Graph Annotation and Search Tool

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

Version vom 25. Februar 2014

Inhalt

1 Graphformat

Ein Graph im Graph Annotation and Search Tool (GAST[1]) besteht aus einer Menge von Knoten und gerichteten Kanten. Knoten und Kanten haben Attribute in der Form von Schlüssel-Wert-Paaren, die sowohl der linguistischen Annotation als auch der Strukturierung und Darstellung des Graphen in GAST dienen. Kanten tragen zusätzlich ein `type`-Attribut.

Ein GAST-Graph ist in Sätze eingeteilt – Einheiten die der Strukturierung und Darstellung dienen, aber natürlich nicht unbedingt (wie auch immer definierten) Sätzen entsprechen müssen. Die Information, zu welchem Satz Knoten und Kanten gehören, ist durch das Attribut `sentence` repräsentiert.

Tokenknoten sind dadurch gekennzeichnet, daß sie ein Attribut `token` tragen, das den Tokentext enthält. Die Token eines Satzes sind außerdem in ihrer Reihenfolge durch Kanten mit `type:t` verbunden (andere Kanten sind `type:g`). Dies dient vor allem der korrekten Darstellung und der Traversierung; für den Benutzer sind diese Kanten unsichtbar, und er kann nicht auf sie zugreifen.

Für Knoten und Kanten gibt es das Attribut `cat`, das innerhalb von GAST keine besondere Bedeutung trägt, das aber besonders dargestellt wird, nämlich ohne Schlüssel und stets zuoberst. Kanten und Knoten tragen keine Nummernattribute. Sie werden für jede generierte Ansicht dynamisch durchnumeriert; diese Nummern dienen der Referenzierung in Annotationsbefehlen und werden in der Form `t23` für Token, `n23` für andere Knoten und `r23` für Kanten angezeigt.

Ein Satz hat einen Knoten mit dem Attribut `cat:meta`, der Informationen trägt, die den ganzen Satz betreffen. Hierbei kann es sich z.B. um Angaben von Quelle, Medium, Sprecher etc. handeln. Der Inhalt dieses Knotens wird im hellgrauen Bereich unter dem

[1] Ähnlichkeiten mit Namen lebender Personen sind rein zufälliger Natur.

Annotationsgraphen und unter dem Text des Satzes (blau dargestellt) in schwarzer Schrift angezeigt.

GAST bietet auch die Möglichkeit, Knoten und Kanten unterschiedlichen Ebenen zuzuordnen. Zur Zeit sind zwei Ebenen implementiert: Eine funktionale/semantische Ebene und eine formale/syntaktische. Zugehörigkeit eines Knoten oder einer Kante zu ersterer wird durch das Attribut `f-layer:t` wiedergegeben, Zugehörigkeit zu letzterer durch `s-layer:t`. Elemente, die dem `f-layer` angehören sind grün dargestellt, Elemente, die dem `s-layer` angehören, blau, Elemente, die beiden angehören, schwarz und Elemente, die keiner Ebene angehören, grau. Token sind keiner Ebene zugeordnet und werden schwarz dargestellt.

Die speziellen Attribute im Überblick:

- `sentence`: Satz, dem Knoten und Kanten angehören
- `token`: enthält den Text von Tokenknoten
- `cat`: privilegiertes Attribut
- `f-layer`: Zugehörigkeit zur funktionalen/semantischen Ebene
- `s-layer`: Zugehörigkeit zur formalen/syntaktischen Ebene

2 Tastaturbefehle

Einige die Ansicht betreffende Funktionen von GAST werden direkt über Tastenbefehle angesprochen. Nachfolgend eine Tabelle der verfügbaren Befehle:

Tastenkombination	Funktion
Graph	
Strg + Umschalt + −/+	Graph verkleinern/vergrößern
Strg + Umschalt + 0	Graph einpassen (bzgl. der Höhe)
Strg + Umschalt + Pfeile	Graph verschieben
Strg + Umschalt + Pos1/Ende	an den linken/rechten Rand des Graphen
Strg + Umschalt + Bild↑/Bild↓	an den oberen/unteren Rand des Graphen
F4	Elementnumerierung im Graphen an-/ausschalten
Fenster	
F1	Hilfefenster zeigen/verbergen
F2	Text und Satzannotationen zeigen/verbergen
F4	Knoten- und Kanten-IDS zeigen/verbergen
F6	Filterfenster zeigen/verbergen
F7	Suchfenster zeigen/verbergen

3 Kommandozeilenbefehle

3.1 Daten und Navigation

3.1.1 Datei laden: `load`

Eine Graphdatei wird mit dem Befehl `load` in den Arbeitsbereich geladen. Der Dateiname wird ohne die Dateiendung `.json` angegeben (enthält er Leerzeichen, so muß er in doppelte Anführungszeichen eingeschlossen werden: `"..."`). Alle Dateien werden aus dem Verzeichnis `data` im GAST-Programmordner geladen. Vor dem Laden werden Daten aus dem Arbeitsbereich von GAST gelöscht. Änderungen, die nicht explizit mit `save` gespeichert wurden, gehen dabei verloren.

