# Flexible Combinatory Categorial Grammar Parsing using the CYK Algorithm and Answer Set Programming

Peter Schüller

Cognitive Robotics Laboratory, Sabancı University, Turkey
peterschueller@sabanciuniv.edu

**Abstract.** Combinatory Categorial Grammar (CCG) is a grammar formalism used for natural language parsing. CCG assigns structured lexical categories to words and uses a small set of combinatory rules to combine these categories in order to parse sentences. In this work we describe and implement a new approach to CCG parsing that relies on Answer Set Programming (ASP) — a declarative programming paradigm. Different from previous work, we present an encoding that is inspired by the algorithm due to Cocke, Younger, and Kasami (CYK). We also show encoding extensions for parse tree normalization and best-effort parsing and outline possible future extensions which are possible due to the usage of ASP as computational mechanism. We analyze performance of our approach on a part of the Brown corpus and discuss lessons learned during experiments with the ASP tools dlv, gringo, and clasp. The new approach is available in the open source CCG parsing toolkit AspCcgTk which uses the C&C supertagger as a preprocessor to achieve wide-coverage natural language parsing.

## 1 Introduction

Parsing is the task of recovering the structure of sentences which is an important task in natural language processing (NLP). Contemporary NLP systems often process input in a 'pipeline' consisting of sequential steps of chunking, part-of-speech tagging, parsing, semantical annotation, and further steps. A widely-used technique in such pipelines is to statistically select a single best result of one stage and feed it to the next one.

However many natural language effects cannot be handled satisfactorily with such an approach, because natural language ambiguities can emerge in various levels of processing and some of them can only be resolved on other levels. For example the sentence

*"John saw the astronomer with the telescope."* (1)

admits two structures, intuitively one where John used a telescope to see the astronomer, and one where John saw an astronomer who had a telescope. On the other hand,

*"John saw the astronomer with the dog."* (2)

cannot only have the structure such that John saw an astronomer who had a dog. These sentences both have a syntactic ambiguity: whether the with-clause modifies 'saw' or 'the astronomer'. In (2) semantic information about "dog", i.e., that it can (usually) not

be used as a "tool for seeing", rules out the structure where 'John performed the action of seeing by means of the dog'. On the other hand (1) can only be disambiguated using contextual information from the world or from previous or following sentences.

These examples show that, to make sense of natural language, a bidirectional integration of natural language processing modules is necessary. Answer Set Programming (ASP) [3,7] is a declarative logic programming formalism which is well-suited to serve as computational formalism for NLP tasks: ASP programs can contain (i) guesses, which support modeling ambiguities of any kind; (ii) definitions of auxiliary concepts, which support modeling processes of natural language (in particular compositionality), and (iii) constraints which support modeling of linguistic constraints on all phenomenological levels.

In this work we describe an efficient encoding for parsing Combinatory Categorial Grammar (CCG) using ASP. CCG is a popular grammar formalism used in natural language parsing, which assigns structured categories to words of a sentence and uses a set of combinatory rules to combine these categories and to parse the sentence. Different from previous work [22] which modeled CCG parsing as action planning we here propose an encoding that is inspired by the CYK algorithm [12, 19, 33] and performs the major computational effort already within instantiation of the program. The combinatorial power of ASP is used for reasoning about parse tree shapes, parse tree normalizations [9, 15, 32], best-effort parsing and further possible extensions, e.g., [23].

Our main contributions are:

- we describe an adaptation of CYK algorithm for CCG parsing and give an encoding which builds a CYK chart during instantiation of the ASP encoding;
- we provide an encoding for enumerating parse trees based on the above encoding;
- we show how normalizing constraints for CCG can be realized as an additional ASP program module;
- we describe an extension of our encoding that supports best-effort parsing, i.e., providing maximal coverage of the input if no full parse tree is possible;
- we report on experiments which show that our approach provides reasonable parsing times and compare the new encoding to the previous approach for CCG parsing with ASP [22] and to the C&C parser;
- we discuss several lessons learned and interesting observations gained from this application of ASP.

An extension of the encodings presented in this work are released as version 0.4 of the open source CCG parsing toolkit ASPCCGTK[1] which uses the C&C supertagger [10] to achieve wide coverage and can visualize multiple CCG parse trees [22].

## 2 Preliminaries

**CCG.** A *Combinatory Categorial Grammar* (CCG) [29] is a tuple $G = (\Sigma, N, S, f, R)$ with $\Sigma$ a finite set of *terminal symbols*, $N$ a finite set of *atomic categories*, $S \in N$ the *start category*, $f$ a function mapping from terminal symbols to complex categories, and

[1] `http://www.kr.tuwien.ac.at/staff/former_staff/ps/aspccgtk/`

$R$ a finite set of *combinatory rules*. *Complex categories* are defined as follows: every atomic category is a complex category; given complex categories $A$ and $B$, $A/B$ and $A\backslash B$ are complex categories; nothing else is a complex category.

A *combinatory rule* (also called *combinator*) is of the form

$$\frac{X_1 \quad \dots \quad X_n}{C}\ \mathfrak{A} \tag{3}$$

where $\mathfrak{A}$ is a symbol indicating the name of the rule and $C, X_1, \dots, X_n$ are categories: we call $X_1, \dots, X_n$ the precondition categories of $\mathfrak{A}$ and $C$ the result category of $\mathfrak{A}$.

