Basic Constraint Grammar Tutorial for CG-3 (Vislcg3)

This text constitutes a fairly complete manual for the CG-3 compiler formalism, but it is also intended as a tutorial for computational linguists who have not so far used Constraint Grammar in their work, or have been using an older implementation, such as Vislcg, CG-2 or CG-1. Thus, usage and linguistic examples are provided throughout the text. For the same reason, grammar architecture, linguistic-expressive issues and application perspectives are discussed where relevant.

1. Command-line usage:

Once installed on your system, the CG-3 compiler can be run command line or as a background service for other programs, reading input from a Unix pipe or file. Basic parameters are a grammar (--grammar) and encoding information (e.g. utf8 or iso-latin). Optionally, specific grammar sections can be activated in isolation (--sections), certain rule types can be inactivated (--no-mappings), and rule actions can be traced (--trace).

- standard call: vislcg3 --grammar *rulesfile,* vislcg3 -g *rulesfile*
- without mapping rules (affects ADD and MAP): --no-mappings
- with rule-number traces for debugging: --trace, -t
- limited number of *n* least heuristic constraint sections: --sections *n, --sections n-m, -s n*
- special mapping prefix (default is '@'), e.g. '§': --prefix §, --prefix '§', -p §, not --prefix='§'
- UTF-8 input (”codepage”): -C UTF-8

Standard morphosyntactic grammars: Ordinarily, input is piped from a lexicon-based morphological multitagger, or a lower level of Constraint Grammar (annotated corpus), but input from probabilistic taggers (Treetagger, TnT, Brill etc.) can also be used, in which case the first rule section typically will be a correction grammar rather than a morphological disambiguation grammar. In order to prevent syntactic rules from interfering with morphological ones (by being run on morphologically not-yet disambiguated input), it is recommended to run vislcg3 twice – first without, then with syntactic mapping. Finally, disambiguated/tagged output can be piped directly to a file, or processed with layout filters or further grammars in other formalisms (constituent grammar, dependency grammar, field grammar etc.).

- cat *textfile* | multitagger | vislcg3 -C UTF-8 --grammar *rulesfile* --no-mappings | vislcg3 -C UTF-8 --grammar *rulesfile* | postfilter > *textfile.cg*
- with tracing: Use --trace after the grammar that you want to debug. Chaining several --trace grammars will work with vislcg2, but give odd results in vislcg3

Multitagger or other input has to deliver so-called verticalized text, i.e. one token pr. line, with non-punctuation tokens followed by a *cohort* of one or more possible analysis, indented, one pr. line. Conventionally, cohort lines start with the lexeme or base-form (in quotes), followed by word class (PoS) and inflexion tags in upper case. Secondary tags, meant to be used as disambiguation context, but not intended for disambiguation themselves, such as subclass, valency and semantic tags, should be placed in <...> brackets between lexeme and word class tags:

```
word form
   "lexeme-1" <valency> .. <semantics> .. POS-1 INFLEXION
   "lexeme-1" <valency> .. <semantics> .. POS-2 INFLEXION
   "lexeme-2" <valency> .. <semantics> .. POS-3 INFLEXION
   "lexeme-2" <valency> .. <semantics> .. POS-4 INFLEXION
```

Output after CG will look like this:

```
"<he>"
   "he" PERS MASC 3S NOM @SUBJ>
"<could>"
   "can" <aux> V IMPF @FS-STA
"<see>"
   "see" <vq> <mv> V INF @ICL-AUX<
"<a>"
   "a" <indef> ART S @>N
"<red>"
   "red" <jcol> <S:14> ADJ POS @>N
"<house>"
   "house" <build>  N S NOM @<ACC
"<$.>"
```

Or, with --trace, to identify rule numbers used:

```
"<he>"
   "he" PERS MASC 3S NOM @SUBJ> MAP:2734
"<could>"
   "can" <aux> V IMPF MAP:1584 @FS-STA ADD:1590 ADD:1595 MAP:2477
"<see>"
   "see" <vq> <mv> V INF ADD:1621 ADD:1637 @ICL-AUX< ADD:1621 ADD:1637 MAP:2194
;  "see" <vq> V PR -3S ADD:1621 REMOVE:5211
;  "see" <vq> V IMP ADD:1621 REMOVE:5211
"<a>"
   "a" <indef> ART S @>N MAP:2161
"<red>"
   "red" <jcol> <S:14> ADJ POS @>N MAP:1758
;  "red" <color> <S:16> <first> N S NOM SUBSTITUTE:1532 SUBSTITUTE:1544 REMOVE:4681
"<house>"
   "house" <build> <HH> <second> <S:135> <nhead> N S NOM SUBSTITUTE:1532
SUBSTITUTE:1545 SELECT:5681 @<ACC SUBSTITUTE:1532 ADD:1722 MAP:2771
;  "house" V PR -3S ADD:1621 REMOVE:5218
;  "house" V IMP ADD:1621 REMOVE:5320
;  "house" V INF ADD:1621 SELECT:5681
"<$.>"
```

Note that removed lines are still shown, but marked with a ';' – allowing easier debugging. Secondary programs can be used to filter this output in various ways. The author, for instance, uses the following:

1. niceline.perl (condenses output to one-line cohorts)

2. hilite_cg 31 '@.*' (colours syntactic @-fields red)

2. The rules file

A vislcg3 rules file consists typically of the following sections:

DELIMITERS (1 line, defines sentence boundaries)

SETS (1 or more sections of set definitions, compiled as one)

CORRECTIONS (1 section of correction rules, replacing tags anywhere in a reading)

MAPPINGS (1 section of mapping rules, adding tags at the end of a reading line)

CONSTRAINTS (1 or more sections of REMOVE or SELECT rules)

END

The CG-3 compiler will use *DELIMITERS* to chunk the text into sentence windows, then define labels for sets of tags or tag lists *(SET* section), and finally apply the rules in the order of occurrence, one section at a time. *CONSTRAINTS* sections will be run iteratively. For special purposes, CG-3 recognizes a *BEFORE-SECTIONS* and a *AFTER-SECTIONS* section, that are only run once, to respectively prepare or post-process an annotation.

