DotCCG and VisCCG: Wiki and Programming Paradigms for Improved Grammar Engineering with OpenCCG

Jason Baldridge†, Sudipta Chatterjee‡,
Alexis Palmer†, and Ben Wing‡

†Dept. of Linguistics, ‡Dept. of Computer Science

University of Texas at Austin

Proceedings of the GEAF 2007 Workshop

Tracy Holloway King and Emily M. Bender (Editors)

CSLI Studies in Computational Linguistics ONLINE

Ann Copestake (Series Editor)

2007

CSLI Publications

http://csli-publications.stanford.edu/

Abstract

We present a suite of tools for simplifying the creation and maintenance of grammars for the OpenCCG parsing and realization system. The core of our approach relies on a terse but expressive textual format, DotCCG, for declaring CCG grammars. It supports powerful string expansions that allow grammar developers to eliminate redundancy in the declaration of both morphology and category definitions. Grammars written in this format are converted into the XML utilized by OpenCCG using the `ccg2xml` utility, which –like a programming language compiler– provides information regarding errors in the grammar, including the type of error and the line number on which it occurs. DotCCG grammars can be edited with VisCCG, a graphical interface which provides visualization of various components of the grammar and allows local editing of information in a manner inspired by wikis. We also report on resources developed to facilitate wide use of the OpenCCG tool suite presented in this paper and on recent uses of the tools in both academic research and classroom environments.

1 Introduction

A major challenge of grammar engineering is enabling users with little computer experience to create complex grammars. Many users encounter significant obstacles and easily get frustrated by trivial syntax errors and non-intuitive formats. At the same time, more experienced users can feel needlessly constrained by grammar engineering aids designed for novice users. Such frustrations slow users down and can result in a focus on mechanics more than on the grammar itself.

This paper presents two contributions for improving current practice in grammar engineering. First, it provides a terse but expressive format for declaring Combinatory Categorial Grammars (CCG) (Steedman, 2000; Steedman and Baldridge, To appear) that utilizes ideas from software engineering for reducing redundancy in CCG grammars. The basic idea is general enough to be used with other formalisms. Second, it describes a wiki-inspired editing interface, VisCCG, that supports grammar visualization while allowing users to directly edit plain text grammars.

The core motivation for these developments is to improve the grammar development cycle for OpenCCG (`openccg.sf.net`) (Hockenmaier et al., 2004; Baldridge and Kruijff, 2002; White and Baldridge, 2003), a parsing and realization system that uses CCG, and to provide a model for facilitating grammar development for both novice and expert grammar writers. OpenCCG has long lacked such an environment despite its use in a number of projects. Grammars developed with VisCCG are compiled into OpenCCG's native XML format, much in the same

[†]We would like to thank Emily Bender, Fred Hoyt, Geert-Jan Kruijff, Mark Steedman, Michael White, students in Jason Baldridge's categorial grammar, computational syntax, and computational linguistics courses at UT Austin in 2006/7, and the participants of the GEAF 2007 workshop for valuable feedback. This research was supported by a Liberal Arts Instructional Technology Grant from the University of Texas at Austin.

manner as wiki pages produce HTML. The goal is to create a grammar engineering environment for CCG that is both easy to *learn* to use and easy to use.

We begin by motivating our work in the context of OpenCCG as well as other grammar engineering platforms. In section 4 we then briefly introduce CCG and OpenCCG and some of the problems with OpenCCG's native XML grammar format. Section 5 discusses DotCCG, followed by an extensive discussion of its parameterized macro mechanisms in section 6. Then we present VisCCG and conclude with a brief discussion of uses of our tools and resources for developing OpenCCG grammars.

2 Motivation

A graphical user interface (GUI) was developed for Grok, OpenCCG's predecessor, but development was ceased as the parsing system itself was improved (see Bierner (2001) and Baldridge (2002) for specific reference to Grok). Developing grammars for OpenCCG has since involved working with unwieldy XML specifications. Our work was initiated to address this (rather large) gap in CCG grammar development.[1] Several aspects of our approach are novel and may be useful in the context of work in other formalisms and/or grammar engineering environments.

The schism between computational definitions and the grammar they are supposed to express has been addressed in various ways, with visualization being a common strategy for more intuitive representations of the grammar. One approach is to develop a GUI for editing objects such as trees and feature structures, such as that of the XTAG system (Doran et al., 2000). The XTAG system included a graphical tree-drawing editor which allowed the user to attach features and labels to nodes of a tree. In such systems, grammar developers usually do not work with the underlying code. A high-level approach like that of the XTAG tree editor is friendly for novice users but can be frustratingly restrictive for experienced users.

An alternative is to develop grammars by working with a low-level format and then visualizing them with a separate GUI which *displays* information. For example, the LKB system (Copestake, 2002) provides extensive, highly configurable displays of various components of grammars written in the Type Description Language. The display functionality in the XLE system for grammar development in the Lexical-Functional Grammar framework (Butt et al., 1998) is similarly informative and configurable. In such systems, however, the developer cannot directly edit the grammar using the GUI. Instead, the plain text grammar is edited and then reloaded to view the effect of the modifications in the graphical representation.

An interesting compromise between visualization and low-level specification can be observed with the use of wikis for creating web content. HTML and XML are cumbersome and unintuitive formats; wiki notation as an alternative has en-

[1] Concurrently with our work, Scott Martin and Michael White at Ohio State University developed a complementary tool called `grammardoc` which produces a set of HTML pages for visualizing OpenCCG grammars. Both `grammardoc` and our tools are distributed with the OpenCCG system.

1	`pay **close** attention`	wiki syntax
2	`pay <b>close</b> attention`	HTML syntax
3	pay **close** attention	display

Figure 1: Wiki-style notation as shorthand for HTML

abled lay users to create web content quickly and effectively. For example, in one common wiki syntax, boldfaced text is indicated with double asterisks around the text. This shorthand (Figure 1, line 1) is then converted into HTML (line 2) and displayed as boldfaced text (line 3). Wikis also make it easy to edit small portions of documents while visualizing the rest, and they provide immediate feedback on the visual outcome of edits. DotCCG provides a similar shorthand notation for OpenCCG's XML, and VisCCG provides user-friendly visualization and editing.

Software engineering provides another source of ideas for improving grammar engineering. Most grammar specifications can be viewed as programming languages particularized to natural language, yet grammar platforms typically do not provide much support for error checking and error messages. Our `ccg2xml` utility compiles DotCCG to OpenCCG's XML and supports such checking in the process, while VisCCG provides feedback in real-time (during editing).

Integrated Development Environments (IDEs) for programming languages can be used to improve productivity for many developers. A key property of IDEs is that they are optional – a developer may use a plain text editor to write programs if they wish. We see VisCCG in this light. It is particularly useful for those who are creating their first grammars. In the classroom setting, we observed that users with less experience working with computers tend to stick with editing their grammars using VisCCG, but many others –particularly those with programming experience– switch over to their favorite text editor (e.g. Emacs or Vi) once they understand the DotCCGformat. The latter would still periodically load their grammars in VisCCG. We see this availability of choice as a highly desirable feature of the new tools we have developed for OpenCCG: the DotCCG format, `ccg2xml`, and VisCCG.