3.1.2 Datei laden: `add`

Mit dem Befehl `add` kann wie mit `load` eine Datei in den Arbeitsbereich geladen werden. Der Unterschied ist, daß der Arbeitsbereich zuvor nicht geleert wird; die neu geladene Datei wird dem Arbeitsbereich hinzugefügt. Achtung: Die Dateien können dann nicht mehr einzeln gespeichert werden, und wenn die gleichen Satznamen sowohl im Arbeitsbereich als auch in der neu geladenen Datei vorhanden sind, werden unter diesen Namen jeweils beide Sätze angezeigt, was zu Problemen bei der weiteren Bearbeitung führt.

3.1.3 Datei speichern: `save`

Mit dem Befehl `save` wird der gesamte Arbeitsbereich in eine Datei gespeichert. Der Dateiname wird wie beim Laden angegeben, die Datei wird im `data`-Ordner gespeichert.

3.1.4 Arbeitsbereich leeren: `clear`

Mit dem Befehl `clear` werde alle Daten aus dem Arbeitsbereich von GAST gelöscht. Änderungen, die nicht explizit mit `save` gespeichert wurden, gehen verloren.

3.1.5 Neuen Satz anlegen: `ns`

Mit `ns` – gefolgt von mit Leerzeichen getrennten anzulegenden Namesräumen – werden ein oder mehrere neue Sätze angelegt. Dabei werden neue Meta-Knoten mit dem entsprechenden `sentence`-Attribut erstellt. Anschließend wird sofort in den ersten angegebenen Satz gewechselt.

3.1.6 Satz löschen: `del`

Mit dem Befehl `del` werden alle Elemente des gegenwärtig angezeigten Satzes gelöscht.

3.1.7 Gehe zu: `s`

Um von Satz zu Satz zu navigieren dient (neben dem Aufklappfeld, das auf Änderungen reagiert) der Befehl `s`, gefolgt vom Namen des anzuzeigenden Satzes.

3.1.8 Graphik exportieren: `export`

Um die Graphik, die GAST für den aktuellen Satz anzeigt, zu exportieren, gibt es den Befehl `export`. Als erstes Argument wird das gewünschte Format angegeben (zur Verfügung stehen `png`, `dot` und `svg`), als zweites Argument der Name der zu erstellenden Datei (ohne Endung; enthält der Name Leerzeichen, so muß er in doppelten Anführungsstrichen angegeben werden). Die Graphik wird im Ordner `exports` gespeichert.

3.1.9 Toolboxdaten importieren

Toolboxdateien können mit dem Befehl `import_toolbox` importiert werden. Als Parameter werden der komplette Dateiname (wenn die Datei im GAST-Programmordner liegt; ansonsten der relative oder absolute Pfad – enthält der Pfad Doppelpunkt oder Leerzeichen, muß er in doppelten Anführungsstrichen angegeben werden) sowie eine Formatbeschreibung erwartet. Die Formatbeschreibung wird im JSON-Format angegeben und besteht aus einer Liste von Listen von Markern. Die Listen sind nach Ebenen sortiert – die höchste (*record*) zuerst – und enthalten die der entsprechenden Ebene zugehörigen Marker (ohne Schrägstich). Dem Marker, auf dessen Grundlage die Token erstellt werden sollen, wird ein Stern vorangestellt. Elemente, die unter der Tokenebene liegen, werden zusammengefügt und in die jeweiligen Token integriert.

Eine Formatbeschreibung für eine Toolboxdatei mit drei Ebenen (Record, Wort, Morphem) könnte z.B. folgendermaßen aussehen:

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

Wie beim Befehl `load` wird der Arbeitsbereich von GAST vor dem Import geleert. Nicht gespeicherte Änderungen gehen verloren.

3.2 Annotationsbefehle

Die Annotationsbefehle von GAST sind darauf ausgerichtet, daß sie möglichst schnell einzugeben sind. Daher ist ihre Syntax entsprechend reduziert: Sie bestehen aus einem kurzen Befehl (oft nur ein Buchstabe) gefolgt von Parametern, die durch Leerzeichen getrennt sind.

3.2.1 Neuer Knoten: `n`

Der Befehl zum Erstellen eines neuen Knotens lautet `n`, gefolgt von den Attributen, die der neue Knoten haben soll, als Schlüssel-Wert-Paare in der Form `schlüssel:wert`.

