deleted with a colon, but without value. You can add and delete annotation in the same command.

The command `a` also serves for annotating sentences and sections. The sentence and section nodes are referenced like `s0`, `s1` etc., where `s0` stands for the current sentence(s), `s1` for the parent section(s) of the current sentence(s), `s2` for the parent(s) of the parent section(s) and so on.

Command `a` in modified BNF:

```
command_a              =   "a " a_parameters
a_parameters           =   a_parameters " " a_parameters
                           element_reference
                           section_node_reference
                           attributes
                           key
element_reference      =   node_reference
                           edge_reference
                           element_sequence
edge_reference         =   "e" number
section_node_reference =   "s" number
element_sequence       =   node_sequence
                           edge_sequence
node_sequence          =   anno_node_sequence
                           token_node_sequence
anno_node_sequence     =   anno_node_reference ".." anno_node_reference
token_node_sequence    =   token_node_reference ".." token_node_reference
edge_sequence          =   edge_reference ".." edge_reference
```

6.3.4 Set layer: `l`

The command `l` sets the layer or layer combination of the given elements and switches to the specified layer/combination for the following operations. As arguments, specify the elements to be affected and the layer shortcut as defined in the layer configuration (cf. Section 5.1). If you don't provide a layer shortcut, the elements are set to belonging to no layer at all, and the current layer is set to *none*.

For layers the given elements are removed from, the annotations will be removed as well. For layers the given elements are added to, there won't be any annotations

initially.

You may also use the command as a simple layer switch (as an alternative to the select field) by giving only the layer shortcut (or none) without any elements.

Command `l` in modified BNF:

```
command_l     =   "l " l_parameters?
l_parameters  =   l_parameters " " l_parameters
                  element_reference
                  layer_shortcut
```

6.3.5 Delete elements: `d`

Elements are deleted with the command `d`, followed by the elements to be deleted. If you delete nodes, the outgoing and ingoing edges are deleted as well. If you delete a token node from the middle of a sentence, the adjacent tokens are joined automatically.

Command `d` in modified BNF:

```
command_d       =   "d " d_parameters
d_parameters    =   d_parameters " " d_parameters
                    element_reference
```

6.3.6 Group nodes under new parent node: `g` or `p`

The grouping command `g` or `p` creates a new parent node for the given nodes. I.e., the command creates a new node and edges that connect the new nodes to the nodes to be grouped. The parameters of this command are the nodes to be grouped and the attributes the newly created node is to bear. The newly created node will belong to the sentence of the node given first in your command (or will be independent if you use `i` in your command); apart from that, the order of nodes and attributes is irrelevant. Like with the command `n`, a layer can be specified.

Command `g`/`p` in modified BNF:

```
command_g           =   ("g " | "p ") g_parameters
g_parameters        =   g_parameters " " g_parameters
                        node_reference
                        node_sequence
                        attributes
                        layer_shortcut
                        "i"
```

6.3.7 Append child node: `h` or `c`

The command `h`/`c` works analogously to the command `g`/`p`, but instead of a parent node a new common child node is created.

Command `h`/`c` in modified BNF:

```
command_h           =   ("h " | "c ") h_parameters
h_parameters        =   h_parameters " " h_parameters
                        node_reference
                        node_sequence
                        attributes
                        layer_shortcut
                        "i"
```

6.3.8 Insert node into edge: `ni`

The command `ni` (*node insert*) allows you to insert a new node into an existing edge. As parameters you specify the edge and the attributes for the new node. A new node will be created, and the given edge will be replaced by two edges with the same annotations, that connect the start node of the original edge with the new node and the new node with the end node of the original edge.

If you specify more than one edge, a new node will be inserted into each of them. The new node will belong to the sentence of the edge's end node (or none of you add the keyword `i`).