English can be parsed with the following combinators [29]: forward and backward application ($>$ and $<$, respectively), forward and backward *composition* ($>\mathbf{B}$ and $<\mathbf{B}$), forward and backward *type raising* ($>\mathbf{T}$ and $<\mathbf{T}$), backward *cross composition*, backward *cross substitution*, and *coordination*.

We limit the presentation of our work to the following set of combinators.

$$\frac{\mathcal{A}/\mathcal{B} \quad \mathcal{B}}{\mathcal{A}}\ > \qquad \frac{\mathcal{A}/\mathcal{B} \quad \mathcal{B}/\mathcal{C}}{\mathcal{A}/\mathcal{C}}\ >\mathbf{B} \qquad \frac{\mathcal{A}}{\mathcal{B}/(\mathcal{B}\backslash\mathcal{A})}\ >\mathbf{T}$$

$$\frac{\mathcal{B} \quad \mathcal{A}\backslash\mathcal{B}}{\mathcal{A}}\ < \qquad \frac{\mathcal{B}\backslash\mathcal{C} \quad \mathcal{A}\backslash\mathcal{B}}{\mathcal{A}\backslash\mathcal{C}}\ <\mathbf{B} \qquad \frac{\mathcal{A}}{\mathcal{B}\backslash(\mathcal{B}/\mathcal{A})}\ <\mathbf{T}$$

These combinatory rules are rule schemas with one or two preconditions, called unary and binary combinators, respectively, in the following. In this work we only consider the syntactic side of these combinators and disregard the semantic operations (following principles of combinatory logic) which are associated with combinators.

We use $\mathcal{A}$, $\mathcal{B}$, $\mathcal{C}$, ... to denote variables in CCG rule schemas. CCG derivation is defined in terms of a function $f$ which maps terminal symbols to CCG categories, and in terms of substitution of adjacent CCG categories by instantiations of combinators. Formally the *CCG derivation relation* $\Rightarrow$ contains for all $\alpha, \beta \in (N \cup \Sigma)^\star$

(i) $\alpha C \beta \Rightarrow \alpha c \beta$ for terminal symbol $c \in \Sigma$ and category $C \in f(c)$ i.e., the terminal symbol $c$ is mapped to category $C$ by $f$, furthermore
(ii) $\alpha C \beta \Rightarrow \alpha X_1 \cdots X_n \beta$ for an instantiation of a combinatory rule $\mathfrak{A} \in R$ of form (3).

The language generated by a CCG $G$ is the set $\{\alpha \in \Sigma^\star \mid S \Rightarrow^\star \alpha\}$ where $\Rightarrow^\star$ is the transitive closure of $\Rightarrow$.

A derivation $S \Rightarrow^\star \alpha$ can be considered as a set of parse trees: "S" is the root of a tree, a tree node is either the category $C$ on the left side of (i) or (ii) above; a tree node generated by (i) has one child which is terminal $c$; a node generated by (ii) has an ordered sequence of children $X_1, \dots, X_n$; in-order traversal of the tree leafs yields $\alpha$.

The problem of *enumerating parse trees* is to obtain all distinct parse trees given $\alpha$.

In the following we will call a natural language input a 'sentence' and the terminal symbols of such an input we will call 'tokens'.[2]

[2] Using the term 'word' can be misleading: what we call 'sentence' is sometimes called 'word', words such as "it's" can become multiple tokens, and punctuation symbols are tokens as well.

*Example 1.* The sentence "The dog bit John" with $f$ such that $f(\text{“}The\text{”}) = \{NP/N\}$, $f(\text{“}dog\text{”}) = \{N\}$, $f(\text{“}bit\text{”}) = \{S\backslash NP,\ (S\backslash NP)/NP\}$, $f(\text{“}John\text{”}) = \{NP\}$ can be derived using combinators $>$ and $<$ as follows:

$$\dfrac{\dfrac{\dfrac{The}{NP/N}\quad\dfrac{dog}{N}}{NP}> \quad \dfrac{\dfrac{\dfrac{bit}{(S\backslash NP)/NP}\quad\dfrac{John}{NP}}{S\backslash NP}>}{S}<$$

where lines below tokens of the sentence show derivations of type (i), i.e., mappings using $f$; lines below show derivations of type (ii), i.e., instantiations of combinators. Note that $f$ provides two categories for "bit" corresponding to the ambiguity between the intransitive and transitive reading of the verb "to bite".

**ASP.** Answer set programming (ASP) [3, 7, 25, 27] is a declarative programming formalism based on the answer set semantics of logic programs [18]. The idea of ASP is to represent a given computational problem by a program whose answer sets correspond to solutions, and then use an answer set solver to generate answer sets for this program. A common methodology in ASP is called GENERATE-DEFINE-TEST [24]: the GENERATE part of a program describes a collection of answer set candidates; the TEST part consists of constraints that eliminate candidates that do not correspond to solutions; the DEFINE part defines concepts in terms of other concepts.

We will present this work using the ASP-Core-2 language [8] of which we introduce a subset on the following example: given as a set of facts of form $edge(X, Y)$ encoding a graph, a typical logic program for solving the 3-colorability problem looks as follows:

$$\begin{gathered} vertex(X) \leftarrow edge(X, \_). \\ vertex(Y) \leftarrow edge(\_, Y). \\ 1 \leq \{\ color(X, red), color(X, green), color(X, blue)\ \} \leq 1 \leftarrow vertex(X). \\ \leftarrow color(X_1, C), color(X_2, C), edge(X_1, X_2). \end{gathered}$$

The first two rules DEFINE $vertex$ in terms of $edge$ ('_' symbols are anonymous variables); the third rule GENERATEs one color for each vertex; the fourth rule performs a TEST: it is a constraint which eliminates candidate solutions where two adjacent vertexes have the same color.