Schlüssel und Wert können entweder als einfache Zeichenkette angegeben werden, wenn sie keines der in der GAST-Annotiersprache verwendeten Steuerzeichen (␣:")[2] enthalten, oder als Zeichenkette in doppelten Anführungsstrichen ("..."), die beliebige Zeichen enthalten darf (doppelte Anführungsstriche müssen mit einem Backslash maskiert werden: \").

Befehl n in modifizierter BNF:

```
befehl_n        =   "n " attribute
attribute       =   attribute " " attribute
                    attribut
attribut        =   schlüssel zeichenkette
schlüssel       =   zeichenkette ":"
zeichenkette    =   zeichen_außer_steuerzeichen+
                    """ beliebiges_zeichen* """
```

3.2.2 Neue Kante: e

Der Befehl zum Anlegen einer neuen Kante lautet e, gefolgt vom Start- und Zielknoten der zu erstellenden Kante und den Attributen, die sie erhalten soll.

Befehl e in modifizierter BNF:

```
befehl_e       =   "e " start_ziel " " start_ziel " " attribute
start_ziel     =   knotenreferenz
                   tokenreferenz
knotenreferenz =   "n" zahl
tokenreferenz  =   "t" zahl
```

3.2.3 Annotieren: a

Der Befehl zum Annotieren beliebiger Elemente ist a, gefolgt von den zu annotierenden Elementen und Attributen, mit denen sie annotiert werden sollen (alle angegeben Elemente werden mit allen angegebenen Attributen versehen). Die Reihenfolge einzelner Elemente und Attribute ist beliebig. Es können auch Sequenzen von Elementen eines Typs (also n, e oder t) durch Verbinden mit zwei Punkten angegeben werden; beispielsweise werden bei Angabe von t3..t7 alle Token von t3 bis t7 annotiert (die Sequenz kann auch umgekehrt angegeben werden, also t7..t3).

[2] Das Zeichen ␣ steht für das Leerzeichen.

Gleichzeitig können mit dem Befehl `a` Attribute gelöscht werden. Dazu wird der zu löschende Schlüssel mit Doppelpunkt, aber ohne Wert angegeben.

Befehl `a` in modifizierter BNF:

```
befehl_a            =   "a " a_parameter
a_parameter         =   a_parameter " " a_parameter
                        elementreferenz
                        attribut
                        schlüssel
elementreferenz     =   knotenreferenz
                        kantenreferenz
                        tokenreferenz
                        metaknotenreferenz
                        elementsequenz
kantenreferenz      =   "e" zahl
metaknotenreferenz  =   "m"
elementsequenz      =   knotensequenz
                        tokensequenz
                        kantensequenz
knotensequenz       =   knotenreferenz ".." knotenreferenz
tokensequenz        =   tokenreferenz ".." tokenreferenz
kantensequenz       =   kantenreferenz ".." kantenreferenz
```

3.2.4 Elemente löschen: `d`

Gelöscht werden Elemente mit dem Befehl `d`, gefolgt von den zu löschenden Elementen. Werden Knoten gelöscht, werden Ein- und Ausgehende Kanten dieses Knotens ebenfalls gelöscht. Beim Löschen von Tokenknoten aus der Mitte eines Satzes wird die Verbindung zwischen den Token rechts und links des gelöschten Tokens wieder hergestellt.

Befehl `d` in modifizierter BNF:

```
befehl_d        =   "d " d_parameter
d_parameter     =   d_parameter " " d_parameter
                    elementreferenz
```

3.2.5 Knoten gruppieren: `g`

Der Gruppierbefehl `g` erstellt einen neuen Mutterknoten für die angegebenen Knoten. D.h. es wird ein neuer Knoten erstellt und Kanten erzeugt, die diesen mit den zu gruppierenden Knoten verbindet. Die Parameter des Befehls sind die zu gruppierenden Knoten und die Attribute, mit denen der neue Knoten annotiert werden soll. Die Reihenfolge von Knoten und Attributen ist irrelevant.

Befehl `g` in modifizierter BNF:

```
befehl_g            =   "g " g_parameter
g_parameter         =   g_parameter " " g_parameter
                        knotenreferenz
                        tokenreferenz
                        knotensequenz
                        tokensequenz
                        attribut
```

3.2.6 Knoten anhängen: `h`

Der Befehl `h` funktioniert wie `g`, mit dem Unterschied, daß anstelle eines Mutterknotens ein neuer gemeinsamer Tochterknoten erstellt wird.