Command `ni` in modified BNF:

```
command_ni          =   "ni " ni_parameters
ni_parameters       =   ni_parameters " " ni_parameters
                        edge_reference
                        attributes
                        layer_shortcut
                        "i"
```

6.3.9 Delete node but preserve connections: `di` and `do`

If you want to delete a node but preserve the connections between parent node(s) and child node(s) of the deleted node, you can use the commands `di` or `do`. These commands delete the specified node and connect each child node to each mother node (this makes sense particularly in a tree-like structure where there is one parent node and many child nodes). `di` (*delete ingoing*) deletes the ingoing edge(s), `do` (*delete outgoing*) deletes the outgoing edge(s).

Commands `di` and `do` in modified BNF:

```
command_di_do       =   "di " anno_node_reference+
                        "do " anno_node_reference+
```

6.3.10 Attach to/detach from sentence: `sa` and `sd`

The command `sa`, followed by one or more independent node references, attaches these nodes to the current sentence (or the first of the current sentences). The command `sd`, in turn, detaches the given nodes from their sentence and makes them independent nodes.

Commands `sa` and `sd` in modified BNF:

```
command_sa          =   "sa " independent_node_ref+
command_sd          =   "sd " dependent_node_ref+
```

6.3.11 Tokenize: `t`, `tb`, `ta`

For creating tokens there are the commands `t`, `tb` and `ta`. The arguments for these commands are a list of words, separated by spaces. These words are inserted as tokens into the current sentence. If the sentence already contains tokens, the command `t` appends the new ones. `tb` (*tokenize before*) and `ta` (*tokenize after*) take a token as their first argument and insert the new tokens before or after, respectively.

The words can be given as bare strings, or, if they contain control characters (`␣:"#`), as string in double quotes (`"..."`; double quotes inside the string have to be escape with a backslash: `\"`).

Commands `t`, `tb`, `ta` in modified BNF:

```
befehl_t        =   "t " words
words           =   words " " words
                    word
word            =   non-control-character+
                    """ character* """

command_tb      =   "tb " token_reference " " words
command_ta      =   "ta " token_reference " " words
```

6.3.12 Undo/Redo: `undo` and `redo` (or `z` and `y`)

You can undo previously issued annotation commands using the command `undo` (or the synonymous command `z`). These annotation commands include those commands that create or delete nodes or edges, as well as creating and deleting of sentences, but not work space-related commands such as `append` or `clear` and not annotation by query (cf. 7.2). A command you have undone can be redone with the command `redo` (or the synonymous command `y`). Undoing and redoing can be used multiple times; but if you issue an annotation command after having undone other commands, the undone commands cannot be redone (linear change history).

You can look at the change history in the log window that is toggled with the F8 key (cf. Section 4.2.4). In this window you can also directly jump to points in the history with a click.

6.3.13 Log in annotator: `annotator` (or `user`)

If you use GraphAnno's multi-annotator feature, you log in as a specific annotator with the command `annotator` (or the synonymous command `user`), followed by the annotator's unique name (as defined in the corresponding window, see 5.5). The logged-in annotator will be shown in the field next to the input line and only the annotator's annotations will be displayed and searched. If you want to log out the current annotator and work on the public annotations again, use the `annotator` command without argument.

7 Query language

7.1 Search

Searching graphs in GraphAnno works by formulating a query that describes a graph fragment and is composed of a set of clauses. The search then finds all subgraphs of the corpus graph that match this description. The clauses that can be used in the description are the `node`, `nodes`, `edge`, `link`, `text`, `meta`, `cond` and `def` clauses. Of these, a query must at least contain one `node` clause or `text` clause, or an unconnected `edge` clause. The various types of clauses will be explained in the following sections.

In a query the clauses are each given on one line in an arbitrary order; indentations and blank lines have no impact on the semantics. Comments may be added as whole lines or after a line; they are preceded by a hash symbol.

Of the following sections, the first one explains how a to-be-matched element is described, which is needed for most of the clause types, the subsequent sections present the available clause types in detail.

