# OpenCCG Realizer Manual

Michael White

March 22, 2013

## Contents



## List of Figures

# 1 About this manual

This manual is a programmer's guide to using the OpenCCG surface realizer in Java applications. You can download and install OpenCCG from its website, `http://openccg.sourceforge.net`. Once you've unpacked the archive, have a look at the `README` file for installation instructions. For a brief introduction to writing grammars for OpenCCG, see the "rough guide" in `docs/grammars-rough-guide.pdf`.

# 2 About the OpenCCG realizer

The OpenCCG realizer [WB03, Whi04a, Whi04b, Whi05] is an open source surface realizer for Steedman's [Ste00a, Ste00b] Combinatory Categorial Grammar (CCG), including the multi-modal extensions to CCG devised by Baldridge and Kruijff [Bal02, BK03a].

Like other chart realizers [Kay96, She97, CCFP99, Moo02], the OpenCCG realizer takes as input a logical form specifying the propositional meaning of a sentence, and returns one or more surface strings that express this meaning according to the lexicon and grammar. A distinguishing feature of OpenCCG is that it implements a hybrid symbolic-statistical chart realization algorithm that combines (1) a theoretically grounded approach to syntax and semantic composition, with (2) the use of integrated language models for making choices among the options left open by the grammar, thereby reducing the need for hand-crafted rules.

To allow language models to be combined in flexible ways—as well as to enable research on how to best combine language modeling and realization—OpenCCG's design includes an extensible API (application programming interface) that allows user-defined functions to be used for scoring partial realizations and for pruning low-scoring ones during the search. The design also includes classes for supporting a range of language models and typical ways of combining them.

The UML class diagram in Figure 1 shows the high-level architecture of the OpenCCG realizer. A realizer instance is constructed with a reference to a CCG grammar (which supports both parsing and realization). The grammar's lexicon has methods for looking up lexical items via their surface forms (for parsing), or via the principal predicates or relations in their semantics (for realization). A grammar also has a set of hierarchically organized atomic types, which can serve as the values of features in the syntactic categories, or as ontological sorts for the discourse referents in the logical

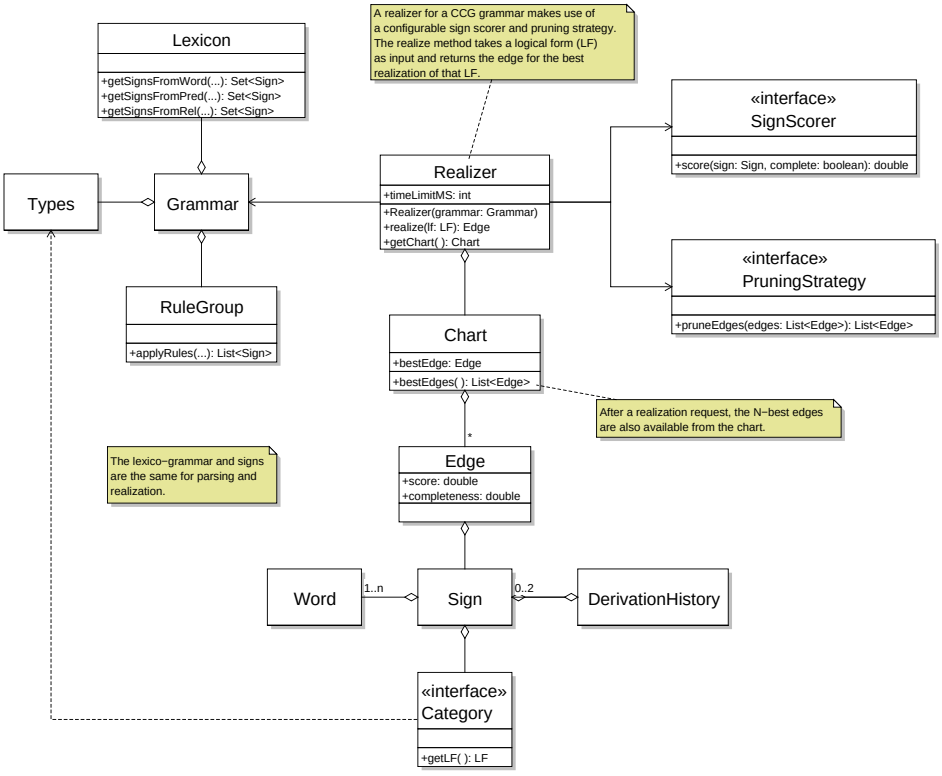



Figure 1: High-level architecture of the OpenCCG realizer

forms (LFs).

Lexical lookup yields lexical signs. A sign pairs a list of words with a category, which itself pairs a syntactic category with a logical form. Lexical signs are combined into derived signs using the rules in the grammar's rule group. Derived signs maintain a derivation history, and their word lists share structure with the word lists of their input signs.

For generality, the realizer makes use of a configurable sign scorer and pruning strategy. A sign scorer implements a function that returns a number between 0 and 1 for an input sign. For example, a standard trigram language model can be used to implement a sign scorer, by returning the probability of a sign's words as its score. A pruning strategy implements a method for determining which edges to prune during the realizer's search. The input to the method is a ranked list of edges for signs that have equivalent categories (but different words); grouping edges in this way ensures that pruning cannot "break" the realizer, i.e. prevent it from finding some grammatical derivation when one exists. By default, an N-best pruning strategy is employed, which keeps the N highest scoring input edges, pruning the rest (where N is determined by the current preference settings).

## 3 Using the realizer

Sample Java code for using the realizer appears in Figure 2. The input is an XML document that contains an `lf` element either as the root or as a child of the root. To create a sample XML document with an acceptable format, you can use the `tccg` tool's `:2xml <filename>` command. Note that the input XML document can be created in any way that is allowed by the JDOM API. For example, if the logical form is created by a Java XSLT-based sentence planner in the same process, the XSLT output can be captured in a JDOM document, and then simply passed by reference to the realizer.

The output of the realizer is typically an XML document, as shown in the figure. In such documents, each word in the output sequence appears in its own element; additionally, any pitch accents and boundary tones appear in separate elements, and any expanded multi-words are indicated. Output documents of this kind can be easily processed into other formats using XSLT. If a simple string output suffices, the `Sign.getOrthography()` method can be used instead.

