Using Constraint Grammar for Chunking

Eckhard Bick
University of Southern Denmark, Odense
eckhard.bick@mail.dk

Abstract

This paper presents and evaluates a novel and flexible chunking method using Constraint Grammar (CG) rules to introduce chunk edges in corpus annotation. Our method exploits pre-existing (non-constituent) morphosyntactic annotation such as part-of-speech or function tags, but can also be made to work on raw text, integrated with other CG modules. The first version of the chunker was developed for German CG-annotated interview data, with a parallel English version derived from the German one, indicating a high degree of language-independence of the rules in the presence of generalized syntactic-functional tags (e.g. subject, object, modifier). Two different approaches are discussed, one for minimal, flat chunking, the other for deep, nested chunking. The system has a reasonable performance and robustness for both, achieving F-scores of 89.1 and 97.4 for nested and minimal chunking, respectively. Xml markup is supported, and with a full set of rules, the tool can be used to convert CG annotation into complete constituent trees in VISL or TIGER format.

Keywords: Chunking, Constraint Grammar, Syntactic Constituent Trees

1 Introduction and related research

In its simplest NLP meaning, chunking can be defined as a shallow parsing method where the edges of syntactic groups are marked, but where the internal structure, head-dependent relations and syntactic function is ignored. When defining the term, Abney (1991) cited psychological (processing) evidence for the linguistic relevance of chunks and used the expression “a single content word surrounded by a constellation of function words, matching a fixed template”. In the original sense, chunks are minimal syntactic units with a recursivity restriction, and chunks of different syntactic type will be shown as chained rather than layered - in particular, no (simple) chunk type will be allowed to span across a (more complex) daughter chunk, and subclauses are not made explicit as chunks. Thus, in minimal chunking np's will not contain postnominal pp's or relative clauses. If a chunk does swallow another chunk, the latter will lose its edges. Depending on linguistic design, this may occur in the handling of prepositions or quantifying adverbials.

Minimal chunking is often used as an intermediate step in NLP, after part-of-speech (POS) tagging, and before deeper structural or functional analysis. Thus, Abney's chunk parser would first create a stream of such minimal chunks, then use an "attacher" to link words within chunks, and chunks to each other in order to create a complete parse tree. Kübler & Hinrichs (2001) use a similar 2-step method, but focus on syntactic function assignment as a vehicle to extend non-recursive chunks to full parse structures on the background of a treebank instance database. In our own approach, we implement a third strategy, where (syntactic) function comes before (syntactic) form, and links are created before chunks. Chunk edges are assigned based on functional relations, and chunking depth becomes a design option rather than a clear, methodologically desired, processing stage distinction. In the context of this paper, we will therefore extend the meaning of chunking to include progressively layered chunking, where nesting is allowed e.g. for np's or object clauses, and where a fully chunked sentence will ultimately be structurally equivalent to a PSG constituent tree.

Chunking is useful for tasks such as term and name extraction (Carreras and Màrquez 2005), information retrieval (Banko et al. 2007), topic screening and others. In such automatic analysis applications, minimal chunking has the obvious advantage of being more robust than layered chunking, being able to avoid complexity issues such as discontinuity, coordination and ellipsis, as well as circumventing free-word-order problems, while still supporting most aspects of the applicative tasks mentioned above. Unlike Abney's original system, the majority of automatic chunkers today are based on machine learning (ML) and trained on manually revised gold annotations such as treebanks. In a CoNLL shared task in 2000, the highest F-score for chunking was 93.5 (Tjong Kim Sang and Buchholz 2000), with relatively little performance variation across different ML techniques. There is evidence that with sufficient training data, similar results can be achieved with ML techniques even without the use of POS information (van den Bosch and Buchholz 2002), while finite state transducers (FST) have an upper bound F-score of 92 for the same task (Jurafsky and Martin 2009). The only rule-based systems in the CoNNL evaluation performed at the bottom of the field, with F-scores of 85.8 and 87.2.

Contrary to these findings, we believe rule-based chunking to have a considerable potential, and have chosen Constraint Grammar for the experiments reported here, a versatile and modular methodology based entirely on linguist-written rules. It is reasonable to assume that differences between rules sets, the expressive power of the rule formalism itself as well as its lexical support may amount to huge performance differences - maybe bigger differences than can be expected from different machine learners using the same training data, and this will be further compounded by the fact that rule-based approaches require a great deal of specialist labour and hence may suffer from project time and manpower constraints. Thus, when writing a chunking grammar, performance will crucially depend on the existence and quality of a morphosyntactic tagger to provide annotated input. What we intend to show and evaluate in this paper is exactly this - how rule-based chunking can mash with and exploit output from a CG tagger, in this case the morphosyntactic stage of the EngGram parser (available online at http://beta.visl.sdu.dk/visl/en/). Since it has already been shown that morphosyntactic CG tagging does support syntactic trees, either through a PSG layer (Bick 2003, for English) or an added external dependency grammar (Bick 2005, for Danish), it is not CG-*based* constituent bracketing as such that is the focus here, but rather the efficiency of our method and the fact that we are exploiting a novel CG feature (relational tags) to perform both dependency-linking and chunking *within* the CG formalism itself rather than as a hybrid add-on technique. This way, all types of existing CG annotation can be seamlessly exploited without loss of information, and with the full expressivity of contextual CG rules.