7.1.1 Element description

An *element description* is the part of a query clause that defines which properties an element (node or edge) must have in order to be matched by a query. Such an element description defines which conditions the attributes and layers of an element should match and, additionally, it may specify conditions on ingoing and outgoing edges (for nodes) as well as conditions on start and end nodes (for edges).

The description of the attributes of the elements to be matched is composed of key-value pairs in the form `key:value` or triples of layer shortcut, key and value in the form `layer:key:value`. These attribute descriptions are joined by the logical operators `!` for *not*, `&` for *and* and `|` for *or* (precedence: `!` > `&` > `|`) as well as (round) parentheses. As a shortcut for disjunctions of attribute descriptions with the same key (and layer), you may use an attribute description of the form `key:value1|value2|...|valueN` (or `layer:key:value1|value2|...|valueN`, respectively).

Attribute descriptions with a layer shortcut match elements that bear the given key-value annotation on the given layer(s); attribute descriptions without a layer shortcut match only elements that bear the given annotation on *all* their layers.

The key of an attribute description can be given as a bare string if it does not contain any of the control characters of the query language (`␣():!&"/?+*{}@#^`), else it has to be given in double quotes (`"xyz"`) and can then contain any character (double quotes have to be escaped with a backslash: `\"`). The layer shortcut must always be a bare string.

The values of the attribute descriptions are strings, that can be given in three different formats. The first format is a simple, unmarked string that may contain all characters except the control characters of the query language (`␣():!&"/?+*{}@#^`). These strings will be matched case-insensitively. The second format is a string in double quotes (`"xyz"`). These strings may contain any character (double quotes have to be escaped with a backslash: `\"`) and will be matched exactly. The third format is a regular expression. A regular expression has to be written in slashes (`/x.z/`) and has to conform to Ruby's regular expression syntax (cf. `http://www.ruby-doc.org/core/Regexp.html`). The regular expressions are not anchored; i.e., for anchoring a regular expression to the start or end of a string you have to use `^` or `$`, respectively. You can find an arbitrary string using an empty regular expression (i.e. `//`).

If you omit a value string, those elements are matched that bear no annotation at all for the given key (and layer). So, e.g., `tns:` matches all elements without a tns annotation, `s:tns:` matches all elements without a tns annotation on the s layer, `!tns:` matches all elements with a tns annotation (on any layer), `tns:|prs` matches all elements with the tns annotation prs or no tns annotation (on all layers).

For matching the layer of an element, you use the shortcut defined for that layer (cf. Section 5.1). Layer shortcuts can be combined using logical operators just like attribute

descriptions. So, if you want to find elements with a specific layer combination you may of course use the respective layer shortcuts joined by `&`, but also the shortcut for that combination.

Token nodes are matched by using the keyword `token`.

For nodes, the element description may, in addition to attribute descriptions and layer shortcuts, contain criteria for ingoing and outgoing edges. These criteria are composed of the keyword `in` or `out`, an optional element description in parentheses and an optional quantifier. The operators `in` or `out` find all ingoing or outgoing edges, respectively, that match the given element description. The quantifier specifies how many edges matching the given description have to be present; its syntax is familiar from regular expressions: `{m,n}` means at least m edges, at most n edges; if you omit the first number, it has the meaning *zero*, omitting the second number means *infinite*. `{n}` means exactly n edges. Additionally there are the shortcuts `?` for `{0,1}`, `*` for `{0,}` and `+` for `{1,}`. What differs from quantifiers in usual regular expressions (and from quantifiers in other contexts of the GraphAnno query language), is that a missing quantifier will be interpreted as `{1,}`.

The operator `link` works in a similar fashion to `in` and `out`, but it does not find ingoing and outgoing edges but (potentially) more complex connections to other nodes. In Section 7.1.5 you may read how these connections may be specified. The rules for quantifiers used together with the `link` operator are the same as for `in` and `out`.

