Specifying Grammars for OpenCCG: A Rough Guide

Cem Bozşahin   Geert-Jan M. Kruijff   Michael White

March 22, 2013

Contents

List of Tables

List of Figures

1 OpenCCG

OpenCCG is an open source natural language processing library written in Java, which provides parsing and realization services based on Mark Steedman's Combinatory Categorial Grammar (CCG) formalism [Ste00]. The library makes use of the multi-modal extensions to CCG devised by Jason Baldridge in his dissertation [Bal02] and in a joint EACL-03 paper with Geert-Jan Kruijff [BK03]. For a concise introduction to CCG with these extensions, see [SB03].

OpenCCG grew out of the Grok system developed by Gann Bierner and Jason Baldridge, and has been refined and extended by Michael White, with further contributions from Cem Bozşahin, Güneş Erkan, Geert-Jan Kruijff, David Reitter and Alexandros Triantafyllidis. Recent development efforts, managed by Michael White, have focused on making the realizer [WB03, Whi04a, Whi04b, Whi05] practical to use in dialogue systems, and improving (somewhat) the grammar development process.

You can download and install OpenCCG from its website, located at `http://openccg.sourceforge.net`. Once you've unpacked the archive, have a look at the `README` file for installation instructions.

2 About this (rough) guide

This guide is intended to provide a brief introduction to writing grammars for OpenCCG. The system is implemented in Java, but you do not need to know Java. OpenCCG provides its own formats for describing grammars, including the combinatory rules, the lexicon (i.e. the lexicalized grammar), feature structures, LF, morphology etc. Two formats are available; one is based on XML and one is a higher-level format that looks similar to C or Java. The syntax of the XML-based format is very simple, but at the same time it can be verbose and hard-to-read. The other format, the so-called "CCG format" (`.ccg`), was specifically designed to be written by hand, and has a richer and more concise syntax. It is a "front-end" format in that it is converted internally to XML before it is actually used by OpenCCG, using the `ccg2xml` tool. As a result, the two formats share many conceptual similarities. **NB:** Note that the XML format is more stable than the `.ccg` format, and in particular, the way in which unification constraints are specified in the `.ccg` format is apt to change.

This manual was originally created before the CCG format existed. As a result, it is primarily geared towards writing grammars directly in XML.

Over time, however, it will be updated to cover the use of the CCG format as well. For the time being, see `src/ccg2xml/README` for documentation of the `.ccg` format.

3 Using the XML-based format

In order to write OpenCCGgrammars directly in the XML-based format, you should be familiar with XML. Actually, all you need to know is that tags can be hierarchically and linearly organized, and that they must be "closed" (by `</tag>` or `/>`) with proper nesting, e.g.

```
<name>
  <surname> Bond </surname> <first> James </first>
  <aka> Jimbo </aka> <aka> Double-Oh (Seven) </aka>
</name>

<feat attr="case" val="acc"/>
```

All the OpenCCG-defined elements and attributes are listed in the XML schema validation files. For reference documentation, you can have a look at these files, which are located in the `$OPENCCG_HOME/grammars/` directory of your installation. For example, `categories.xsd` describes the tags that go into OpenCCG categories.

For more advanced use of OpenCCG, it helps to know XSLT.

3.1 XML-based grammar architecture in OpenCCG

A run-time grammar for OpenCCG typically consists of five primary files, with the following canonical names:

`grammar.xml` Specifies the name of the grammar, and lists the names of the other files. This file may also specify XSLT transformations to use in converting LFs to/from XML, and/or properties of a custom tokenizer (see `grammar.xsd` for details).

`lexicon.xml` Specifies *lexical families*. A lexical family specifies one or more related categories, with their associated feature structures and logical forms. Lexical families are loosely based on the notion of *tree families* in XTAG.

`morph.xml` Specifies the *words* of the grammar. Each word is related to a lexical family through the part-of-speech tag of the word. If a family is a closed class, we specify explicitly with a family what words are its members.

`rules.xml` Specifies which combinatory rules are available to the grammar. For the purpose of this document, we assume that application, type-raising, and composition (harmonic as well as crossed) are available. Unary type changing rules are also placed into `rules.xml`.

`types.xml` **(optional)** Specifies the syntactic and semantic type/sort hierarchies. Unlike HPSG, only atomic types are supported in OpenCCG. Multiple-inheritance is allowed [Erk03].

Standard practice is to store these files in a directory under the `grammars` directory of the OpenCCG distribution. Besides the above files, it is also a good idea to have a `testbed.xml` file. A testbed is a list of test expressions, where we specify for each expression the number of parses (≥ 0) the grammar should yield, and optionally the intended LF.

4 Words and categories

4.1 Lexical families

Traditionally, the lexicon for a categorial grammar specifies for each word its own category. In OpenCCG, categories are instead organized into lexical *families*, which are related to whole sets of words. (As mentioned earlier, the idea of families we employ here is loosely based on the notion of *tree families* in XTAG.) This makes it possible to avoid giving the same specification over and over again in a lexicon.

The simplest way in which words can be related to families is through their parts of speech: for a word we have to specify its part of speech, and for a family we have to specify the part of speech a word has to have for the family to be applicable. To control the applicability of a family, we can also declare it to be *closed*. A closed family is not applicable to *every* word that has the appropriate part of speech, but only to those words (stems) that are listed with the family as its members. Note that a closed family does not exactly correspond to the notion of a closed class word, as open class words (especially verbs) are often listed as members of closed families, in order to assign them appropriate subcategorization frames.

To illustrate, let's look at some examples from the `tiny` sample grammar. A family is defined within the following element:

```
<family name="Noun" pos="N">
  <entry name="Primary">
    :
  </entry>
</family>

<family name="ProNP" pos="Pro" closed="true">
  <entry name="Primary">
    :
  </entry>
  <member stem="pro1"/>
  <member stem="pro2"/>
  <member stem="pro3f"/>
  <member stem="pro3m"/>
  <member stem="pro3n"/>
</family>
```

These two families are for nouns and pronominal NPs, as their *name* attributes indicate; they have parts of speech `N` and `Pro`, respectively, given by the *pos* attribute. The pronominal NP family has `closed="true"`, indicating that it's a closed family. The members are `pro1` . . . `pro3n`, where `pro1` is an abstract stem for the first person pronouns *I*, *we*, *me*, *us*, and so on.