Befehl `h` in modifizierter BNF:

```
befehl_h            =   "h " h_parameter
h_parameter         =   h_parameter " " h_parameter
                        knotenreferenz
                        tokenreferenz
                        knotensequenz
                        tokensequenz
                        attribut
```

3.2.7 Tokenisieren: `t` und `ti`

Zum Eingeben von Token gibt es die Befehle `t` und `ti`. Diese Befehle nehmen eine Folge von durch Leerzeichen getrennten Wörtern als Argument und fügen sie als neue Token in den aktuellen Satz ein. Der Befehl `t` fügt die neuen Token dabei ans Ende der ggf. schon bestehenden an, `ti` nimmt als erstes Argument noch eine Tokenreferenz und fügt die neuen Token davor ein.

Die Wörter können als einfache Zeichenkette angeben werden, oder, wenn sie Steuerzeichen (␣:") enthalten, als Zeichenkette in doppelten Anführungsstrichen ("..."; doppelte Anführungsstriche müssen dann mit einem Backslash maskiert werden: \").

Befehle `t` und `ti` in modifizierter BNF:

```
befehl_t        =   "t " wörter
wörter          =   wörter " " wörter
                    wort
wort            =   zeichen_außer_steuerzeichen+
                    """ beliebiges_zeichen* """

befehl_ti       =   "ti " tokenreferenz " " wörter
```

3.2.8 Ebenen setzen: `l`

Mit dem Befehl `l` wird, alternativ zum Aufklappfeld, die Ebene gesetzt, in der sich die nachfolgend erstellten Elemente befinden sollen. `f` steht hier für die funktionale/semantische Ebene, `s` für die formale/morphosyntaktische, und `fs` für beide.

Befehl `l` in modifizierter BNF:

```
befehl_l    =   "l " ("f"|"s"|"fs")
```

4 Abfragesprache

4.1 Suche

Das Prinzip der Graphsuche von GAST besteht darin, mit einer Menge von Klauseln ein Graphfragment zu beschreiben. Bei der Suche werden dann alle Teilgraphen des durchsuchten Korpusgraphen gefunden, die dieser Beschreibung entsprechen. Es stehen die Klauseln `node`, `nodes`, `edge`, `link`, `text`, `meta`, `cond` und `def` zur Verfügung, wobei eine Anfrage mindestens eine `node`- oder `text`-Klausel oder eine unverbundene `edge`-Klausel enthalten muß. Die Klauseln werden in den folgenden Abschnitten im einzelnen beschrieben.

Die einzelnen Klauseln werden in beliebiger Reihenfolge in jeweils eine eigene Zeile geschrieben; Einrückungen und Leerzeilen werden nicht interpretiert. Kommentare werden durch Voranstellen eines Doppelkreuzes # markiert.

4.1.1 node

Die node-Klausel beschreibt einen Knoten, der im Graphfragment genau einmal vorkommen soll. Die Klausel besteht aus dem Schlüsselwort node, einer optionalen ID und einer Attributbeschreibung.

Die ID besteht aus einem @ gefolgt von einer Zeichenkette, die aus alphanumerischen Zeichen und dem Unterstrich besteht. Unter der ID kann der Knoten in anderen Teilen der Suchanfrage referenziert werden.

Die Attributbeschreibung besteht aus Schlüssel-Wert-Paaren der Form schlüssel:wert, die mit den logischen Operatoren ! für „nicht“, & für „und“ und | für „oder“ (Bindungsstärke: ! > & > |) sowie Klammerung mit runden Klammern verknüpft sind. Als Abkürzung für Disjunktionen von Schlüssel-Wert-Paaren mit dem gleichen Schlüssel steht die Form schlüssel:wert1|wert2|...|wertn zur Verfügung.

Der Schlüssel eines Schlüssel-Wert-Paares kann entweder als einfache Zeichenkette angegeben werden, wenn er keines der in der Abfragesprache verwendeten Steuerzeichen (␣():!&/?+*{}@#^) enthält, oder als Zeichenkette in doppelten Anführungsstrichen ("xyz"), die beliebige Zeichen enthalten darf (doppelte Anführungsstriche müssen mit einem Backslash maskiert werden: \").

Die Werte der Schlüssel-Wert-Paare sind Zeichenketten, die auf dreierlei Art und Weise angegeben werden können. Die erste Variante sind einfache umarkierte Zeichenketten, die alle Zeichen außer den in der Abfragesprache verwendeten Steuerzeichen (␣():!&/?+*{}@#^) enthalten dürfen. Diese Zeichenketten werden bei der Suche ohne Beachtung von Groß- und Kleinschreibung verglichen. Die zweite Variante sind Zeichenketten in doppelten Anführungsstrichen ("xyz"). Diese dürfen beliebige Zeichen enthalten (doppelte Anführungsstriche müssen mit einem Backslash maskiert werden: \") und werden unter Beachtung von Groß- und Kleinschreibung verglichen. Die dritte Variante sind reguläre Ausdrücke. Diese werden in Schrägstrichen angegeben (/x.z/ und gehorchen den Regeln für reguläre Ausdrücke in Ruby (siehe http://www.ruby-doc.org/core/Regexp.html). Die regulären Ausdrücke sind nicht verankert; um den Ausdruck am Anfang bzw. Ende einer Zeichenkette zu verankern müssen also ^ bzw. $ verwendet werden; eine beliebige Zeichenkette kann mit // gefunden werden.