The operators `in`, `out` and `link` are meaningless if used on the description of an edge; they always match with a count of 1 when used on edges.

For edges, the element description may contain, additionally to attribute descriptions and layer shortcuts, properties of the start or end node, specified with the keyword `start` or `end`, respectively, followed by the node description in parentheses.

The operators `start` and `end` are meaningless in node descriptions; they always match when used on nodes.

The element description in modified BNF:

```
element_description = element_description " & " element_description
                      element_description " | " element_description
                      "!" element_description
                      "(" element_description ")"
                      attribute
                      layer_shortcut
                      edge_criterion
                      node_criterion
                      "token"
attribute           = (layer_shortcut ":")? key attribute_value ("|" attribute_value)*
```

```
layer_shortcut      = alnum+
key                 = string ":"
attribute_value     = char_except_control_char+
                      """ character* """
                      "/" regular_expression "/"
                      ""
string              = char_except_control_char+
                      """ character* """
edge_criterion      = "in" ("(" element_description")")? quantifier?
                      "out" ("(" element_description")")? quantifier?
                      "link" ("(" connection ")") quantifier?
quantifier          = "?" | "*" | "+" | "{" number? ("," number?) "}"
node_criterion      = "start" ("(" element_description ")")
                      "end" ("(" element_description ")")
```

7.1.2 `node`

The `node` clause describes a node that should occur once in the graph fragment. The clause is composed of the keyword `node`, an optional ID and an element description.

The ID consists of a `@` followed by a string that consists of alphanumeric characters, including the underscore, with the additional restriction that the first character must not be a digit. Using this ID, the node can be referenced in other parts of the query.

The `node` clause in modified BNF:

```
node_clause     =   "node" id? " " element_description
id              =   "@" (letter | "_") alphanumeric_char*
```

Examples:

- Search for all nodes that bear the category S or VP and that are no tokens:

  ```
  node cat:S|VP & !token
  ```

- Search for all nodes of category VP or tokens bearing the pos value verb:

  ```
  node cat:VP | token & pos:verb
  ```

- Search for all nodes of category S that have at least two outgoing AUX edges:

  ```
  node cat:S & out(cat:AUX){2,}
  ```

- Search for all nodes of category S that directly dominate at least one node bearing the pos value pro:

  ```
  node cat:S & out(end(pos:pro))
  ```

7.1.3 `nodes`

The `nodes` clause describes a set of nodes that should be present in the graph fragment. The set may end up being empty if it is only the target of a `link` clause or `edge` clause. The `nodes` clause has the same syntax as the `node` clause (except for the keyword, of course).

The `nodes` clause in modified BNF:

```
nodes_clause    =   "nodes" id? " " element_description
```

7.1.4 `edge`

The `edge` clause may be used in two different ways. First, you can use it for searching for single edges. Then the clause consists of the keyword `edge`, an optional ID (under which the edge can be references in the exporting function) and an attribute description for edges as explained above in 7.1.2. Second, you can use the `edge` clause for stating the there should be an edge with the specified properties between two nodes or sets of nodes (specified via `node` or `nodes`, respectively) of the graph fragment. When used in this way, you have to provide the IDs of start and end of the edge after the (optional) ID of the edge itself.

Conditioned by the optional ID and the various usages, the keyword `edge` my be followed by zero up to three IDs. The interpretation of these IDs follows from ther number and their order. One ID: ID of the edge itself; two IDs: start and end of the edge; three IDs: ID of the edge, IDs of start and end.

The `edge` clause in modified BNF:

```
edge_clause     =   "edge" id? (id id)? " " element_description
```

Examples:

- Search for all edges that indicate the syntactic function subject:

  ```
  edge synfunc:subj
  ```

- Search for all nodes of cateory S, each with the set of nodes of category NP that are linked via an edge of category S, A or P:

  ```
  node  @s    cat:S
  nodes @np   cat:NP
  edge  @s@np cat:S|A|P
  ```

7.1.5 `link`

A `link` clause specifies how two nodes or node sets of the graph fragment should be connected. The connection can be specified flexibly as a chain of nodes and edges using quantifiers and disjunctions similar to a regular expression. The `link` clause consists of the keyword `link`, the specification of the start and end nodes (via IDs as described for the `edge` clause above) and the description of the connection.