2 Adding Chunk Edges

Both the output and input of our chunkers follow the verticalized, 1-token-per-line format common in Constraint Grammar annotation. We have developed two different methods to add chunking information to this format, with different methodological advantages, which can then both be filtered into a common xml-style format. Both methods exploit recent improvements in the open-source CG3 compiler (http://beta.visl.sdu.dk/cg3.html), allowing so-called cohort insertion and named, bidirectional relational tags.

2.1 The Cohort Insertion Method

Constraint Grammar compilers traditionally use a fixed tokenization, where each token may have several readings (a so-called cohort), but without the possibility of changing the number, order or span of the tokens themselves. In our own implementation, however, we provide for the possibility of adding, moving and removing tokens, exploiting this feature for the insertion of chunk edges. In its simplest version, our insertion chunker uses 20 CG rules, first 12 rules (a,b) to insert different types of chunk-opening brackets (named for phrase type), then 8 rules (c) to insert matching chunk-closing brackets.

The first two examples open np chunks by adding a marker cohort before (left of) the first np element. Rule (a) looks for prenominal modifiers (@>N) or potential np-heads, i.e. nouns or pronouns/numbers provided that the latter do not have a function marking as prenominals (@>N) or predicatives (@PRED, @SC, @OC) which would indicate an adjectival reading. In order to make sure that the found np-element is in fact the np's left edge, there is a NOT condition excluding further prenominals (@>N) to the left, as well as adverbial pre-adjects (@>A) that might pre-modify the premodifiers themselves (e.g. *'very @>A high @>N taxes'*). The NEGATE condition, finally, provides for the exception of coordinated premodifiers[1]. Rule (b) addresses np-

[1] In this version of the chunker, we follow the principle, applied in the CoNLL 2000 shared task on chunking, that lower level coordination within a phrase is treated as chunk-internal, while coordination of phrase heads is treated as chunk-external.

internal adverbial adjects (@>A) in a similar way, again taking into account possible coordination of pre-modifiers.

(a) ADDCOHORT ("<$np>" "CHUNK" NP) BEFORE @>N OR N/PROP/PRON OR DET/NUM/PERS - @>N - @PRED - @SC - @OC (NOT -1 @>A OR @>N) (NEGATE -1 IT LINK -1 @>N) ;

(b) ADDCOHORT ("<$np>" "CHUNK" NP) BEFORE @>A (*1 @>N BARRIER NON-ADV - KC) (NOT -1 @>A OR @>N) (NEGATE -1 IT LINK -1 @>A - PRP OR @>N) ;

Similar rules exist for the other chunk types: adjective phrase (adjp), adverbial phrase (advp), prepositional phrase (pp), verb phrase (vp) and the minor classes of conjp (conjunction phrase), prt (particles), intj (interjection) that usually contain only a single word. Once a chunk is opened, a corresponding rule can insert an ENDCHUNK marker token after the last element in the chunk. Thus, rule (c) looks for potential np-heads with a NOT condition against them functioning as pre-modifiers (@>N) themselves (as would be the case in English noun chain compounds). To ensure exact bracket matching and as a safety measure, there is a condition looking left (*-1) for the corresponding chunk-opening token with a BARRIER condition for overlaps, i.e. other CHUNK markers.

(c) ADDCOHORT ("<$/np>" "ENDCHUNK" NP) AFTER N/PROP/PRON OR N/PROP/PRON OR DET/NUM/PERS - @>N - @PRED - @SC - @OC (NOT 0 @>N) (*-1 CHUNK-NP BARRIER CHUNK) ;

2.2 The Relation-Adding Method

The cohort insertion method is a very simple method, and works well and robustly for minimal chunking. However, in the face of more complex annotation needs, it has the shortcoming of not marking chunk heads as opposed to other chunk elements, and it is less well-suited for layered chunking, because of the risk of crossing brackets. The CG-compiler may lack sufficient structural information simply because opening and closing brackets are only inserted and not paired by links. Thus, in layered chunking, closing brackets in particular may accumulate after the same token, and bracket order will simply be by (inverse) rule order, making it very difficult for the grammarian to control this order, not least because CG rules can be reiterated if contexts change from false to true due to other rules being applied, and because opening and closing brackets have opposite ordering needs. To further complicate things, a more fine-grained, head-marking chunking scheme may run into cases of discontinuity, with a need for partial closing and re-opening bracket types raising ambiguity issues in complex cases.