In each family, we define one or more entries, using an *entry* element. Each entry defines a category with accompanying feature structure and logical form. Each entry is given a name; we usually give the main entry `name="Primary"`. An example of a family with multiple (ok, two) entries appears below. The first entry is named `DTV`, for ditransitive verb, as it specifies a category with two NP complements; the second entry is named `NP-PPfor`, as it specifies a category with an NP complement followed by a PP complement headed by *for*. In both cases, the extra complement plays the role of Beneficiary in the semantics, motivating the grouping of these two entries into a single family.

```
<family name="DitransitiveBeneficiaryVerbs" pos="V" closed="true">
  <entry name="DTV">
    :
  </entry>
```

```
    <entry name="NP-PPfor">
      :
    </entry>
    <member stem="buy"/>
    <member stem="rent"/>
  </family>
```

4.2 Categories

Within an entry we define a category. A category can either be atomic or complex (i.e. a function). The example below illustrates how we specify an atomic category using the *atomcat* element, giving its label as a value of the attribute *type*.

```
<family name="Noun" pos="N">
  <entry name="Primary">
    <atomcat type="n">
      :
    </atomcat>
  </entry>
</family>
```

We can assign a feature structure to an atomic category using the *fs* element. The *fs* element has an *id* attribute so that we can explicitly reference the feature structure, when needed.

```
<atomcat type="n">
  <fs id="2"> .. </fs>
  :
</atomcat>
```

We can add individual features using *feat* elements. In their simplest form, a feature has an *attr* specifying the attribute and a *val* giving the value of the attribute.

```
<fs id="2">
  <feat attr="num" val="sg"/>
</fs>
```

Now, since we don't want all nouns to be singular, we can instead declare the value of the `num` feature to be a variable, as follows:

Rules	OpenCCG	MMCCG
application only	`*`	$\star$
associative	`^`	$\diamond$
permutative	`x`	$\times$
permutative right	`x>`	$\times\triangleright$
permutative left	`<x`	$\triangleleft\times$
associative permutative right	`>`	$\triangleright$
associative permutative left	`<`	$\triangleleft$
all rules	`.`	$\bullet$

Table 1: Slash modes

```
<atomcat type="n">
  <fs id="2">
    <feat attr="num"> <featvar name="NUM"/> </feat>
    :
  </fs>
  :
</atomcat>
```

Here the *featvar* element introduces a variable with `name="NUM"` as the value of the feature. Note that this feature specification serves as an implicit declaration that nouns have a `num` feature. As such, it interacts with the *inheritsFrom* mechanism for default unification, as will be explained below. With basic categories such as this one, it is a good idea to specify all relevant features.

An entry can also specify a *complex* category, i.e. a function. For that, we use the *complexcat* element. This element is essentially a list, enumerating the result category and its arguments in the order as given by a Steedman-style category. Argument categories may be atomic or complex (i.e., creating a higher-order function); the result category must be atomic (see [Bal02] for discussion).

For each argument we give the slash using a *slash* element that has attributes *mode* and *dir* to specify what kind of slash we are dealing with. The available slash modes are given in Table 1. (Note that in XML, the angle brackets `<` and `>` must be escaped as `&lt;` and `&gt;`, respectively.) See [Bal02][p. 100] and [BK03] for discusion of the slash modes in multimodal CCG. Slashes may also have variables over modes, and may be inert, as discussed in [Bal02][Ch. 8].

```
<complexcat>
  <atomcat type="s">
    <fs id="1"> .. </fs>
  </atomcat>
  <slash dir="\" mode="&lt;"/>
  <atomcat type="np">
    <fs id="2"> <feat attr="case" val="nom"/> .. </fs>
  </atomcat>
  <slash dir="/" mode="&gt;"/>
  <atomcat type="np">
    <fs id="3"> <feat attr="case" val="acc"/> .. </fs>
  </atomcat>
  :
</complexcat>
```

$$\mathsf{s}_{\langle 1\rangle}\backslash \mathsf{np}_{\langle 2\rangle nom}/\mathsf{np}_{\langle 3\rangle acc}$$

Figure 1: Transitive verb category

Figure 1 shows how the category for a transitive verb can be defined; at the bottom of the figure is a more human-friendly notation for the category. The result category is s. There are two argument categories, an np with accusative case to the right, and an np with nominative case to the left. In the human notation, the feature structure id's are shown subscripted in angle brackets, followed by the features themselves. Note that when the intended feature is evident from the feature value, the feature name is left off; also, when the slash mode is consistent with the slash direction (e.g. $\triangleright$ and /), the mode is not shown, as in [Bal02].

4.3 Words

Since pronouns retain case marking in English, the case requirements on the arguments of a transitive verb have the effect of determining which pronouns can appear in which positions. For example, the first person pronoun *I* is allowed in subject position, while *me* is allowed in object position, but not vice-versa.

This naturally leads us to how we specify properties of words in the `morph.xml` file. For each word, we have to give its wordform and part of

```
<entry pos="Pro" word="I" stem="pro1" macros="@1st @sg @nom .."/>
<entry pos="Pro" word="me" stem="pro1" macros="@1st @sg @acc .."/>
<entry pos="Pro" word="we" stem="pro1" macros="@1st @pl @nom .."/>
<entry pos="Pro" word="us" stem="pro1" macros="@1st @pl @acc .."/>
:
<macro name="@nom">
  <fs id="2" attr="case" val="nom"/>
</macro>
<macro name="@acc">
  <fs id="2" attr="case" val="acc"/>
</macro>
```

$$\begin{array}{rcl} \textit{I} & \vdash & \mathsf{np}_{\langle 2\rangle \mathit{1st,sg,nom}} \\ \textit{me} & \vdash & \mathsf{np}_{\langle 2\rangle \mathit{1st,sg,acc}} \\ \textit{we} & \vdash & \mathsf{np}_{\langle 2\rangle \mathit{1st,pl,nom}} \\ \textit{us} & \vdash & \mathsf{np}_{\langle 2\rangle \mathit{1st,pl,acc}} \end{array}$$

Figure 2: Case macros

speech, as follows:

```
<entry pos="Prep" word="for"/>
```

If the word's stem differs from its form, the stem must be listed too:

```
<entry pos="N" word="policemen" stem="policeman" .. />
```

To add further information, such as case, we use *macros*, as illustrated in Figure 2. In the figure, the entries for the first person pronouns are given, along with their syntactic macros, specified by the *macros* attribute. The case macros, named `@nom` and `@acc`, appear next in the figure, defined by the *macro* elements. These macros set the case feature on the category associated with the word (via its part of speech), by accessing the feature structure with id 2 and setting the value of the `case` feature to `nom` and `acc`, respectively. (The number macros `@sg` and `@pl` are analogous.) The effects of the macros are shown at the bottom of the figure, where the word forms for the first person pronouns are paired with their associated categories, which differ in their number and case values.