The description of the connection consists of a sequence of `edge` and `node` terms. These terms are composed of the keyword `edge` or `node` and an optional description of the element in parentheses. This description is an attribute description as explained for the `node` clause above. The element description may be followed by an ID, under which the found elements can be references in the exporting function (but not in the search query!).

For alternatives you may use the *or* operator `|`; its precedence is lower than that of the sequence. You may use parentheses for adjusting the precedence. You may also use quantifiers (as described under **??** on page 32) after elements or elements grouped in parentheses. These quantifiers are interpreted neither as greedy nor as non-greedy – all matching connections will be found and counted as separate matches.

Corresponding to the nature of a graph, a connection always consists of an alternating sequence of nodes and edges (starting and ending with an edge). When specifying a connection, however, it is not obligatory to provide an alternating sequence; only the first edge must not be omitted. If you provide two elements of the same type (i.e. edges or nodes) in succession, the search engine will accept any element of the other type in between. E.g., the connection description `edge(a:b) edge(c:d)` will match an edge bearing the attribute `a:b`, en unspecified node an then another edge, bearing the attribute `c:d`.

When you use `link` in a node attribute (e.g. in a `node` clause), do not provide start and end node. The start node then is the node that is searched for; the end node is the last node that is described in the connection description or, if the connection description ends with an edge, an unspecified node.

The `link` clause in modified BNF:

```
link_clause     =   "link" id id " " connection
connection      =   connection " " connection
                    connection "|" connection
                    "(" connection ")"
                    connection quantifier
                    "edge" ("(" element_description ")")? id?
                    "node" ("(" element_description ")")? id?
```

Examples:

– Search for all graph fragments that are composed of a node of category P and a node of cateogry S, where the former dominates the latter via an edge of category EX:

```
node @p cat:P
node @s cat:S
link @p@s edge(cat:EX)
```

– Search for a node of catogory S, all nodes of category NP that are dominated by the S node and all tokens that are dominated by these NP nodes:

```
node @s cat:S
nodes @np cat:NP
nodes @tok token
link @s@np edge+
link @np@tok edge+
```

– Search for a node of catogory S and all tokens that are dominated by the S node via an NP node (this yields the same graph fragments as the last example, but without assigning an ID to the NP nodes):

```
node @s cat:S
nodes @tok token
link @s@tok edge+ node(cat:NP) edge+
```

– Search for a node of catogory NP and the next dominating node of category S (i.e., without intervening S nodes):

```
node @np cat:NP
node @s cat:S
link @s@np edge node(!cat:S)*
```

7.1.6 `text`

With the `text` clause you can find a series of token nodes by specifying their text. Additionally you may specify other attributes the token nodes should bear. The `text` clause is composed of the keyword `text`, an optional ID and the description of a text fragment.

The description of the text fragment consists of a sequence of word descriptions that describe the token text and an optional attribute description in parentheses for each token. For specifying the token text there are three posiibilities as is described in 7.1.2 for the values of key-value pairs: bare strings, quoted strings and regular expressions; the attribute descriptions follow the same rules as in 7.1.2, too.

For the description of the text fragment you may use the operator `|` for *or* (precedence lower than sequence) as well as parentheses and quantifiers (these are non-greedy in the text search). Additionally you can append an ID to a subgroup of your text frag-

ment that enables referencing these nodes in other clauses. Quantifiers and IDs have a higher precence than sequence and disjunction; the order of quantifier and ID after the same subgroup does not matter. The optional ID after the keyword `text` captures all nodes that are found in by the clause. The fragment may be anchored to the start or end of a sentence by `^s`; it is not possible to search for text accross sentences.