Des weiteren kann die Attributbeschreibung Kriterien für ein- und ausgehende Kanten enthalten. Diese bestehen aus dem Schlüsselwort in bzw. out, einer optionalen Attributbeschreibung in runden Klammern und einem optionalen Quantor. Der Operator in bzw. out findet alle ein- bzw. ausgehenden Kanten mit den angegebenen Attributen, der Quantor gibt an, wieviele Kanten der spezifizierten Art vorhanden sein dürfen und ist (syntaktisch) wie bei regulären Ausdrücken definiert: {m,n} für mindestens m-mal, höchstens n-mal; bei Auslassung der ersten Zahl wird 0 angenommen, bei Auslassung der zweiten unendlich. {n} steht für genau n mal. Des weiteren gibt es die Abkürzungen ? für {0,1}, * für {0,} und + für {1,}. Anders als von regulären Ausdrücken gewohnt (und anders als beim Auftreten von Quantoren in anderen Kontexten der

GAST-Abfragesprache), wird das Fehlen eines Quantors hier als `{1,}` interpretiert.

Für die Kanten wiederum können – zusätzlich zu den einfachen Attributen – über die Schlüsselwörter `start` bzw. `end` und Attributbeschreibungen in runden Klammern auch Eigenschaften des Start- bzw. Zielknoten angegeben werden.

Ähnlich wie `in` und `out` funktioniert `link`, jedoch werden damit nicht nur ein- und ausgehenden Kanten abgefragt, sondern (ggf.) komplexere Verbindungen zu anderen Knoten. Wie diese Verbindungen spezifiziert werden, ist in 4.1.4 beschrieben. Auch für `link` gelten die Regeln für Quantoren, wie für `in` und `out` beschrieben.

Die `node`-Klausel in modifizierter BNF:

```
node-klausel    =   "node" id? " " knotenattribute
id              =   "@" alphanumerisches_zeichen+
knotenattribute =   knotenattribute " & " knotenattribute
                    knotenattribute " | " knotenattribute
                    "!" knotenattribute
                    "(" knotenattribute ")"
                    attribut
                    kantenkriterium
attribut        =   zeichenkette ":" attributwert ("|" attributwert)*
zeichenkette    =   zeichen_außer_steuerzeichen+
                    """ beliebiges_zeichen* """
attributwert    =   zeichen_außer_steuerzeichen+
                    """ beliebiges_zeichen* """
                    "/" regulärer_ausdruck "/"
kantenkriterium =   "in" ("(" kantenattribute ")")? quantor?
                    "out" ("(" kantenattribute ")")? quantor?
                    "link" ("(" verbindung ")") quantor?
quantor         =   "?" | "*" | "+" | "{" zahl? ("," zahl?) "}"
kantenattribute =   kantenattribute " & " kantenattribute
                    kantenattribute " | " kantenattribute
                    "!" kantenattribute
                    "(" kantenattribute ")"
                    attribut
                    knotenkriterium
knotenkriterium =   "start" ("(" knotenattribute ")")
                    "end" ("(" knotenattribute ")")
```

Beispiele:

- Suche alle Knoten, die die Kategorie S oder VP haben und keine Token sind:
  `node cat:S|VP & !token://`

- Suche alle Knoten, die von der Kategorie VP sind oder Token mit dem pos-Wert verb:
  `node cat:VP | token:// & pos:verb`
- Suche alle Knoten der Kategorie S, die mindestens zwei ausgehende AUX-Kanten haben:
  `node cat:S & out(cat:AUX){2,}`
- Suche alle Knoten der Kategorie S, die mindestens einen Knoten mit dem pos-Wert pro dominieren:
  `node cat:S & out(end(pos:pro))`

4.1.2 `nodes`

Die `nodes`-Klausel beschreibt eine Menge von Knoten, die im Graphfragment enthalten sein sollen; die Menge kann jedoch auch leer sein. Die `nodes`-Klausel hat, abgesehen vom Schlüsselwort, die gleiche Syntax wie die `node`-Klausel.