3 Combinatory Categorial Grammar

CCG is a lexicalized grammar formalism that has attracted both linguistic and computational interest. It has a universal rule component that drives the combination of categories and their semantics to provide compositional analyses for sentences. Categories may be either atomic elements or (curried) functions which specify the canonical linear direction in which they seek their arguments. Some simplified example lexical entries are given below:

Olivia := np *the* := $\mathsf{np}/_{\star}\mathsf{n}$

Finn := np *saw* := $(\mathsf{s}\backslash\mathsf{np})/\mathsf{np}$

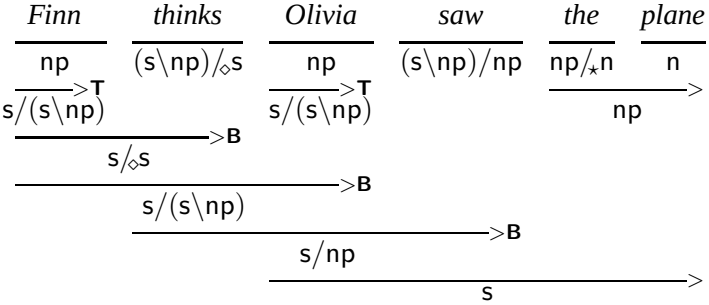
plane := n *thinks* := $(\mathsf{s}\backslash\mathsf{np})/_{\diamond}\mathsf{s}$

The most basic rules are forward (>) and backward (<) application. CCG also utilizes rules based on the composition (**B**), type-raising (**T**), and substitution (**S**) combinators of combinatory logic. The rules of CCG are:[2]

(>)	$X/_{\star}Y \quad Y \Rightarrow X$	(<)	$Y \quad X\backslash_{\star}Y \Rightarrow X$
(>**B**)	$X/_{\diamond}Y \quad Y/_{\diamond}Z \Rightarrow X/_{\diamond}Z$	(<**B**)	$Y\backslash_{\diamond}Z \quad X\backslash_{\diamond}Y \Rightarrow X\backslash_{\diamond}Z$
(>$\mathbf{B}_{\times}$)	$X/_{\times}Y \quad Y\backslash_{\times}Z \Rightarrow X\backslash_{\times}Z$	(<$\mathbf{B}_{\times}$)	$Y/_{\times}Z \quad X\backslash_{\times}Y \Rightarrow X/_{\times}Z$
(>**T**)	$X \Rightarrow Y/(Y\backslash X)$	(<**T**)	$X \Rightarrow Y\backslash(Y/X)$

Each rule is keyed to a modality; this allows lexical items to selectively utilize some rules but not others. For example, the $/_{\star}$ slash on the category for *the* keeps the composition rules from causing ungrammatical word order permutations within English noun phrases. See Baldridge (2002) and Baldridge and Kruijff (2003) for full explication of the computational and linguistic significance of modalities.

Though the application rules do the majority of the work, the others are crucial for building the non-standard constituents for which categorial grammars are well-known. With these rules and the categories given above, we can provide an incremental derivation for a sentence such as 'Finn thinks Olivia saw the plane':



The constituent s/np derived above for 'Finn thinks Olivia saw' is also used in analyses for relative clauses like 'the plane that [Finn thinks Olivia saw]' and right-node raising sentences like '[Kestrel heard] and [Finn thinks Olivia saw] the plane'.

There has been a great deal of work in computational linguistics using CCG over the past two decades, and there is an even greater degree of activity in recent years. A major development was the creation of CCGbank (Hockenmaier and Steedman, 2007), which has allowed the creation of fast and accurate probabilistic CCG parsers for producing deep dependencies (Hockenmaier, 2003; Bos et al., 2004; Clark and Curran, 2007). CCG has also been used to induce semantic parsers from sentences paired with logical forms (Zettlemoyer and Collins, 2007).

Work with OpenCCG represents another major branch of CCG research. It is used for testing and developing syntactic and semantic analyses (Bierner, 2001; Baldridge, 2002; Kruijff and Baldridge, 2004; Gerstenberger and Wolksa, 2005) and for research into CCG parsing and realization (Hockenmaier et al., 2004; White and Baldridge, 2003; White, 2006b; White et al., 2007). It performs parsing/realization in the systems of a number of projects, many of which are given in Figure 2. Most of these are dialog systems, including natural language interfaces for robots (CoSy, JAST, and INDIGO) and MP3 systems (SAMMIE).

[2]We exclude substitution here for space reasons. An example is >**S**: $(X/_{\diamond}Y)/_{\diamond}Z \quad Y/_{\diamond}Z \Rightarrow X/_{\diamond}Z$.

Project	References/Website
AdaRTE	Rojas-Barahona (2007) http://www.labmedinfo.org/research/adarte/adarte.htm
COMIC	Foster and White (2005, 2007); Nakatsu and White (2006); White (2006a) http://www.hcrc.ed.ac.uk/comic/
CoSy	Kruijff et al. (2007) http://www.cognitivesystems.org
CrAg	Isard et al. (2006) http://www.hcrc.ed.ac.uk/crag/
DIALOG	Wolska and Kruijff-Korbayová (2004); Benzmüller et al. (2007) http://www.ags.uni-sb.de/~dialog/
FLIGHTS	Moore et al. (2004)
INDIGO	http://www.ics.forth.gr/indigo/
JAST	Rickert et al. (2007) http://www.euprojects-jast.net/
Methodius	Isard (2007) http://www.ltg.ed.ac.uk/methodius/
SAMMIE	Becker et al. (2006) http://www.talk-project.org

Figure 2: Example projects that use OpenCCG for parsing and realization.

4 OpenCCG's XML Format

The underlying native specification format of OpenCCG is XML. Grammatical information is split across six interdependent files, some of which define components that were directly inspired by XTAG (Doran et al., 2000). Each file defines a major component of the grammar, including (a) a structured lexicon containing families of lexical entries, (b) a morphological database pairing words with their stems and morphological features, (c) morphological macros instantiating feature values on lexical entries, (d) a hierarchy of typed features, (e) a set of parameterized CCG rules, and (f) a testbed of sentences used for simple regression testing.

As an example of what is involved in creating lexical entries in OpenCCG, Figure 3 shows a fragment of the XML lexicon, morphology, and typed-feature files for an Ojibwe[3] grammar. This fragment defines a noun family that has a single lexical category, which contains three lexical items: *gaago* 'porcupine', *kwe* 'woman', and *mzinig* 'book'. Each lexical item inflects with four forms: singular proximate, singular obviative, plural proximate, and plural obviative. The inflectional suffixes vary according to the stem. *Gaago* and *kwe* are of animate gender, while *mzinig* is inanimate. A basic feature hierarchy is defined, consisting of person (2nd, 1st, 3rd, non3rd), number (singular, plural), gender (animate, inanimate), and obviation status (proximate, obviative). Note that the majority of the XML for defining the feature hierarchy has been truncated for space reasons.