The realization algorithm is implemented by the `realize(LF)` method. As in the chart realizers cited earlier, the algorithm makes use of a chart and

```
// load grammar, instantiate realizer
URL grammarURL = ...;
Grammar grammar = new Grammar(grammarURL);
Realizer realizer = new Realizer(grammar);

// configure realizer with trigram backoff model
// and 10-best pruning strategy
realizer.signScorer = new StandardNgramModel(3, "lm.3bo");
realizer.pruningStrategy = new NBestPruningStrategy(10);

// ... then, for each request:

// get LF from input XML
Document inputDoc = ...;
LF lf = realizer.getLfFromDoc(inputDoc);

// realize LF and get output words in XML
Edge bestEdge = realizer.realize(lf);
Document outputDoc = bestEdge.sign.getWordsInXml();

// return output
... outputDoc ...;
```

Figure 2: Example realizer usage

an agenda to perform a bottom-up dynamic programming search for signs whose LFs completely cover the elementary predications in the input logical form. The algorithm's details and a worked example appear in [Whi04a, Whi04b]. To see a full realization trace, you can use ccg-realize to realize an LF stored in an XML file (e.g. one created using tccg). As shown in Figure 2, the `realize(LF)` method returns the edge for the best realization of the input LF, as determined by the sign scorer. After a realization request, the N-best complete edges—or more generally, all the edges for complete realizations that survived pruning—are also available from the chart. To access these edges, you can invoke `realizer.getChart().bestEdges()`.

The search for complete realizations proceeds in one of two modes, anytime and two-stage (packing/unpacking). In the anytime mode, a best-first search is performed with a configurable time limit (which may be a limit on how long to look for a better realization, after the first complete one is found). With this mode, the scores assigned by the sign scorer determine the order of the edges on the agenda, and thus have an impact on realization speed. In the two-stage mode, a packed forest of all possible realizations is created in the first stage; then in the second stage, the packed representation is unpacked in bottom-up fashion, with scores assigned to the edge for each sign as it is unpacked, much as in [Lan00]. In both modes, the pruning strategy is invoked to determine whether to keep or prune newly constructed edges. For single-best output, the anytime mode can provide signficant time savings by cutting off the search early; see [Whi04b] for discussion. For N-best output—especially when a complete search (up to the edges that survive the pruning strategy) is desirable—the two-stage mode can be more efficient.

# 4 Scoring signs

The classes for implementing sign scorers appear in Figure 3. In the diagram, classes for n-gram scoring appear towards the bottom, while classes for combining scorers appear on the left, and the class for avoiding repetition appears on the right.

## 4.1 Standard n-gram models

The `StandardNgramModel` class can load standard n-gram backoff models for scoring, as shown earlier in Figure 2. Such models can be constructed with the SRILM toolkit [Sto02], as described in Section 4.7; in principle, other toolkits could be used instead, as long as their output could be converted

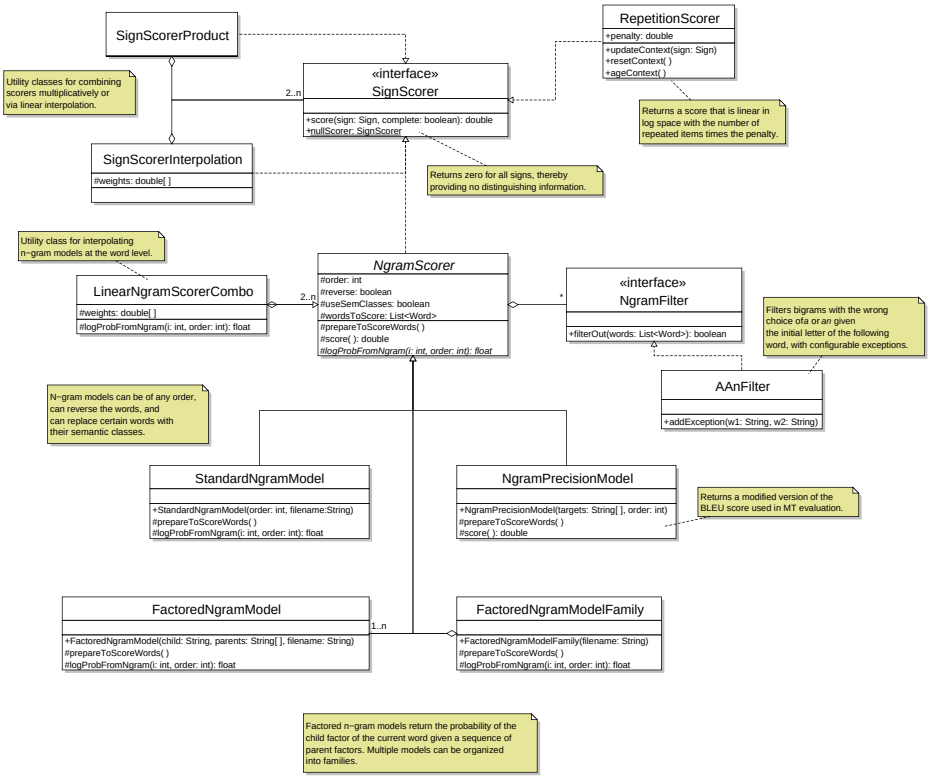



Figure 3: Classes for scoring signs

into the same file formats. Since the SRILM toolkit has more restrictive licensing conditions than those of OpenCCG's LGPL license, OpenCCG includes its own classes for scoring with n-gram models, in order to avoid any necessary runtime dependencies on the SRILM toolkit.

The n-gram tables are efficiently stored in a trie data structure (as in the SRILM toolkit), thereby avoiding any arbitrary limit on the n-gram order. To save memory and speed up equality tests, each string is interned (replaced with a canonical instance) at load time, which accomplishes the same purpose as replacing the strings with integers, but without the need to maintain a separate mapping from integers back to strings. For better generalization, certain words may be dynamically replaced with the names of their semantic classes when looking up n-gram probabilities. Words are assigned to semantic classes in the lexicon, and the semantic classes to use in this way may be configured at the grammar level. Note that [OR02] and [Rat02] make similar use of semantic classes in n-gram scoring, by deferring the instantiation of classes (such as *departure city*) until the end of the generation process; our approach accomplishes the same goal in a slightly more flexible way, in that it also allows the specific word to be examined by other scoring models, if desired.