Set sections contain LIST definitions of sets, written as lists of OR'ed tags or tag chains (in parentheses). Once defined, sets may be combined into new sets with a SET definition, using *set union* and *set subtraction*. Unlike earlier CG compilers, CG3 does in fact allow SET definitions anywhere in the grammar, so a grammarian may choose to keep the more general definitions together in a separate section, while keeping rule- or task-specific SET definitions close to the rule or rule section that uses them.

Mapping and **Correction sections** have MAP/ADD and SUBSTITUTE rules, respectively. These rules are applied in strict sequential order. In earlier CG implementations, MAP/ADD rules could not "see" in their context conditions what earlier mapping rules had mapped, but this limitation has been lifted in CG3, so it is now possible for instance, to let argument mapping rules refer to previously mapped main verb and auxiliary tags.

Constraint sections will be interpreted as heuristicity batches, with safer rules in the first sections, and more heuristic rules in later sections. Each section is repeated until no further of its rules can be instantiated (i.e. meet their context conditions), then the next section is run and the first section re-run after second-section disambiguation to check for changed contexts. After that, a third section is run, and the lower ones rerun: 1 – 2 1 – 3 1 2 – 4 1 2 3 ... etc. The --sections n flag can limit this reiteration to sections 1 through n , or even select a range (or ranges) of sections, e.g 3-7,9-10.

Since CG-3 allows any rule type to occur anywhere in the grammar, a neutral *SECTION* header has been introduced to optionally replace the traditional *CONSTRAINTS, MAPPINGS* and *CORRECTIONS* headers.

Unlike its predecessors CG-2 and Vislcg, CG-3 always applies rules in the order they occur in the grammar, and will try to apply a given rule to all cohorts in the window before moving on to the next rule.

Each set definition or rule is terminated with a semicolon, but can run over several lines. As in several programming languages, the #-symbol marks the rest of a line as a comment.

3. The individual elements and functors of a CG grammar file

3.2. Delimiters

The vislcg compiler applies rules within a certain context window, defined by *delimiters*. Typically, delimiters will be sentence boundary markers (i.e. punctuation), but paragraphs, corpus section markers or even specific stop-words could be used. Rules can refer to the boundaries with the reserved symbols >>> (left boundary) and <<< (right boundary).

DELIMITERS = “<.>” “<!>” "<?>" ;

The example defines a full stop, exclamation mark or question mark as a delimiter. Note that punctuation notation follows word form notation, with quotes and angle brackets. For the sake of headlines and running input in general, it can be recommended to include a ”hard-break” introduced by a preprocessor, e.g. ”<¶>”.

CG-3 can keep several windows in memory at any time in order to facilitate cross-window scanning from contextual tests (using the W operator, with a default span of ±2 windows). It is also possible to break a window into smaller windows on-the-fly with a DELIMIT rule anywhere in the grammar.

3.3. Set definitions

In both their targets and context conditions, CG rules can refer not only to words, lexemes and tags, but also *sets* of words, lexemes or tags, or even combinations of these three types. Two kinds of set definitions are used:

(a) *LIST* set-name =

followed by a list of tags or tag combinations (the latter in parentheses), separated by spaces. The list constitutes the set, and a rule targeting a set is equivalent to a batch of rules targeting each set element separately. Note that tag combinations are not, unlike CG2, order sensitive at the moment, i.e. (<vi> <vt>) equals (<vt> <vi>). Therefore, if you wish to make this distinction, you should define composite tags or create them with a preprocessor, e.g. <vi-vt> and <vt-vi>.

(b) *SET* set-name =

defining a new set as a mathematical operation on existing sets. Sets used in a SET definition, must occur earlier in the grammar. Tags can be used as sets on the fly by enclosing them in parentheses.

A set element can be:

- a tag, word form or lexeme, e.g. N [for noun], "<bought>" [word form] or "buy" [lexeme]
- a combination of (1), as a kind of "snapshot" from a reading, in parentheses. The snapshot may have "holes" (i.e. interfering tags appearing in the reading but not in the set element). For instance, (N M P) [for *noun masculine plural*], or (“eat” INF).

In a SET definition (b), sets can be combined with the following operators:

set union: OR or **|** , e.g. set1 OR set2 OR (tag3) OR (N F S)

concatenation (cartesian product): + , e.g. set1 + set2, yields all possible combinations of the 2 sets' elements. Thus, a concatenation of LIST set1 = V and LIST set2 = INF GER PCP covers all non-finite verb forms: (V INF) (V GER) (V PCP).

negation (match set difference): - , e.g. set1 *but not* set2, means set1 as long as the reading in question does not contain elements from set2. Thus, rather than as just a removal of set2 elements form the set1 list (i.e. *defining list difference,* as used in Tapanainen's cg2), vislcg3 interprets the minus operation as a kind of NOT condition, so the presence of a set2 element in a reading will block and override the presence of a set1 reading. Thus, (N) - (P) means non-plural nouns. If needed for compatibility reasons or the like, the old, narrow list difference operator can still be had, using the ∆ symbol (U+2206).

The + and - operators have precedence over OR.

failfast: The ^ symbol can be used in both set operations (e.g. A ^ C OR B) and set definitions (LIST = a b ^c). A set or list element prefixed by ^ will block instantiation of the entire set if matched in a given reading, even if other elements of the set would otherwise make the set compatible with the cohort line in question.

Note that all set operators, as well as the parenthesis convention for creating sets on-the-fly, can be used in targets and context conditions of rules.

tag inversion: ! (exclamation mark) is used as a tag-prefix and means *”all but ..”* or *”but not”*, much like the ^ fail fast prefix. However, ! is used in tag strings and context parentheses, while ^ is used in set definitions or set operations. Thus, the latter (^) blocks OR'ed lists, while the former (!) blocks AND'ed lists, if instantiated in a reading. (V !PAS), for instance, matches all verb forms that are not passive, independently of e.g. tense and mood.

The formalism has a built-in magic set, (*), to denote ”everything”. The (*) set is an easy way to navigate step-wise left or right in LINK'ed contexts, e.g. LINK -1 (*) LINK 1 ... to include the 0 position in an unbounded search (useful in complex vp's). The magic (*) set can also be used to negate a set, e.g. (*) - (N) for all tokens that are not nouns. (*) - N is formally equivalent to (!N), but faster for the compiler to match.