Variables are universally quantified over rules; a logic program is generally evaluated by (i) grounding it, i.e., instantiating all variables with terms that contain no variables, and (ii) searching for an answer set using methods related to SAT solving [17]. In this work we will use uninterpreted function symbols, e.g., in addition to constants programs can contain function terms. We use this to represent non-atomic CCG categories: $r(\text{“S”}, \text{“S”})$ and $l(r(\text{“S”}, \text{“NP”}), \text{“NP”})$ represent $S/S$ and $(S/NP)\backslash NP$, respectively.

Additionally we use count aggregates as literals in rule bodies: intuitively the literal $2 \leq \#count\ \{\ X\ :\ pred(X, Y)\ \} \leq 4$ is true iff the set of substitutions for variable $X$ such that $pred(X, Y)$ is true in the answer set candidate has cardinality 2, 3, or 4.

For a detailed description of ASP-Core-2 and for semantics of ASP we refer to [8].

## 3 Realizing CCG parsing with CYK in Answer Set Programming

A sentence $\alpha$ with $n$ tokens is presented to our CCG parser encoding as a set of facts of the form $catFor(C, P)$ where $C$ specifies the CCG category and $P$ the token position: $P \in \{1, \ldots, n\}$. Intuitively these categories are obtained from statistical tagging; a parser must select a single category at each position for generating a parse tree. Given a sentence $\alpha$ we denote by $inp(\alpha)$ its encoding in terms of token categories.

*Example 2 (ctd.).* "The dog bit John" has $n = 4$ tokens and $inp(\text{"The dog bit John"}) =$
$= \{catFor(l(\text{"NP"}, \text{"N"}), 1), catFor(\text{"N"}, 2), catFor(l(\text{"S"}, \text{"NP"}), 3),$
$catFor(r(l(\text{"S"}, \text{"NP"}), \text{"NP"}), 3), catFor(\text{"NP"}, 4)\}$

**CYK for CCG.** The CYK algorithm was originally proposed for parsing Context Free Grammar in Chomsky Normal Form [12, 19, 33], i.e., grammars with rules of the form $A \Rightarrow BC$ (corresponding to binary CCG combinators) and $A \Rightarrow c$ (corresponding to application of the function $f$) only, where $A$, $B$, and $C$ are nonterminals, and $c$ is a terminal. An adaptation for parsing grammars which also contain rules of the form $A \Rightarrow B$ (corresponding to CCG type raising) has been discussed in [21], in [30] specific problems of parsing CCG with CYK are discussed.

Algorithm 1 shows an adaptation of CYK to parse CCG in the spirit of [21]; Figure 1 visualizes a CYK chart and possible combinator applications for $n = 3$ tokens.

### 3.1 Building a CYK Chart via ASP Grounding

The ASP encoding we present in the following realizes Algorithm 1 such that a CYK chart for the given input is computed during program instantiation. To that end we present a non-ground encoding where identifiers starting with a capital letter denote variables and '_' denotes anonymous variables.

We represent contents of chart cells $C \in \mathcal{T}_{I,J}$ using predicate $grid(I, J, C)$. Corresponding to line 2 of Alg. 1 we fill diagonal cells $\mathcal{T}_{D,D}$ with $f(a_D)$ using rule

$$grid(D, D, C) \leftarrow catFor(C, D). \tag{4}$$

We track applicability of unary (lines 2 and 10) and binary (line 8) combinators using predicates $applicableU$ and $applicableB$, resp.: $applicableU(R, I, J, C', C)$ represents that combinator $R$ can be applied to category $C$ in cell $\mathcal{T}_{I,J}$ and yields category $C'$ in the same cell; $applicableB(R, I, J, H, C', X, Y)$ represents that combinator $R$ can be applied to categories $X$ and $Y$ in cells $\mathcal{T}_{H,J}$ and $\mathcal{T}_{I,H+1}$, resp., and yields category $C'$ in $\mathcal{T}_{I,J}$ (see also Fig. 1). We next give examples for encoding combinators $>\mathbf{T}$ and $>$:

$$applicableU(\text{">}\mathbf{T}\text{"}, I, J, r(B, l(B, A)), A) \leftarrow grid(I, J, A), raiseCategory(A, B). \quad \left[\frac{A}{B/(B\backslash A)}\ >\mathbf{T}\right] \tag{5}$$

$$applicableB(\text{">"}, I, J, H, A, r(A, B), B) \leftarrow grid(H, J, r(A, B)), grid(I, H+1, B). \quad \left[\frac{A/B \quad B}{A}\ >\right] \tag{6}$$

We define categories resulting from applicable rules to be part of the chart using rules

$$\begin{aligned} grid(I, J, C) &\leftarrow applicableU(\_, I, J, C, \_). \\ grid(I, J, C) &\leftarrow applicableB(\_, I, J, \_, C, \_, \_). \end{aligned} \tag{7}$$

**Algorithm 1:** CYK adapted for CCG Parsing

**Input**: CCG $G{=}(\Sigma, N, S, f, R)$; token sequence $\alpha = a_1, \ldots, a_n \in \Sigma^\star$ with $n \geq 1$

1 **for** $d = 1, \ldots, n$ **do** // initialize diagonal cells $(d, d)$ using $f$ and unary combinators in $R$

