GraphAnno

An annotation and query tool
for graph-based linguistic annotations

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

Version of May 17, 2016

Contents

1 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 1.1, Section 1.2 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.

1.1 Graph model

As already said, GraphAnno represents all data in a graph, consisting of differend 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*. 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.

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

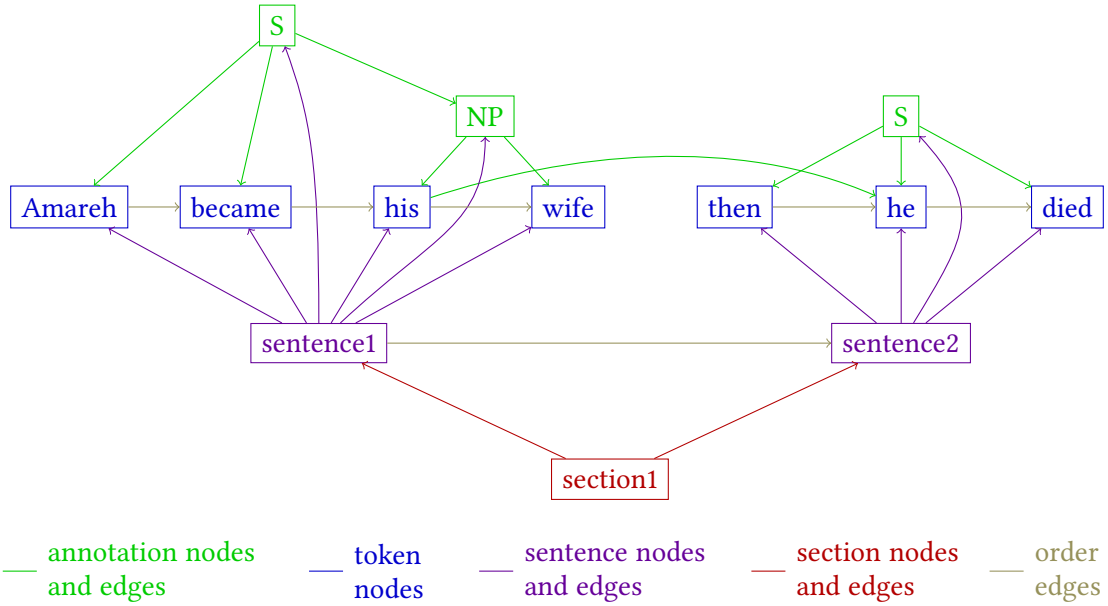

Figure 1: The GraphAnno data model – an example

in the navigation window and for referencing.

GraphAnno also provides the possibility to assign annotation nodes and edges to different layers. The fact that a node or edge is affiliated to a certain layer is represented by that element bearing an annotation with the key corresponding to the layer and the value `t` (for *true*). This representation allows for elements to belong to more than one layer, with the consequence that layers my overlap in an arbitrary manner. Nodes and edges of different layers may be displayed in different colors and aligned in different ways (see 3.1 for how to configure the layers and their diplay).

1.2 Serialization format

For persisting, the GraphAnno data is serialized as a JSON[1] object. This object has three obligatory keys: `nodes` and `edges` with an array of nodes and an array of edges, respectively, and `version` with the format version number (integer). This is shown in Figure 2. Furthermore, the JSON object has a number of optional attributes for configuration.

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

Figure 2: A basic GraphAnno graph JSON object

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

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. Optionally, a node bears the attribute `attr` with its annotations (object). The `attr` object contains arbitrary keys with string values.

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, 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": <object>
}
```

Figure 3: A GraphAnno node JSON object