All of these problems can be addressed simultaneously by exploiting another non-traditional CG feature, named relations, which we originally added for the sake of anaphora treatment and discourse structure. Using relational tags, chunk edges can either be linked to each other or to the chunk head, and in principle carry all information needed to configure a complete, classical constituent tree. In this approach, given sufficient structural information in the CG input annotation, chunking provides a conversion method between different functional dependency grammar on the one hand, and labeled constituent trees on the other. Users will be able to apply standard xml tools to manipulate, search, evaluate or visualize the resulting chunk structures because chunk brackets can be expressed as xml opening and closing markers.

We developed the relation chunker in the context of a joined annotation project for German and English transcribed speech corpora, and some design options are therefore project-specific, such as the decision to allow discontinuity (crossing branches), to provide for separate coordination chunks, and to only mark multi-word chunks. However, it has to be borne in mind that in a rule-based CG system, it is relatively easy to change such design parameters, without the need of manual re-annotation of a training corpus. In particular, the chunk type of single-word constituents is implicit in their word-class, and could be added with one rule per type.

The relations chunker uses 59 rules to establish relations between a constituent head and its leftmost and rightmost descendents (dependents, dependents of dependents etc.). A typical rule pair is shown below:

(a) ADDRELATIONS (np-head-l) (np-start) TARGET (*) (c @>N OR @N<&) TO (llScc (*)) ;

(b) ADDRELATIONS (np-head-r) (np-stop) TARGET (*) (c @>N OR @N<&) (r:np-head-l (*)) TO (rrScc (*)) ;

Rule (a) adds a left-edge relation between any (*) target word with a pre-nominal (@>N) or postnominal (@N<) modifier child (c), and the leftmost (ll) of its descendents[2] (cc - children & children's children). The S (self) provision allows for the head itself forming the chunk's edge, and the modifier condition prevents 1-word chunks. An ADDRELATIONS rule allows for two asymmetric relation names, given in the first two brackets of the rule. Here, *np-head-l* (np-head-leftlooking) is the relation name tagged on the head, and *np-start* is the name for the same relation seen from, and tagged on, the leftmost dependent.

Rule (b) adds the corresponding right-edge relation to the rightmost (rr) descendents (cc) of np-heads. Matching bracket counts are ensured by adding the condition that the target already has to carry a pre-existing *np-head-l* tag.

2.2.1 Discontinuity

The rules described above will identify external chunk edges by locating leftmost and rightmost descendents of a given head, but they cannot cope with internal edges caused by crossing dependency branches (constituent discontinuity). Therefore, further rules are needed, like the np-examples below. Rule (a) marks the end of a discontinuity "hole", with a head-edge relation named *np-head-ld* (left-oriented discontinuity edge) on the head, and *np-stop-d* (discontinuity stop edge) on the (right-located) internal edge dependent. The rule works by identifying existing, ordinary right edges (r:np-head-r (*)) and looks left of these (LINK *-1) for arguments, adverbials (@ARG/ADVL) or verbs (VV), implying that such function tags would break the continuity of an np chunk if they are not explicitly marked as embedded, i.e. if neither the break candidate itself (S) or any of its ancestors or parent (*p) is identical with the rule target (NOT *pS _TARGET_). If a chunk-breaker is found, the rule backtracks (x) to the outer edge and from there looks left (**-1xA) for the *last* word (**) that *does* have the rule target as parent-ancestor, or is identical with it (*pS), and attaches (A) the relation here.

(a) ADDRELATIONS (np-head-ld) (np-stop-d) TARGET (*) (c @>N OR @N<&) (r:np-head-r (*) LINK *-1 @ARG/ADVL OR VV LINK NOT *pS _TARGET_) TO (r:np-head-r (*) LINK *-1X @ARG/ADVL OR VV LINK **-1xA ALL-ORD LINK *pS _TARGET_) ;

[2] In principle, a leftmost ancestor dependency chain (llcc) can be quite complex because a right daughter dependent may have crossing left granddaughter dependents that are further left than the head itself or its left daughters.

Correspondingly, rule (b) marks a relation between a head and a left-located internal edge, named *np-head-rd* (right-oriented discontinuity edge) on the head, and *np-start-d* (discontinuity start edge) on the edge dependent. Again, to ensure matching bracket counts, the rule checks if the matching *head-ld* relation tag is already present on the head. This way, the head accumulates information about all edges controlled by it, while the chunk edge tokens themselves carry only one tag containing chunk type and head ID[3].