2 $\quad \mathcal{T}_{d,d} := f(a_d) \cup \{\mathcal{B}' \mid \mathcal{A}' \in f(a_d)$ and $\frac{\mathcal{A}'}{\mathcal{B}'}$ is an instantiation of a combinator $\frac{\mathcal{A}}{\mathcal{B}}$ in $R\}$

3 **for** $i = 2, \ldots, n$ **do** // iterate columns $i$ from left to right

4 $\quad$ **for** $j = i-1, \ldots, 1$ **do** // iterate rows $j$ from bottom to top

5 $\quad\quad \mathcal{T}_{i,j} := \emptyset$

6 $\quad\quad$ **for** $h = j, \ldots, i-1$ **do** // iterate distance of source cells from left and from top

7 $\quad\quad\quad$ **foreach** *combinator* $\frac{\mathcal{B} \quad \mathcal{C}}{\mathcal{A}}$ *in* $R$ **do**

8 $\quad\quad\quad\quad$ **if** $\mathcal{B}' \in \mathcal{T}_{h,j}$ *and* $\mathcal{C}' \in \mathcal{T}_{i,h+1}$ *and* $\mathcal{A}', \mathcal{B}', \mathcal{C}'$ *can instantiate* $\mathcal{A}, \mathcal{B}, \mathcal{C}$ **then**

9 $\quad\quad\quad\quad\quad \mathcal{T}_{i,j} := \mathcal{T}_{i,j} \cup \{\mathcal{A}\}$

10 $\quad\quad \mathcal{T}_{i,j} := \mathcal{T}_{i,j} \cup \{\mathcal{B}' \mid \mathcal{A}' \in \mathcal{T}_{i,j}$ and $\frac{\mathcal{A}'}{\mathcal{B}'}$ is an instantiation of a combinator $\frac{\mathcal{A}}{\mathcal{B}}$ in $R\}$

11 **if** $S \in \mathcal{T}_{1,n}$ **then return** *yes* **else return** *no*

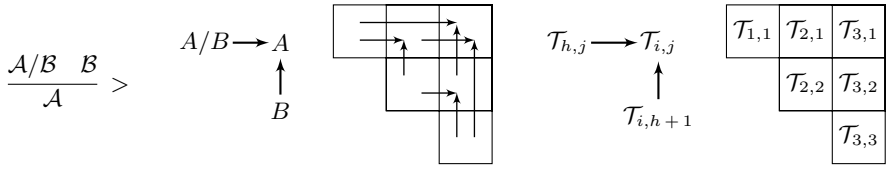


**Fig. 1.** CYK chart visualization for input sentence with 3 tokens and for $>$ combinator.

This concludes the deterministic non-ground program $\Pi_{CYK}$ which consists of rules (4) to (7). Intuitively this encoding defines applicability from chart cells as lines 2, 8 and 10 of Alg. 1 do, furthermore it defines that categories that result from applying combinators are again part of chart cells.

Given a sentence $\alpha$, program $inp(\alpha) \cup \Pi_{CYK}$ has a single answer set $I$ which represents the CYK chart for $\alpha$ as produced by lines 1 to 10 of Alg. 1.

For space reasons we here do not present type conversion (e.g., $N \Rightarrow NP$) and punctuation rules (e.g., to handle commas) as described in [11, Appendix A]; these features are necessary for wide-coverage parsing and they are implemented in ASPCCGTK.

**Language Membership.** We can check whether an input is part of the language by checking whether cell $(n, 1)$ contains category $S$, e.g., using the following constraint:

$$\leftarrow not\ grid(n, 1, \text{“S”}). \tag{8}$$

or by performing an ASP query (see Section 4 on Magic Set experiments).

### 3.2 Enumerating Parse Trees

We next describe an encoding for enumerating all parse trees of a given input. We achieve this by (i) guessing which applicable combinators are applied and (ii) restricting

the guess to a tree such that the chosen combinators form edges, input tokens are leaves, and category "$S$" in cell $(n, 1)$ is the root node.

For each applicable combinator we guess whether it is applied or not:

$$\begin{array}{c} 0 \leq \{\ applyB(R, I, J, H, X, Y, Z)\ \} \leq 1 \leftarrow applicableB(R, I, J, H, X, Y, Z). \\ 0 \leq \{\ applyU(R, I, J, X, Y)\ \} \leq 1 \leftarrow applicableU(R, I, J, X, Y). \end{array} \tag{9}$$

To ensure that the above guess induces a parse tree, we first define reachability of categories in cells from other categories via applied combinators.

$$\begin{array}{c} reach(H, J, C_{left}) \leftarrow reach(I, J, C), applyB(\_, I, J, H, C, C_{left}, \_). \\ reach(I, H+1, C_{down}) \leftarrow reach(I, J, C), applyB(\_, I, J, H, C, \_, C_{down}). \\ reach(I, J, C_{sameCell}) \leftarrow reach(I, J, C), applyU(\_, I, J, C_{sameCell}, C) \end{array} \tag{10}$$

We ensure that the guess is restricted to parse using the following rules

$$reach(n, 1, \text{“S”}) \leftarrow \tag{11}$$

$$\leftarrow valid(I, I), \#count\{\ C : reach(I, I, C), catFor(C, I)\ \} \leq 0 \tag{12}$$

$$\begin{array}{l} \leftarrow applyB(\_, I, J, \_, C, \_, \_), not\ reach(I, J, C) \\ \leftarrow applyU(\_, I, J, C, \_), not\ reach(I, J, C) \end{array} \tag{13}$$

$$\begin{array}{l} \leftarrow valid(I, J), 2 \leq \#count\{\ C : applyU(\_, I, J, C, \_)\ \} \\ \leftarrow valid(I, J), 2 \leq \#count\{\ C : applyB(\_, I, J, \_, C, \_, \_)\ \} \end{array} \tag{14}$$

These rules define the root category "S" to be reachable (11), require that each word is reachable (12), disallow unreachable combinators to be applied (13), and disallow more than one binary (resp., unary) combinator application in one cell (14).