As discussed in [Whi04b], with dialogue systems like COMIC n-gram models can do an excellent job of placing underconstrained adjectival and adverbial modifiers—as well as boundary tones—without resorting to the more complex methods investigated for adjective ordering in [SH99, Mal00]. For instance, in examples like those in (1), they correctly select the preferred positions for *here* and *also* (as well as for the boundary tones), with respect to the verbal head and sister dependents:

(1) a. Here$_{L+H*}$ LH% we have a design in the classic$_{H*}$ style LL% .
b. This$_{L+H*}$ design LH% here$_{L+H*}$ LH% is also$_{H*}$ classic LL% .

We have also found that it can be useful to use reverse (or "right-to-left") models, as they can help to place adverbs like *though*, as in (2):

(2) The tiles are also$_{H*}$ from the Jazz$_{H*}$ series though LL% .

In principle, the forward and reverse probabilities should be the same—as they are both derived via the chain rule from the same joint probability of the words in the sequence—but we have found that with sparse data the estimates can differ substantially. In particular, since *though* typically appears at the end of a variety of clauses, its right context is much more predictable than its left context, and thus reverse models yield more accurate estimates of its likelihood of appearing clause-finally.

### 4.2 N-gram scorers

The `StandardNgramModel` class is implemented as a subclass of the base class `NgramScorer`. All `NgramScorer` instances may have any number of `NgramFilter` instances, whose `filterOut` methods are invoked prior to n-gram scoring; if any of these methods return true, a score of zero is immediately returned. The `AAnFilter` provides one concrete implementation of the `NgramFilter` interface, and returns true if it finds a bigram consisting of *a* followed by a vowel-inital word, or *an* followed by a consonant-initial word, subject to a configurable set of exceptions that can be culled from bigram counts. We have found that such n-gram filters can be more efficient, and more reliable, than relying on n-gram scores alone; in particular, with *a/an*, since the unigram probability for *a* tends to be much higher than that of *an*, with unseen words beginning with a vowel, there may not be a clear preference for the bigram beginning with *an*.

The base class `NgramScorer` implements the bulk of the `score` method, using an abstract `logProbFromNgram` method for subclass-specific calculation of the log probabilities (with backoff) for individual n-grams. The `score` method also invokes the `prepareToScoreWords` method, in order to allow for subclass-specific pre-processing of the words in the given sign. With `StandardNgramModel`, this method is used to extract the word forms or semantic classes into a list of strings to score. It also appends any pitch accents to the word forms or semantic classes, effectively treating them as integral parts of the words.

Since the realizer builds up partial realizations bottom-up rather than left-to-right, it only adds start of sentence (and end of sentence) tags with complete realizations. As a consequence, the words with less than a full $n-1$ words of history are scored with appropriate sub-models. For example, the first word of a phrase is scored with a unigram sub-model, without imposing backoff penalties.

Another consequence of bottom-up realization is that both the left- and right-contexts may change when forming new signs from a given input sign. Consequently, it is often not possible (even in principle) to use the score of an input sign directly in computing the score of a new result sign. If one could make assumptions about how the score of an input sign has been computed—e.g., by a bigram model—one could determine the score of the result sign from the scores of the input signs together with an adjustment for the word(s) whose context has changed. However, our general approach to sign scoring precludes making such assumptions. Nevertheless, it is still possible to improve the efficiency of n-gram scoring by caching the log prob-

```
// configure realizer with 4-gram forward and reverse backoff
// models, interpolated with equal weight
NgramScorer forwardModel = new StandardNgramModel(4, "lm.4bo");
NgramScorer reverseModel = new StandardNgramModel(4, "lm-r.4bo");
reverseModel.setReverse(true);
realizer.signScorer = new SignScorerInterpolation(
    new SignScorer[] { forwardModel, reverseModel }
);
```

Figure 4: Example interpolated n-gram model

ability of a sign's words, and then looking up that log probability when the sign is used as the first input sign in creating a new combined sign—thus retaining the same left context—and only recomputing the log probabilities for the words of any input signs past the first one. (With reverse models, the sign must be the last sign in the combination.) In principle, the derivation history could be consulted further to narrow down the words whose n-gram probabilities must be recomputed to the minimum possible, though `NgramScorer` only implements a single-step lookup at present.[1] Finally, note that a Java `WeakHashMap` is used to implement the cache, in order to avoid an undesirable buildup of entries across realization requests.

### 4.3 Interpolation

Scoring models may be linearly interpolated in two ways. Sign scorers of any variety may be combined using the `SignScorerInterpolation` class. For example, Figure 4 shows how forward and reverse n-gram models may be interpolated.

With n-gram models of the same direction, it is also possible to linearly interpolate models at the word level, using the `LinearNgramScorerCombo` class. Word-level interpolation makes it easier to use cache models created with maximum likelihood estimation, as word-level interpolation with a base model avoids problems with zero probabilities in the cache model. As discussed in [BIOW05], cache models can be used to promote alignment with a conversational partner, by constructing a cache model from the bigrams in the partner's previous turn, and interpolating it with a base model.[2]

[1] Informal experiments indicate that caching log probabilities in this way can yield an overall reduction in best-first realization times of 2-3% on average.

[2] At present, such cache models must be constructed with a call to the SRILM toolkit; it would not be difficult to add OpenCCG support for constructing them though, since

```
// configure realizer with 4-gram backoff base model,
// interpolated at the word level with a bigram maximum-likelihood
// cache model, with more weight given to the base model
NgramScorer baseModel = new StandardNgramModel(4, "lm.4bo");
NgramScorer cacheModel = new StandardNgramModel(2, "lm-cache.mle");
realizer.signScorer = new LinearNgramScorerCombo(
    new SignScorer[] { baseModel, cacheModel },
    new double[] { 0.6, 0.4 }
);
```

Figure 5: Example word-level interpolation of a cache model

Figure 5 shows one way to create such an interpolated model.