(b) ADDRELATIONS (np-head-rd) (np-start-d) TARGET (*) (c @>N OR @N<&) TO (r:np-head-ld (*) LINK **1A ALL LINK *pS _TARGET_) ;

2.3 Language independence

It is an interesting question to what degree a function-first, linking-before-chunking approach will lead to a higher degree of language independence in a chunking grammar. We believe this to be the case because the use of higher-level categories, at least if notationally unified, insulates the chunking grammar from language specific differences such as agreement features and word order. Though we do not yet have data for less related languages, our German and English grammars provide empirical support for this assumption. Thus, the original German layered chunker did work without rule modifications for English, and even the final, optimized English grammar (59 rules) needed rule changes or additions almost exclusively in areas, where the German grammar still had coverage problems, specifically coordination (10 rules) and vp discontinuity (6 rules). Only one rule had to be amended in a truly language-specific way, to account for discontinuous, fronted arguments of stranded prepositions in English, and 8 default bracket closing rules were added to check for matching brackets. Most importantly, all of the above English changes could be reexported into the German grammar almost as is, and even the language-specific stranded-preposition rule would do no harm - rather, it would simply not apply. As long as function tags and dependencies are defined in a unified way, this might be true for many other language pairs, too: Unimpeded by morphological or topological constraints, a pure function/relation reference in a chunking rule will either have the desired effect in the other language, or none at all.

3 Format conversions

Because all information is encoded locally as tags on tokens, Constraint Grammar output is easy to parse for format conversion programs, allowing such filters to extract information from several levels of tagging at the same time, with only one regular expression match. This way information can be made explicit that would otherwise be stated only implicitly, and html tags (for visualization), sgml tags (for corpus segmentation) or xml tags (for external tools) can be inserted before or after certain trigger tags or tag combinations. For the speech corpus annotation project, xml-style encoding of chunking information was the desired target format. With the cohort insertion method this amounted to simply turning chunk edge cohorts into <...> lines (Fig. 1), while the filter program for the relation-adding method had to insert xml tags before or after tokens carrying chunk edge labels (Fig. 3). Because this method was used to produce multi-layered chunking, the filter program also had to keep track of the xml tag nesting, i.e. arrange brackets in correct matching order, whereas the minimal chunks produced with the insertion

[3] Of course, if a head is situated leftmost or rightmost in its chunk, it will carry both types of tags.

method were simply defined - following CoNLL conventions - as non-overlapping and non-recursive, avoiding any bracketing complexities:

```
<chunk form="advp">
 So [so] <*> <aquant> ADV @ADVL>
<chunk form="/advp">
<chunk form="np">
 anyone [anyone] INDP S NOM @SUBJ>
<chunk form="/np">
<chunk form="np">
 who_ [who] <sam-> <rel> INDP S/P @SUBJ>
<chunk form="/np">
<chunk form="vp">
 _s [be] <-sam> <mv> V PR 3S @FS-N<
<chunk form="/vp">
<chunk form="adjp">
 familiar [familiar] ADJ POS @<SC
<chunk form="/adjp">
<chunk form="pp">
 with [with] PRP @A<
<chunk form="/pp">
<chunk form="vp">
 playing [play] <mv> V PCP1 @ICL-P<
<chunk form="/vp">
<chunk form="np">
 different [different] ADJ POS @>N
 types [type] <ac-cat> <idf> N P NOM @<ACC
<chunk form="/np">
<chunk form="pp">
 of [of] PRP @N<
<chunk form="/pp">
<chunk form="np">
 games [game] <game> <idf> N P NOM @P<
<chunk form="/np">
<chunk form="pp">
 through [through] PRP @<ADVL
<chunk form="/pp">
<chunk form="np">
 a [a] <indef> ART S @>N
 console [console] <tool> <idf> N S NOM @P<
<chunk form="/np">
<chunk form="vp">
 will [will] <aux> V PR @FS-STA
 be [be] <mv> V INF @ICL-AUX<
<chunk form="/vp">
...
```

FIGURE 1: Minimal chunks, insertion method, xml format

The format does not explicitly mark heads, but because we followed the CoNLL standard in only allowing left dependents, this does not amount to any loss of information - the head is simply the rightmost/last constituent of a multi-word minimal chunk.

For the sake of evaluation and comparability, we also provide a denser, non-xml format, with the <B> (beginning-of) and <I> (inside-of) tags used in the CoNLL evaluation. This is achieved by a couple of short CG rules, where (a) extracts the chunk type ("<.(.+)>"r) as a bracketed regular expression variable from an immediately preceding (-1) CHUNK opening cohort and remaps it as a <B> tag (<C:B-$1>), while rule (b) adds <I> tags with chunk type-information extracted from immediately preceding <B>- or <I>-tagged words. A third rule (c) maps an <O> tag (outside-of-chunk) to all remaining words. Since minimal chunking regards 1-word constituents as chunks, and all word classes are mapped onto chunk types, all instances of <O> amount to annotation errors in the CG input.

```
So             <C:B-advp> ADV @ADVL>             near          <C:B-pp> PRP @<ADVL
you            <C:B-np> PERS 2S/P NOM @SUBJ>     completely    <C:B-np> ADV @>A
might          <C:B-vp> V IMPF @FS-STA           *different    <C:I-np> ADJ POS @>N
*be            <C:I-vp> V INF @ICL-AUX<          *things       <C:I-np> N P NOM @P<
near           <C:B-pp> PRP @<SA                 that          <C:B-np> INDP P @SUBJ>
some           <C:B-np> DET S/P @>N              are           <C:B-vp> V PR -1/3S @FS-N<
*universities  <C:I-np> N P NOM @P<              completely    <C:B-adjp> ADV @>A
but            <C:B-conjp> KC @CO                *unrelated    <C:I-adjp> ADJ POS @<SC
you            <C:B-np> PERS 2S/P NOM @SUBJ>     to            <C:B-pp> PRP @A<
```