As another example, Figure 3 shows how the person macros are used (together with the number macros) in setting up person and number agreement constraints with various forms of the verb *buy*. Note that the tense

```
<entry pos="V" word="buy" macros="@pres @non-3rd @sg"/>
<entry pos="V" word="buys" stem="buy" macros="@pres @3rd @sg"/>
<entry pos="V" word="buy" macros="@pres @pl"/>
<entry pos="V" word="bought" stem="buy" macros="@past"/>
:
<macro name="@1st"> <fs id="2" attr="pers" val="1st"/> </macro>
<macro name="@2nd"> <fs id="2" attr="pers" val="2nd"/> </macro>
<macro name="@3rd"> <fs id="2" attr="pers" val="3rd"/> </macro>
<macro name="@non-3rd">
  <fs id="2" attr="pers" val="non-3rd"/>
</macro>
```

$$\begin{array}{rcl}
\textit{buy} & \vdash & \mathsf{s}_{\langle 1\rangle}\backslash\mathsf{np}_{\langle 2\rangle non\text{-}3rd,sg,nom}/\mathsf{np}_{\langle 3\rangle acc} \\
\textit{buys} & \vdash & \mathsf{s}_{\langle 1\rangle}\backslash\mathsf{np}_{\langle 2\rangle 3rd,sg,nom}/\mathsf{np}_{\langle 3\rangle acc} \\
\textit{buy} & \vdash & \mathsf{s}_{\langle 1\rangle}\backslash\mathsf{np}_{\langle 2\rangle pl,nom}/\mathsf{np}_{\langle 3\rangle acc} \\
\textit{bought} & \vdash & \mathsf{s}_{\langle 1\rangle}\backslash\mathsf{np}_{\langle 2\rangle nom}/\mathsf{np}_{\langle 3\rangle acc}
\end{array}$$

Figure 3: Person macros

macros `@pres` and `@past` do not contribute syntactic features; instead they contribute semantic features to the logical form (cf. Section 5). Additionally, note that the macro `@non-3rd` supplies a syntactic person value that is compatible with both `1st` and `2nd`, as specified in `types.xml` (cf. Section 6).

It is important to note that macro instantiation does not involve unification: macros set feature values regardless of any value that might already be present for the feature in the feature structure. Conceivably, it would be convenient on occasion (though computationally more expensive) to use unification, rather than overwriting, during macro instantiation, but there is no support for doing so at present.

4.4 Unification

Speaking of unification, let's examine the role it plays in enforcing subject-verb agreement. The category for the definite article is given in Figure 4. The definite article is compatible with both singular and plural nouns, but it must retain this number information for subject-verb agreement to work. Propagating number information is accomplished here by setting the feature structure id to be same on both the `np` result category and on the argument category `n` (i.e., we have `id="2"` in both cases). Figure 5 provides an illustration, showing how *the teacher* ends up as a singular `np`, while *the teachers*

```
<complexcat>
  <atomcat type="np">
    <fs id="2"> <feat attr="pers" val="3rd"/> .. </fs>
  </atomcat>
  <slash dir="/" mode="^"/>
  <atomcat type="n">
    <fs id="2"> .. </fs>
  </atomcat>
</complexcat>
```

$$\textit{the} \;\vdash\; \mathsf{np}_{\langle 2\rangle\, \mathit{3rd}} /_{\diamond} \mathsf{n}_{\langle 2\rangle}$$

Figure 4: The definite article

$$\begin{array}{ccc}
\textit{the} & \textit{teacher} & \textit{buys}\\
\cline{1-1}\cline{2-2}\cline{3-3}
\mathsf{np}_{\langle 2\rangle\, \mathit{3rd}} /_{\diamond} \mathsf{n}_{\langle 2\rangle} & \mathsf{n}_{sg} & \mathsf{s}\backslash\mathsf{np}_{\mathit{3rd},sg,nom}/\mathsf{np}_{acc}\\
\cline{1-2}
\multicolumn{2}{c}{\mathsf{np}_{\langle 2\rangle\, \mathit{3rd},sg}} & {>}\\
\cline{1-2}
\multicolumn{2}{c}{\mathsf{s}_{\langle 1\rangle}/\mathsf{s}_{\langle 1\rangle}\backslash\mathsf{np}_{\langle 2\rangle\, \mathit{3rd},sg}} & {>}\mathbf{T}\\
\hline
\multicolumn{3}{c}{\mathsf{s}_{\langle 1\rangle}/\mathsf{np}_{acc}} \\
\multicolumn{3}{r}{{>}\mathbf{B}}
\end{array}$$

$$\begin{array}{ccc}
\textit{the} & \textit{teachers} & \textit{*buys}\\
\cline{1-1}\cline{2-2}\cline{3-3}
\mathsf{np}_{\langle 2\rangle\, \mathit{3rd}} /_{\diamond} \mathsf{n}_{\langle 2\rangle} & \mathsf{n}_{pl} & \mathsf{s}\backslash\mathsf{np}_{\mathit{3rd},sg,nom}/\mathsf{np}_{acc}\\
\cline{1-2}
\multicolumn{2}{c}{\mathsf{np}_{\langle 2\rangle\, \mathit{3rd},pl}} & {>}\\
\cline{1-2}
\multicolumn{2}{c}{\mathsf{s}_{\langle 1\rangle}/\mathsf{s}_{\langle 1\rangle}\backslash\mathsf{np}_{\langle 2\rangle\, \mathit{3rd},pl}} & {>}\mathbf{T}\\
\hline
\multicolumn{3}{c}{***} \\
\multicolumn{3}{c}{\mathsf{s}_{\langle 1\rangle}/\mathsf{np}_{acc}} \\
\multicolumn{3}{r}{{>}\mathbf{B}}
\end{array}$$

Figure 5: Unification and subject-verb agreement

```
<complexcat>
  <atomcat type="pp">
    <fs inheritsFrom="3"> <feat attr="lex" val="[*DEFAULT*]"/> </fs>
  </atomcat>
  <slash mode="&lt;" dir="/"/>
  <atomcat type="np">
    <fs id="3"> <feat attr="case" val="acc"/> </fs>
  </atomcat>
</complexcat>
```

$$\textit{for} \ \vdash \ \mathsf{pp}_{for,acc,X} \,/_{\triangleleft}\, \mathsf{n}_{acc,X}$$

Figure 6: Default unification with case marking prepositions

ends up as a plural one.[1] Type raising the `np` and forward composing it with *buys* requires it to unify with the backwards `np` argument of the verb, and in particular, requires the number feature to be singular. Since this is only the case with *the teacher*, the last step in the derivation of *the teachers *buys* will be blocked by a unification failure.

Coindexing two feature structures ensures that all their features will take on the same values. There are times, however, when we want two feature structures to just mostly take on the same values, except for one or two particular features. To support such cases, OpenCCG includes a limited form of default unification, specified by the *inheritsFrom* attribute of a feature structure element.

The *inheritsFrom* mechanism is implemented by compiling out the default unification of two feature structures into individual feature equations at the time of lexical lookup. This works as follows. First, any features appearing on the target category, but not the result category, are copied over. Then, for every feature that has been observed in the grammar for the result category—except for any features that already appear there—a feature equation is added, i.e. the feature is set to the same variable on both the target and result categories.