Developing grammars directly in XML is time-consuming and error prone. XML was designed as a format to standardize communication of data among computers, not for direct editing by humans. Furthermore, OpenCCG's XML format contains many redundancies and interdependencies, leading to errors when a change is made in one place and not propagated elsewhere. For example, the association between the part of speech `N` and the three lexical items is declared in the lexicon file and in multiple places throughout the morphology file. The declarations of multiple inflected forms of the same stem are also highly repetitive and fail to express any generalizations over the forms. Finally, the features attached to

[3]Ojibwe is an Algonquian language of the upper Great Lakes region and southeastern Ontario.

Ojibwe lexicon file

```
<family name=‘‘N’’ pos=‘‘N’’ closed=‘‘true’’>
  <entry name=‘‘Entry-1’’>
    <atomcat type=‘‘n’’>
      <fs id=‘‘1’’>
        <feat attr=‘‘index’’>
          <lf>
            <nomvar name=‘‘X’’/>
          </lf>
        </feat>
      </fs>
      <lf>
        <satop nomvar=‘‘X’’>
          <prop name=‘‘[‘DEFAULT’]’’/>
        </satop>
      </lf>
    </atomcat>
  </entry>
  <member stem=‘‘mzinigna’’/>
  <member stem=‘‘gaago’’/>
  <member stem=‘‘kwe’’/>
</family>
```

Ojibwe morphology file

```
<entry word=‘gaago’ macros=‘@3rd @sg @prox @anim’ pos=‘N’ stem=‘gaago’/>
<entry word=‘gaagon’ macros=‘@3rd @sg @obv @anim’ pos=‘N’ stem=‘gaago’/>
<entry word=‘gaagog’ macros=‘@3rd @pl @prox @anim’ pos=‘N’ stem=‘gaago’/>
<entry word=‘gaagong’ macros=‘@3rd @pl @obv @anim’ pos=‘N’ stem=‘gaago’/>
<entry word=‘mzinigna’ macros=‘@3rd @sg @prox @inan’ pos=‘N’ stem=‘mzinig’/>
<entry word=‘mzinignan’ macros=‘@3rd @sg @obv @inan’ pos=‘N’ stem=‘mzinig’/>
<entry word=‘mzinignag’ macros=‘@3rd @pl @prox @inan’ pos=‘N’ stem=‘mzinig’/>
<entry word=‘mzinignang’ macros=‘@3rd @pl @obv @inan’ pos=‘N’ stem=‘mzinig’/>
<entry word=‘kwe’ macros=‘@3rd @sg @prox @anim’ pos=‘N’ stem=‘kwe’/>
<entry word=‘kwewan’ macros=‘@3rd @sg @obv @anim’ pos=‘N’ stem=‘kwe’/>
<entry word=‘kwen’ macros=‘@3rd @pl @prox @anim’ pos=‘N’ stem=‘kwe’/>
<entry word=‘kwenwan’ macros=‘@3rd @pl @obv @anim’ pos=‘N’ stem=‘kwe’/>

<macro name="@anim">
  <fs id="1" attr="GEND" val="anim"/>
</macro>
<macro name="@inan">
  <fs id="1" attr="GEND" val="inan"/>
</macro>
...
```

Ojibwe typed-feature file

```
  <type name="GEND"/>
  <type name="anim" parents="GEND"/>
  <type name="inan" parents="GEND"/>
  <type name="OBV"/>
  <type name="prox" parents="OBV"/>
  <type name="obv" parents="OBV"/>
...
```

Figure 3: XML specifying an Ojibwe noun family containing three lexical items.

```
feature {
  gend<1>: anim inan;
  pers<1>: 1st 2nd 3rd;
  num<1>: sg pl;
  obv<1>: prox obv;
}

family N {
  entry: n<1>[X]: X(*);
}

def noun(stem, obv-end, pl-end, gend) {
  word stem:N {
    stem: 3rd sg prox gend;
    stem.obv-end: 3rd sg obv gend;
    stem.pl.end: 3rd pl prox gend;
    stem.obv-end.pl-end: 3rd pl obv gend;
  }
}

noun(gaago, n, g, anim)
noun(mzinigna,n, g, inan)
noun(kwe, wan, n, anim)
```

Figure 4: DotCCG equivalent of the Ojibwe XML fragment given in Figure 3.

inflected forms need to be declared both in the morphology and typed-feature files.

5 DotCCG: shorthand for OpenCCG

DotCCG was created to overcome the deficiencies of direct XML input of grammars.[4] It is a human-friendly format which seeks to eliminate redundancy and boost expressiveness while requiring far fewer lines of code than raw XML. It was designed to be concise, flexible, and easy to use, and specifically intended for direct input and editing using a text editor. The grammar is placed in a single `.ccg` file, with declarations in any order and freely grouped or separated. All of the XML required by OpenCCG is generated by passing the `.ccg` file through `ccg2xml`, a program written in Python and implemented using PLY.[5] Handling the dependencies in this way greatly reduces the burden on the grammar developer and increases the grammar's modularity and maintainability. Figure 4 shows the full DotCCG equivalent of the Ojibwe XML fragment.

DotCCG was designed with an emphasis on making the grammar specification language as tolerant and expressive as possible. The general feel of DotCCG syntax is like C, Java, or Perl. However, the syntax is very forgiving on the usage of commas, semicolons, and other terminators and separators. In fact, this punctuation can

[4] An existing solution using XSLT transformations is available (Bozşahin et al., 2006) but requires significant technical expertise.

[5] PLY, available at `http://www.dabeaz.com/ply/`, is a package that provides functionality equivalent to Lex and YACC.

be omitted as long as no syntactic ambiguity will result.[6] This eliminates one of the major stumbling blocks grammar engineers typically face when adjusting to an unfamiliar format. Although DotCCG looks similar to a traditional programming language, the format is intended for use by non-programmers as well as programmers. Its semantics are on a higher level than most programming languages, and it consistently favors expressiveness and ease-of-use over rigid formatting. It is lenient in its handling of commas and other punctuation, and most syntactic elements can be omitted if not needed, with sensible default behavior.

The five sections of DotCCG grammars are described below. Each section is implemented within the `.ccg` file with a series of declarations.

Features — Declaring features allows for simple specification of and reference to features in lexical entries and categories. For example, the Ojibwe grammar fragment shown above creates a simple feature structure with person, number, gender and obviation features. The character in angle brackets following the name of the feature is required by OpenCCG and relates to its mechanism for unifying feature values across lexical categories. Features in DotCCG can also be nested and allow for multiple inheritance.