could	<C:B-vp> V IMPF @FS-STA	you	<C:B-np> PERS 2S/P ACC @P<
*end	<C:I-vp> V INF @ICL-AUX<	as=well	<C:B-advp> ADV @<ADVL
up	<C:B-prt> ADV @MV<	.	

FIGURE 2: Minimal chunks, insertion method, B/I/O format

Like most Constraint Grammars, our input CG marks a number of multi-word expressions (MWEs) as tokens, in particular certain complex prepositions, conjunctions and adverbs, as well as MWEs in the productive category of names. An example is the last word in Fig. 2 ('as well'), which however could simply be expanded by splitting on space, letting the first part of the MWE inherit the MWE tag, and adding <I> tags to all other MWE parts.

Fig. 3 shows a fully layered chunk tree for the same data used in Fig. 1, with layering depth shown as indented dots (. . .), to improve readability. CG lemma, inflexion, pos, function and dependency tags are retained, the latter employing a sentence-internal numbering scheme (#n->m tags). The relational, chunk edge-linking CG tags would be redundant, and are not shown after conversion into xml-format. In order to facilitate linguistic corpus searches, the xml chunk lines carry explicit feature-attribute pairs for (chunk) form, (head) function and head ID. The latter are - unlike dependency IDs - numbered across the whole corpus, because our CG compiler in principle allows relations across sentence boundaries, allowing for co-referent resolution, discourse annotation or text-level chunking.

```
<chunk form="fcl" function="STA" head="167">
. So          [so] <*> <aquant> ADV @ADVL> #1->16 ID:153
. <chunk form="np" function="SUBJ" head="154">
. . anyone  [anyone] INDP S NOM @SUBJ> #2->15 ID:154
. . <chunk form="fcl" function="N<" head="156">
. . . who_   [who] <clb> <sam-> <rel> INDP S/P @SUBJ> #3->4 ID:155
. . . _s       [be] <-sam> <mv> V PR 3S @FS-N< #4->2 ID:156
. . . <chunk form="adjp" function="SC" head="157">
. . . . familiar          [familiar] <close-2> <acquainted> ADJ POS @<SC #5->4 ID:157
. . . . <chunk form="pp" function="A<" head="158">
. . . . . with               [with] PRP @A< #6->5 ID:158
. . . . . <chunk form="icl" function="P<" head="159">
. . . . . . playing         [play] <mv> V PCP1 @ICL-P< #7->6 ID:159
. . . . . . <chunk form="np" function="ACC" head="161">
. . . . . . . different      [different] ADJ POS @>N #8->9 ID:160
. . . . . . . types          [type] <ac-cat> <idf> <nhead> N P NOM @<ACC #9->7 ID:161
. . . . . . . <chunk form="pp" function="N<" head="162">
. . . . . . . . of            [of] PRP @N< #10->9 ID:162
. . . . . . . . games       [game] <game>  <idf> <nhead> N P NOM @P< #11->10 ID:163
. . . . . . . </chunk form="pp" function="N<" head="162">
. . . . . . </chunk form="np" function="ACC" head="161">
. . . . . . <chunk form="pp" function="ADVL" head="164">
. . . . . . . through       [through] <advl-fs> PRP @<ADVL #12->7 ID:164
. . . . . . . <chunk form="np" function="P<" head="166">
. . . . . . . . a               [a] <indef> ART S @>N #13->14 ID:165
. . . . . . . . console     [console] <tool> <idf> <nhead> N S NOM @P< #14->12 ID:166
. . . . . . . </chunk form="np" function="P<" head="166">
. . . . . . </chunk form="pp" function="ADVL" head="164">
. . . . . </chunk form="icl" function="P<" head="159">
. . . . </chunk form="pp" function="A<" head="158">
. . . </chunk form="adjp" function="SC" head="157">
. . </chunk form="fcl" function="N<" head="156">
. </chunk form="np" function="SUBJ" head="154">
. <chunk form="vp" function="P" head="167">
. . will      [will] <aux> <cjt-first> V PR @FS-STA #15->0 ID:167
. . be        [be] <mv> V INF @ICL-AUX< #16->15 ID:168
. </chunk form="vp" function="P" head="167">
... (37 lines)
</chunk form="fcl" function="STA" head="167">
```