### 4.4 N-gram precision models

The `NgramPrecisionModel` subclass of `NgramScorer` computes a modified version of the Bleu score used in MT evaluation [PRWZ01]. Its constructor takes as input an array of target strings—from which it extracts the n-gram sequences to use in computing the n-gram precision score—and the desired order. Unlike with the Bleu score, rank order centroid weights (rather than the geometric mean) are used to combine scores of different orders, which avoids problems with scoring partial realizations which have no n-gram matches of the target order. For simplicity, the score also does not include the Bleu score's bells and whistles to make cheating on length difficult.

We have found n-gram precision models to be very useful for regression testing the grammar, as an n-gram precision model created just from the target string nearly always leads the realizer to choose that exact string as its preferred realization. These models can also be useful for evaluating the success of different scoring models in a cross-validation setup, though with high quality output, manual inspection is usually necessary to determine the importance of any differences between the preferred realization and the target string. Finally, note that n-gram precision models can be used as a quick-and-dirty substitute for standard n-gram models, if one does not have time to install and use the SRILM toolkit.

these models do not require smoothing.

### 4.5 Factored language models

A factored language model [BK03b] is a new kind of language model that treats words as bundles of factors. To support scoring with such models, OpenCCG represents words as objects with a surface form, pitch accent, stem, part of speech, supertag, and semantic class. Words may also have any number of further attributes, such as associated gesture classes, in order to handle in a general way elements like pitch accents that are "coarticulated" with words.

To represent words efficiently, and to speed up equality tests, all attribute values are interned, and the `Word` objects themselves are interned via a factory method. Note that in Java, it is straightforward to intern objects other than strings by employing a `WeakHashMap` to map from an object key to a weak reference to itself as the canonical instance. (Using a weak reference avoids accumulating interned objects that would otherwise be garbage collected.)

With the SRILM toolkit, factored language models can be constructed that support *generalized parallel backoff*: that is, backoff order is not restricted to just dropping the most temporally distant word first, but rather may be specified as a path through the set of contextual parent variables; additionally, parallel backoff paths may be specified, with the possibility of combining these paths dynamically in various ways. In OpenCCG, the `FactoredNgramModel` class supports scoring with factored language models that employ generalized backoff, though parallel backoff is not yet supported, as it remains somewhat unclear whether the added complexity of parallel backoff is worth the implementation effort. Typically, several related factored language models are specified in a single file and loaded by a `FactoredNgramModelFamily`, which can multiplicatively score models for different child variables, and include different sub-models for the same child variable.

To illustrate, let us consider a simplified version of the factored language model family used in the COMIC realizer. This model computes the probability of the current word given the preceding ones according to the formula shown in (3), where a word consists of the factors word (W), pitch accent (A), gesture class (GC), and gesture instance (GI), plus the other standard factors which the model ignores:

$$
\begin{aligned}
&P(\langle W, A, GC, GI\rangle \,|\, \langle W, A, GC, GI\rangle_{-1} \ldots) \approx \\
&\quad P(W \,|\, W_{-1}W_{-2}A_{-1}A_{-2}) \times \\
&\quad P(GC \,|\, W) \times \\
&\quad P(GI \,|\, GC)
\end{aligned} \tag{3}
$$

In (3), the probability of the current word is approximated by the probability of the current word form given the preceding two word forms and preceding two pitch accents, multiplied by the probability of the current gesture class given the current word form, and by the probability of the current gesture instance given the current gesture class. Note that in the COMIC grammar, the choice of pitch accent is entirely rule governed, so the current pitch accent is not scored separately in the model. However, the preceding pitch accents are taken into account in predicting the current word form, as perplexity experiments have suggested that they do provide additional information beyond that provided by the previous word forms.

The specification file for this model appears in Figure 6. The format of the file is a restricted form of the files used by the SRILM toolkit to build factored language models. The file specifies four models, where the first, third and fourth models correspond to those in (3). With the first model, since the previous words are typically more informative than the previous pitch accents, the backoff order specifies that the most distant accent, `A(-2)`, should be dropped first, followed by the previous accent, `A(-1)`, then the most distant word, `W(-2)`, and finally the previous word, `W(-1)`. The second model is considered a sub-model of the first—since it likewise predicts the current word—to be used when there is only one word of context available (i.e. with bigrams). Note that when scoring a bigram, the second model will take the previous pitch accent into account, whereas the first model would not. For documentation of the file format as it is used in the SRILM toolkit, see [KBD+02].

Like `StandardNgramModel`, the `FactoredNgramModel` class stores its n-gram tables in a trie data structure, except that it stores an interned factor key (i.e. a factor name and value pair, or just a string, in the case of the word form) at each node, rather than a simple string. During scoring, the `logProbFromNgram` method determines the log probability (with backoff) of a given n-gram by extracting the appropriate sequence of factor keys, and using them to compute the log probability as with standard n-gram models. The `FactoredNgramModelFamily` class computes log probabilities by delegating to its component factored n-gram models (choosing appropriate sub-models, when appropriate) and summing the results.

### 4.6 Avoiding repetition

While cache models appear to be a promising avenue to promote lexical and syntactic alignment with a conversational partner, a different mechanism appears to be called for to avoid "self-alignment"—that is, to avoid the

```
## Simplified COMIC realizer FLM spec file

## Trigram Word model based on previous words and accents, dropping accents first,
##   with bigram sub-model;
## Unigram Gesture Class model based on current word; and
## Unigram Gesture Instance model based on current gesture class

4

## 3gram with A
W : 4 W(-1) W(-2) A(-1) A(-2) w_w1w2a1a2.count w_w1w2a1a2.lm 5
  W1,W2,A1,A2  A2 ndiscount gtmin 1
  W1,W2,A1  A1 ndiscount gtmin 1
  W1,W2  W2 ndiscount gtmin 1
  W1  W1 ndiscount gtmin 1
  0   0  ndiscount gtmin 1

## bigram with A
W : 2 W(-1) A(-1) w_w1a1.count w_w1a1.lm 3
  W1,A1  A1  ndiscount gtmin 1
  W1  W1  ndiscount gtmin 1
  0   0   ndiscount gtmin 1

## Gesture class depends on current word
GC : 1 W(0) gc_w0.count gc_w0.lm 2
  W0  W0 ndiscount gtmin 1
  0   0  ndiscount gtmin 1

## Gesture instance depends only on class
GI : 1 GC(0) gi_gc0.count gi_gc0.lm 2
  GC0  GC0 ndiscount gtmin 1
  0 0
```