By $\Pi_{Tree}$ we denote rules (9) to (14). $\Pi_{Tree}$ follows the classical GENERATE-DEFINE-TEST approach: it guesses a subset of applicable combinators (9), defines a notion of reachability (10)-(11), and restricts the guess to certain trees (12)-(14).

With $\Pi_{Tree}$ we can enumerate parse trees: given an input sentence $\alpha$ the answer sets of program $inp(\alpha) \cup \Pi_{CYK} \cup \Pi_{Tree}$ correspond 1-1 to the CCG parse trees of $\alpha$.

Note that we do not use $f$ in our encoding, instead we use $f$ to generate $inp(\alpha)$; this is because $f$ corresponds to statistical tagging which is handled outside of our encoding.

**Parse Tree Normalization.** CCG generates *spurious* parse trees which are not of interest because they provably lead to the same linguistic interpretation as other parse trees; this can only be avoided by *normalization* of parse trees [9, 15, 32] which is performed by constraining the shape of CCG parse trees depending on the type of combinator used to create each tree node. For example, using the category resulting from $>\mathbf{T}$ as the first prerequisite of $>$ can always be replaced by a single application of $<$. Fortunately such constraints can be represented easily in ASP, the above normalization is encoded as

$$\leftarrow applyB(\text{“}>\text{”}, I, J, H, \_, L, \_), applyU(\text{“}>\mathbf{T}\text{”}, I, H, L, \_). \tag{15}$$

Such normalizations can eliminate an exponential number of spurious parse trees [15]. Thanks to the modularity of our encoding we can maintain multiple sets of normalizing constraints (e.g., to normalize towards left- or right-branching) and simply add them to our program when needed without changing rules in the encoding.

**Best-Effort CCG Parsing.** If a sentence has no parse tree it can be useful to obtain a best-effort parse forest. This can be done with respect to various optimization criteria, e.g., finding a minimum of root nodes for a set of parse trees which contains all tokens of an input or ignoring a minimum of input tokens. Our parser encoding allows us to perform such best-effort parsing with only small modifications. For example we can create a parser that enumerates (i) complete parse trees if one exists, otherwise (ii) partial parse trees with a minimum number of root nodes such that all input tokens are reachable. This is achieved by replacing (11) with the following set of rules:

$$\begin{aligned} 0 \leq \{\ guessReach(I,J,C)\ \} \leq 1 \leftarrow\ & grid(I,J,C). \\ & reach(I,J,C) \leftarrow guessReacch(I,J,C). \\ \#minimize\{1 : guessReach(I,J,C)\}.& \end{aligned} \tag{16}$$

## 4 Experimental Evaluation

For performance evaluation we used Section A of the Brown corpus [16] which is a freely available English language corpus. We selected Section A because it contains newspaper articles and the C&C supertagger we use for tagging the input is trained on a newspaper corpus. Section A contains 4611 sentences in total, Table 1 groups these sentences in terms of their length, e.g., the corpus contains 684 sentences with a length between 11 and 15 words (inclusive) where the average sentence length is 13.

Our experiments were performed similar as the C&C parser operates when parsing a sentence: we obtain tags of probability class $\beta = 0.075$, 0.03, and 0.001 from the C&C supertagger, then we run our encoding on categories obtained with $\beta \geq 0.075$, if this does not yield a parse tree we retry with $\beta \geq 0.03$, then with $\beta \geq 0.001$, and we register failure if even this does not yield a parse trees.

Where not otherwise indicated we used GRINGO version 3.0.5 and CLASP[3] version 2.1.1 for experiments; some experiments were performed with DLV[4] version 2012-12-17. We used a timeout of 300 seconds and enumerated up to 100 parse trees for each sentence, this was done in single threaded mode on a Linux server with 32 2.4GHz Intel® E5-2665 CPU cores and 64GB memory.

Table 1 reports the number of CCG tags required to parse a sentence (if no parse was found the value for $\beta = 0.001$), the number of parse trees obtained for each sentence, and the percentage of sentences where a parse tree was found. For example, sentences with 11-15 words required 25 tags for parsing on average, each sentence yielded on average 38 parse trees, and 84.9% of sentences yielded at least one parse tree.

**Comparison with planning approach.** We compare our approach (CYK+ASP) with the ASP formulation for CCG parsing that uses planning [22]. The CYK algorithm is a (dynamic programming) approach, therefore in a CYK chart partial parse trees can be reused between complete parse trees. This is not possible in the planning approach which requires the notion of time to define an order of combinator applications.