As an example, Figure 6 shows the category for "case marking" prepositions, i.e. those prepositions which are assumed to play a purely syntactic role. The `pp` result category has a `lex` feature which is instantiated by the

[1]Note that feature structure id's are only shown when relevant to unification. Also, in OpenCCG derivations, the id's are actually mapped to "fresh" ones after lexical lookup, to avoid any accidental coindexations across different lexical items.

```
<complexcat>
  <atomcat type="s"> .. </atomcat>
  <setarg>
    <slash dir="/" mode="&gt;"/>
    <atomcat type="np">
      <fs> <feat attr="case" val="nom"/> .. </fs>
    </atomcat>
    <slash dir="/" mode="x"/>
    <atomcat type="np">
      <fs> <feat attr="case" val="gen"/> .. </fs>
    </atomcat>
  </setarg>
  :
</complexcat>
```

nanghuhuli ("catches") $\vdash$ $\mathsf{s}\{/\mathsf{np}_{nom}, /_{\times}\mathsf{np}_{gen}\}$

Figure 7: Tagalog transitive verb with set args

stem of the actual lexical item, as specified by the keyword `"[*DEFAULT*]"`. Its remaining features are "inherited from" the feature structure with id 3, i.e. the one for the argument `np`, as specified by `inheritsFrom="3"`. When the category for a case-marking preposition (such as *for* here) is instantiated, a feature equation is established between the `pp` and `np` categories for the `index` feature (whose purpose will be discussed in the next section); the value of the `case` feature is also copied over. Thus, in the figure, both the `pp` result category and the `np` argument category have an index variable `X` (and accusative case), while only the result PP has a `lex` feature, with `for` as its value.[2]

4.5 Set args

To conclude our discussion of specifying syntactic categories, we should mention the availability of set args and dollar variables in OpenCCG. Set args enable us to define categories that allow arguments to appear in any order,

[2]Since the case feature is arguably superfluous on the result PP, one could avoid it by just including an explicit feature equation for the `index` variable. Generally though, it's simpler and less error-prone to use the *inheritsFrom* mechanism than to manually include all the relevant feature equations.

```
<complexcat>
  <atomcat type="s"> .. </atomcat>
  <slash/>
  <dollar name="1"/>
  <slash dir="\" mode="*"/>
  <complexcat>
    <atomcat type="s"> .. </atomcat>
    <slash/>
    <dollar name="1"/>
  </complexcat>
  <slash dir="/" mode="*"/>
  <complexcat>
    <atomcat type="s"> .. </atomcat>
    <slash/>
    <dollar name="1"/>
  </complexcat>
</complexcat>
```

$$\textit{and} \ \vdash \ \mathsf{s}\$_1\backslash_{\star}\mathsf{s}\$_1/_{\star}\mathsf{s}\$_1$$

Figure 8: Category with dollar variables for sentential coordination

as illustrated in Figure 7 for the Tagalog verb *nanghuhuli ("catches")*. In the figure, both the nominative and genitive arguments must appear to the right of the verb, but their relative order is unconstrained. Note that the nominative argument is given the more powerful associative and permutative slash, while the genitive argument is given the permutative-only slash; see [Bal02][Ch. 7] for discussion, and [BS05] for further examples.

4.6 Dollar variables

Dollar variables range over a stack of arguments, and can be useful in defining categories for conjunctions, type-raised categories for quantifiers, and categories for unary rules (cf. [Bal02][Ch. 8]). Figure 8 shows the category for *and* that allows a range of clausal categories to be coordinated—e.g., transitive verbs, verb phrases, or subject-verb constituents, in right-node raising—as discussed in [Whi04a].

5 Logical forms

5.1 Hybrid logic dependency semantics

To associate meanings with categories, we need to take care of two things: the structure of the meaning (logical form) itself, and the relation between the category and that meaning. Usually the latter comes down to specifying how the meanings of arguments are to be fit into the logical form.

As logical forms we use *hybrid logical terms* that specify semantic dependency structures; for details, see [Kru01, BK02, WB03, Whi04a]. (If you're wondering where the λ's have gone, see Section 5.6.)

We give the logical form of a category using the *lf* element:

```
<complexcat>
  :
  <lf> .. </lf>
</complexcat>
```

The *lf* element must always appear at the end of a category specification (whether atomic or complex).

The simplest logical form is of the form $@_X\phi$, with ϕ a proposition. We interpret X as the *discourse referent* of the proposition. For the proposition itself, we can follow linguistic tradition and use a word's stem to represent its meaning, except that we'll use boldface rather than prime notation (i.e., we'll represent the meaning of *word* as **word** rather than *word'*). We achieve exactly this effect using the keyword `"[*DEFAULT*]"`, as shown below:

```
<lf>
  <satop nomvar="X:sem-obj">
    <prop name="[*DEFAULT*]"/>
  </satop>
</lf>
```

The *satop* element introduces a satisfaction operator @, along with a nominal variable, or *nomvar*, X. (In the grammar, we use logic variables rather than concrete instantiations for nominals; during parsing or realization, OpenCCG instantiates these variables dynamically.) Nominals can have types (or sorts) associated with them, as will be explained further in Section 6; here, X is allowed to be any subtype of semantic object. The *prop* element specifies the proposition.

```
[lexicon.xml]
  <family name="Noun" pos="N">
    <entry name="Primary">
      <atomcat type="n">
        <fs id="2">
          <feat attr="num"> <featvar name="NUM"/> </feat>
          <feat attr="index">
            <lf> <nomvar name="X"/> </lf>
          </feat>
        </fs>
        <lf>
          <satop nomvar="X:sem-obj">
            <prop name="[*DEFAULT*]"/>
          </satop>
        </lf>
      </atomcat>
    </entry>
  </family>

[morph.xml]
  <entry pos="N" word="flower" class="thing" macros="@sg @sg-X"/>
  :
  <macro name="@sg-X">
    <lf>
      <satop nomvar="X">
        <diamond mode="num"> <prop name="sg"/> </diamond>
      </satop>
    </lf>
  </macro>
```

$$\textit{flower} \ \vdash \ \mathsf{n}_{\langle 2\rangle sg, X:\mathrm{thing}} \ : \ @_{X:\mathrm{thing}}(\mathbf{flower} \wedge \langle \textsc{num}\rangle \mathrm{sg})$$

Figure 9: Noun with logical form

5.2 The syntax-semantics interface

To establish the interface between syntactic structure, as defined by the category, and the logical form, we co-index categories with nominals in the logical form, by adding an attribute `index` to the feature structure of each category and giving that attribute the corresponding nominal as value. To illustrate, Figure 9 (top) shows the complete category definition for nouns. Note how the feature structure includes a feature named `index`, whose value is a logical form just consisting of the nominal variable X—that is, a variable with the same name as the one introduced in the satisfaction operator of the LF further down.[3]

In the middle of Figure 9, the entry for the noun *flower* in `morph.xml` is shown. This entry includes `thing` as the semantic class (i.e. its semantic type), which is unified with the type of the nominal head X during lexical instantiation. The entry also includes the macros `@sg` and `@sg-X`, where the former adds the syntactic feature of singular number, while the latter adds the semantic feature of singular number on the nominal X (as shown below the entry). At the bottom of the figure is the complete category that results from lexical lookup and instantiation of *flower*.