Figure 6: Example factored language model family specification

```
// set up n-gram scorer and repetition scorer
String lmfile = "ngrams/combined.flm";
NgramScorer ngramScorer = new FactoredNgramModelFamily(lmfile, true);
ngramScorer.addFilter(new AAnFilter());
RepetitionScorer repetitionScorer = new RepetitionScorer();

// combine n-gram scorer with repetition scorer
realizer.signScorer = new SignScorerProduct(
    new SignScorer[] { ngramScorer, repetitionScorer }
);

// ... then, after each realization request,
Edge bestEdge = realizer.realize(lf);

// ... update repetition context for next realization:
repetitionScorer.ageContext();
repetitionScorer.updateContext(bestEdge.getSign());
```

Figure 7: Example combination of an n-gram scorer and a repetition scorer

repetitive use of words and phrases. As a means to experiment with avoiding repetition, OpenCCG includes the `RepetitionScorer` class. This class makes use of a configurable penalty plus a set of methods for dynamically managing the context. It returns a score of $10^{-c_r \times p}$, where $c_r$ is the count of repeated items, and $p$ is the penalty. Note that this formula returns 1 if there are no repeated items, and returns a score that is linear in log space with the number of repeated items otherwise.

A repetition scorer can be combined multiplicatively with an n-gram model, in order to discount realizations that repeat items from the recent context. Figure 7 shows such a combination, together with the operations for updating the context. By default, open class stems are the considered the relevant items over which to count repetitions, though this behavior can be specialized by subclassing `RepetitionScorer` and overriding the `updateItems` method. Note that in counting repetitions, full counts are given to items in the previous words or recent context, while fractional counts are given to older items; the exact details may likewise be changed in a subclass, by overriding the `repeatedItems` method.

### 4.7 Building language models with the SRILM toolkit

You can use OpenCCG's regression testing tool, ccg-test, to help build and test language models built with the SRILM toolkit. By default, running ccg-test will use the grammar in the current directory to parse and realize the default regression file, `testbed.xml`, using an n-gram precision model constructed for each test item. Using the appropriate command-line options, it is also possible to export the text of the test items in order to construct an n-gram model with the SRILM toolkit, and then use the resulting model in testing the realizer.

To display the syntax of ccg-test's command-line options, you can invoke it with the the `-h` option, as shown in (4a). To export the text of the test items to a text file, you use the `-text` option, as in (4b). The next step is to use SRILM's `ngram-count` tool to build an n-gram language model. In (4c),[3] `ngram-count` is used to build a 4-gram backoff model, `n.4bo`, from the text file `tb.txt`, using Ristad's "natural" discounting method [Ris95]. For small test sets, we have found that Ristad's method works better than the default one (Good-Turing). Note that in (4c), the `-unk` option is used to reserve some probability for unknown words; the `-gt<N>min 1` options (for N=1 to 4) specify to keep all 1-counts; and the `-ndiscount<N>` options (for N=1 to 4) specify the use of natural discounting for unigrams through 4-grams. Finally, to test the resulting language model, you use ccg-test's `-ngramorder` and `-lm` options, as shown in (4d).

(4) a.
```
ccg-test -h
```

b.
```
ccg-test -text tb.txt
```

c.
```
ngram-count -order 4 -unk -text tb.txt -lm n.4bo
-gt1min 1 -gt2min 1 -gt3min 1 -gt4min 1
-ndiscount1 -ndiscount2 -ndiscount3 -ndiscount4
```

d.
```
ccg-test -noparsing -ngramorder 4 -lm n.4bo
```

To perform a simple 2-fold cross-validation, ccg-test includes options for exporting or testing just the even or odd test items. The command in (5a) shows how you can export just the text of the even-numbered test items. Note that the `-textsc` option specifies that the text be exported using semantic class replacement, i.e. with certain words replaced with their semantic classes; the classes to use for this purpose are specified using the `replacement-sem-classes` attribute of the `tokenizer` element in the `grammar.xml` file. The next step is to build a language model just as before;

[3] This command, and the ensuing ones, should be entered on one line.

the abbreviated command appears in (5b). Finally, you can test the language model on just the odd-numbered items, as in (5c), where the `-lmsc` option specifies that semantic class replacement should be employed when scoring realizations with the model. Naturally, you can switch the `-even` and `-odd` flags, and adjust the text and language model names, to test realization on the even-numbered items, using a language model trained from the odd-numbered ones.

(5) a. `ccg-test -even -textsc tb-sc.even.txt`

b. `ngram-count -order 4 -unk -text tb-sc.even.txt -lm n-sc.even.4bo ...`

c. `ccg-test -noparsing -odd -ngramorder 4 -lmsc n-sc.even.4bo`

An example of building a factored language model appears next. In (6a), the text of the test items is exported, where each word appears with all its factors, and word forms are replaced with semantic classes when appropriate. In (6b), SRILM's `fngram-count` is used to create a factored language model from the spec file named `spec.flm`. (Note that the various individual language model files are listed in the spec file.) Finally, (6c) shows how the factored language model can be tested in `ccg-test`.

(6) a. `ccg-test -textfsc tb-fsc.txt`

b. `fngram-count -factor-file spec.flm -text tb-fsc.txt -lm -unk`

c. `ccg-test -noparsing -flmsc spec.flm`

## 5 Pruning Strategies

The classes for defining edge pruning strategies appear in Figure 8. As mentioned in Section 2, an N-best pruning strategy is employed by default, where N is determined by the current preference settings. It is also possible to define custom strategies. To support the definition of a certain kind of custom strategy, the abstract class `DiversityPruningStrategy` provides an N-best pruning strategy that promotes diversity in the edges that are kept, according to the equivalence relation established by the abstract `notCompellinglyDifferent` method. In particular, in order to determine which edges to keep, a diversity pruning strategy clusters the edges into a ranked list of equivalence classes, which are sequentially sampled until the limit N is reached. If the `singleBestPerGroup` flag is set, then a maximum of one edge per equivalence class is retained.