Table 1 reports performance of the planning approach: experiments show that the CYK approach scales much better than planning, especially for larger sentences. Performance is similar only for the shortest group of sentences, for sentences of length 21-25

[3] `http://potassco.sourceforge.net/`

[4] `http://www.dlvsystem.com/`

| Group: words in sentence | # | 1-10 | 11-15 | 16-20 | 21-25 | 26-30 | 31-35 | 36-40 | 41+ |
|---|---|---|---|---|---|---|---|---|---|
| Sentences in group | # | 983 | 684 | 779 | 704 | 526 | 396 | 258 | 281 |
| Words in sentence | avg # | 5 | 13 | 17 | 22 | 27 | 32 | 37 | 48 |
| CCG categories[†] | avg # | 14 | 25 | 37 | 48 | 59 | 77 | 87 | 122 |
| Parse Trees | avg # | 6 | 38 | 65 | 81 | 80 | 80 | 79 | 72 |
| Sentences with parse tree | % | 64.2 | 84.9 | 84.7 | 85.9 | 82.1 | 81.1 | 79.1 | 73.0 |
| CYK+ASP parse time | avg sec | 0.6 | 0.7 | 1.2 | 1.9 | 3.5 | 6.7 | 11.9 | 42.0 |
| CYK+ASP timeouts | # | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 8 |
| Planning+ASP parse time | avg sec | 0.8 | 4.3 | 19.1 | 62.1 | 110.1 | 135.7 | 156.5 | 205.5 |
| Planning+ASP timeouts | # | 0 | 0 | 4 | 51 | 137 | 157 | 118 | 149 |

[†] Category set with smallest $\beta$ value that is sufficient for finding a parse tree, $\beta = 0.001$ if no parse tree can be found with tags provided by the C&C supertagger.

**Table 1.** Performance comparison on Section A (newspaper) of Brown corpus using C&C for tagging. Times and timeouts are for the task of enumerating up to 100 parse trees per sentence.

the CYK approach gives an answer within 1.9 seconds while planning requires more than one minute. Moreover, the planning approach suffers from timeouts already with sentences of length 16-21 while the CYK approach has no timeout for any sentence smaller than 41 tokens.

The CYK approach requires a maximum of 1.5GB of memory with an average of 1.1GB over all sentences, where as the planning approach requires up to 5.8GB of memory with an average of 3.8GB.

**Parse Effort Profile.** Figure 2 gives a diagram of parsing time for all sentences in our benchmark, first grouped by the $\beta$ value required to find a parse tree, then by the time required to enumerate up to 100 parse trees. E.g., 2596 sentences obtain a parse tree with $\beta = 0.075$ and 3639 out of 4611 sentences obtain a parse tree with $\beta \geq 0.001$. We plot total time required (dashed red) and solver time required (solid blue).

The graph shows that parse time is not distributed evenly among the sentences in the benchmark: a majority of sentences can be parsed in a comparatively short time, in particular for sentences with the most probable tags ($\beta = 0.075$), while there are few sentences in the benchmark that take a disproportionately high amount of time. There is only a weak correlation between difficulty of a sentence and its length or amount of tags (not shown), therefore additionally plotting the amount of tags and/or the length of each sentence in the figure would make it unreadable. Furthermore we see that solving time is negligible compared to grounding time.

Additionally we measured the time for grounding $\Pi_{CYK}$ and the time for grounding $\Pi_{CYK} \cup \Pi_{Tree}$ (not shown here); these times are nearly the same independent from the length of the input. This shows that the main computational effort is due to $\Pi_{CYK}$.

**Comparison with C&C.** The popular C&C parser is designed as a highly efficient CCG wide coverage parser [11, 13] and it operates on the same C&C tagger output as our parser. This makes it suitable for a comparison: in [13] C&C is reported to parse the whole section 00 of CCGBank (1913 sentences with a similar distribution of sentence

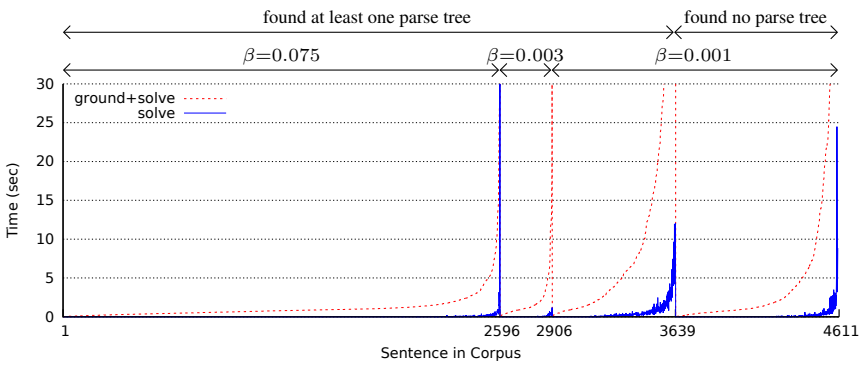


**Fig. 2.** Total time and solve time for parsing Section A of the Brown corpus, grouped by the tagger $\beta$ value required to parse each sentence and sorted by parse time (timeout 300 seconds).

lengths as in the Brown corpus) within less than 100 seconds on a slower computer than ours. Therefore our approach clearly cannot compete with the performance of C&C. However, the aim of this work was not to build the fastest parser, but to build a flexible parser with reasonable performance that can return multiple parse trees and can easily be extended with reasoning capabilities that go beyond what C&C can do.

**Stratification and DLV vs GRINGO+CLASP.** Apart from benchmarking with GRINGO and CLASP we also considered DLV. Our first observation was that DLV does not recognize our encoding to be finitely groundable due to rule (5) and other raising rules which contain a higher level of function symbol nesting in their head than in the body. However due to the $raiseCategory$ predicate (which does not depend on the $applicableU$ predicate in the head of (5)) the program clearly has a finite instantiation and using DLV with the option `-nofinitecheck` leads to a finite grounding.