3.4. Constraints

Constraint rules are ordered in sections, usually in order to separate safer rules (to be used earlier) from more heuristic rules (to be used later). One and the same grammar can be run at different levels of heuristicity by using the --sections n flag when calling Vislcg3, meaning that only the first (=safest) n constraint sections of the grammar will be used.

A CG rule has the following general form, with [] brackets indicating optional elements:

["<Wordform>"] ***OPERATION*** *TARGET [[**IF]** (CONTEXT-1) (CONTEXT-2) ...] ;*

Consider the following examples:

(a) REMOVE VFIN IF (-1 ART) ;

(b) REMOVE (N) IF (-1 (PERS NOM)) ;

(a) will remove finite verb readings (the target) from a cohort, if the one immediately to the left (-1) contains an article tag, while (b) will remove noun readings in the presence of an immediately preceding personal pronoun in the nominative, thus disambiguating nominal-verbal ambiguities like *hit* in *”the hit/they hit”*.

Note that the target VFIN is a defined set (e.g. consisting of tense or mode tags), while the target (N) is a simple tag, declared as a set on-the-fly by using parentheses.

3.4.1 OPERATION:

(a) **REMOVE**

Removes a reading from a cohort, if it contains a TARGET'ed tag – unless this reading is the last surviving reading. In the case of morphological or PoS tag this means that one (entire) reading line, in a cohort of readings for a given token, will be removed – for instance the reading line "comer" V PR 1S IND will be removed from the analysis cohort of "como", if either the V (verb) or PR (present tense) tags are TARGET'ed by a successful REMOVE rule, leaving the "como" ADV reading to survive. If the target is a MAP'ed tag with the predefined prefix (for syntax, usually a @-tag), it is removed from the reading line, and if it is the only or last surviving MAP'ed tag, the whole reading line will be removed (unless it is the last reading line in the cohort). If you explicitly wish to allow the removal of a last reading, you can do so using the rule option UNSAFE, i.e. *REMOVE UNSAFE (TAG) IF ....,* or globally through the --unsafe flag (which can be overridden locally by the SAFE option).

(b) **SELECT**

Selects a reading, if it contains a TARGET'ed tag. In practice, selection is equivalent to a removal of all *other* readings. In the case of @-tag target, the reading line is cleared of all other @-tags.

In ordinary mode, each operation will immediately affect (narrow) the contexts of subsequent rules. This progressive and interactive disambiguation is one of the core strengths of the CG methodology and should not be overridden lightly, but if you do want to have a rule look at already-deleted contexts, you can do so with the LOOKDELETED option. A softer variant is the LOOKDELAYED option that will only look at information removed by a previous rule that has been specifically scheduled for delayed removal with the DELAYED option. Individual contexts (rather than a whole rule) can be made to see deleted material with the D-operator, and delayed-removed material with the d-operator, e.g. (*-1d N-HUM). Finally, there is an IMMEDIATE option to override a global --delayed flag.

3.4.2 WORDFORM:

Optional part of a rule, restricting the rule to the word form in question. Since the operation is case sensitive, preprocessing (lower-casing) is necessary, if a rule targeting e.g. an English noun also is to apply if the noun occurs in sentence-initial position. VISL grammars use lower-casing of initials, storing the uppercase information as a tag (<*>) instead.

WORDFORM may only be 1 wordform, and must not contain set operatons or tags. Otherwise, the WORDFORM condition works like a context condition for position 0 (self).

3.4.3 TARGET:

Obligatory part of a rule. A target is always a set, either a predefined set from the SETS section, or a tag string defined as a set on-the-fly by using parentheses, e.g. NOMINAL (defined by LIST = N ADJ PCP) or (N) or (N F P). Using predefined sets as targets, effectively fuses what in the CG-1 formalism was a same-context batch of multiple rules, into one rule:

SELECT NOMINAL IF (-1C DET) ;

(same as 3 rules targeting (N), (ADJ) and (PCP) separately).

3.4.4 CONTEXT:

One or more contexts can be used, but (heuristic) rules without any context are allowed, too. Each context is enclosed in parentheses. Contexts are applied as AND-linked conditions, i.e. all conditions of a given rule must be true ("instantiated") for the rule to apply. A context condition may contain the following elements:

(1) An obligatory **position marker,** consisting of a number indicating relative distance in tokens. The default (positive number) is a right context, while a negative number indicates a left context. A context can be negated by using **NOT** in front of the position marker. For *LINK'ed* contexts (cp. below), **NOT** negates (*only)* the immediate context conditions (to which the adjacent position marker applies), while **NEGATE** is used to ”open a negation bracket”, where the remaining *(LINK'ed)* contexts are negated as a whole.

(2) A**n asterisk (*),** prefixed[1] to the position marker number means "unbounded context". In this case, a context condition has to be true all the way to the left (-) or right (+) sentence boundary – even if the context search should cross the TARGET position **(position 0)**[2]. A positive unbounded context condition is instantiated at the closest possible position – unless a **double asterisk (**)** is used, which will allow instantiation at the second or a later occurrence. Later instantiation is relevant only in the presence of LINK'ed contexts (which might not be true of the first, but rather of a later occurrence of the original condition). An **at-sign (@)** in front of a position number means absolute

1 Prefixing is the traditional convention, but theoretically the asterisk may appear anywhere in the position block.
2 In CG1, unbounded searches were not allowed to back-cross the 0-position, so in order to facilitate porting of older grammars, CG3 support a --no-pass-origin flag to emulate this behaviour.

context, e.g. @1 for the first token/cohort, @2 for the second, and @-2 for the second-but-last token/cohort in the sentence.

(3) In CG-3 it is possible to search for the same context both left and right at the same time. This is called the **nearest neighbour** test, and is expressed by using the ”magic” position 0, e.g. (NOT *0 VFIN) to exclude finite verbs in the whole sentence

(4) An obligatory **context condition** consists of a (position-restricted) set (or set-ified tags or tag sequences). As elsewhere, sets may be combined by set operators: **OR** (or '|', union), **+** (concatenation in <u>one and the same</u> reading line). The old Vislcg **AND** (or '+', intersection, both tags in the same cohort, but not necessarily in the same reading), has been deprecated in favour of the equivalent *LINK 0*.