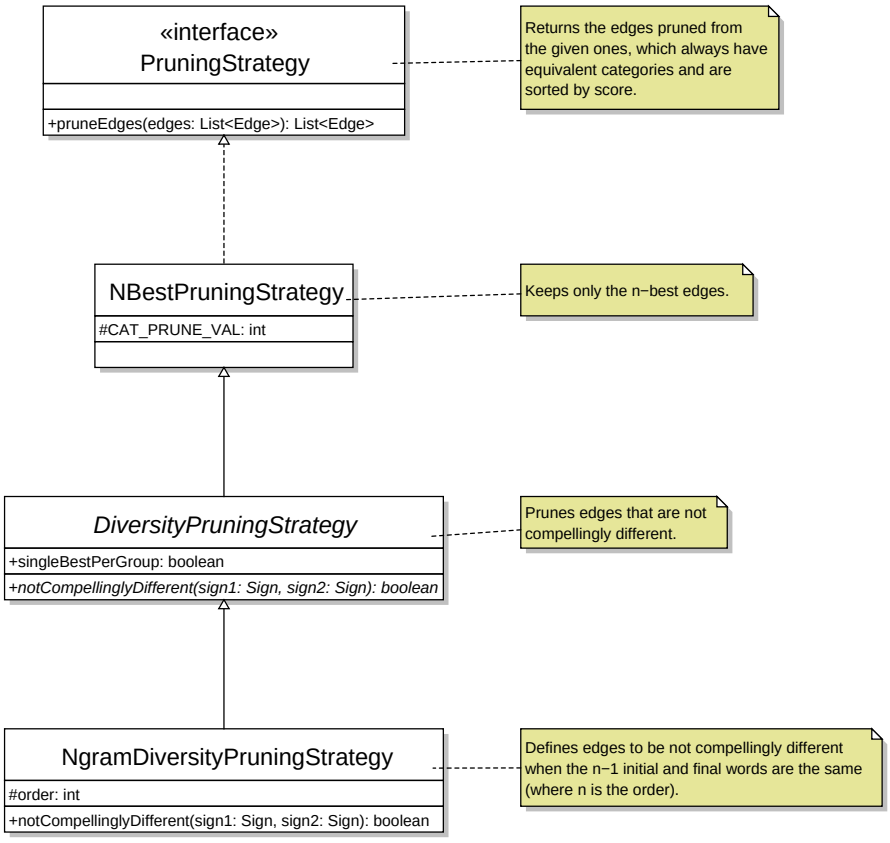



Figure 8: Classes for defining pruning strategies

```
// configure realizer with gesture diversity pruner
realizer.pruningStrategy = new DiversityPruningStrategy() {
  /**
   * Returns true iff the given signs are not compellingly different;
   * in particular, returns true iff the words differ only in their
   * gesture instances. */
  public boolean notCompellinglyDifferent(Sign sign1, Sign sign2) {
    List words1 = sign1.getWords(); List words2 = sign2.getWords();
    if (words1.size() != words2.size()) return false;
    for (int i = 0; i < words1.size(); i++) {
      Word w1 = (Word) words1.get(i); Word w2 = (Word) words2.get(i);
      if (w1 == w2) continue;
      if (w1.getForm() != w2.getForm()) return false;
      if (w1.getPitchAccent() != w2.getPitchAccent()) return false;
      if (w1.getVal("GC") != w2.getVal("GC")) return false;
      // nb: assuming that they differ in the val of GI at this point
    }
    return true;
  }
};
```

Figure 9: Example diversity pruning strategy

As an example, the COMIC realizer's diversity pruning strategy appears in Figure 9. The idea behind this strategy is to avoid having the N-best lists become full of signs whose words differ only in the exact gesture instance associated with one or more of the words. With this strategy, if two signs differ in just this way, the edge for the lower-scoring sign will be considered "not compellingly different" and pruned from the N-best list, making way for other edges whose signs exhibit more interesting differences.

OpenCCG also provides a concrete subclass of `DiversityPruningStrategy` named `NgramDiversityPruningStrategy`, which generalizes the approach to pruning described in [Lan00]. With this class, two signs are considered not compellingly different if they share the same $n-1$ initial and final words, where $n$ is the n-gram order. When one is interested in single-best output, an n-gram diversity pruning strategy can increase efficiency while guaranteeing no loss in quality—as long as the reduction in the search space outweighs the extra time necessary to check for the same initial and final words—since any words in between an input sign's $n-1$ initial and final ones cannot affect the n-gram score of a new sign formed from the input sign. However, when N-best outputs are desired, or when repetition scoring is employed, it is less

clear whether it makes sense to use an n-gram diversity pruning strategy; for this reason, a simple N-best strategy remains the default option.

## 6 Disjunctive logical forms

In applications, to specify the desired space of possible paraphrases, one may either provide an input logical form that underspecifies certain realization choices, or include explicit disjunctions in the input LF (or both). In our experience, we have found disjunctive LFs—inspired by those found in [She97]—to be an important capability, especially as one seeks to make grammars reusable across applications.