Die `nodes`-Klausel in modifizierter BNF:

```
nodes-klausel    =   "nodes" id? " " knotenattribute
```

4.1.3 `edge`

Die `edge`-Klausel hat zwei Anwendungen. Zum einen kann sie verwendet werden, um einzelne Kanten zu suchen. Dann besteht die Klausel aus dem Schlüsselwort `edge`, einer optionalen ID, unter die die Kante in der Ausgabe referenziert werden kann, und einer Attributbeschreibung für Kanten wie in 4.1.1 beschrieben. Zum anderen dient die `edge`-Klausel dazu, anzugeben, daß zwischen zwei Knoten bzw. Knotenmengen des Graphfragments (mit `node` bzw. `nodes` spezifiziert) eine Kante mit den angegebenen Eigenschaften existieren soll. Dazu werden nach der (optionalen) ID der Kante die IDs von Start und Ziel der Kante angegeben.

Durch die optionale ID und die verschiedenen Verwendungsmöglichkeiten kann das Schlüsselwort `edge` von null bis drei IDs gefolgt sein. Die Interpretation dieser IDs ergibt sich aus ihrer Anzahl und der Reihenfolge. Eine ID: ID der Kante selber; zwei IDs: Start und Ziel der Kante; drei IDs: ID der Kante, Start und Ziel.

Die `edge`-Klausel in modifizierter BNF:

```
edge-klausel    =   "edge" id? (id id)? " " kantenattribute
```

Beispiele:

- Suche alle Kanten die die syntaktische Funktion Subjekt anzeigen:
  `edge synfunc:subj`
- Suche alle Knoten der Kategorie S, jeweils mit der Menge von Knoten der Kategorie NP, die über eine Kante der Kategorie S, A oder P verbunden sind:
  ```
  node @s cat:S
  nodes @np cat:NP
  edge @s@np cat:S|A|P
  ```

4.1.4 `link`

Die `link`-Klausel beschreibt, wie zwei Knoten oder Knotenmengen des Teilgraphen verbunden sein sollen. Dabei kann als Verbindung eine Kette von Knoten und Kanten ähnlich einem regulären Ausdruck beschrieben werden. Die `link`-Klausel besteht aus den Schlüsselwort `link`, der Angabe von Start- und Endknoten der Verbindung in der Form `@id1@id2` und der Beschreibung der Verbindung.

Die Verbindungsbeschreibung besteht aus einer Abfolge von `edge`- und `node`-Elementen. Diese bestehen aus dem jeweiligen Schlüsselwort (`edge` bzw. `node`) und optional einer Angabe von Bedingungen, die das Element erfüllen muß, in runden Klammern. Dabei handelt es sich um eine Attributbeschreibung wie oben für `node` angegeben. Gefolgt werden kann die Elementbeschreibung von einer ID, unter der die gefundenen Elemente in der Ausgabe (nicht in der Suche!) referenziert werden können.

Für Alternativen steht der Operator `|` „oder“ zur Verfügung; bezüglich der Bindungsstärke steht er unter der Abfolge. Klammerung ist mit runden Klammern möglich. Des weiteren können Quantoren verwendet werden. Diese werden weder als gierig noch als genügsam interpretiert; alle passenden Verbindungen werden gefunden und als separate Treffer gewertet.

Eine Verbindung besteht – der Natur eines Graphen entsprechend – stets aus einem Wechsel von Knoten und Kanten (beginnend und endend mit jeweils einer Kante). Bei der Angabe von einer Verbindung darf jedoch darauf verzichtet werden, stets Knoten und Kanten im Wechsel anzugeben; nur die erste Kante darf nicht ausgelassen werden. Stehen zwei Elemente des gleichen Typs (also `edge` oder `node`) hintereinander, so wird bei der Suche dazwischen ein unspezifiziertes Element des jeweils anderen Typs eingeschoben. Die Verbindungsbeschreibung `edge(a:b) edge(c:d)` beispielsweise findet eine Kante mit dem Attribut `a:b`, dann einen beliebigen Knoten und dann eine Kante mit dem Attribut `c:d`.

Wird `link` als Knotenattribut (z.B. in einer `node`-Klausel) verwendet, fällt die Angabe von Start- und Endknoten weg. Startknoten ist dann der gesuchte Knoten, Endknoten der letzte in der Verbindung spezifiziert Knoten bzw., wenn die Verbindungsbeschreibung mit einer Kante endet, ein Knoten mit beliebigen Eigenschaften.

Die `link`-Klausel in modifizierter BNF:

```
link-klausel    =   "link" id id " " verbindung
verbindung      =   verbindung " " verbindung
                    verbindung "|" verbindung
                    "(" verbindung ")"
                    verbindung quantor
                    "edge" ("(" kantenattribute ")")? id?
                    "node" ("(" knotenattribute ")")? id?
```

Beispiele:

- Suche alle Graphfragmente, die aus einem Knoten der Kategorie P und einem der Kategorie S bestehen, wobei ersterer letzteren über eine Kante der Kategorie EX dominiert:
  ```
  node @p cat:P
  node @s cat:S
  link @p@s edge(cat:EX)
  ```
- Suche einen Knoten der Kategorie S, alle Knoten der Kategorie NP, die dieser dominiert, und alle Token, die von den NPn dominiert werden:
  ```
  node @s cat:S
  nodes @vpn cat:NP
  nodes @tok token://
  link @s@npn edge+
  link @npn@tok edge+
  ```
- Suche einen Knoten der Kategorie S und alle Token, die von diesem über einen NP-Knoten dominiert werden (bis auf die IDs äquivalent zum vorigen Beispiel):
  ```
  node @s cat:S
  nodes @tok token://
  link @s@tok edge+ node(cat:NP) edge+
  ```

4.1.5 `text`

Die `text`-Klausel dient dazu, eine Abfolge von Token-Knoten zu finden. Dabei ermöglicht die Textsuche sowohl die Suche nach reinem Text als auch nach weiteren Attributen der Token-Knoten. Die `text`-Klausel besteht aus dem Schlüsselwort `text`, einer optionalen ID und der Beschreibung eines Textfragments, das mit `^s` am Anfang bzw. Ende des Textes eines Satzes verankert werden kann.

Die Beschreibung des Textfragments besteht aus einer Abfolge von Wortbeschreibungen, die aus einer Zeichenkette, die den Tokentext beschreibt (drei Varianten wie oben unter `node` für die Werte in Schlüssel-Wert-Paaren beschrieben), und einer optionalen Angabe von Attributen (Attributebeschreibung wie oben unter `node`) in runden Klammern zusammengesetzt ist. Für die Beschreibung des Textfragments stehen wie

bei der Verbindungsbeschreibung der Operator | für „oder“ (Bindungsstärke schwächer als die der Sequenz), Klammerung und Quantoren zur Verfügung, wobei die Quantoren bei der Textsuche genügsam sind. Zusätzlich kann Textfragmenten eine ID nachgestellt werden, um die gefundenen Knoten in anderen Klauseln zu referenzieren. Quantoren und IDs haben eine höhere Bindungsstärke als Sequenz und Disjunktion; die Reihenfolge von Quantor und ID hinter dem selben Textfragment ist beliebig. Die optionale ID nach dem Schlüsselwort gilt für alle Knoten der `text`-Klausel. Das gesuchte Textfragment kann mit `^s` am Beginn bzw. Ende eines Textes (Text eines Satzes) verankert werden.

Die `text`-Klausel in modifizierter BNF:

```
text-klausel    =   "text" id? " " "^s"? textfragment "^s"?
textfragment    =   textfragment " " textfragment
                    textfragment "|" textfragment
                    "(" textfragment ")"
                    textfragment quantor
                    textfragment id
                    wort
wort            =   attributwert ("(" knotenattribute ")")?
```

Beispiel:

- Suche alle Sätze, in denen das Wort das Nomen „Säge“ an dritter Stelle steht:
  `text ^s //{2,2} Säge(pos:n)`
- Suche zwei Vorkommen von „er“, die von einer beliebigen Anzahl Wörter getrennt sind, wobei das erste „er“ und alle folgenden Wörter bis zum zweiten „er“ von einem Knoten vom cat S dominiert werden:
  ```
  node @s cat:S
  text (er //*)@t er
  link @s@t edge*
  ```

4.1.6 `meta`

Die `meta`-Klausel schränkt die Menge der zu durchsuchenden Sätze ein. Über diese Klausel können Eigenschaften angegeben werden, die der `meta`-Knoten eines Satzes (ein Knoten mit dem entsprechenden `sentence`-Wert und dem Attribut `cat:meta`) haben muß. Die Klausel besteht aus dem Schlüsselwort `meta` und einer Attributbeschreibung wie oben unter `node` beschrieben.

Die meta-Klausel in modifizierter BNF:

```
text-klausel    =   "meta " attribute
```

4.1.7 cond

Die cond-Klausel gibt Bedingung an, die das Graphfragment erfüllen muß und wirkt wie ein Filter. Die Klausel besteht aus dem Schlüsselwort cond und der Bedingung in Ruby-Kode. Die Knoten und Knotenmengen werden dabei durch die vergebenen IDs referenziert. Bei der Referenzierung ist zu beachten, daß es sich bei den mit node und edge gefundenen Knoten und Kanten um einzelne Elemente, bei den mit nodes, text und link gefundenen Knoten und Kanten hingegen um Arrays von Elementen handelt.

Auf die Attribute der Knoten wird in der Form .attr['schlüssel'] zugegriffen; für die Attribute sentence, token und cat stehen Abkürzungen der Form .sentence, .token und .cat zur Verfügung. Über die Methode .meta kann auf den meta-Knoten des Satzes, zu dem das Elementgehört, zugegriffen werden. Bei der Verwendung von Attributwerten ist zu beachten, daß es sich bei diesen stets um Zeichenketten handelt; sollen sie als Zahlen behandelt werden, müssen sie mit .to_i bzw. .to_f umgewandelt werden.