Words — Word declarations associate lexical items with particular categories and features as well as specifying morphological information. The following are two examples for English, one showing a simple word `the` of family `Det`, and the other showing a pseudo-word `pro1` of family `Pro` and semantic class `animate`, with various surface realizations according to case and number:

(1)
```
word the:Det;
word pro1:Pro(animate) {
  I: 1st sg nom;
  me: 1st sg acc;
  we: 1st pl nom;
  us: 1st pl acc; }
```

Word declarations are commonly placed inside of expansions, as in the `noun` expansion in the Ojibwe fragment. See section 6 for further discussion.

Rules — This section specifies the rules allowed or disallowed in the particular grammar. The CCG rules enabled by default are the forward and backward varieties of application, harmonic composition, and crossed composition. Substitution rules must be invoked explicitly. OpenCCG supports the modalities of Baldridge and Kruijff (2003), so the applicability of the rules is controlled by the use of these modalities on slashes in categories.

Type-raising can be invoked and restricted to particular argument and result categories. For example, the following declaration adds the rule $\mathsf{np} \Rightarrow \mathsf{s\$}/(\mathsf{s\$}\backslash\mathsf{np})$:

(2)
```
typeraise + $: np => s;
```

Type-changing rules can also be added. The following would be one way of implementing pro-drop in a grammar ($\mathsf{s}_{\mathsf{fin}}\backslash\mathsf{np}_{\mathsf{nom}}$ changes to $\mathsf{s}_{\mathsf{fin}}$):

[6] The only situation where separators are required occurs in arguments to textual expansions, which can consist of arbitrary text.

(3)
```
typechange: s[finite]\np[nom] => s[finite] ;
```

Lexicon/Categories — Lexical families consist of one or more category declarations and optional specification of lexical items which are members of that family. For example, in English the lexical family `Det` has just a single category: $\mathtt{np}/_{\diamond}\mathtt{n}$. The family for dative alternation verbs, though, has two possible categories, one for the double object construction and one for the pp-complement construction.

There are two types of intransitive verbs in Ojibwe, those with an animate subject (`VAI`) and those with an inanimate one (`VII`). The category declarations for these two families are shown below.[7] Features are enclosed in square brackets, and the final term, after the second colon, is the semantic representation.

(4)
```
family VAI {
  entry: s<8>[E] | n<1>[anim X]: E:action (* <actor>X:sem-obj); }
family VII {
  entry: s<8>[E] | n<1>[inan X]: E:action (* <actor>X:sem-obj); }
```

Testbed — The testbed contains a list of constructions and the number of parses the grammar is expected to find for each construction. The testbed facility provides for simple regression testing, e.g. whether the expected number of parses are obtained and whether sentences can be reverse realized from their parse results.[8]

(5)
```
testbed {
  wiisniwag gaagog: 1;     ## the porcupines eat
  wiisniwag mzinignan: 0;  ## *the books eat }
```

6 Expansions with DotCCG

6.1 Introduction to expansions

Most grammar engineering systems provide mechanisms to reduce redundancy. These support the expression of various levels of generalization while providing power and flexibility. For example, XLE has macros and parameterized rules, and the LKB uses types to capture lexical and syntactic regularities. DotCCG offers parameterized string-rewrite functions that we call **expansions**.

We chose expansions as our primary abstraction mechanism because they are flexible and easy to use. The definitions directly specify their expansions and mirror what will be inserted and processed when an expansion call is made. The lack of a need to "program" data makes expansions easy to use for non-programmers. Furthermore, expansions can abstract over *any* portion of a text, regardless of whether such a usage was anticipated in the initial design of the grammar. A programmed mechanism, by contrast, either has to impose a uniform structure on all specifications or have separate mechanisms to handle each type of structure.

[7] The numbers in angle brackets represent the feature structure ID assigned to the category. These are global for the grammar: this is one of the main weaknesses of OpenCCG grammar specification.

[8] The sentences given here are not surface forms but rather idealizations of Ojibwe sentences prior to phonological processes.

Our expansions are quite similar to XLE macros and parameterized rules, but with greater syntactic flexibility, fewer constraints, and increased string manipulation capabilities. The expansions allow DotCCG to handle quite complex morphology without having to interface with external morphological analyzers. Of course, there are many advantages to interfacing with existing tools such as morphological analyzers, and XLE grammars have been successfully interfaced with finite-state analyzers (Kaplan et al., 2004). Along with the flexible syntax, of course, comes a reduced level of control over expansions, for good and for ill. Unlike XLE, for example, no error occurs if not all input arguments appear in the output specified for the expansion. While this may allow a user to write expansions with unexpected consequences, it gives the expansions a broader range of possible functionalities.

A disadvantage to our solution is that expansions are a meta-theoretic construct and as such are not visible in the underlying grammar framework itself. By the time OpenCCG sees the grammar, all expansions have taken place, and there is no record of how the expanded structures were constructed. Thus, it may be hard to debug a problem occurring in a group of deeply nested expansions,[9] and injudicious use of expansions can lead to quite obfuscated code.

A simplified version of an expansion contained in Figure 4 is given in (6). It defines a parameterized expansion named `noun`, with two formal parameters `stem` and `gend`. Calling this expansion with `noun(gaago, anim)` produces the expanded text given in (7).

(6)
```
def noun(stem, gend) {
  word stem:N {
    stem: 3rd sg prox gend;
    stem.n: 3rd sg obv gend;
    stem.g: 3rd pl prox gend;
    stem.ng: 3rd pl obv gend;
}}
noun(gaago, anim)
```

(7)
```
word gaago:N {
     gaago: 3rd sg prox anim;
     gaagon: 3rd sg obv anim;
     gaagog: 3rd pl prox anim;
     gaagong: 3rd pl obv anim;
}
```

Occurrences of formal parameters inside of the expanded text have been replaced with their actual values, and strings separated by a period have been concatenated.

6.2 Nested expansions for complex morphology

Expansions in conjunction with word declarations make it easy to express arbitrarily complicated morphology. They are used extensively in DotCCG grammars. Expansions can be nested inside of each other without restriction, allowing almost any pattern of syncretism to be factored out with little or no repetition.