As an illustration of disjunctive logical forms, consider the semantic dependency graphs in Figure 10, which are taken from the COMIC[4] multimodal dialogue system.[5] Given the lexical categories in the COMIC grammar, the graphs in Figure 10(a) and (b) fully specify their respective realizations, with the exception of the choice of the full or contracted form of the copula.[6] To generalize over these alternatives, the disjunctive graph in (c) may be employed. This graph allows a free choice between the domain synonyms *collection* and *series*, as indicated by the vertical bar between their respective predications. The graph also allows a free choice between the ⟨Creator⟩ and ⟨GenOwner⟩ relations—lexicalized via *by* and the possessive, respectively—connecting the head $c$ (*collection* or *series*) with the dependent $v$ (for *Villeroy and Boch*); this choice is indicated by an arc between the two dependency relations.[7] Finally, the determiner feature (⟨det⟩the) on $c$ is indicated as optional, via the question mark.[8]

It is worth pausing at this point to observe that in designing the COMIC grammar, the differences between (a) and (b) could perhaps have been collapsed. However, such a move would make it more difficult to reuse the grammar in other applications—and indeed, the core of the grammar is shared with the FLIGHTS system [MFLW04]—as it would presuppose that

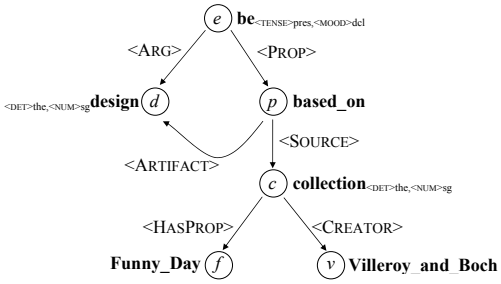
[4] `http://www.hcrc.ed.ac.uk/comic/`

[5] To simplify the exposition, the features specifying information structure and deictic gestures have been omitted, as have the semantic sorts of the discourse referents.

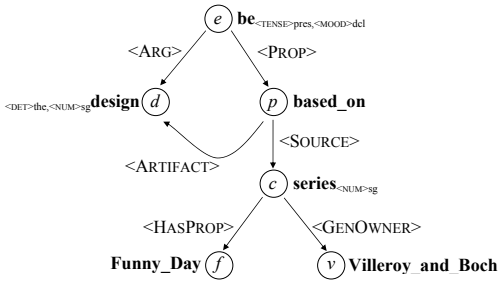
[6] Note that to be consistent with the distributed grammar, the predicate **based_on** should actually be **based-on**; this discrepancy has been corrected in the subsequent figures.

[7] Note that the arc and vertical bar are just presentation devices; there is no difference in the underlying implementation.

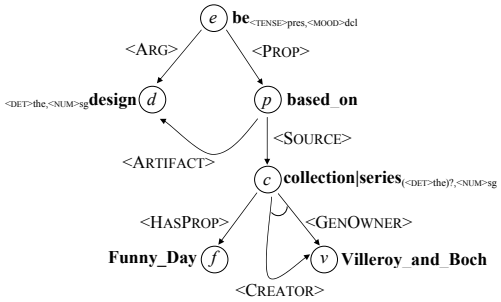
[8] Another option would be to include the determiner feature in the alternative with the ⟨Creator⟩ relation, but that would make the graph harder to draw and would not illustrate optionality.




(a) Semantic dependency graph for *The design (is|'s) based on the Funny Day collection by Villeroy and Boch.*




(b) Semantic dependency graph for *The design (is|'s) based on Villeroy and Boch's Funny Day series.*




(c) Disjunctive semantic dependency graph covering (a)-(b), i.e. *The design (is|'s) based on (the Funny Day (collection|series) by Villeroy and Boch | Villeroy and Boch's Funny Day (collection|series)).*

Figure 10: Example semantic dependency graphs.

$@_e(\textbf{be} \wedge \langle \textsc{tense} \rangle \text{pres} \wedge \langle \textsc{mood} \rangle \text{dcl} \wedge$
$\quad \langle \textsc{Arg} \rangle (d \wedge \textbf{design} \wedge \langle \textsc{det} \rangle \text{the} \wedge \langle \textsc{num} \rangle \text{sg}) \wedge$
$\quad \langle \textsc{Prop} \rangle (p \wedge \textbf{based-on} \wedge$
$\qquad \langle \textsc{Artifact} \rangle d \wedge$
$\qquad \langle \textsc{Source} \rangle (c \wedge \textbf{collection} \wedge \langle \textsc{det} \rangle \text{the} \wedge \langle \textsc{num} \rangle \text{sg} \wedge$
$\qquad\quad \langle \textsc{HasProp} \rangle (f \wedge \textbf{Funny\_Day}) \wedge$
$\qquad\quad \langle \textsc{Creator} \rangle (v \wedge \textbf{V\&B}))))$

(a)

$\vdots$

$@_e(\textbf{be} \wedge \langle \textsc{tense} \rangle \text{pres} \wedge \langle \textsc{mood} \rangle \text{dcl} \wedge$
$\quad \langle \textsc{Arg} \rangle (d \wedge \textbf{design} \wedge \langle \textsc{det} \rangle \text{the} \wedge \langle \textsc{num} \rangle \text{sg}) \wedge$
$\quad \langle \textsc{Prop} \rangle (p \wedge \textbf{based-on} \wedge$
$\qquad \langle \textsc{Artifact} \rangle d \wedge$
$\qquad \langle \textsc{Source} \rangle (c \wedge \langle \textsc{num} \rangle \text{sg} \wedge (\langle \textsc{det} \rangle \text{the})? \wedge$
$\qquad\quad (\textbf{collection} \veebar \textbf{series}) \wedge$
$\qquad\quad \langle \textsc{HasProp} \rangle (f \wedge \textbf{Funny\_Day}) \wedge$
$\qquad\quad (\langle \textsc{Creator} \rangle \boxed{v} \veebar \langle \textsc{GenOwner} \rangle \boxed{v}))))$
$\wedge\ @_v(\textbf{Villeroy\_and\_Boch})$

(c)

Figure 11: HLDS for examples in Figure 10.

these paraphrases should always available in the same contexts. An example where the disjunctively specified paraphrases have applicable contexts that are more clearly limited appears in (7):

(7) (This design | This one | This) (is|'s) (classic | in the classic style) | Here we have a (classic design | design in the classic style).

This example shows some of the phrasings that may be used in COMIC to describe the style of a design that has not been discussed previously. The example includes a top-level disjunction between the use of a deictic NP *this design* | *this one* | *this* (with an accompanying pointing gesture) followed by the copula, or the use of the phrase *here we have* to introduce the design. While these alternatives can function as paraphrases in this context, it is difficult to see how one might specify them in a single underspecified (and application-neutral) logical form.