FIGURE 3: Layered "maximal" chunking, relational method, xml format

As can be seen from the example, this implementation of layered chunking does provide for right branching (e.g. postnominal pp's), and it can also handle discontinuity, marking left and right halves of a discontinuous chunk by adding right or left hyphens, respectively, to the chunk's form attribute:

```
<chunk form="fcl" function="STA" headid="2" head="does">
. <chunk form="pp-" function="SA" headid="5" head="from">
. . Where  [where] <clb> <*> <interr> <aloc> ADV @>>P #1->5 ID:1
. </chunk form="pp-" function="SA" head="5" head="from">
. <chunk form="vp-" function="STA" headid="2" head="does">
. . does   [do] <chunk-head> <aux> V PR 3S @FS-STA #2->0 ID:2
. </chunk form="vp-" function="P" head="2" head="does">
. it       [it] PERS NEU 3S NOM @<SUBJ #3->4 ID:3
. <chunk form="-vp" function="STA" headid="2" head="does">
. . come   [come] <move> <mv> V INF @ICL-AUX< #4->2 ID:4
. </chunk form="-vp" function="STA" head="2" head="does">
. <chunk form="-pp" function="SA" headid="5" head="from">
. . from   [from] <chunk-head> <prp-strand> PRP @<SA #5->4 ID:5
. </chunk form="-pp" function="SA" head="5" head="from">
</chunk form="fcl" function="STA" head="2" head="does">
$?         [?] PU @PU #6->0 ID:6
```

FIGURE 4: Layered chunking, discontinuity

For the layered, maximal chunking we followed the VISL convention, avoiding non-branching nodes and bracketing 1-word chunks only in the case of discontinuity (http://beta.visl.sdu.dk/VTB-design.html). If, for instance for np extraction purposes, single nouns were to be bracketed, this could be easily achieved by adding one simple rule applying to all nouns without an np head relation tag, mimicking the behavior of the minimal chunker on this point[4]. Because our layered bracketing de facto amounts to complete constituent trees, we were able to build conversion filters for both the VISL and TIGER formats[5] (figures 5 & 6), allowing corpus users to take advantage of the numerous tools available for these formats, such as visualizers, editors and search tools.

A1 STA:fcl	=======DN:adj("different" POS) different =======H:n("type" <idf> P NOM) types

[4] Likewise, all other word classes could of course be made to spawn 1-word chunks of a corresponding type.

[5] For clarity, a number of secondary tags was removed from the VISL non-terminal brackets. Similarly, the lemma, morphology and extra fields were removed from the TIGER non-terminal lines.

```
=fA:adv("so" <aquant>) So
=S:np
==H:pron-indef("anyone" S NOM) anyone
==DN:fcl
===S:pron-rel("who" <sam-> <rel> S) who_
===P:v-fin("be" <-sam> PR 3S) _s
===Cs:adjp
====H:adj("familiar" POS) familiar
====DA:pp
=====H:prp("with") with
=====DP:icl
======P:v-pcp1("play") playing
======Od:np
```

```
=======DN:pp
========H:prp("of") of
========DP:n("game" <idf> P NOM) games
======fA:pp
=======H:prp("through") through
=======DP:np
========DN:art("a" <indef> S) a
========H:n("console" <idf> S NOM) console
=,
=P:vp
==Vaux:v-fin("will" <cli> PR) will
==Vm:v-inf("be") be
=Cs:adj("content" POS) content
```

FIGURE 5: VISL constituent trees

```
<terminals>
  <t id="s1_1" word="So" pos="adv"/>
  <t id="s1_2" word="anyone" pos="pron-indef" />
  <t id="s1_3" word="who_" pos="pron-rel"/>
  <t id="s1_4" word="_s" pos="v-fin"/>
  <t id="s1_5" word="familiar" pos="adj"/>
  <t id="s1_6" word="with" pos="prp"/>
  <t id="s1_7" word="playing" pos="v-pcp1"/>
  <t id="s1_8" word="different" pos="adj"/>
  <t id="s1_9" word="types" pos="n"/>
  <t id="s1_10" word="of" pos="prp"/>
  <t id="s1_11" word="games" pos="n"/>
  <t id="s1_12" word="through" pos="prp"/>
  <t id="s1_13" word="a" pos="art"/>
  <t id="s1_14" word="console" pos="n"/>
  <t id="s1_15" word="," pos="pu"/>
  <t id="s1_16" word="will" pos="v-fin"/>
  <t id="s1_17" word="be" pos="v-inf"/>
  <t id="s1_18" word="content" pos="adj"/>
  <t id="s1_19" word="." pos="pu"/>
</terminals>
```

```
<nonterminals>
  <nt id="s1_500" cat="s">
    <edge label="STA" idref="s1_501"/></nt>
  <nt id="s1_501" cat="fcl">
    <edge label="PU" idref="s1_19"/></nt>
  <nt id="s1_502" cat="np">
    <edge label="H" idref="s1_2"/>
    <edge label="DN" idref="s1_503"/></nt>
  <nt id="s1_503" cat="fcl">
    <edge label="S" idref="s1_3"/>
    <edge label="P" idref="s1_4"/>
    <edge label="Cs" idref="s1_504"/></nt>
  <nt id="s1_504" cat="adjp">
    <edge label="H" idref="s1_5"/>
    <edge label="DA" idref="s1_505"/></nt>
  <nt id="s1_505" cat="pp">
     <edge label="H" idref="s1_6"/>
    <edge label="DP" idref="s1_506"/></nt>
  <nt id="s1_506" cat="icl">
    <edge label="P" idref="s1_7"/></nt>
</nonterminals>
```