Beispiele:

- Suche zwei S-Knoten, die über eine ad-Kante verbunden sind und den gleichen Wert für tns haben:
  ```
  node @s1 cat:S
  node @s2 cat:S
  link @s1@s2 cat:ad
  cond @s1.attr['tns'] == @s2.attr['tns']
  ```
- Suche alle S-Knoten, die mindestens drei Token dominieren:
  ```
  node @s cat:S
  nodes @tok token://
  link @s@tok edge+
  cond @tok.length >= 3
  ```

4.1.8 def

Mit def besteht die Möglichkeit, für die Suche Makros zu definieren. Dabei wird ein Name angegeben, unter dem das Makro in den Suchklauseln angesprochen werden kann, und eine Attribut- oder Verbindungsbeschreibung, die durch den Namen vertreten wird. Die Attribut-/Verbindungsbeschreibung ist aufgebaut wie für node und link in 4.1.1 bzw. 4.1.4 beschrieben.

Die Makrodefinition `def` in modifizierter BNF:

```
makrodefinition =   "def " name " " (kantenattribute | knotenattribute | verbindung)
name            =   alphanumerisches_zeichen+
```

Beispiele:

- Suche zwei Knoten mit `lemma:cum` und `pos:G-`, die über eine Kante verbunden sind, und von denen einer ein Token ist:
  ```
  def cum lemma:cum & pos:G-
  node @s1 cum
  node @s2 cum & token://
  link @s1@s2 edge
  ```

4.2 Datenexport

Die GAST-Abfragesprache biete auch Funktionalität zum Exportieren von Suchergebnissen als CSV-Dateien an. Mit den nachfolgen beschriebenen Klauseln kann angegeben werden, wie Infomationen der in einer zuvor durchgeführten Suche gefundenen Teilgraphen ausgegeben werden sollen. Die Daten jedes gefundenen Teilgraphen werden in eine Zeile der CSV-Datei geschrieben, mit `sort` können die Ergebnisse sortiert werden, mit `col` wird angegeben, welche Spalten mit welchen Werten angelegt werden sollen. Als erste Spalte wird stets eine fortlaufende Numerierung der Ergebnisse mit ausgegeben.

4.2.1 `sort`

Die `sort`-Klausel dient dazu, die Ausgabe der gefundenen Teilgraphen zu sortieren. Es können mehrere `sort`-Klauseln angegeben werden, wobei weiter unten angegeben Klauseln nur ausgewertet werden, wenn weiter oben angegebenen Klauseln keine Reihenfolge zwischen zwei Teilgraphen ergeben. Eine `sort`-Klausel wird in Ruby-Kode formuliert und muß einen Wert ergeben, der für die Sortierung verglichen werden soll. Wie bei `cond` werden die gefundenen Knoten und Kanten über die in der Suche vergebenen IDs referenziert. Bei Verwendung von Attributwerten ist zu beachten, daß es sich bei diesen stets um Zeichenketten handelt; sollen sie als Zahlen verglichen werden, müssen sie mit `.to_i` bzw. `.to_f` umgewandelt werden.

Beispiele:

- Sortiere die Ergebnisse nach Satznamen, bzw. nach Tokennummer (die Methode `.tokenid` gibt die Stelle des Tokens im Satz, beginnend mit 0, aus), wenn sie dem

gleichen Satz angehören (durch die Suchanfrage sei gegeben: ein Token mit der ID `@t1`):

```
sort @t1.attr['sentence']
sort @t1.tokenid
```

4.2.2 `col`

Jede `col`-Klausel steht für eine zu exportierende Spalte. Sie hat als ersten Parameter den Spaltentitel (der keine Leerzeichen enthalten darf), gefolgt von Ruby-Kode, der den auszugebenden Wert ergibt. Knoten und Kanten werden wie gehabt über ihre in der Suche vergebene ID referenziert, dabei ist zu beachten, daß es sich bei den mit `node` gefundenen Knoten um einzelne Knoten, bei den mit `nodes`, `text` und `link` gefundenen Knoten hingegen um Arrays von Knoten handelt.

Zugriff auf Attribute erfolgt wie unter 4.1.7 für `cond` beschrieben. Es gibt jedoch noch weitere für die Ausgabe nützliche Methoden: `.tokens` gibt die über syntaktische Kanten dominierten Token als Liste aus, `.text` deren Text als Zeichenkette (die einzelnen Tokentexte mit Leerzeichen getrennt). `.sentence_tokens` gibt alle Token des Satzes, dem das Element angehört als Liste aus, `.sentence_text` wiederum deren Text. `.position` gibt die Position eines Knotens als Durchschnitt der Positionen seiner dominierten Token aus.