Note that it is possible to fill in the semantic head's proposition with something other than the stem. To do so, we can specify a *pred* to use in place of the stem, when listing a stem as a member of a family:

```
<family closed="true" pos="V" name="TV">
  <entry name="primary">
    :
  </entry>
  :
  <member stem="nanghuhuli" pred="catch"/>
  :
</family>
```

With this specification, the proposition **catch** will appear in the logical form for the Tagalog verb *nanghuhuli*:

(1) *nanghuhuli* $\vdash$ $\mathsf{s}_E\{/\mathsf{np}_{nom,X}, /_{\times}\mathsf{np}_{gen,Y}\}$:
$@_E(\textbf{catch} \wedge \langle\textsc{Actor}\rangle X \wedge \langle\textsc{Patient}\rangle Y)$

[3]It suffices to put all the semantic type restrictions on the nominals in the LF; i.e., it's not necessary to put them on the `index` variables as well.

```
<complexcat>
  <atomcat type="s">
    <fs id="1">
      <feat attr="index"><lf><nomvar name="E"/></lf></feat>
    </fs>
  </atomcat>
  <slash dir="\" mode="&lt;"/>
  <atomcat type="np">
    <fs id="2"> ..
      <feat attr="index"><lf><nomvar name="X"/></lf></feat>
    </fs>
  </atomcat>
  <slash dir="/" mode="&gt;"/>
  <atomcat type="np">
    <fs id="3"> ..
      <feat attr="index"><lf><nomvar name="Y"/></lf></feat>
    </fs>
  </atomcat>
  <lf>
    <satop nomvar="E:action">
      <prop name="[*DEFAULT*]"/>
      <diamond mode="Actor">
        <nomvar name="X:animate-being"/>
      </diamond>
      <diamond mode="Patient">
        <nomvar name="Y:sem-obj"/>
      </diamond>
    </satop>
  </lf>
</complexcat>
```

$$\begin{array}{lll} \textit{bought} & \vdash & \mathsf{s}_{\langle 1\rangle E:\text{action}} \backslash \mathsf{np}_{\langle 2\rangle nom, X:\text{animate-being}} / \mathsf{np}_{\langle 3\rangle acc, Y:\text{sem-obj}} \ : \\ & & @_{E:\text{action}}(\mathbf{buy} \wedge \langle \text{TENSE} \rangle \text{past}) \wedge \\ & & @_{E:\text{action}}(\langle \text{ACTOR} \rangle X\!:\text{animate-being}) \wedge \\ & & @_{E:\text{action}}(\langle \text{PATIENT} \rangle Y\!:\text{sem-obj}) \end{array}$$

Figure 10: Transitive verb with logical form

```
<family name="Det" pos="Det" closed="true" indexRel="det">
  <entry name="Primary">
    <complexcat>
      <atomcat type="np">
        <fs id="2"> ..
          <feat attr="index"><lf><nomvar name="X"/></lf></feat>
        </fs>
      </atomcat>
      <slash dir="/" mode="^"/>
      <atomcat type="n">
        <fs id="2">
          <feat attr="index"><lf><nomvar name="X"/></lf></feat>
        </fs>
      </atomcat>
      <lf>
        <satop nomvar="X:sem-obj">
          <diamond mode="det"><prop name="[*DEFAULT*]"/></diamond>
        </satop>
      </lf>
    </complexcat>
  </entry>
  <member stem="a"/>
  <member stem="the"/>
  <member stem="some"/>
</family>
```

$$\textit{the} \ \vdash \ \mathsf{np}_{\langle 2\rangle \mathit{3rd}, X:\text{sem-obj}} /_{\diamond} \mathsf{n}_{\langle 2\rangle X:\text{sem-obj}} \ : \ @_{X:\text{sem-obj}}(\langle \textsc{det}\rangle \text{the})$$

Figure 11: Determiner with logical form

5.3 Dependency relations

To introduce dependency relations—like those we just saw for the Tagalog verb *nanghuhuli*—we use the *diamond* element. For example, to state that an English transitive verb has a logical form with an ⟨Actor⟩ and a ⟨Patient⟩, we can specify the category shown in Figure 10. At the bottom of the figure, this category appears instantiated for the verb *bought*. Note that for ease of display, the logical form has been partially flattened, with each dependency relation appearing on a separate line. (In OpenCCG, logical forms are automatically flattened prior to parsing or realization.)

5.4 Function words

Finally, Figure 11 illustrates how we can specify the meaning of function words, i.e. words that have no independent meaning. The example gives a specification for determiners, which add a semantic feature ⟨DET⟩ to the meaning of the nominal head they modify. The noun itself provides the meaning through co-indexation with X, the root nominal of the logical form for the determiner. At the bottom of the figure is the category instantiated for the definite article *the*.

To support the realization of function words, the semantic relation or feature introduced by the word must be declared using the *indexRel* attribute on the *family* element.[4] For example, in Figure 11 we have `indexRel="det"`, which indicates that the ⟨DET⟩ feature should be used to trigger the lookup of the appropriate determiner. (Normally, semantic relations or features are introduced as part of the meaning of content words.) When function words are semantically null—e.g., with case-marking prepositions in the `tiny` grammar—the keyword `*NoSem*` should be given as the value of the *indexRel* attribute.

5.5 Relation sorting

It is possible to specify the order in which to display relations appearing at the same level in the logical forms. By default, relations are sorted alphabetically, with a few exceptions, e.g. that ⟨RESTR⟩ should appear before ⟨BODY⟩. You can customize the order in which relations appear using the optional *relation-sorting* element in `lexicon.xml`. See the `lexicon.xsd` schema for details.

5.6 From CCG to OpenCCG: Taking care of lambdas

In CCG, logical forms are normally given using terms from the lambda calculus, e.g.

(2) *buy* $\vdash$ $(\mathsf{s}\backslash\mathsf{np}_{nom})/\mathsf{np}_{acc}$: $\lambda x_2 x_1.\mathrm{buy}' x_2 x_1$

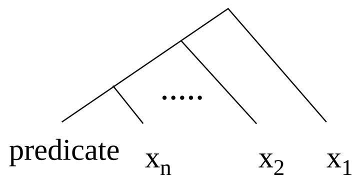
(3) *nanghuhuli* $\vdash$ $\mathsf{s}\{/\mathsf{np}_{nom}, /_{\times}\mathsf{np}_{gen}\}$: $\lambda\{x_1, x_2\}.\mathrm{catch}' x_2 x_1$

In (2), x_2 corresponds to the outermost argument, $/\mathsf{np}_{acc}$, and x_1 to the innermost one, $\backslash\mathsf{np}_{nom}$. Example (3) uses set-lambda notation with the fol-

[4] In principle, it would be possible for OpenCCG to figure out when a semantic relation or feature should be used for indexing purposes, but the possibility of adding further semantic content via macros makes it non-trivial to do so.

lowing convention (cf. [BS05]): the lambda operator binds a set of variables which are paired with the set of arguments in left-to-right order. Thus, in the second example above, x_1 corresponds to $/\mathsf{np}_{nom}$, and x_2 to $/_{\times}\mathsf{np}_{gen}$.

The interpretation of the CCG λ-terms above is as in (4), where the argument x_i c-commands x_{i+j} for $j = 1, 2, \cdots n - i$, at the level of predicate-argument structure (c-command is called LF-command in CCG for that reason). This is how (and where) CCG defines binding constraints.