(5) A **C (careful)** condition attached to the position number means that the context condition has to be a safe (i.e. the only) reading of the cohort in question. For instance, (-1C N) denotes an unambiguous noun one position to the left (i.e. left adjacent). A word with both a noun (N) and a verb (V) reading in this position would not fulfill the context condition. Note that in connection with an unbounded search, a C condition may make the search “jump” ambiguous occurrences of the same context, a potentially unintended behaviour that however can be blocked by a BARRIER condition for the same context (cp. (7) below), i.e. *1 (X) BARRIER (X). In other words, without the BARRIER *1C (X) behaves like **1 (X) LINK NOT 0 (*) - (X).

(6) An optional **linked context,** where the word **LINK** chains 2 contexts (within the same context parenthesis). The second, linked context condition is written in the same fashion as the first one, but its relative position is calculated from the instantiated first context rather than the rule target. In other words, each LINK resets the context position to 0. In this way, it is possible to create arbitrarily long chains of LINK'ed context conditions. In practice, most links in a chain point to the same side (i.e. either right or left), but a change of direction is perfectly in order.

(7) An optional **barrier context,** where the word **BARRIER** is used right after an unbounded context (*-context). A barrier context blocks the preceding context search, if the barrier condition is instantiated before the unbounded context can be instantiated. As usual, barrier contexts may consist of sets, set-ified tags or set combinations, but do not need a position marker. For instance, (*1 VFIN BARRIER CLB) looks for a finite verb (VFIN) anywhere to the right (*1), but only if there is no interfering clause boundary (CLB) in between. A subordinator or comma would thus block further VFIN-searching. The BARRIER keyword can be used in careful mode, too **(CBARRIER)**, where only unambiguous readings will block the search. For NEGATEd contexts, CBARRIER is the recommended option.

(8) In order to continue a context search across window boundaries, use **Span Left** (<) and **Span Right** (>) as a pre- or postfix for the position block, e.g. <*-1 (left) or >*1 (right). Using **'W'** instead of the arrows will allow a span to search in both directions. As a default, the span covers 2 windows left and 2 windows right of the focus window, but the number can be set arbitrarily with the --num-windows command line flag. For instance (*-1 >>> LINK -1W (”<:>”) LINK -1 V-QUOTE) will check if the preceding sentence ends in a colon, after a quoting verb, making the second sentences a quotation ”object” of the first.

3.5. Mappings

A MAPPING-rule has the following general layout:

OPERATION *<u>(MAPTAG-1 MAPTAG-2 ...)</u> (TARGET)* ***IF*** *(CONTEXT-1) ... (CONTEXT-n)*

The following rule, for instance,

```
MAP (@SUBJ> @ACC>) TARGET N OR (PERS NOM)
        IF (NOT *-1 NON-PRE-N) (1C VFIN) ;
```

will map potential subject and accusative object tags onto nouns an personal pronouns in the nominative, if there are no non-prenominals to the left (i.e. if the np in question is the first in the sentence), and if a safe (C) verb follows immediately to the right.

Mapping rules add tags to a cohort line (i.e. reading), if that line contains a certain TARGET tag or matches a certain TARGET set, and if certain optional CONTEXTs are fulfilled. Context conditions are expressed as in the CONSTRAINT section, and sets are used and constructed in the usual way. Any kind of tag may be added. However, only mapped tags with a special mapping-prefix (by default, @) will be treated as real *mapped_tags*. *Mapped_tags* are traditionally syntactic tags, added and disambiguated *on the surviving cohort line* after morphological disambiguation (the line itself representing a PoS/inflexion reading), but can be used in many other ways:

1. Mapping of secondary tags that may be necessary for morphological disambiguation, but are context dependent and cannot, therefore, be added from the lexicon. Candidates are auxiliary/main verb markers, punctuations correction, marking of multi-word expressions.
2. Mapping of higher level tags, like semantic role tags or named entity tags.
3. Early syntactic mapping, before morphological disambiguation, to exploit syntactic constraints for indirect part-of-speech constraining.

During disambiguation, @tags will be cut down to the last reading *on a given line*. If there is only one reading line in the cohort, this last @tag is untouchable[3], otherwise the whole reading line dies together with its last @tag. When calling a grammar with Vislcg3, the @-prefix may be changed by using the --prefix ... flag.

The following OPERATIONs are allowed in mapping rules:

- MAP: This is the general mapping operator. It is a feature of the special @tags, that MAP rules cannot apply if the targeted cohort line already contains one or more @tags (from an earlier MAP rule or the lexicon). Thus, if ambiguity is desired, the @tags in question have to be MAP'ed at the same time (i.e. by the same rule). In order to allow further mapping, ADD rules have to be used instead of MAP rules.
- **ADD**: Mapping of @tags is performed independently of the presence of other @tags on the cohort line. Thus, @-mapping may continue until a MAP rule "closes" the @tag-list for a given cohort line. Hoever, @tags in the input, from the lexicon or an earlier CG module, will block both MAP, ADD and REPLACE readings.

---

3 This restriction may be overridden, if necessary (REMOVE UNSAFE or UNMAP), and special rule flags will allow the grammarian to refer to deleted MAP readings.

- **REPLACE**: This is a CG-2 operator retained in Vislcg and CG-3, but often disused by grammarians in favour of the new and more powerful SUBSTITUTE operator. REPLACE deletes all tags but the first one (normally the lexeme tag), and adds the mapped tags instead.

Unlike constraint rules, mapping-rules are applied exactly once and mapping rule sections are not rerun together with higher order constraint sections. And since mapping rules are applied before other rules, they are located together within a MAPPING section (also called BEFORE-SECTIONS in CG-3)..

3.6. Corrections/Substitutions

Correction rules (or – more neutrally – substitution rules) were originally introduced to correct faulty input – for instance from a probabilistic tagger or an earlier CG, or in a spell/grammar checker – by replacing tags with other tags. Deletion can be handled by nil-replacements (*), and insertion by replacing a tag with an appended version containing both the old and the new, inserted tag.