The `text` clause in modified BNF:

```
text_clause     =   "text" id? " " "^s"? text_fragment "^s"?
text_fragment   =   text_fragment " " text_fragment
                    text_fragment "|" text_fragment
                    "(" text_fragment ")"
                    text_fragment quantor
                    text_fragment id
                    word
word            =   attribute_value ("(" element_description")")?
```

Examples:

- Search for all sentences that contain the verb “permit” in third position:

  ```
  text ^s //{2,2} permit(pos:v)
  ```

- Search for two occurrences of “he” that are separated by an arbitrary number of words, where the first “he” and all following words before the second “he” are dominated by an S node:

  ```
  node @s cat:S
  text (he //*)@t he
  link @s@t edge+
  ```

7.1.7 `meta`

The `meta` clause restricts the set of sentences to be searched. In this clause you can declare properties the sentence or its containing sections should have. Where there are conflicting annotations in the section hierarchy, the respective attribute of the lower section node is considered.

The clause is coposed of the keyword `meta` and an attribute description (like an element description, but only referring to annotations, not to ingoing or outgoing edges).

The `meta` clause in modified BNF:

```
meta_clause     =   "meta " attributes
attributes      =   attributes" & " attributes
                    attributes" | " attributes
                    "!" attributes
                    "(" attributes ")"
                    attribute
```

7.1.8 `cond`

With the `cond` clause you can specify conditions the graph fragment has to fulfil. It works as a filter on the set of graph fragments found based on the rest of the query. The clause is composed of the keyword `cond` and the condition in Ruby code. The nodes and node sets are referenced with the IDs used in the other clauses of the query. When using IDs, note that the nodes and edges found by `node` and `edge` are single elements, whereas those found by `nodes`, `text` and `link` are arrays of elements.

In the condition, you may access the attributes of elements using square brackets like this: `['layer_shortcut', 'key']` (where `layer_shortcut` mustn't be a layer combination) or this: `['key']`. The first variant gives you the value for the given key on the given layer; the second variant gives you the value for the given key if it has the same value on all layers or `nil` if it hasn't. For the attributes `token` and `cat` you can use as a shortcut the accessor methods `.token` and `.cat`. The layer(s) of an element are accessed via the method `.layers`; this method returns an array of the shortcuts of the layers the element belongs to. Via the method `.sentence` you can access the sentence node the element belongs to. When using attribute values, bear in mind that these are always strings; you may cast them to numbers with `.to_i` or `.to_f` for integers or floating point numbers, respectively.

Examples:

- Search for two S nodes that are linked via an ad edge and bear the same value for tns:

  ```
  node @s1 cat:S
  node @s2 cat:S
  link @s1@s2 cat:ad
  cond @s1['tns'] == @s2['tns']
  ```

- Search for all S nodes that dominate at least three tokens:

  ```
  node @s cat:S
  nodes @tok token
  link @s@tok edge+
  cond @tok.length >= 3
  ```

7.1.9 `def`

With `def` you can define makros for use in your query. After the keyword `def` you specify a name for the makro and an attribute description or connection description that will be filled in for your makro. The attribute or connection description is formulated as described in 7.1.1 or 7.1.5, respectively.

Bear in mind that a makro will be interpreted as it was in parentheses, i.e. it will be evaluated first. E.g., if you define a makro `cat:S | cat:VP` and combine it with the condition `tns:prs` using the operator `&`, this will not be evaluated as `cat:S | cat:VP & tns:prs` with the `&` preceding the `|` but as `(cat:S | cat:VP) & tns:prs`.