As an example, a large fragment of Classical Arabic, including all noun, verb, adjective and pronoun morphology and correct handling of resumptive pronouns in relative clauses, was implemented in an 800-line `.ccg` file (about 20% of which is comments). It produces a vocabulary with more than 1100 words. The following portion shows how some of the complexities of present-tense verbs can be handled:

[9] To help alleviate this, `ccg2xml` provides options to debug expansion problems, such as displaying the text after expansion processing.

```
# Arabic verb fragment.  We are omitting a great deal: dual number,
# jussive mood, all past tense forms, doubled verbs, etc.

# All present-tense verbs can be reduced to four forms (five, counting the
# dual), plus prefixes.

def gen-pres(mood, fsing, fsing-fem, fplur-masc, fplur-fem) {
  # A special phonological rule collapses adjacent glottal stops: e.g.
  # _a_kulu -> _aakulu.  We implement using regsub() -- see below.
  _ . regsub('^([aiu])_', '\1\1', fsing): pres, mood, 1st, sg;
  t.fsing: pres, mood, 2nd, m, sg;
  t.fsing-fem: pres, mood, 2nd, f, sg;
  y.fsing: pres, mood, 3rd, m, sg;
  t.fsing: pres, mood, 3rd, f, sg;

  n.fsing: pres, mood, 1st, pl;
  t.fplur-masc: pres, mood, 2nd, m, pl;
  t.fplur-fem: pres, mood, 2nd, f, pl;
  y.fplur-masc: pres, mood, 3rd, m, pl;
  y.fplur-fem: pres, mood, 3rd, f, pl;
}

# Most verbs can be reduced to two stems (one for feminine plural and one
# for all other cases), with a specific set of endings, which vary between
# indicative and subjunctive.

def two-form-pres-indic(formv, formc) {
  gen-pres(indic, formv.u, formv.iina, formv.uuna, formc.na)
}
def two-form-pres-subj(formv, formc) {
  gen-pres(subj, formv.a, formv.ii, formv.uu, formc.na)
}

# The basic Arabic verb conjugations are strong, second-weak, doubled, and
# third-weak.  Strong verbs have one stem, while second-weak and doubled
# (not included here) have two.  Second-weak verbs have many subtypes, so
# we require that each verb give both stems.

def strong-pres(form) {
  two-form-pres-indic(form, form)
  two-form-pres-subj(form, form)
}
def 2nd-weak-pres(formv, formc) {
  two-form-pres-indic(formv, formc)
  two-form-pres-subj(formv, formc)
}

# Third-weak verbs merge stem and endings, and have three subtypes, ending
# in -aa, -ii, or -uu in the base form.

def 3rd-weak-pres-aa(form) {
  gen-pres(indic, form.aa, form.ayna, form.awna, form.ayna)
  gen-pres(subj, form.aa, form.ay, form.aw, form.ayna)
}
def 3rd-weak-pres-ii(form) { ... }       # Omitted to save space
def 3rd-weak-pres-uu(form) { ... }       # Omitted to save space

# Here we provide expansions for the various conjugations. (These are
# appropriate for a full verb paradigm, including both present and past
# tense, but the past-tense expansion has been commented out.) Each lexical
# entry specifies the past-tense stem (which is used to form the verb's
# "dictionary form"), some properties (valency and English translation), a
```

```
# present-tense stem, and any other required info.  Second-weak verbs have
# two stems for each of present and past, while third-weak verbs specify
# the past (ay/aw/ii) and present (ii/uu/aa) subtypes.

def strong-verb(past, props, pres) {
  word past: props {
    strong-pres(pres)
  }
}
def 2nd-weak-verb(pastv, props, pastc, presv, presc) {
  word pastv: props {
    2nd-weak-pres(presv, presc)
  }
}
def 3rd-weak-verb(past_stem, props, past_type, pres_stem, pres_type) {
  word past_stem . past_type: props {
  # Note how we are dynamically constructing the expansion call!
    3rd-weak-pres- . pres_type(pres_stem)
  }
}

# Here we declare the actual verbs.  These are identical to how they appear
# in the full grammar, where each one expands to 52 individual forms.

strong-verb(katab, TransV(pred=write), aktub)
2nd-weak-verb(kaan, TransV(pred=be), kun, akuun, akun)
3rd-weak-verb(_a9T, DitransV(pred=give), ay, u9T, ii)
```

Note that Arabic verbs are formed in a complex fashion involving prefixes, suffixes, and internal stem changes. In general, there are different stems for past and present, and many verbs have two stems in each tense. The endings also vary in complicated ways among different moods and classes. By the judicious use of nested expansions, however, we can reduce each lexical entry down to a very small size, where only the class and underivable stem forms are given. The following table shows the indicative and subjunctive moods generated for the three sample verbs: *kataba* ‘write’ (strong verb), *kaana* ‘be’ (2nd-weak verb; note the short vowel in *yakunna*), and *’a9Taa* ‘give’ (3rd-weak verb).

	kataba.IND	kataba.SBJ	kaana.IND	kaana.SBJ	’a9Taa.IND	’a9Taa.SBJ
1sg	’aktubu	’aktuba	’akuunu	’akuuna	’a9Taa	’a9Taa
2sg.m	taktubu	taktuba	takuunu	takuuna	ta9Taa	ta9Taa
2sg.f	taktubiina	taktubii	takuuniina	takuunii	ta9Tayna	ta9Tay
3sg.m	yaktubu	yaktuba	yakuunu	yakuuna	ya9Taa	ya9Taa
3sg.f	taktubu	taktuba	takuunu	takuuna	ta9Taa	ta9Taa
1pl	naktubu	naktuba	nakuunu	nakuuna	na9Taa	na9Taa
2pl.m	taktubuuna	taktubuu	takuunuuna	takuunuu	ta9Tawna	ta9Taw
2pl.f	taktubna	taktubna	takunna	takunna	ta9Tayna	ta9Tayna
3pl.m	yaktubuuna	yaktubuu	yakuunuuna	yakuunuu	ya9Tawna	ya9Taw
3pl.f	yaktubna	yaktubna	yakunna	yakunna	ya9Tayna	ya9Tayna

6.3 Expansions and built-in functions

Expansions are made even more powerful by three built-in expansion functions, which provide the full power of regular-expression matching and replacement. **`regsub`(PATTERN, REPLACEMENT, TEXT)** returns TEXT, but with all occurrences of PATTERN (a regular expression) replaced with REPLACEMENT (a standard regular expression substitution expression, including backreferences

to captured text). **`ifmatch(PATTERN, TEXT, IF-TRUE, IF-FALSE)`** matches regular expression PATTERN against TEXT, returning IF-TRUE if it matches and IF-FALSE otherwise. **`ifmatch-nocase`** functions similarly, but the matching is case-insensitive.

An example of the usage of these functions is computing English plurals:

(8)
```
def pluralize(Word) {
 ifmatch('^.*[aeiou][oy]\$', Word, Word . s,
  ifmatch('^.*([sxoy]|sh|ch)\$', Word,
    regsub('^(.*)y\$', '\1i', Word) . es,
       Word . s))}
```

This definition handles both *-s* and *-es* endings, including words ending with *-y*. It will correctly map *cat, box, boy, lady* into *cats, boxes, boys, ladies*, respectively.