The general shape of a correction rule is the following:

SUBSTITUTE (TAG-1) (TAG-2) TARGET (TAG-3) IF (CONTEXT-1) ... (CONTEXT-2)

Here, TAG-1 is replaced with TAG-2 in cohort lines that contain the target tag TAG3 with (optional) context conditions structured in the usual fashion. As usual, on-the-fly sets (as in the example) can be used on par with predefined or combined sets. For instance, the following rule:

SUBSTITUTE (PRON) (CONJ) TARGET (”that”) IF (-1 T:pp LINK -1 <v-speak>)

corrects a pronoun-”that” reading (e.g. from a probabilistic PoS tagger) into a conjunction.-”that” reading, if ”that” is preceded by a prepositional-phrase template (T:pp) and a speech verb: *He had <u>promised on an earlier occasion that</u> he would not interfere.*

Substitution rules are also a work-around for changing tag lines after ”closure” with a @-tag. Thus, even @tags themselves can be changed, removed or appended in this way:

SUBSTITUTE (@SUBJ) (@SUBJ @ACC) TARGET (N) (-1 >>>) (1 (PERS NOM)) ;

(a previously safe noun subject is assigned object ambiguity at sentence initial position if the next word to the right is a personal pronoun in the nominative: 'Fish I do like.')

For deletions, use SUBSTITUTE (deletable) (*) TARGET ...[4]

For adding secondary tags, use SUBSTITUTE (PoS) (<secondary> PoS) TARGET ..., where *PoS* is a part-of-speech tag. Since the PoS tag is conventionally the first primary (morphological) tag in a cohort line, this will ensure that the new secondary tag is placed correctly between lexeme/lemma tag and morphological tags.

Ordinaryly, SUBSTITUTE rules are used in the mapping section, to be run once. However, they can be used in ordinary sections, and will then be repeated like ordinary rules. This may be the desired

4 The (*) denotes a magical ”deletion tag”, that could also simply be empty, ().

behaviou, if a SUBSTITUTE depends on a safe context which will only materialize as disambiguation progresses. However, when using SUBSTITUTE rules in ordinary sections, you will have to make sure that endless loops are prevented, since two SUBSTIUTE rules, such as a deleting and an inserting one, may cancel each other's effects.

It is possible to substitute a tag chain rather than a single tag, but conflicts may arise between the grammarian's expectation of fixed word order on the one hand, and CG-3's internally free tag order on the other hand. If two non-adjacent tags are included together in the from part of a substitution rule, you should expect all tags between them to disappear, too.

4. Dependency

Traditional Constraint Grammar syntax can be described as flat dependency syntax, with directional attachment markers at least at the group level, and in newer grammars also at the clause constituent level, allowing postprocessing with a dependency generator. CG-3 is the first public Constraint Grammar implementation that allows direct reference to dependency links, as well as from-scratch insertion of dependency arcs.

Dependency tags have the form *#n->m* or *#n→m*, where *n* is the daughter token id and *m* is the mother token id. Annotated input data has to adhere to this convention in order to be accessible to the CG rules.

There are 3 possible dependency references, to be used instead of the ordinary position markers in contexts conditions:

p (parent, mother)
c (child, daughter)
s (sibling)

ADD (§AG) TARGET @SUBJ (p V-HUM LINK c @ACC LINK 0 N-NON-HUM) ;

(Add an AGENT tag to a subject reading if its parent verb is a human verb that in turn has a child accusative object that is a non-human noun.)

In order to add dependency annotation to ”virgin” input, the operators SETPARENT and SETCHILD are used together with a TO target. Thus,

```
SETPARENT @FS-<ACC (*-1 (”que”) BARRIER CLB
    TO (**-1 <mv>  LINK 0 V-COGNITIVE)  (NOT 1 @<ACC);
```

will link a finite object clause (@FS-<ACC) with a que-complementizer to a main verb (<mv>) anywhere to the left (**-1) if the latter is a cognitive verb (V-COG) and is not followed by an ordinary direct object (@<ACC). If the sub-clause and main-clause verbs have the token id's #10 and #5, the result will be the following dependency tag:

... VFIN ... @FS-<ACC #10->5

Note that both the SET-target and the TO-target can have their own **independent** context conditions, counting from their respective positions as zero. Attachment will thus be made to the final match of the first context (i.e. parenthesis) after TO, while any further contexts after TO will relate to attachment position as zero, *not* to the original zero of the SET-target.

CG3 has a built-in check against dependency loops, preventing SETCHILD from attaching if doing so would create a loop. Instead, the rule will search onward for a valid, free parent that does not form a loop relation with the target. A corresponding precaution is valid for SETPARENT. This behaviour can be overridden with the ALLOWLOOP or the NEAREST options. The former will opt for the last matching TO-target, the latter for the first.

Dependency relation operators can be combined with a number of options:

- ***** (Deep scan) allows a child- or parent-test to continue searching along a straight line of descendants and ancestors, respectively, until the test condition is matched or until the end of a relation chain is reached. Departing from subjects or objects, for instance, '*p VFIN', will find the finite verb in the parent verb chain, even if the subject or object itself is linked to a non-finite main verb.
- **ALL** or **C**[5] requires a child- or sibling-relation to match *all* children or *all* siblings, respectively. Note that this is different from the ordinary C (= safe) option which applies to readings. Thus 'cC ADJ' or 'ALL c ADJ' means 'only adjectives as children' – e.g. no articles or pp's, while 'c (*) LINK 0C ADJ' means 'any one daughter with an unambiguous adjective reading[6].
- **NONE** or **NOT**[7] has the opposite effect of ALL - it means, that *no* child, or *no* sibling, may match. Note that the option will negate the *whole* dependency. Thus, 'NONE c @>N' means that there is no premodifier child, i.e. that all children are not premodifiers. If you want to find *a (just one)* daughter that does not match, the format is 'c (*) LINK NOT 0 @>N', and the context will be true even if there is another daughter that *is* a premodifier.
- **S** (Self) can be combined with c, p or s to look at the current target as well. For example, 'c @SUBJ LINK cS HUM' looks for a human subject np – where either the head noun (@SUBJ) itself is human, or where it has a modifier that is tagged as human.

Dependency tags may be referred to even if they are imported as part of the import cohorts, created either by a non-CG module or by an earlier CG module in a grammar pipe. In this case newly assigned dependencies will override old ones, but the input token numbering (and hence, sentence separation) will be maintained in spite of the fact that the CG3 compiler internally uses a running token numbering across sentence boundaries.