In an early version of the encoding, computing the set of possible categories of a token was performed with a non-stratified rule. With this encoding, DLV performed consistently better than GRINGO for the task of grounding. Replacing this rule by a few stratified rules changed the situation: now DLV produced a slightly smaller grounding in about the same time, but GRINGO became so much faster that it consistently performed better than DLV. (All results in this paper were produced using GRINGO+CLASP and the stratified encoding.) We conclude that efficiency of GRINGO is very sensitive to program structure while for DLV we could not observe this in our benchmarks.

**Queries and Magic Sets.** As $\Pi_{CYK}$ is stratified it is possible to use Magic Sets [1] for efficient query evaluation, e.g., the query '$grid(n, 1, \text{“}S\text{”})?$' checks whether a parse tree exists. Such a check is important to see whether a given set of tags is sufficient for finding a parse tree, or whether the $\beta$ value needs to be reduced to obtain more tags. Unfortunately, using DLV with Magic Set for the above query led to much longer grounding times than using DLV without Magic Set; the reason is not clear to us yet.

**Grounding vs Solving.** Finally we experimented with putting some of the tree normalization constraints, e.g., (15), already into the $\Pi_{CYK}$ encoding. This requires to define exceptions to rule applicability, therefore the CYK encoding becomes more compli-

cated (it is still stratified). The result of this experiment is a reduced grounding size and (with GRINGO) a significantly increased grounding time. As the time spent in grounding and solving of $\Pi_{Tree}$ (including all normalization constraints) is negligible, we reverted to the simpler CYK encoding with larger and faster grounding. We conclude that eliminating solutions in solving can perform significantly better than making a program more complex in order to eliminate those solutions already in grounding, even if the complex program has a smaller instantiation.

## 5 Related Work

CCG-based systems OPENCCG [31] and TCCG [4,5] (implemented in the LKB toolkit) can provide multiple parse trees for a given sentence. Both use chart parsing algorithms with CCG extensions such as modalities or hierarchies of categories. While OPENCCG is primarily geared towards generating sentences from logical forms, TCCG targets parsing. However, both implementations require lexicons with specialized categories.

The wide-coverage CCG parser C&C [9, 10] relies on machine learning techniques for tagging an input sentence with CCG categories as well as for computing the single most likely parse tree with an efficient chart algorithm. In ASPCCGTK we reuse the CCG supertagger of C&C to obtain CCG categories, we also compare ASPCCGTK to C&C performance.

The Grail parser [26] is based on multi-modal categorial grammar (which is able to represent CCG) and contains a graphical user interface for 'interactive parsing'. Grail uses theorem proving techniques based on the Lambek calculus, this makes it very expressive but slow in some cases; therefore in some cases the user must support the search for a parse tree in the user interface. Compared to our work, Grail is more general and has different aims, e.g., being a tool for learning about Lambek calculus.

Transforming context free grammars (CFGs) in Chomsky Normal Form (CNF) using the CYK algorithm and parsing them using SAT solvers has been studied under the name "GRAMMAR constraint" [20, 28], including recent work based on ASP [14]. Results indicate that SAT and ASP solving can perform well for parsing using the CYK algorithm. Two important differences between these studies and our work are: (i) CFGs use atomic categories and a large set of rules that forms the grammar, while CCG uses structured categories and a small set of rule schemas, hence performance observations might not directly carry over and encodings must be significantly different; moreover (ii) our work is about natural language parsing while GRAMMAR studies experiment with artificial grammars that encode solutions to Shift Scheduling problems.

Parsing CCG with CYK is not polynomial if categories are represented explicitly, however recording only changes of categories can make it polynomial [30]; we here represent categories explicitly and consider a more involved encoding as future work.

## 6 Conclusion

We have presented an encoding for parsing CCG in ASP which — as opposed to a previous approach, and as opposed to usual ASP methodology — puts the major effort of computation into instantiation of the representation. This increased effort of grounding

allows the search for an answer set to be fast, empirical results show that the new approach consistently outperforms the former approach that used planning. Experiments show that our approach provides reasonable performance for using it in practice.

The possibility to trade search effort for grounding effort is due to the CYK algorithm which has been around for a long time and can be realized in ASP in a natural way. This approach of gaining efficiency goes against the declarativity of ASP, because our encoding effectively prescribes a way of grounding that reproduces the data structure generated by CYK. Nevertheless the result is a parsing framework that profits from the declarative nature of ASP because reasoning modules that operate on parse trees (i.e., normalization, semantic disambiguation) can be tightly and modularly integrated with the parser without significant changes to the parser encoding.

**Future Work.** In the future we want to adapt ASPCCGTK to become compatible with Boxer [6] which is a tool for creating semantic representations for sentences in first order logic. Integration with Boxer opens new possibilities for NLP tasks where multiple readings of a sentence must be considered, e.g., for Recognizing Textual Entailment (RTE) or Semantic Evaluation (SemEval) Challenges.

We have done preliminary work on disambiguation of parse trees using semantic information [23], e.g., from FRAMENET [2], such that the large number of parse trees (our experiments enumerated up to 100 trees per sentence) can be reduced to those trees which are consistent with semantic restrictions (see examples in the introduction). In the future we want to continue work in that direction.

If better efficiency becomes an issue, using techniques from [30] and computing the CYK chart in C++ and enumerating parse trees with constraints in ASP are possibilities.

Finally our CYK encoding could be useful for benchmarking ASP grounders.

**Acknowledgments.** We thank Yuliya Lierler for fruitful discussions related to the topics of this work. We thank the anonymous reviewers for their constructive comments. Peter Schüller is supported by TUBITAK Research Fellowship 2216.