Expansions in combination with `regsub` can also be used to handle complex cases such as infixation in Tagalog, where verbs can take on a number of different voice affixes that single out a particular participant in an event (Kroeger, 1993). For example, the stem *bili* 'buy' can take the inflected forms *bumili* (actor), *binili* (object), *binilhan* (dative), *ipinambili* (instrumental), *ibinili* (benefactive), and *kabibili* (recent-perfective). The following DotCCG fragment demonstrates this, breaking the stem into two parts to allow for infixation and using `regsub` to handle reduplication in *kabibili* and the deletion of *i* and insertion of *h* in *binilhan*:[10]

(9)
```
def reduplicate (Word) { regsub('^(..)(.*)$', '\1\1\2', Word) }

def regular_verb (St1, St2, LF) {
  St1 . um . St2                                :VerbAV (pred=LF);
  St1 . in . St2                                :VerbOV (pred=LF);
  St1 . in . regsub('^(.*)i$', '\1h', St2) . an :VerbDV (pred=LF);
  ipinam . St1 . St2                            :VerbIV (pred=LF);
  i . St1 . in . St2                            :VerbBV (pred=LF);
  ka . reduplicate(St1 . St2)                   :VerbRP (pred=LF);
}

regular_verb (b, ili, buy);
```

6.4 Expansions for inheritance-like effects

In grammar engineering, inheritance is often used to eliminate redundancy by allowing partial definitions to be used as a base upon which further definitions are built. Inheritance (including defaults) is in fact one of the core aspects of the LKB system (in that it uses the Type Description Language) which allows complex linguistic signs to be built elegantly with a series of incremental declarations using inheritance. Villavicencio (2002) utilizes inheritance in the LKB to create a categorial grammar which defines the transitive verb and sentential complement categories as extensions of the intransitive verb category, ditransitives as extensions of transitives, and so on.

[10] Tagalog verbal morphology in general is of course much more complex than for this one stem, but this shows in principle how such patterns can be captured.

OpenCCG does not provide support for inheritance in general, but the XML format does provide special declarations to allow the inheritance patterns used by Villavicencio (Baldridge, 2002). Interestingly, expansions provide an alternative way to achieve this effect:

(10)
```
def iv_cat (PostSyn, MoreSem) {
  s[E] \ np[X nom] PostSyn: E(* <Subject>X MoreSem)
}
def tv_cat (PreSyn, PostSyn, MoreSem) {
  iv_cat(PreSyn / np[Y acc] PostSyn, <DirectObject>Y MoreSem)
}
family IntransV(V) {
  entry: iv_cat(,);
}
family TransV(V) {
  entry: tv_cat(,,);
}
family DitransV(V) {
  entry: tv_cat(            , / np[Z acc] , <IndirectObject>Z);
  entry: tv_cat(/ pp[Z acc] ,             , <IndirectObject>Z);
}
```

This shows the declaration of a parameterized expansion, `iv_cat`, which defines a category (and its semantics) while leaving variables embedded in it that allow further syntactic and semantic arguments to be added. The `tv_cat` definition in turn builds on `iv_cat`, allowing arguments to be inserted either before or after the direct object. The `DitransV` family makes use of this, providing entries that implement both double-object and PP-shifted forms of a ditransitive verb.

An important aspect of OpenCCG that supports this sort of inheritance in the semantics is the use of hybrid logics (Baldridge and Kruijff, 2002) for representing logical forms as a flattened set of elementary predications.[11]

Expansions provide a very flexible means to generalize not only how words are defined (morphology), but also how categories are constructed. The space savings (in terms of the amount of grammar code which a grammar engineer is confronted with) can be orders of magnitude in size: for example, the 16 DotCCG lines given above translate into 200+ (harder to maintain) lines in OpenCCG's XML.

Of course, constructing words and categories in this way can make it difficult to see exactly what the lexicon looks like directly in DotCCG. VisCCG, described in detail in the next section, is able to display—at various levels of granularity—the resulting lexicon, both the words and the categories that are available, *while* the grammar is being edited for faster development and debugging.

7 VisCCG: wiki-style GUI editing

DotCCG provides a great deal of power to the grammar engineer with or without a GUI. However, for many users, a GUI is still an important means for using a grammar platform effectively, and visualization can help even the advanced developer

[11] Similar representations, e.g. Minimal Recursion Semantics, would work equally well in this regard.

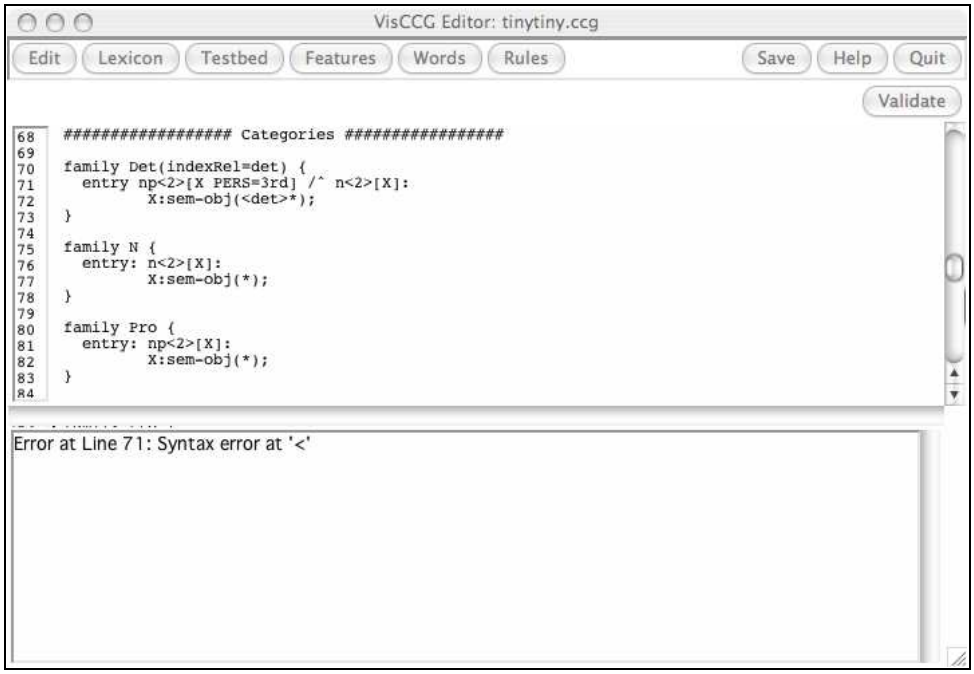


Figure 5: Debugging with CCG

see the structure and definitions of a grammar more effectively. VisCCG takes a wiki-like approach, which enables grammar visualization while never taking the developer too far from the underlying definitions. The goal is to allow new users to begin using the system very quickly without constraining advanced users within the bounds of purely-graphical editing (as opposed to textual editing in conjunction with visualization).

When starting new grammars, it is often useful to iron out nuances of the lexicon, rules or morphology before expanding the grammar significantly. VisCCG allows users to begin with a few essential aspects such as rules and features and then visualize and debug them even without a complete grammar. This adheres to the software engineering paradigm of rapid application development. Individual sections can be edited and visualized independently, enhancing the maintainability of the grammars.