FIGURE 6: TIGER treebank format

4 Evaluation

4.1 Minimal Chunker evaluation

There are several aspects in the evaluation of a CG-based chunker. First of all, in descriptive terms, it is interesting to see how well a function-based medium-level CG annotation can be converted into a constituent-based chunking parse which basically amounts to the task of computing (syntactic) form from (syntactic) function. Second, because a CG rule set is malleable and allows incremental improvements, it is important for development to identify specific error patterns and error triggers. For minimal chunking in particular, which as a shallow parsing technique will usually be performed on raw input rather than on corpora with linguist-revised grammatical tagging, it is important to evaluate the annotation chain as a whole, not just the chunker as such, and to identify error triggers at all levels.

We therefore evaluated the minimal chunker together with a morphosyntactic (POS and function) run of the underlying CG, on a 3563-word section of the English interview corpus, in the B/I/O format. Since error inspection was important for us, and no funding was available to create a multi-annotator gold standard, evaluation was done by output correction alone, with a corresponding risk of parser-friendly bias. For the complete run, recall was 97.4% and precision 97.5%, the difference being due to 3 MWE tokenization errors and 5 <O> (out-of-chunk) errors. While the latter are caused by the chunking grammar itself, the former is partly triggered by transcription conventions in the corpus, where noun-verb contractions were not recognized (persona's = persona is, who've = who have). Of the main body of errors, i.e. <B> and <I> errors, about 25% were pure chunking errors, where chunk form was correct, but segmentation faulty, caused almost always by function tag errors in the underlying CG. In the remaining 75%, chunk form was wrong, indicating underlying POS errors. Table 1 shows a confusion matrix for this error type.

gold: tagged:	np	adjp	advp	vp	pp	conjp	intj	prt
np	-	4	2	**5**	0	1	3	0
adjp	**6**	-	3	0	0	0	3	0
advp	**5**	**8**	-	0	0	1	0	0
vp	**4**	3	1	-	0	0	0	0
pp	0	0	3	0	-	1	0	0
conjp	3	0	1	0	1	-	0	0
intj	0	0	0	0	0	0	-	0
prt	0	0	0	0	0	0	0	-
sum/all	**18**/1630	15/101	10/254	5/956	1/316	3/236	6/50	0/21
relative	1.1%	**14.9**%	3.9%	0.5%	0.3%	1.3%	**12.0**%	0%

TABLE 1: Confusion table for chunk form types, minimal chunking

As can be seen, the most common form error was adjective phrases tagged as adverb phrases. Across confusion types, as a class, np's had the highest error frequency, but in relative terms the most error-prone classes were adjp's and interjections. Particles (verb-integrated adverbs) were recognized 100%, and vp and pp errors were very rare.

4.2 Maximal Chunker evaluation

For the the maximal/layered chunker (constituent tree generator), two evaluation runs were performed, one with a complete analysis chain, the other with a morphosyntactic gold corpus as input, where PoS and function tags had been hand-corrected, and where only dependency links had to be added automatically. For the former, the same (interview) data were used as for the minimal chunking evaluation, the latter consisted of 102 random Journalese sentences (1817 words) from the Leipzig Corpora Collection (http://corpora.informatik.uni-leipzig.de/download.html)[6].

With errors in only 12 out of 1055 multi-word[7] chunks (1,1%), the gold input run demonstrated the effectiveness of the CG chunking method in isolation, especially when taking into account that 3/4 of the errors were attachment errors directly attributable to (live) dependency grammar rather than the chunker itself. The only chunk missing outright was a coordination chunk, and 2 of the 3 chunk bracketing errors that were not due to attachment problems, also involved coordination chunks. In a certain sense, it can be concluded that CG chunking on top of syntactic CG analysis is more a format conversion than an independent layer of annotation - in other words, it is (almost) information-equivalent to CG dependency annotation, with most new information being contained in the latter already, making performance a direct function of the performance of the underlying morphosyntactic CG parser. Thus, the only chunk type in our grammar that does not really "trust" its dependency input, is coordination, where rules work with matching form and function tags rather than dependency links alone, taking into account the relatively high dependency error rate for this category.