## References

1. Alviano, M., Faber, W., Greco, G., Leone, N.: Magic sets for disjunctive datalog programs. Tech. rep., Università della Calabria, Dipartimento di Matematica (2009)
2. Baker, C.F., Fillmore, C.J., Lowe, J.B.: The Berkeley FrameNet Project. In: 36th Annual Meeting of the Association for Computational Linguistics and 17th International Conference on Computational Linguistics. pp. 86–90 (1998)
3. Baral, C.: Knowledge Representation, Reasoning, and Declarative Problem Solving. Cambridge University Press (2003)
4. Beavers, J.: Documentation: A CCG implementation for the LKB. Tech. rep., Stanford University, Center for the Study of Language and Information (2003)
5. Beavers, J.: Type-inheritance combinatory categorial grammar. In: International Conference on Computational Linguistics (COLING'04) (2004)
6. Bos, J.: Wide-coverage semantic analysis with boxer. In: Semantics in Text Processing (STEP). pp. 277–286. College Publications (2008)
7. Brewka, G., Eiter, T., Truszczynski, M.: Answer set programming at a glance. Commun. ACM 54(12), 92–103 (2011)
8. Calimeri, F., Faber, W., Gebser, M., Ianni, G., Kaminski, R., Krennwallner, T., Leone, N., Ricca, F., Schaub, T.: ASP-Core-2 input language format (2012)

9. Clark, S., Curran, J.R.: Log-linear models for wide-coverage CCG parsing. In: SIGDAT Conference on Empirical Methods in Natural Language Processing (EMNLP-03) (2003)
10. Clark, S., Curran, J.R.: Parsing the WSJ using CCG and log-linear models. In: 42nd Annual Meeting of the Association for Computational Linguistics (ACL). pp. 104–111 (2004)
11. Clark, S., Curran, J.R.: Wide-coverage efficient statistical parsing with CCG and log-linear models. Computational Linguistics 33(4), 493–552 (2007)
12. Cocke, J., Schwartz, J.T.: Programming Languages and Their Compilers. Courant Institute of Mathematical Sciences, New York (1970)
13. Djordjevic, B., Curran, J.R.: Efficient combinatory categorial grammar parsing. In: Proceedings of the 2006 Australasian Language Technology Workshop (ALTW). pp. 3–10 (2006)
14. Drescher, C., Walsh, T.: Modelling grammar constraints with answer set programming. In: International Conference on Logic Programming. vol. 11, pp. 28–39 (2011)
15. Eisner, J.: Efficient normal-form parsing for combinatory categorial grammar. In: 34th Annual Meeting of the Association for Computational Linguistics. pp. 79–86. ACL (1996)
16. Francis, W.N., Kucera, H.: Brown corpus manual. Letters to the Editor 5(2), 7 (1979)
17. Gebser, M., Kaufmann, B., Neumann, A., Schaub, T.: Conflict-driven answer set solving. In: International Joint Conference on Artificial Intelligence. pp. 386–392 (2007)
18. Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. In: Proc. International Logic Programming Conference and Symposium (ICLP). pp. 1070–1080 (1988)
19. Kasami, T.: An efficient recognition and syntax analysis algorithm for context-free languages. Tech. Rep. AFCRL-65-758, Air Force Cambridge Research Laboratory (1965)
20. Katsirelos, G., Narodytska, N., Walsh, T.: Reformulating global grammar constraints. In: CPAIOR. pp. 132–147 (2009)
21. Lange, M., Leiß, H.: To CNF or not to CNF? An efficient yet presentable version of the CYK algorithm. In: Informatica Didactica 8. Universität Potsdam (2009)
22. Lierler, Y., Schüller, P.: Parsing Combinatory Categorial Grammar via planning in Answer Set Programming. In: Correct Reasoning. LNCS, vol. 7265, pp. 436–453. Springer (2012)
23. Lierler, Y., Schüller, P.: Towards a tight integration of syntactic parsing with semantic disambiguation by means of declarative programming. In: Erk, K., Koller, A. (eds.) 10th International Conference on Computational Semantics (2013)
24. Lifschitz, V.: Answer set programming and plan generation. Artif. Intel. 138, 39–54 (2002)
25. Marek, V., Truszczyński, M.: Stable models and an alternative logic programming paradigm. In: The Logic Programming Paradigm: a 25-Year Perspective, pp. 375–398. Springer (1999)
26. Moot, R.: Proof Nets for Linguistic Analysis. Ph.D. thesis, Utrecht Institute of Linguistics OTS (2002)
27. Niemelä, I.: Logic programs with stable model semantics as a constraint programming paradigm. Annals of Mathematics and Artificial Intelligence 25, 241–273 (1999)
28. Quimper, C.G., Walsh, T.: Decomposing global grammar constraints. In: Constraint Programming (CP). pp. 590–604 (2007)
29. Steedman, M.: The syntactic process. MIT Press, London (2000)
30. Vijay-Shanker, K., Weir, D.J.: Polynomial time parsing of combinatory categorial grammars. In: 28th Annual Meeting of the Association for Computational Linguistics. pp. 1–8 (1990)
31. White, M., Baldridge, J.: Adapting chart realization to CCG. In: European Workshop on Natural Language Generation (EWNLG'03) (2003)
32. Wittenburg, K.: Predictive combinators: a method for efficient processing of combinatory categorial grammars. In: 25th Annual Meeting of the Association for Computational Linguistics (ACL). pp. 73–80 (1987)
33. Younger, D.H.: Recognition and parsing of context-free languages in time $n^3$. Information and Control 10(2), 189–208 (Feb 1967)