5. Other relational links than syntactic dependency

5 Because of the confusability of C, ALL is now the recommended form.

6At some stage, we intend to reserve 'C' to mean 'LINK 0C ADJ', so the combination 'ALL cC ADJ' will be meaningful.

7 NONE is the recommended form, and NOT is deprecated here, because it may be confused with ordinary NOT

The default relation between tokens is the dependency relation, but the CG-3 formalism also allows to add secondary dependencies, or other relations like anaphora relations, discourse relations, secondary ”semantic” dependencies etc. This is done using *named* relations:

one-way relations:
SETRELATION (name) TARGET targetset [context] TO link-context [IF context]
two-way relations:
SETRELATIONS (name) (name) TARGET targetset [context] TO link-context [IF context]

For instance, the following will set an **”identity”** relation from a relative pronoun to a noun occurring earlier in the sentence:

SETRELATION (identity) TARGET (<rel>) TO (*-1 N) ;

This will yield the following as an additional tag on the pronoun reading: ID:n R:identity:m, where *R:* introduces the relation name, *n* is the ID of the pronoun, and *m* the ID of the noun. The two-way operator SETRELATIONS, with two label brackets, one for each end of the relation arc, can be used to mark a given relation on both ends with different names, e.g. an *experiencer-stimulus* relation.

SETRELATION(S) overrides (removes) any previous relation(s) with the same name. For multiple relations of the same name, use ADDRELATION (one-directional) and ADDRELATIONS (two-directional) instead.

In order to remove relations individually, use REMRELATION or REMRELATIONS.

Note that CG3 can use contexts across several sentence windows, and thus assign long range relations, such as cross-sentence subject anaphora (cp. Bick 2010[8]).

Just like dependencies, other relation types will be visible to ordinary rules, and can be referred to in subsequent contextual tests. To do so, use 'r:name' for the positional test, e.g. (r:identity HUM) to check if a pronouns antecedent is human. The ALL or NONE operators are also allowed with relational tests, and may be useful, depending on the type of relations you are working with.

8 Bick, Eckhard (2010), A Dependency-based Approach to Anaphora Annotation, in: (eds.) *Extended Activities Proceedings, 9th International Conference on Computational Processing of the Portuguese Language* Apr. 27-30. Porto Alegre, Brazil. pp. xxx. ISSN 2177-3580 (Extended, original version: http://visl.sdu.dk/~eckhard/pdf/PROPOR2010_anaphora_submit.pdf)

6. Interfacing with other descriptive systems and parsing methodology

One of the design goals of CG3, and a motivation for continued development, is the desire to allow Constraint Grammar to not only support its native descriptive paradigms, functional dependency grammar and topological methods, but also to emulate other descriptive systems and their parsing methodologies. From a principled point of view, three competitors have to be considered – (a) generative grammar with a constituent tree notation, (b) unification grammar and (c) probabilistic and machine learning.

CG3 addresses (a) in two ways: First, its deep dependency annotation simply allows the transformation into constituent trees, creating various treebank formats – like the one used in the PENN treebank – on the fly. An example of a program performing such a transformation, is the author's *dep2tree,* supporting the VISL convention of constituent trees notation. Second, the core idea of generative parsing – constituent bracketing – can in part be simulated using so called *TEMPLATEs*.

On the other hand, CG3 was inspired by unification grammar (b) to allow the unification of set variables across targets and contexts. Finally, probabilistic methods (c) have also been accommodated for in the new formalism, by allowing reference to numerical tags expressing statistical information learned from raw or annotated corpora.

7. Templates

TEMPLATEs are labels for complex contexts conditions, which – once defined – can then be used by many different rules, or even in other templates. For instance, an np could be defined as

(a) TEMPLATE np = (? ART LINK 1 N) OR (? ART LINK 1 ADJ LINK 1 N)

(b) TEMPLATE np = ([ART, N]) OR ([ART,ADJ,N])

(c) TEMPLATE np = ? ART LINK *1 N BARRIER NON-PRE-N

and then referenced as

(*1 VFIN LINK *1 T:np).

Note that templates can be defined either as a consecutive list of sets (b), in angular brackets, or as a LINK'ed context (a), CG-style, in ordinary context brackets. Though the former is more reminiscent of generative rewriting rules, the latter is more powerful, since it will allow complex unbounded links (* - links) and thus can cover more cases in one and the same expression (c).

The linguistic motivation behind templates is to allow direct references to constituent units, just as in generative grammar. Thus, classical *constituent templates* are designed to reduce constituents to ”terminals” – on par with cohort tags and sets, and will be used like the latter, forming contexts together with an ordinary external position marker, as in the example above.

However, CG-internally, templates could also simply be interpreted as shorthand (variables) for context parentheses, so-called *context templates*. As such, they logically need to allow *internal*, predefined positions, as in the following example for a human verb-template, where the motivation is not a constituent definition, but simply to integrate two context alternatives into one[9], and to label the result with one simple variable.

9 In traditional CG, this OR'ed expression could not even be expressed in one rule, let alone referenced as one label.

TEMPLATE v-hum = (c @SUBJ + HUM) OR (*1 (”that” KS) BARRIER V)

Compiler-internally, both template types are processed in a similar way, which is why constituent templates have question marks or 0-positions as place holders for an external position marker, which will be inserted into the template by the compiler at run-time (”position override”).

Constituent templates allow a direct conceptual transfer from generative rules. Thus, a simple generative np grammar:

np = adjp? n pp? ;

adjp = adv? adj ;

pp = prp np ;

could be expressed in CG3 as:

TEMPLATE adjp = ((ADJ) OR (ADV LINK 1 ADJ)) ;

TEMPLATE pp = (PRP LINK 1 N) OR (PRP LINK 1 adjp LINK 1 N) ;

TEMPLATE np = ((N)

OR (T:adjp LINK 1 N)

OR (T:adjp LINK 1 N LINK 1 T:pp)

OR (N LINK 1 T:pp)) ;

Note that order is important - if a template uses another template that has not been defined yet, the compiler will die. As a consequence of this, so far, CG3 does not allow cross-definition recursion in templates[10]. Therefore, either a pp definition cannot itself contain an np template, or an np definition cannot itself contain a pp template, i.e. in our example, the following cannot be used:

not possible: TEMPLATE pp = (PRP LINK 1 T:np) ;

Like sets, templates can be combined, or even defined on-the-fly inside a rule, though at the time of writing this feature has only be implemented for *context templates*, not for *constituent templates*. Note that a separate set of parentheses is required for each level of template definition definition. For instance, if you want a rule to act on either a left-hand human subject or a right hand object clause (not possible in cg2), you can do so with:

SELECT (<v-hum>) ((*-1C HUM + @SUBJ>) OR (*1C @FS-<ACC)).