```
{
  "id":<integer>,
  "start":<integer>,
  "end":<integer>,
  "type": "a" | "s" | "p" | "o",
  "attr":<object>
}
```

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

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.
- Each 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.
- 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.

2 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 2.1). 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 2.2. Finally, you can use keyboard shortcuts for some operations like, e.g., toggling windows, navigating or zooming. These shortcuts are listed in Section 2.3.

2.1 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 4.

2.2 Windows

For some purposes GraphAnno has dedicated static windows. These windows can be toggled, dragged around and resized. Their position, size and display status (shown or hidden) is saved in the browser (in a cookie) or, optionally, in the corpus file (see 3.6) so you have the windows positioned as needed when you open your corpus. The different windows are described in the following.

2.2.1 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. When you want to choose multiple sentences or sections, you can select the elements with single clicks and confirm your selection with the enter key (or cancel with esc). For selecting you can also use the ctrl and shift keys and the up/down arrow keys. Note that you may only choose elements of the same level (i.e., the same column in the window).

For a quicker navigation from sentence to sentence, you may want to use the keyboard shortcuts listed in 2.3; moreover, you can use the command `s` from the command line (see 4.1.7).

2.2.2 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 5.1.1. 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 3.1); *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.

2.2.3 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 5. 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 3.1), and in the navigation window all sentences 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 the next paragraph).

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.

2.2.4 Log

GraphAnno records the change history of an annotation session (or of the whole corpus; you can configure this in the file settings, see 3.6). 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` and not annotation by query (cf. Section 2.2.3 and 5.2).

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 3.5), 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 4.2.10.

2.3 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
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
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

3 Configuration

3.1 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.[2] *Default color* applies to all nodes and edges that are 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).

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. *Attribute* is the attribute that will be set to `t` for elements that belong to the layer in question. The *shortcut* is an identifier that can be used for the annotation of elements (cf. 4.2.1 or 4.2.11); this shortcut may only consist of alphanumeric characters and underscores and must not have the same structure as element references (i.e., `t`, `n` or `e` followed by a number, or `m`; cf. 4.2.3). *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 heigher 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 *attributes* 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.

Under *search makros* you can enter predefined graph-specific search makros as described in Section 5.1.8. These makros are then available for search queries in the graph. In the search makros text area, the definitions (`def ...`) have to be enterd each on its own line.

3.2 Permitted annotations / tagset

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

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

tagset. Only the keys and values defined here are permitted for the annotation of elements, illicit input is answered with an error message. Already existing annotations are not affected by changes of the permitted keys and values.

If you do not specify any keys and values, all annotations are permitted. If you specify keys, only these keys are allowed for annotations. If you leave the value field for a key empty, then all values are permitted for this key. If you want to restrict the possible values for a key, you may enter these into the value field of the key in question, separated by spaces.

The notation of the values is subject to the same rules as in the query language (cf. Section 5.1.1, p. 22): Simple values are entered without further markup; if a value contains special characters (see p. 22) it has to be enclosed in double quotes (`"..."`). Additionally – just like in the query language (p. 22) – 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.

3.3 Annotation makros

For a graph you save annotation makros, that facilitate the annotation with frquently needed attribute combinations. With the command `makros` you open a window where you can define these makros. In the fields on the left 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 3.1); 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 4.2.1 for details).

3.4 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.

3.5 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.

3.6 File settings

The command `file` opens a window, in which you can specify the settings for the file in which the work space 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. 4.2.10) be saved, so that it stays available accross settings? (Attention, this option can lead to very big files and long 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.)

4 Command line commands

4.1 Data and navigation

4.1.1 Load file: `load`

With the command `load` you load a graph file into the work space. By default, the file 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). 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). Provide the file name with or without the extension `.json`; the path has to be enclosed in double quotes (`"..."`) if it contains spaces.

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 loaded file is shown next to the input line on the bottom of the user interface.

4.1.2 Append file: `append`

The command `append` appends the contents of the given file (specify the file as described for `load`) to the work space.

4.1.3 Save file: `save`

The command `save` saves the work space to a GraphAnno file in the `data` directory. The file name has to be entered in the same way as with the command `load`. If there is a filename indicated next to the input line, the work space can be saved to this file without specifying the file name. Attention: no warning is issued when an existing file will be overwritten.

4.1.4 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 file will be indicated anymore.

4.1.5 Create new sentence: `ns`

With the command `ns` – followed by one or more sentence names separated by spaces – you can create a new sentence. The command creates sentence nodes with the corresponding `name` attribute; afterwards you are directed to the first newly created sentence.

4.1.6 Delete sentence: `del`

The command `del` deletes the current sentence including the sentence node. If you enter one or more sentence names as arguments, it is not the current sentence that will be deleted but the sentences that bear one of the given names. A further possibility is to enter a regular expression (in slashes); in that case, all sentences whose names match the given regular expression will be deleted.

4.1.7 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.

4.1.8 Export graphics: `image`

With the command `image`, you can export the graphic that GraphAnno is displaying for the current sentence. The first argument is the desired format. All formats supported by Graphviz are available, e.g. `dot`, `eps`, `pdf`, `png` or `svg`. See under `http://www.graphviz.org/content/output-formats` for the complete list. The second argument is the name of the new image file (without extension; put the name in double quotes if it contains spaces). The image will be saved in GraphAnno's `images` directory.

4.1.9 Export corpus: `export`

Using the command `export`, you can export the contents of the work space as a corpus in another format. The first argument is the format (at present the only fully functional format is `sql` for the import in GraphInspect; the format `paula` is theoretically available, but it is heavily restricted with respect to layers), the second argument is the name of the corpus to be saved. For `paula` you can optionally specify the name of the corpus document to be created as third argument. The exported corpus will be saved in GraphAnno's `exports` directory.

4.1.10 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 preformatted texts. First, you have to enter a string that will be used for segmenting the sentences. The preset is `\n` [3] 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

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

next field is for an annotation command (see 4.2.3) 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.

4.1.11 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"]]
```