In the raw text run, the maximal chunker suffered from the accumulated error rate of all CG analysis modules, and did not perform as well. A 1389 word section was used, containing 635 (multi-word) chunks. 17 chunks were not recognized, 4 chunks were in excess and 58 chunks had wrong bracketing[8]. This amounts to a recall of 88.2%, a precision of 90.0% and a balanced F-score[9] of 89.1.

Because both the CG dependency grammar and the chunk form assignment relied on morphosyntactic tags, erroneous head PoS or erroneous dependent function will lead to both wrong attachment and wrong form assignment, so form tag errors in correctly bracketed chunks were extremely rare, and category confusion was otherwise mainly triggered by wrong morphosyntactic tagging. Again, coordination errors figured prominently, and over 50% of undetected chunks were coordination chunks.

5 Conclusions

We have shown that Constraint Grammar rules constitute an efficient method for syntactic chunking. In a full CG suite, together with a morphosyntactic annotation module, between 89% and 97.5% of chunks will be correctly recovered for raw English text, representing the extremes of minimal chunking (no right np-branching and no nesting) on the one hand, and full layered constituent chunking on the other. While both chunking modules are quite rule-efficient (with 20 and 59 rules, respectively), only the minimal chunker works on morphosyntactic tags alone, while the layered chunker (which in its deepest version is a constituent parser rather than a chunker in the traditional sense of the word) uses an intermediate step of dependency attachment (279 rules). Still, even the combined error percentage of dependency and chunker is very low

[6] A direct comparison on the same data could have been interesting, but within the timeframe of the project, it was not possible to create a CG gold corpus for the Interview data.

[7] If single words were counted as chunks, no new error information would be added because all errors would constitute integration into multi-word chunks and should thus already be marked as multi-word chunk errors.

[8] For discontinuous chunks, both parts were counted as chunks, and the only discontinuity error was therefore counted as two.

[9] Defined as the harmonic mean of recall and precision.

(just over 1%) if seen in isolation, i.e. on correct function tag input, so the performance gap between minimal and layered chunking seems to be caused not so much by the chunking rules themselves, but rather by the fact that the deeper the nesting, the more morphosyntactic errors will trigger bracketing errors.

We believe, from a grammar writer's perspective, that the chunkers - especially the minimal chunker - are fairly language independent, because they run on an input level where syntactic function categories provide a "language-insulating" level of abstraction (provided a common notational system is used), but this assumption needs to be verified by future evaluation with generic rules and a larger set of languages, including languages that are typologically more different than English and German.

For practical reasons of availability, we tested gold input performance on a different genre (news) than the raw input runs (interview data), but optimally both runs and both chunking levels should be evaluated across different genre in a comparable way. This could also shed light on the question whether rule-based chunking is either more or less genre-sensitive than machine learning methods, and - if relevant - how much individual rules contribute to genre sensitivity, allowing the use of limited, genre-specific grammar patches.

References

Abney, Steven (1991). Parsing by Chunks. In: *Principle-Based Parsing*. Kluwer Academic Publishers, pp. 257-278

Banko, M., M.J. Cafarella, S. Soderland, M. Broadhead and O. Etzioni (2007). Open Information Extraction from the Web. In: *Proceedings of the 20th IJCAI,* pp. 2670-2676

Bick, Eckhard (2003). A CG & PSG Hybrid Approach to Automatic Corpus Annotation. In: Kiril Simow & Petya Osenova (eds.), *Proceedings of SProLaC2003* (at Corpus Linguistics 2003, Lancaster), pp. 1-12

Bick, Eckhard (2005). Turning Constraint Grammar Data into Running Dependency Treebanks, In: Civit, Montserrat & Kübler, Sandra & Martí, Ma. Antònia (red.), *Proceedings of TLT 2005 (4th Workshop on Treebanks and Linguistic Theory, Barcelona, December 9th - 10th, 2005)*, pp. 19-27

Carreras, X and L. Màrquez (2003). Phrase Recognition by Filtering and Ranking with Perceptrons. In: P*roceedings of RANLP-2003,* pp. 78-85.

Jurafsky, D. and J. H. Martin (2009). *Speech and Language Technology,* Pearson Education, p. 490.

Kübler, Sandra and Erhard W. Hinrichs (2001). From chunks to function-argument structure: A similarity- based approach. In *Proceedings of ACL-EACL 2001* (Tolouse, France), pp. 338-345.

Tjong Kim Sang, E. and S. Buchholz. (2000). Introduction to the CoNLL-2000 shared task: Chunking. In *Proceedings of CoNLL-2000 and LLL-2000* (Lisbon, Portugal), pp. 127– 132.

Van den Bosch, Antal and Sabine Buchholz (2002). Shallow parsing on the basis of words only: A case study . In *Proceedings of the 40th Meeting of the Association for Computational Linguistics* (ACL'02), pp. 433-440.