The makro definition `def` in modified BNF:

```
makro_definition =   "def " name " " makro
makro            =   element_description
                     connection
name             =   alnum+
```

Examples:

- Search for two nodes with `cat:S` or `cat:VP` that are linked via one edge and where the second one is in present tense:

  ```
  def svp cat:S | cat:VP
  node @s1 svp
  node @s2 svp & tns:prs
  edge @s1@s2
  ```

7.2 Annotation

After performing a search you can automatically annotate elements of the found graph fragments using the IDs used in your search query. For this purpose you can use the annotation commands described in 6.3 (except for `t`).

Because of the other using context you have take account of some changes:

- Instead of the node and edge references in the form of `n7`, `t8` or `e13` you use the IDs defined in your search query (in the form of `@s1` or the like).
- There is no annotation layer set. You should use a layer switch in your first annotation command (if needed).
- When using the command `n`, it is obligatory to specify a node that determines the sentence the new node will belong to.

- As the IDs can stand for single elements as well as for sets of elements, you have to note some particularities: For the command `e` you have to give exactly two IDs; if these are single element IDs, a single edge will be created; if they are one single element IDs and one set ID or two set IDs, an edge will be created for each of the node combinations from the two IDs. For the tokenizing commands `tb` and `ta` the first or last token of the set, respectively, will be used if you give a set ID. In all other commands you can give an arbitrary number of set IDs or single element IDs. All given elements then will be treated as *one* set of elements.
- When specifying attributes, you can use the elements found in your search. For this purpose, write the value string in double quotes and interpolate a part of the string in Ruby code (in which you can use your IDs) wrapped in braces with a prepended hash sign (as you also do in plain Ruby). As an example, a search/annotation combination that annotates all nodes of category S with their dominated text (terminated with a full stop) could look like this:

```
node @s cat:S
a @s text:"#{@s.text}."
```

7.3 Data export

The GraphAnno query language also gives you the possiblity to export search results as CSV files. With the clauses `col` and `sort`, described in the following, you can specify which information about the graph fragments found in your search should be exported and in which form. The data of each found graph fragment will be written in one line of the CSV file; with `sort` you can sort the matches and with `col` you specify which columns with which values are to be created. There will always be a first column with a consecutive numbering of the matches.

7.3.1 `col`

Each `col` clause stands for a column in the CSV data to be exported. Its first parameter is the column title (which must not contain spaces); it is followed by Ruby code that yields the value to emit. Nodes and edges are referenced using the IDs defined in your search query. Again, bear in mind that the nodes and edges found by `node` and `edge` are single elements, whereas those found by `nodes`, `text` and `link` are arrays of elements.

You can access the attributes and layers as described in 7.1.8 for `cond`. There are also other useful methods for exporting data: E.g., `.tokens` returns a list of the dominated tokens (you may also give a more specific link description string as argument of the method); `.text` works in the same way but returns the token text as a string. `.sentence_tokens` returns a list of all token nodes of the sentence the element belongs to, `.sentence_text`, again, their text.

When working with the multi-annotator feature (cf. 5.5), the direct access to the attributes (using square brackets) will return the annotations of the current annotator. If you want to access the annotations of other annotators, use the method `.private_attr`, which takes as argument the name of the annotator, e.g., `@s.private_attr('thomas')['cat']`). For the public annotations, use the method `.public_attr` (e.g., `@s.public_attr['cat']`).

7.3.2 `sort`

The `sort` clause can be used for sorting the matches. When you use multiple `sort` clauses, clauses specified later will only be evaluated when the clauses specified further up do not yield an order.

The `sort` clause uses Ruby code which yields the value that will be compared for sorting the matches. Like with `col`, the nodes and edges are references via the IDs defined in your search query. Again, bear in mind that attribute values are always strings and have to be casted to numbers (`.to_i` or `.to_f`) when you expect them to be compared as numbers.

Examples:

- Sort the matches by sentence name or, if they belong to the same sentence, token ID (the method `.tokenid` returns the position of the token in its sentence, starting with 0). Given by the search query: the ID `@t1` referring to a token:

```
sort @t1.sentence.name
sort @t1.tokenid
```