(4)


As we have seen earlier in this section, OpenCCG uses hybrid logic dependency semantics (HLDS) terms, rather than λ-terms, in its logical forms. For example, the HLDS terms for (2) and (3) appeared in Figure 10 and in (1). These terms differ from their λ-counterparts in a couple of ways. First, in semantic construction, argument binding is accomplished through unification, rather than via function application. And second, predicates are typically connected to their arguments via semantic roles, such as ⟨Agent⟩ and ⟨Patient⟩—though nothing prevents relations such as ⟨Arg1⟩ and ⟨Arg2⟩ from being used instead. Using semantic roles can be more convenient for applications, and makes it possible to capture semantic similarities across argument structure alternations. The downside is that it makes it impossible to enforce binding constraints. In principle, relations encoding semantic roles (e.g. ⟨Agent⟩ and ⟨Patient⟩) could be combined with ones for argument structure (e.g. ⟨Arg1⟩ . . . ⟨ArgN⟩) in the same HLDS logical form, though this has not yet been tried.

6 Types

Types (aka sorts) allow for some abstraction, generalization and specialization in an OpenCCG grammar. Unlike HPSG, OpenCCG only employs atomic types. These types may be used as restrictions on syntactic or semantic feature variables, or given as values of syntactic or semantic features. Multiple-inheritance is allowed (see [Erk03] for further information).

Types are kept in the types file, usually named `types.xml`. This file is optional, which means that features can be untyped (actually, all features will be considered to be of type `top` in this case, which is the only predefined type).

```
<!-- person vals -->
<type name="pers-vals"/>
<type name="3rd" parents="pers-vals"/>
<type name="non-3rd" parents="pers-vals"/>
  <type name="1st" parents="non-3rd"/>
  <type name="2nd" parents="non-3rd"/>
```

Figure 12: Hierarchy of syntactic person values

```
<!-- ontological sorts -->
<type name="sem-obj"/>
  <type name="phys-obj" parents="sem-obj"/>
    <type name="animate-being" parents="phys-obj"/>
      <type name="person" parents="animate-being"/>
    <type name="thing" parents="phys-obj"/>
  <type name="situation" parents="sem-obj"/>
    <type name="change" parents="situation"/>
      <type name="action" parents="change"/>
    <type name="state" parents="situation"/>
```

Figure 13: Hierarchy of semantic types/sorts

In Section 4.3, Figure 3, we saw how the value of the person feature `non-3rd` could be used to define a category compatible with both `1st` and `2nd` person singular subjects. This definition relies on the following specification of person values in the `tiny` grammar's `types.xml` file, listed in Figure 12. As this example shows, types are defined using a *type* element and are required to have a *name*. They may also have a space-separated list of one or more *parent* types. Indenting may be used to show the primary type-subtype hierarchy:[5] here, `1st` and `2nd` are subtypes of `non-3rd`, while `non-3rd` and `3rd` are subtypes of `pers-vals`. If no parent types are listed—as with `pers-vals`—the type is implicitly a subtype of `top`.

The OpenCCG type system does not distinguish between syntactic and semantic types; it is up to the grammar designer to ensure their systematic use. For example, if there is to be a syntactic hierarchy and a semantic hierarchy of types, it is a good idea to define a 'top object' for each (or

[5]Since multiple parents are allowed, nesting of elements is not used to define type-subtype relationships.

at least for one of them), e.g. `sem-obj` as the root of the semantic type hierarchy, as illustrated in Figure 13. The types in this figure are the ones assumed by the transitive verb category given in Figure 10. With these types, OpenCCG can parse and realize *he bought a flower*, but not **a flower bought he*, since *flower* has type `thing`, and `thing` is not compatible with the type `animate-being`, as is required for the ⟨Actor⟩ of a **buy** action.

7 Rules

The rules file, typically named `rules.xml`, specifies the combinatory rules for a grammar. The rule specifications for the `tiny` grammar appear below:

```
<!-- Application -->
<application dir="forward"/>
<application dir="backward"/>

<!-- Harmonic Composition -->
<composition dir="forward" harmonic="true"/>
<composition dir="backward" harmonic="true"/>

<!-- Crossed Composition -->
<composition dir="forward" harmonic="false"/>
<composition dir="backward" harmonic="false"/>

<!-- Type-raising -->
<typeraising dir="forward" useDollar="false"/>
<typeraising dir="backward" useDollar="true"/>
<typeraising dir="backward" useDollar="true">
  <arg><atomcat type="pp"/></arg>
</typeraising>
```

In addition to the rules shown here, it is also possible to have substitution rules, as well as additional type raising rules. By default, the argument and result categories for a type raising rule are `np` and `s`, respectively. To create a type raising rule using different categories, you can use an *arg* and/or *result* element to specify the desired atomic category. For example, a backward type raising rule for prepositional phrases is included as the last rule above.

Theoretically speaking, CCG combinatory rules are universal; *any* lexicalized grammar has access to them if it uses in the lexical categories the modalities licensed by the rules (see [SB03] for further information). The rules file can also incorporate rules that are language specific. To illustrate, let's consider the case of pro drop in Turkish. Turkish is a pro-drop language, which means that subjects of finite clauses can be dropped because morphology of the finite verb already indicates the subject:

	Ben	uyu-du-m/*-n
(a)	I	sleep-PAST-1SG/*-2SG
	'I slept.'	

	Uyu-du-m
(b)	sleep-PAST-1SG
	'(I) slept.'

Pro drop can be modelled in different ways. For example, one can write a lexical rule to generate the derived lexical entries of finite verbs *in* the lexicon, so that every finite verb has two lexical entries, one derived from the other—e.g. (1) below, which asserts $\mathsf{s}_{\mathit{fin}}\backslash\mathsf{np}_{\mathit{acc}}$ and $\mathsf{s}_{\mathit{fin}}$ entries in the lexicon from $\mathsf{s}_{\mathit{fin}}\backslash\mathsf{np}_{\mathit{nom}}\backslash\mathsf{np}_{\mathit{acc}}$ and $\mathsf{s}_{\mathit{fin}}\backslash\mathsf{np}_{\mathit{nom}}$, etc.

$$\mathsf{s}_{\mathit{fin}}\backslash\mathsf{np}_{\mathit{nom}}\$_1 \Rightarrow \mathsf{s}_{\mathit{fin}}\$_1 \tag{1}$$

This strategy has theoretical and practical implications. Theoretically, it assumes that *all* morphology is confined to the lexicon, including inflectional morphology, which is usually regarded as part of syntax (see e.g. [Boz02] for its implications for the transparency of syntax-semantics correspondence). OpenCCG does not assume that there are only words in the lexicon; anything that can bear a category (words, affixes, clitics) can be a lexical item.[6]

On the practical side, it assumes that all inflected forms of the verb are listed in the lexicon. For morphologically rich languages such as Turkish, this amounts to around 2^8 entries per verb because Turkish has 8 inflections in the verb paradigm, all of which are optional.