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.

4.1.12 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 3.1), `tagset` for permitted annotations / tagset (see 3.2). 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.

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

These commands open the windows for the settings that were described in Section 3. `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.

4.2 Annotation commands

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.

The commands require to be in a sentence, i.e., you have to create a sentence first (using the command `ns`, see 4.1.5) if the work space is empty.

4.2.1 New node: `n`

The command for creating a new node is `n`, followed by the attributes the new node is to bear as key-value pairs in the format `key:value`. Key and value can be given either as simple string (if it doesn't contain any of the special characters used in the annotation language: `␣:"#`)[4]) or as string in double quotes (`"..."`), that may contain any character (double quotes themselves have to be excaped with a backslash: `"...\"..."`).

You can also use the shortcut from your previously defined annotation makros (see Section 3.3). 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 to which the new node is to belong (if you don't, it will belong to the layer set in the layer dropdown field). For this purpose you use the shortcuts defined in the layer configuration (cf. Section 3.1). The use of these shortcuts also has the effect of a switch, insofar as it sets the layer for the following operations (like the command `l`, see 4.2.11).

Command `n` in modified BNF:

```
command_n           =   "n " attributes
attributes          =   attributes " " attributes
                        attribute
                        annotation_shortcut
                        layer_shortcut
attribute           =   key string
key                 =   string ":"
string              =   character_not_special+
                        """ character* """
```

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

```
alnum               =   letter | digit | "_"
annotation_shortcut =   alnum+
layer_shortcut      =   alnum+
```

4.2.2 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. Like with `n`, the specification of a layer is possible.

Command `e` in modified BNF:

```
command_e       =   "e " start_end " " start_end " " attributes
start_end       =   node_reference
                    token_reference
node_reference  =   "n" number
token_reference =   "t" number
```

4.2.3 Annotate: a

The command for annotation 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; also layer shortcuts can be used). The order of the elements and attributes is free. You can also annotate sequences of elements of the same type (i.e., `n`, `e` or `t`) 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`).

At the same time you can use the command `a` for deleting attributes. In order to do so, enter the key to be deleted with colon, but without value.

Command `a` in modified BNF:

```
command_a           =   "a " a_parameters
a_parameters        =   a_parameters " " a_parameters
                        element_reference
                        attributes
                        key
element_reference   =   node_reference
                        edge_reference
                        token_reference
                        meta_node_reference
                        element_sequence
```

```
edge_reference      =   "e" number
meta_node_reference =   "m"
element_sequence    =   node_sequence
                        token_sequence
                        edge_sequence
node_sequence       =   node_reference ".." node_reference
token_sequence      =   token_reference ".." token_reference
edge_sequence       =   edge_reference ".." edge_reference
```

4.2.4 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
```

4.2.5 Group nodes under new parent node: `g` oder `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 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
                        token_reference
                        node_sequence
                        token_sequence
                        attribute
                        layer_shortcut
```

4.2.6 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
                        token_reference
                        node_sequence
                        token_sequence
                        attribute
                        layer_shortcut
```

4.2.7 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

Command `ni` in modified BNF:

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

4.2.8 Delete node but preserve connections: `di` und `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 " node_reference+
                        "do " node_reference+
```