Here, two on-the-fly templates are OR'ed and encapsulated with a common set of parentheses. The bidirectional *0 search is a special case of this, but only works with identical conditions both left and right: REMOVE (VFIN) (*0C VFIN BARRIER CLB) ;

When using templates together with (external) BARRIER's, the template can be thought of as one token – meaning that right-looking contexts with a template (*1 T:x BARRIER ...) will be interpreted against the left edge of the template, while left-looking contexts (*-1 T:x BARRIER ...) will be interpreted against the right edge of the template so as to avoid internal, unpredictable parts of the template itself to trigger the BARRIER condition. Similarly LINK'ed conditions after a template will depart from different edges of the template instantiation in the sentence, according to whether the

10 The only type of recursion currently supported is direct recursion withing the same definition, e.g. TEMLATE n-chain = (N) OR (N LINK 1 N) OR (N LINK 1 T:n-chain)

template was found by a right-looking or left-looking search. In the latter, a LINK will continue from the left edge, in the former it will depart from the template's right edge[11].

8. Set unification

CG3 allows the use of sets as to-be-unified variables, prefixing $$ before the set name. All occurrences of such a set in a given rule will be unified to mean the same set member, and the rule operation will only apply if the set does have a member that satisfies all occurrences of the set in both target and contexts at the same time. Note that in ordinary mode the set is instantiated the first time it is met by the rule parser: If this is not in the target but in a context position, the rule may be interpreted unintuitively because the rule compiler does not normally respect context ordering, trying to optimize it for other goals such as speed. Therefore, if a $$-set only occurs in contexts (and not in the target), the KEEPORDER option should be used.

The following is an example for the unification of semantic roles (agent, patient, theme and location), in a grammar with '§' as the mapping-prefix:

```
LIST ROLE = §AG §PAT §TH §LOC ;
SELECT $$ROLE (-1 KC) (-2C $$ROLE) ;
```

Set unification could be simulated in CG2 only at the cost of considerable rule explosion. Thus, our example would have to be written with 4 rules instead of one:

```
SELECT (§AG) (-1 KC) (-2C (§AG)) ;
SELECT (§PAT) (-1 KC) (-2C (§PAT)) ;
SELECT (§TH) (-1 KC) (-2C (§TH)) ;
SELECT (§LOC) (-1 KC) (-2C (§LOC)) ;
```

So far, unification tags can only be used in targets and contexts, but not mapped:

not possible: MAP $$ROLE TARGET (N) IF (-1 KC) (-2 N + $$ROLE) ;

For many languages, a frequently used low-level unification concerns gender, number and case in np's:

```
LIST GNC = (M S NOM) (F S NOM) (M P NOM) (F P NOM) (M S ACC) (F S ACC) (M P ACC) (F P ACC) ; # with M = male, F = female, S = singular, P = plural, NOM = nominative ACC = accusative
SELECT ADJ + $$GNC (*1C N + $$GNC BARRIER NON-ATTR) ;
```

Top level set unification: In some cases, especially ontology tag types, one may want to unify not individual tags, but groups of tags, e.g. match hyponyms through their hypernym umbrella tag (or what WordNet would call synsets). The following is an example involving VISL's semantic prototype tags:

```
LIST <hum> = <H> <Hprof> <Hfam> <Htitle> <Hideo> ;
LIST <animal> = <A> <Azo> <Aorn> <Aich> <Aent> ;
LIST <tool> = <tool> <tool-cut> <tool-shoot> <tool-tie> ;
LIST <food> = <food> <food-h> <fruit> <drink> ;
SET SEMS = <hum> OR <animal> OR <tool> OR <food> ;
```

11 The ”relevant edge behaviour” of templates with external BARRIERs and LINKs is internally achieved in a somewhat different way, but is a functionally correct description of surface rules.

Such higher level sets, consisting not of tags or tag strings, but of a combination of other sets, can also be unified, but have to be marked with a && prefix (rather than a $$ prefix) when used in rules:

```
LIST @<FUNC = @<ACC @<SC @<OC @<SUBJ ;
SELECT $$@<FUNC (0 &&SEMS)  (*-1 (“and”) BARRIER NON-PRE-N LINK -1C $$<FUNC LINK 0 &&SEMS) ; # selects a left pointing coordinated function (rather than opting for a clause break and a right-pointing function), if 2 np's are of the same semantic class and separated only by “and”
```

The above is equivalent to 4 separate rules using $$<hum>, $$<animal>, $$<tool> and $$<food>, respectively, instead of the top level SEMS set.

9. Numerical matches

For the first time, the new CG-3 formalism allows flexible integration of statistical data, frequency thresholds and confidence values directly in the Constraint Grammar framework – a feature CG2 and Vislcg could only approximate through the use of <Rare> sets and heuristic section chunking of grammars. The option should be paving the way for hybrid systems integrating both hand-crafted linguistic rules and raw probabilistic corpus data.

A numerical is of the type <TYPE:number>, where *TYPE* is a label, and *number* is an integer assigned to *TYPE*. Examples are lexical frequencies drawn from a corpus, or confidence values in a spell or grammar checker. CG contexts and targets can make use of the numerical tags with either =, < or > plus the combinations >= and <=. Thus, on a relative lexical probability scale between 0 and 100:

```
REMOVE (<f<10> N) (0 (<f>60> V)) (1 N) ;
```

will remove noun readings with a lower-than-10% probability in the presence of a higher-than-60% probability for a verb reading, if there is another noun candidate immediately to the right.

Minimum and maximum values can be selected or removed from a cohort by using MIN and MAX, respectively. In its simplest form, this feature can be used as a last heuristics, after ordinary rules:

```
SELECT (<f=MAX>) ;
```

or

```
REMOVE (<f=MIN>) ;
```

Note that the former will also remove readings with no f-value given, while the latter will keep them, following the intuitively most likely interpretation of the rules purpose.