An alternative is to add a unary rule for pro drop to `rules.xml`. This rule will apply "on the fly", that is, it can apply to lexical or combinatorially-derived inflected verb forms. The rule, which implements (1), may be specified as shown in Figure 14 (we omit the $ variable for simplicity). Unary

[6]Currently, OpenCCG does not have any mechanism to enforce the *Lexical Integrity Principle* of [BM95], which basically states that words are islands as far as syntax is concerned, e.g. it is not possible to extract out of a word. [Boz02] proposes that different attachment characteristics of words and bound morphemes can be factored into CCG's lexical entries and combinatory rules (the latter simply projects them onto surface grammar), in effect rendering LIP as a phonological principle, but this is an open problem for now.

```
  <typechanging name="pd">
    <arg>
      <complexcat>
        <atomcat type="s">
          <fs id="1">
            <feat attr="v-form" val="finite"/>
          </fs>
        </atomcat>
        <slash dir="\"/>
        <atomcat type="n">
          <fs> <feat attr="case" val="nom"/> </fs>
        </atomcat>
      </complexcat>
    </arg>
    <result>
      <atomcat type="s"> <fs id="1"/> </atomcat>
    </result>
  </typechanging>
</rules>
```

Figure 14: Unary rule for pro drop

rules are defined using a *typechanging* element, since such rules must change the type of the argument category—otherwise, nothing would prevent the rule from applying again and again to its own output.

8 Trying it out

Once you've configured and built OpenCCG per the `README` file, you're ready to try out the grammar testing tools. You can experiment with the grammars described in `SAMPLE_GRAMMARS` or make one of your own. In the latter case, it will be easier if you create your grammar in its own subdirectory of the `grammars` directory.

There are (at present) three command line tools for trying grammars out: `tccg`, `ccg-test` and `ccg-realize`.

8.1 `tccg`

The `tccg` tool (for "text CCG") is for interactively testing a grammar. Its (primary) usage is

```
  tccg (<grammarfile>)
```

The default grammar file name is `grammar.xml`. You can try it out by going to the `grammars/tiny` directory and running `tccg`, like so:[7]

```
D:\Mike\dev\openccg\grammars\tiny>tccg
Loading grammar from URL: file:/D:/Mike/dev/openccg/grammars
/tiny/grammar.xml
Grammar 'tiny' loaded.

Enter strings to parse.
Type ':r' to realize selected reading of previous parse.
Type ':h' for help on display options and ':q' to quit.
You can use the tab key for command completion,
Ctrl-P (prev) and Ctrl-N (next) to access the command history,
and emacs-style control keys to edit the line.

tccg>
```

Typing in `:h` shows all the available commands. For example, `:derivs` turns on the display of derivations when you parse an expression:

[7]Examples like this one may have an occasional extra line break to improve readability.

```
tccg> :derivs
tccg> the teacher buys
3 parses found.

Parse 1: s/np
------------------------------
(lex)  the :- np/^n
(lex)  teacher :- n
(>)    the teacher :- np
(>T)   the teacher :- s/@i(s\@inp)
(lex)  buys :- s\np/np
(>B)   the teacher buys :- s/np

tccg>
```

Here we see a (simplified) vertical display of the derivation seen earlier in Figure 5. (If you have LaTeX installed, it's also possible to see derivations like those in Figure 5 using the `:vison` command, but note that its current behavior is a bit flaky.) Only the first parse is shown; the other two parses, for the ditransitive and np pp_{for} categories of the verb, can be seen by turning on all derivations with the `:all` command. To see the features on the categories, you can use the `:feats` command, optionally with a subset of features to show.

Logical forms can be shown with the `:sem` command:

```
tccg> :noderivs
tccg> :sem
tccg> she bought the policeman a flower
1 parse found.

Parse: s :
  @b1:action(buy ^
             <tense>past ^
             <Actor>(p1:animate-being ^ pro3f ^
                     <num>sg) ^
             <Beneficiary>(p2:person ^ policeman ^
                           <det>the ^
                           <num>sg) ^
             <Patient>(f1:thing ^ flower ^
                       <det>a ^
                       <num>sg))
tccg>
```

To see the realizations for this logical form (i.e., the one from the previous parse), use the `:r` command:

```
tccg> :nosem
tccg> :r
[1.000] she bought the policeman a flower :- s
[0.167] she bought a flower for the policeman :- s
tccg>
```

Realizations are ordered by their n-gram similarity to the previously entered expression. You can have a look in the `morph.xml` file for more words to form expressions with.

Note that the settings of the various options available in `tccg`; use `:reset` to undo all these settings and return to the default ones.

8.2 ccg-test

The `ccg-test` tool is for regression testing, and also provides options for timing the realizer. Its (primary) usage is

```
  ccg-test (-noparsing|-norealization)
           (-g <grammarfile>)
           (<regressionfile>)
```

By default, `ccg-test` will use the grammar in the current directory and the default regression file, `testbed.xml`.

Note that you can set realizer options, such as its time limit, in `tccg`, e.g. by issuing the command `:tl 1000` (for time limit 1000 ms.), and this value will persist and be used by `ccg-test`.

8.3 ccg-realize

The `ccg-realize` tool provides a sample interface to the realizer (see the underlying `opennlp/ccg/Realize.java` file), and can be an aid in debugging realization. It loads a grammar, runs the realizer on an input XML file, and logs its processing to an output text file (or to system out). Its usage is

```
  ccg-realize (-g <grammarfile>) <inputfile> (<outputfile>)
```

You can create input files for `ccg-realize` using the `:2xml` option in `tccg`.

9 Building grammars

OpenCCG comes with various utilities to help you build the files used by the runtime system—and to validate their contents—rather than writing them entirely by hand. The utilities take advantage of the `ccg-build` front end to the Apache Ant (http://ant.apache.org) build tool. In principle, `ccg-build` allows you to organize your files in any way you like to produce the runtime grammar files.

9.1 Validating the grammar files

You can use `ccg-build` to validate the grammar files against their XML schemas. To do so, you need to have a `build.xml` file in your grammar directory, which contains build tasks for the Apache Ant tool to carry out. The `tiny` grammar directory contains a build file which just validates the runtime files, as shown below:

```
D:\Mike\dev\openccg\grammars\tiny>ccg-build
Buildfile: build.xml

init:
     [echo] ----------- OpenCCG ------------

grammar:
     [echo] Validating grammar.xml, lexicon.xml, morph.xml,
            rules.xml and types.xml

BUILD SUCCESSFUL
Total time: 4 seconds
```

If there are any errors, validation with `ccg-build` gives relatively informative error messages. Loading a grammar into `tccg` will perform some further checks, but note that loading a grammar with errors usually means `tccg` croaks—outputting only (less informative) stack traces—so it's good practice to validate any changes you make to your grammars prior to running `tccg`.