VisCCG allows the user to begin a new grammar with a template that organizes the modules of the grammar. This simplifies bootstrapping of grammar development and also helps maintain a de facto standard for grammars developed using the system – though users are free to deviate from it if they wish. More importantly, as the grammar evolves over time with perhaps multiple people contributing to and refining the grammar, the subsection to be edited is easily localized.

IDEs for programming languages provide detailed debugging information for syntax errors in source code. Similarly, VisCCG identifies syntax errors in the DotCCG source and highlights them for users to fix, as illustrated in figure 5.

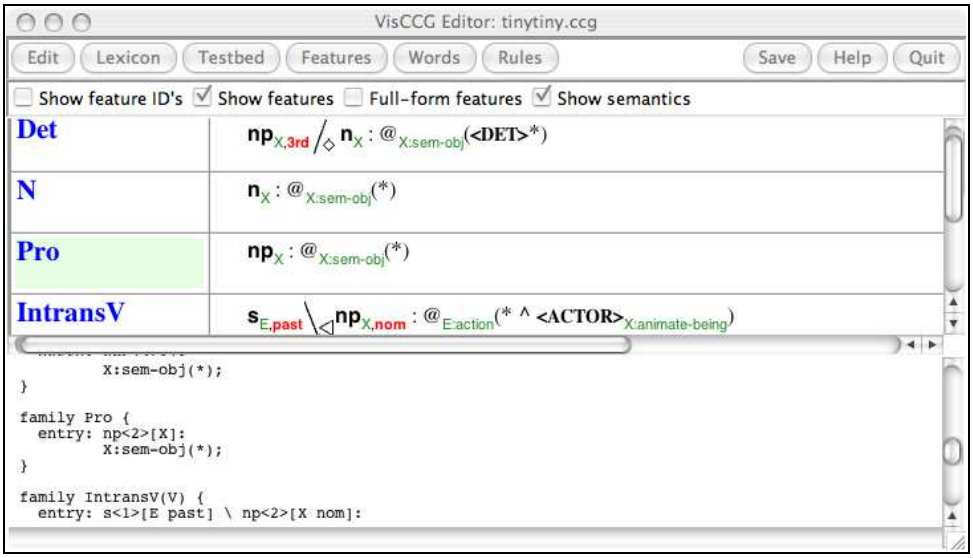


Figure 6: Local editing in Lexicon mode. The *Pro* family has been selected for editing from the graphical display (the top pane); this opens the grammar file for editing at the location which specifies the family (the lower pane).

The line numbers displayed beside the source help localize and isolate individual errors. This capability alone dramatically improves development time, even for experienced developers.

The visualization of a grammar is often very different from what we can express in text. VisCCG enables users to view the grammar at various levels of granularity, allowing the user to spot errors and generalizations easily and without needing to view unrelated information, such as details of features or semantics. As with wikis, VisCCG allows a user to locally edit a small part of the grammar. This is made possible by the terseness of DotCCG, which itself is made possible by the fact that CCG categories can be concisely specified in a linear format. VisCCG additionally allows editing to occur while the user continues to view the graphical representation of the grammar. This feature allows seamless editing of one category definition in the 'Lexicon' tab while other categories are visualized at the desired granularity. Also, the results of such an edit are immediately visible, allowing the user to try out various features before saving changes. An example of editing the 'Pro' family is illustrated in Figure 6.

VisCCG has many different modes of visualization. The initial screen is a basic editor that allows the user to develop their grammar from scratch. The 'Testbed' tab also the user to input new test sentences, and the 'Feature' tab provides a straightforward means of editing the feature hierarchy. The 'Words' tab lists all available lexical items as well as their various inflected forms. This is especially useful for checking the output of expansions, and in particular expansions which produce words based on stems and morphological regularities. This rich set of capabilities

enables the user to update the grammar with a tight editing and visualization cycle. These capabilities also ease the process of grammar development by allowing the user to focus on particular sections, while being able to switch back to any other view easily.

8 Uses of and resources for DotCCG and VisCCG

VisCCG has been used so far in both graduate and undergraduate classes to teach both CCG and grammar engineering. Even students with little computational background were able to use the tools effectively with just a single lab session. Previous courses that used the XML format proved it to be frustrating for students, and required many sessions for them to use at all (and certainly not master). This experience was in fact the genesis of DotCCG.

For teaching purposes and to facilitate wider use of VisCCG, we have developed a wiki[12] which focuses on the various computational and linguistic resources available for learning to use and for using the system. These resources include tutorials, links to software download sites, and access to a number of grammars which have been developed using VisCCG. Among these are small (in many cases tiny) grammars for Tagalog, Ojibwe, French, and Hungarian, as well as some small-domain English grammars. Though no truly broad-coverage grammar has been developed with our new tools to date, they are already being used to develop grammars used in some of the projects listed in Figure 2, including AdaRTE, INDIGO, and Methodius.

We see a number of interesting directions for development of the tools discussed in this paper. In addition to refining the presentation of the various components of the grammar, it would be extremely useful to be able to run the OpenCCG parser from inside VisCCG. It would also be interesting to expand the grammar initialization process to include something like the customization questionnaire used in the Grammar Matrix (Bender and Flickinger, 2005).

9 Conclusion

We have presented an overview and motivation of our work on a set of tools for improving grammar engineering for OpenCCG. The approach is two-pronged in that it improves textual representations of CCG grammars via the DotCCG format and it allows the information in such grammars to be visualized with VisCCG. VisCCG furthermore supports wiki-style editing that enables users to edit small sections of the grammar while visualizing the rest and to see the results of their edits immediately. However, the use of VisCCG for editing is optional – DotCCG grammars can be edited with any plain-text editor as well. The simplicity, flexibility and power

[12] `http://comp.ling.utexas.edu/wiki/doku.php/openccg`

of DotCCG and the optional availability of VisCCG is crucial for supporting the needs of both new and advanced users.

References

Baldridge, Jason. 2002. *Lexically Specified Derivational Control in Combinatory Categorial Grammar*. Ph. D.thesis, University of Edinburgh.

Baldridge, Jason and Kruijff, Geert-Jan. 2003. Multi-Modal Combinatory Categorial Grammar. In *Proceedings of EACL*, Budapest, Hungary.

Baldridge, Jason and Kruijff, Geert-Jan M. 2002. Coupling CCG and Hybrid Logic Dependency Semantics. In *Proceedings of ACL*.

Becker, Tilman, Blaylock, Nate, Gerstenberger, Ciprian, Kruijff-Korbayov, Ivana, Korthauer, Andreas, Pinkal, Manfred, Pitz, Michael, Poller, Peter and Schehl, Jan. 2006. Natural and intuitive multimodal dialogue for in-car applications: The SAMMIE system. In *Proceedings of the ECAI Sub-Conference on Prestigious Applications of Intelligent Systems (PAIS 2006)*, Riva del Garda, Italy.

Bender, Emily M. and Flickinger, Dan. 2005. Rapid Prototyping of Scalable Grammars: Towards Modularity in Extensions to a Language-Independent Core. In *Proceedings of the 2nd International Joint Conference on Natural Language Processing IJCNLP-05 (Posters/Demos)*, Jeju Island, Korea.