4.2.9 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
```

4.2.10 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. 5.2). 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 2.2.4). In this window you can also directly jump to points in the history with a click.

4.2.11 Set layer: `l`

As an alternative to the select field, you can set the layer used for the subsequently created elements with the command `l`. For the layers you use the shortcuts defined in the layer configuration (cf. 3.1).

Command `l` in modified BNF:

```
command_l   =   "l " layer_shortcut
```

4.2.12 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 3.5). 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.

5 Query language

5.1 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.

5.1.1 `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 attribute 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 attribute description is composed of key-value pairs of the form `key:valu`, that 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 key-value pairs with the same key, you may use a key-value pair description of the form `key:value1|value2|...|valueN`.

The key of a key-value pair can be given as a simple 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 values of the key-value pairs 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. `//`).

In the place of a key-value pair you may also use the keyword `token`, which matches token nodes.

In addition to key-value pairs, the attribute description may contain criteria for ingoing and outgoing edges. These criteria are composed of the keyword `in` or `out`, an optional attribute description in parentheses and an optional quantifier. The operator `in` or `out` find all ingoing or outgoing edges, respectively, that bear the given attributes. 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,}`.

For the description of the edges (that are given in parentheses) you may specify – additionally to the attribute descriptions – properties of the start or end node using the keyword `start` or `end`, respectively, followed by the node description in parentheses.

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 5.1.4 you can read how these connections may be specified. The rules for quantifiers explained above are the same as for `in` and `out`.

The `node` clause in modified BNF:

```
node_klausel    =   "node" id? " " node_attributes
id              =   "@" (letter | "_") alphanumeric_char*
node_attributes =   node_attributes " & " node_attributes
                    node_attributes " | " node_attributes
                    "!" node_attributes
                    "(" node_attributes ")"
                    attribute
                    edge_criterion
                    "token"
attribute       =   string ":" attribute_value ("|" attribute_value)*
string          =   char_except_control_char+
                    """ character* """
attribute_value =   char_except_control_char+
                    """ character* """
                    "/" regular_expression "/"
edge_criterion  =   "in" ("(" edge_attributes ")")? quantifier?
                    "out" ("(" edge_attributes ")")? quantifier?
                    "link" ("(" connection ")") quantifier?
quantifier      =   "?" | "*" | "+" | "{" number? ("," number?) "}"
edge_attributes =   edge_attributes " & " edge_attributes
                    edge_attributes " | " edge_attributes
                    "!" edge_attributes
                    "(" edge_attributes ")"
                    attribute
                    node_criterion
node_criterion  =   "start" ("(" knotenattribute ")")
                    "end" ("(" knotenattribute ")")
```

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))
  ```

5.1.2 `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? " " node_attributes
```

5.1.3 `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 5.1.1. 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)? " " edge_attributes
```

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
  ```

5.1.4 `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 22) 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" ("(" edge_attributes ")")? id?
                    "node" ("(" node_attributes ")")? 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):

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

5.1.5 `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 5.1.1 for the values of key-value pairs: bare strings, quoted strings and regular expressions; the attribute descriptions follow the same rules as in 5.1.1, 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 fragment 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_fragmentquantor
                    text_fragment id
                    word
word            =   attribute_value ("(" node_attributes")")?
```

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+
  ```

5.1.6 `meta`

The `meta` clause restricts the set of sentences to be searched. In this clause you can declare properties the sentence node should have. The clause is coposed of the keyword `meta` and an attribute description as described in 5.1.1 for `node`.

The `meta` clause in modified BNF:

```
text-clause     =   "meta " attributes
```

5.1.7 `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: `['key']`; for the attributes `token` and `cat` you can use as a shortcut the accessor

methods `.token` and `.cat`. 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 string; 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
```

5.1.8 `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 for `node` and `link` in 5.1.1 and 5.1.4, 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            =   edge_attributes
                     node_attributes
                     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
```

5.2 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 4.2 (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`, you have 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 IDs. 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}."
  ```

5.3 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.

5.3.1 `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 as described in 5.1.7 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. 3.5), 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']`).

5.3.2 `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
```