9.2 Using a `dict.xml` file

Rather than creating a `morph.xml` file directly, you can employ a `dict.xml` file, which groups word forms by their stems and parts of speech, and lists the closed families for a given stem. A `dict.xml` files usually works together with a file called `lexicon-base.xml`, which does not contain *member* entries for families. From the `dict.xml` file—which also contains macro definitions—the `morph.xml` file can be generated automatically, with proper hooks to a derived `lexicon.xml` file, using `ccg-build`. (See the `dict.xsd` schema in the `grammars` directory for a complete description.)

In short, the simplest way to use a `dict.xml` file with `ccg-build` is to prepare a family of categories in the file named `lexicon-base.xml` without *member* entries, and to group stems and their word forms in a file called `dict.xml`, along with macro definitions.

A sample entry from the `cem-english` `dict.xml` file appears below:

```
  <entry stem="eat" pos="V">
    <member-of family="IV"/>
    <member-of family="TV"/>
    <word form="eat" macros="@nonfin"/>
    <word form="eats" macros="@pres @+3rd-agr @sg-agr"/>
    <word form="ate" macros="@past"/>
    <word form="eaten" macros="@past-part"/>
  </entry>
```

Note that the stem *eat* is declared intransitive and transitive without duplication in the morph or lexicon files. If you run `ccg-build` as follows,

```
cem-english> ccg-build grammar
Buildfile: build.xml

init:
     [echo] ----------- OpenCCG ------------

grammar:
     [echo] Adding family members from dict.xml to
            lexicon-base.xml, yielding lexicon.xml

     [echo] Extracting morph items from dict.xml to morph.xml

     [echo] Validating grammar.xml, lexicon.xml, morph.xml,
            rules.xml and types.xml

BUILD SUCCESSFUL
Total time: 5 seconds
```

there will be two lexical assignments for every word form of *eat*, one for its intransitive use, and one for transitive.

9.3 Reducing redundancy with XSLT

XSLT is a language for transforming XML documents. Two XSLT transformations, `add-family-members.xsl` and `extract-morph.xsl`, are used by `ccg-build` to handle `dict.xml` files. You can also use XSLT transformations to reduce redundancy in the lexico-grammar specifications.

For example, the `worldcup` grammar illustrates a couple of ways of using XSLT to improve the specification of lexical families. With this grammar, the `lexicon-base.xml` file is generated from an XSLT transformation, called `lexicon-base.xsl`. This file begins with the definition of variables for the various atomic categories used in later complex category and family definitions. The variable names follow the format

```
<cat>(.<id>)?(.from-<id>)?(.<index>?)(.<feat-descriptions>)*
```

i.e., the category label, followed optionally by the `id`, the `inheritsFrom` id, the `index` variable, and any further feature descriptions. This convention allows one to see what atomic categories are already in use, and to determine the contents of an atomic category at a glance. For example, `np.3.Y.acc` is the name of the category with label `np`, id `3`, index `Y`, and the case value `acc`; `np.2.X.default` is similar, but has default variables for all features other than the `index`.

Once variables have been declared, they can be referenced further on using `xsl:copy-of` statements. For example, the variable `np.2.X.default` is referenced seven times in `lexicon-base.xsl`. In this way, if a change to `np.2.X.default` is desired, it can be made in one place in the file, rather than seven.

Another XSLT mechanism employed in `lexicon-base.xsl` is a named templated called `extend` which serves to append one element as the last child of another. This mechanism is used to associate logical forms with syntactic categories, as well as to create new categories from existing ones. For example, in

```
<xsl:variable name="wh-det.subj">
  <xsl:call-template name="extend">
    <xsl:with-param name="elt"
                    select="xalan:nodeset($wh-np.subj)/*"/>
    <xsl:with-param name="ext"
```

```
                        select="$fslash-n.2.X"/>
    </xsl:call-template>
  </xsl:variable>
```

the category for a subject type-raised *wh*-determiner, like *which*, is created by extending the category of a subject type-raised *wh*–noun phrase (e.g. *who*) with an extra nominal argument, `fslash-n.2.X` (declared earlier as a forward slash plus a category with label `n`, id `2` and index `X`).[8] In this way, any changes made to the category for *wh*-determiners will carry over to the category for *wh*–noun phrases.

As XSLT is a powerful and extensible XML transformation language, there are many further possibilities for using it in grammar development—limited only by your imagination (and hacking ability).

References

[Bal02] Jason Baldridge. *Lexically Specified Derivational Control in Combinatory Categorial Grammar.* PhD thesis, School of Informatics, University of Edinburgh, 2002. Available from http://homepages.inf.ed.ac.uk/jbaldrid/dissertation/.

[BK02] Jason Baldridge and Geert-Jan M. Kruijff. Coupling CCG and hybrid logic dependency semantics. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics (ACL 2002)*, Philadelphia, Pennsylvania, 2002.

[BK03] Jason Baldridge and Geert-Jan M. Kruijff. Multi-modal combinatory categorial grammar. In *Proceedings of the 10th Conference of the European Chapter of the Association for Computational Linguistics (EACL 2003)*, Budapest, Hungary, 2003.

[BM95] Joan Bresnan and Sam A. Mchombo. The Lexical Integrity Principle: Evidence from Bantu. *Natural Language and Linguistic Theory*, 13:181–254, 1995.

[Boz02] Cem Bozsahin. The combinatory morphemic lexicon. *Computational Linguistics*, 28(2):145–176, 2002.

[BS05] Cem Bozsahin and Mark Steedman. Lexicalized asymmetry and syntactic projection. Manuscript, University of Edinburgh, 2005.

[8]The `xalan:nodeset` function makes it possible to use a variable as a parameter to a named template; its use won't be necessary in future versions of XSLT.

[Erk03] Güneş Erkan. A type system for CCG. Master's thesis, Middle East Technical University, Ankara, 2003. Available from http://www.LcsL.metu.edu.tr/ftp/theses/erkan-ms-03.pdf.gz.

[Kru01] Geert-Jan M. Kruijff. *A Categorial-Modal Logical Architecture of Informativity: Dependency Grammar Logic & Information Structure.* PhD thesis, Faculty of Mathematics and Physics, Charles University, Prague, Czech Republic, 2001.

[SB03] Mark Steedman and Jason Baldridge. Combinatory Categorial Grammar. Tutorial paper, available from http://homepages.inf.ed.ac.uk/jbaldrid/ccg.pdf, 2003.

[Ste00] Mark Steedman. *The Syntactic Process.* The MIT Press, Cambridge Mass., 2000.

[WB03] Michael White and Jason Baldridge. Adapting chart realization to CCG. In *Proceedings of the Ninth European Workshop on Natural Language Generation*, Budapest, Hungary, 2003.

[Whi04a] Michael White. Efficient Realization of Coordinate Structures in Combinatory Categorial Grammar. *Research on Language and Computation*, 2004. To appear.

[Whi04b] Michael White. Reining in CCG Chart Realization. In *Proceedings of the Third International Conference on Natural Language Generation, INLG-04*, 2004.

[Whi05] Michael White. Designing an extensible API for integrating language modeling and realization. In *Proc. ACL-05 Workshop on Software*, 2005.