Benzmüller, Christoph, Horacek, Helmut, Kruijff-Korbayova, Ivana, Pinkal, Manfred, Siekmann, Jörg and Wolska, Magdalena. 2007. Natural Language Dialog with a Tutor System for Mathematical Proofs. In Ruqian Lu, Jörg Siekmann and Carsten Ullrich (eds.), *Cognitive Systems*, volume 4429 of *LNAI*, Springer.

Bierner, Gann. 2001. *Alternative Phrases: Theoretical Analysis and Practical Applications*. Ph. D.thesis, Division of Informatics, University of Edinburgh.

Bos, Johan, Clark, Stephen, Steedman, Mark, Curran, James R. and Hockenmaier, Julia. 2004. Wide-Coverage Semantic Representations from a CCG Parser. In *Proceedings of COLING-04*, pages 1240–1246.

Bozşahin, Cem, Kruijff, Geert-Jan M. and White, Michael. 2006. Specifying Grammars for OpenCCG: A Rough Guide. http://openccg.sf.net/.

Butt, Miriam, King, Tracy Holloway, Niño, María-Eugenia and Segond, Frédérique. 1998. *A Grammar Writer's Cookbook*. Stanford, CA: CSLI.

Clark, Stephen and Curran, James. 2007. Wide-Coverage Efficient Statistical Parsing with CCG and Log-Linear Models. *Computational Linguistics* 33(4).

Copestake, Ann. 2002. *Implementing Typed Feature Structure Grammars*. Stanford, CA: CSLI Publications.

Doran, Christine, Hockey, Beth Ann, Sarkar, Anoop, Srinivas, B. and Xia, Fei. 2000. Evolution of the XTAG System. In Anne Abeillé and Owen Rambo (eds.), *Tree Adjoining Grammars: Formalisms, Linguistic Analysis and Processing*, pages 371–404, Stanford, CA: CSLI Publishing.

Foster, Mary Ellen and White, Michael. 2005. Assessing the impact of adaptive generation in the COMIC multimodal dialogue system. In *Proceedings of the IJCAI 2005 Workshop on Knowledge and Reasoning in Practical Dialogue Systems*, Edinburgh.

Foster, Mary Ellen and White, Michael. 2007. Avoiding repetition in generated text. In *Proceedings of ENLG*, Schloss Dagstuhl.

Gerstenberger, Ciprian-Virgil and Wolksa, Magdalena. 2005. Introducing Topological Field Information into CCG. In *Proceedings of the 10th ESSLLI Student Session*, pages 62–74, Edinburgh, UK.

Hockenmaier, Julia. 2003. Parsing with Generative Models of Predicate-Argument Structure. In *Proceedings of ACL*.

Hockenmaier, Julia, Bierner, Gann and Baldridge, Jason. 2004. Extending the coverage of a CCG System. *Research in Language and Computation* 2, 165–208.

Hockenmaier, Julia and Steedman, Mark. 2007. CCGbank: A Corpus of CCG Derivations and Dependency Structures Extracted from the Penn Treebank. *Computational Linguistics* 33(3), 355–396.

Isard, Amy. 2007. Choosing the Best Comparison Under the Circumstances. In *Proceedings of the International Workshop on Personalization Enhanced Access to Cultural Heritage (PATCH07)*, Corfu, Greece.

Isard, Amy, Brockmann, Carsten and Oberlander, Jon. 2006. Individuality and Alignment in Generated Dialogues. In *Proceedings of INLG-06*, pages 22–29.

Kaplan, R. M., Maxwell, J. T., King, T. H. and Crouch, R. S. 2004. Integrating Finite-state Technology with Deep LFG Grammars. In *Proceedings of Combining Shallow and Deep Processing for NLP, ESSLLI 2004*.

Kroeger, Paul. 1993. *Phrase Structure and Grammatical Relations in Tagalog*. Stanford: CSLI Publications.

Kruijff, Geert-Jan and Baldridge, Jason. 2004. Generalizing Dimensionality in Combinatory Categorial Grammar. In *Proceedings of COLING-04*.

Kruijff, Geert-Jan M., Zender, Hendrik, Jensfelt, Patric and Christensen, Henrik I. 2007. Situated Dialogue and Spatial Organization: What, Where... and Why? *International Journal of Advanced Robotic Systems* 4(2).

Moore, Johanna D., Foster, Mary Ellen, Lemon, Oliver and White, Michael. 2004. Generating tailored, comparative descriptions in spoken dialogue. In *Proceedings of FLAIRS 2004*, Miami Beach.

Nakatsu, Crystal and White, Michael. 2006. Learning to Say It Well: Reranking Realizations by Predicted Synthesis Quality. In *Proceedings of COLING-ACL 2006*.

Rickert, Markus, Foster, Mary Ellen, Giuliani, Manuel, By, Tomas, Panin, Giorgio and Knoll, Alois. 2007. Integrating language, vision and action for human robot dialog systems. In *Proceedings of HCI International 2007*, Beijing.

Rojas-Barahona, Lina M. 2007. Adapting Combinatory Categorial Grammars in a Framework for Health Care Dialogue Systems. In *Proceedings of the 11th Workshop on the Semantics and Pragmatics of Dialogue (DECALOG 2007)*, pages 187–188.

Steedman, Mark. 2000. *The Syntactic Process*. MIT Press/Bradford Books.

Steedman, Mark and Baldridge, Jason. To appear. Combinatory Categorial Grammar. In Robert Boersley and Kersti Börjars (eds.), *Nontransformational Syntax: A Guide to Current Models*, Blackwell.

Villavicencio, Aline. 2002. *The Acquisition of a Unification-Based Generalised Categorial Grammar*. Ph. D.thesis, University of Cambridge.

White, Michael. 2006a. CCG Chart Realization from Disjunctive Inputs. In *Proceedings of INLG-06*.

White, Michael. 2006b. Efficient Realization of Coordinate Structures in Combinatory Categorial Grammar. *Research on Language and Computation* 4(1), 39–75.

White, Michael and Baldridge, Jason. 2003. Adapting Chart Realization to CCG. In *Proceedings of ENLG*.

White, Michael, Rajkumar, Rajakrishnan and Martin, Scott. 2007. Towards Broad Coverage Surface Realization with CCG. In *Proceedings of the Workshop on Using Corpora for NLG: Language Generation and Machine Translation (UC-NLG+MT)*, Copenhagen.

Wolska, Magdalena and Kruijff-Korbayová, Ivana. 2004. Analysis of Mixed Natural and Symbolic Input in Mathematical Dialogs. In *Proceedings of ACL*, pages 25–32.

Zettlemoyer, Luke and Collins, Michael. 2007. Online Learning of Relaxed CCG Grammars for Parsing to Logical Form. In *Proceedings of EMNLP-CoNLL 2007*.