10. Regular expressions

Another innovation in CG-3 is the use of regular expressions for word forms, base forms and secondary tags (<...> angle-bracketed tags). An interpretation of a tag as a regular expression is forced by appending a 'r' after the tag:

- ".*ize"r to match certain transitive verbs in English
- <[HA].*>r to match semantic prototype tags for *animates*, i.e. *humans (e.g. <Hprof>)* and *animals (e.g. <Aorn>).*

Another literal string modifier, used in the same fashion as 'r', is 'i', indicating case-insensitivity. The two can be combined, e.g. ".....”ir .

11. Grammar-text interaction

There are numerous possibilities in the CG-3 formalism to influence the interaction of the CG grammar and its input data. Thus, parameters can be set in the grammar or command-line for

- triggering the use or non-use of certain rule sections
- changing window delimiters on the fly
- naming and referencing rules
- setting external, corpus-driven parameters on the fly, such as domain or genre

For these and other options, please refer to our CG-3 page on http://beta.visl.sdu.dk/constraint_grammar.html where you can also find a CG laboratory interface to test some of the options in this tutorial.

12. Tracing

Tracing of rule applications on a text texts allows the grammarian to debug his grammar.

As a default, rules are traced using their line number, but an optional rule name can be added to each rule operator with a colon, e.g. REMOVE:rule_name. With the --trace-name-only command line option, line numbers will be suppressed for named rules.

In ordinary tracing mode, removed lines will still be shown, but prefixed with a semicolon. To prevent this behaviour, and see only surviving lines, use --trace-no-removed.

13. Binary grammars

The CG3 rule compiler can build *binary grammars*, rather than parse rules from scratch each time. The two main advantages of binary grammars are (a) speed and (b) data protection, for commercial applications.

vislcg3 -C UTF-8 -g *rulesfile* --grammar-only --grammar-bin *binfile*

A Perl support tool, *cg3-autobin.pl* – to be used instead of the ordinary *vislcg3* command, with the same command line options – will compile a grammar to binary form the first time and re-use that on subsequent runs for the speed boost.

14. Sample rules file

The following is a sample file for a Portuguese Constraint Grammar. with the classical sections of delimiters, sets, mappings and constraints. Note that the mappings sections is sandwiched between two constraint sections – the first for part of speech and morphology, the second for syntax, the idea being that mapping rules would apply to partially disambiguated cohorts rather than all readings. However, since MAP rules are run before ordinary disambiguation rules, the placement of the MAPPING section by itself does not have the desired effect. Rather, it is necessary to run the same grammar twice, as consecutive modules with different section options, allowing mapping and post-mapping disambiguation only in the second round:

cat cohort-file | vislcg3 -g grammar --sections=1 --no-mappings | vislcg3 -g grammar

For more complex grammars, this effect is usually achieved by using a multi-grammar architecture, chaining separate grammars for different levels of annotation. In this case, set definition sections can be shared with the INCLUDE option, which will cause the compiler to load an external file as if it had been pasted in on the INCLUDE line:

INCLUDE set-section;

****

```
DELIMITERS =  "<$.>" "<$!>" "<$?>" "<$\;>" "<$¶>" ; # sentence window
```

SETS

```
LIST ALL = N PROP ADJ DET PERS SPEC ADV V PRP KS KC IN ; # all word classes (but
  not punctuation)

LIST NOMINAL = N PROP ADJ (PCP2 STA) ; # nominals, i.e. potentieal nominal heads

LIST PRE-N = ART DET NUM ADJ STA ; # prenominals

LIST NON-PRE-N = (*) - PRE-N ;

LIST NON-PRE-N/ADV = (*) - PRE-N - (ADV) ;


LIST P = P S/P ; # plural

SET PRE-N-P = PRE-N + P ; # plural prenominals, equivalent to (ART DEF) (DET P)
  (DET S/P) (NUM P) ADJ (PCP2 STA) ;


LIST VFIN = PR PAST IMP ;

LIST VV = PR PAST INP INF AKT PAS ;


LIST CLB = "<,>" KS (ADV <rel>) (ADV <interr>) ; # clause boundaries
```

```
LIST V-SPEAK = <vq> <Vcog> <speak> "answer" "say" "tell" ; # speech verbs
LIST @MV = @FMV @IMV &MV ; # main verbs


CONSTRAINTS


REMOVE (N P) IF (-1C PRE-N) (NOT -1 PRE-N-P) ; # remove a plural noun reading if
   there is a safe prenominal to the left that is not compatible with a plural
   reading


REMOVE VFIN OR INF (-1C ART OR (GEN)) ;
REMOVE VFIN OR INF (*-1C ART OR (GEN) BARRIER NON-PRE-N/ADV) ; # remove finte verb
   and infinitive readings if there is an article to the left


REMOVE VFIN IF (*1 VFIN BARRIER CLB OR (KC) LINK *1 VFIN BARRIER CLB OR (KC)) ; #
   remove a finite verb reading if there are to more finite verbs to the right none
   of them barred by a clause boundary (CLB) and coordinating conjunction (KC).


"<that>" SELECT (KS) (*-1 V-SPEAK BARRIER ALL - (ADV)) ; # select the conjunction
   reading for the word form 'that', if there is a speech-verb to the left with
   nothing but adverbs in between.
```

MAPPINGS

```
MAP (@SUBJ> @ACC>) TARGET (N NOM) (NOT *-1 NON-PRE-N/ADV) (*1C VFIN) ;
MAP (@<SUBJ @<ACC @<SC) TARGET (N NOM) (*-1 VVC BARRIER NON-PRE-N/ADV) ;
MAP (@P< @>N) TARGET (N NOM) (*-1C PRP BARRIER NON-PRE-N/ADV) ;
MAP (@>N) TARGET (N GEN) ;
MAP (@SUBJ> @<ACC @P< @<SC @>N) TARGET (N NOM) ;
```

CONSTRAINTS

```
REMOVE (@SUBJ>) IF (NOT *1 VFIN) ; # remove a forward subject if there's no finite
   verb to the right
REMOVE (@SUBJ>) IF (*1 CLB BARRIER VFIN) ; # remove a forward subject if there's no
   finite verb to the right
```