Graphs such as those in Figure 10 are represented internally using Hybrid Logic Dependency Semantics (HLDS), as in Figure 11. In HLDS, as can be seen in Figure 11(a), each semantic head is associated with a nominal that identifies its discourse referent, and heads are connected to their dependents via dependency relations, which are modeled as modal relations; modal relations are also used to represent semantic features (in which case the relation is to an anonymous node). In (c), two new operators are introduced to represent periphrastic alternatives and optional parts of the meaning, namely $\veebar$ and $(\cdot)?$, for exclusive-or and optionality, respectively. To indicate that a nominal represents a reference to a node that is considered a shared part of multiple alternatives, the nominal is annotated with a box, as exemplified by $\boxed{v}$. This notion of shared references is needed during the logical form flattening stage of the realization algorithm in order to determine which elementary predications are part of each alternative.

To specify inputs to the realizer, an XML representation of HLDS terms may be employed; alternatively, the more intuitive XML graph representation illustrated in Figure 12 and Figure 13 may be used, with an automatic translation converting such representations to HLDS. As can be seen in Figure 12, the nodes and dependency relations in the graph are represented by `node` and `rel` elements. Note that `node` elements that represent subordinated, reentrant references to a node use an `idref` attribute, as exemplified by the `Artifact` relation to the `node` element with `idref="d"`. Figure 13 shows how periphrastic alternatives and optional parts of the meaning are specified using the `one-of` and `opt` elements, respectively. Where the alternatives involve attributes of a node, an `atts` element is used to provide the

```
<node id="e" pred="be" tense="pres" mood="dcl">
  <rel name="Arg">
    <node id="d" pred="design" det="the" num="sg"/>
  </rel>
  <rel name="Prop">
    <node id="p" pred="based-on">
      <rel name="Artifact"> <node idref="d"/> </rel>
      <rel name="Source">
        <node id="c" pred="collection" det="the" num="sg">
          <rel name="HasProp">
            <node id="f" pred="Funny_Day"/>
          </rel>
          <rel name="Creator">
            <node id="v" pred="Villeroy_and_Boch"/>
          </rel>
        </node>
      </rel>
    </node>
  </rel>
</node>
```

Figure 12: XML for example (a) Figure 10.

lexical predications or semantic features in question. Finally, note that `node` elements that represent references to a node that is considered a shared part of multiple alternatives is marked with the `shared="true"` attribute, as is the case here with the references to the dependent node $v$ (for *Villeroy and Boch*).

## References


[Bal02] Jason Baldridge. *Lexically Specified Derivational Control in Combinatory Categorial Grammar*. PhD thesis, School of Informatics, University of Edinburgh, 2002.

[BIOW05] Carsten Brockmann, Amy Isard, Jon Oberlander, and Michael White. Variable alignment in affective dialogue. In *Proc. UM-05 Workshop on Affective Dialogue Systems*, 2005.

[BK03a] Jason Baldridge and Geert-Jan Kruijff. Multi-Modal Combinatory Categorial Grammar. In *Proc. ACL-03*, 2003.

[BK03b] Jeff Bilmes and Katrin Kirchhoff. Factored language models and general parallelized backoff. In *Proc. HLT-03*, 2003.

```
<node id="e" pred="be" tense="pres" mood="dcl">
  <rel name="Arg">
    <node id="d" pred="design" det="the" num="sg"/>
  </rel>
  <rel name="Prop">
    <node id="p" pred="based-on">
      <rel name="Artifact"> <node idref="d"/> </rel>
      <rel name="Source">
        <node id="c" num="sg">
          <opt> <atts det="the"/> </opt>
          <one-of>
            <atts pred="collection"/>
            <atts pred="series"/>
          </one-of>
          <rel name="HasProp">
            <node id="f" pred="Funny_Day"/>
          </rel>
          <one-of>
            <rel name="Creator">
              <node idref="v" shared="true"/>
            </rel>
            <rel name="GenOwner">
              <node idref="v" shared="true"/>
            </rel>
          </one-of>
        </node>
      </rel>
    </node>
  </rel>
</node>
<node id="v" pred="Villeroy_and_Boch"/>
```

Figure 13: XML for example (c) Figure 10.

[CCFP99] John Carroll, Ann Copestake, Dan Flickinger, and Victor Poznański. An efficient chart generator for (semi-) lexicalist grammars. In *Proc. EWNLG-99*, 1999.

[Kay96] Martin Kay. Chart generation. In *Proc. ACL-96*, 1996.

[KBD+02] Katrin Kirchhoff, Jeff Bilmes, Sourin Das, Nicolae Duta, Melissa Egan, Gang Ji, Feng He, John Henderson, Daben Liu, Mohamed Noamany, Pat Schone, Richard Schwartz, and Dimitra Vergyri. Novel Approaches to Arabic Speech Recognition: Report from the 2002 Johns-Hopkins Summer Workshop, 2002.

[Lan00] Irene Langkilde. Forest-based statistical sentence generation. In *Proc. NAACL-00*, 2000.

[Mal00] Robert Malouf. The order of prenominal adjectives in natural language generation. In *Proc. ACL-00*, 2000.

[MFLW04] Johanna Moore, Mary Ellen Foster, Oliver Lemon, and Michael White. Generating tailored, comparative descriptions in spoken dialogue. In *Proc. FLAIRS-04*, 2004.

[Moo02] Robert C. Moore. A complete, efficient sentence-realization algorithm for unification grammar. In *Proc. INLG-02*, 2002.

[OR02] Alice H. Oh and Alexander I. Rudnicky. Stochastic natural language generation for spoken dialog systems. *Computer, Speech & Language*, 16(3/4):387–407, 2002.

[PRWZ01] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: a Method for Automatic Evaluation of Machine Translation. Technical Report RC22176, IBM, 2001.

[Rat02] Adwait Ratnaparkhi. Trainable approaches to surface natural language generation and their application to conversational dialog systems. *Computer, Speech & Language*, 16(3/4):435–455, 2002.

[Ris95] Eric S. Ristad. A Natural Law of Succession. Technical Report CS-TR-495-95, Princeton Univ., 1995.

[SH99] James Shaw and Vasileios Hatzivassiloglou. Ordering among premodifiers. In *Proc. ACL-99*, 1999.

[She97] Hadar Shemtov. *Ambiguity Management in Natural Language Generation*. PhD thesis, Stanford University, 1997.

[Ste00a] Mark Steedman. Information structure and the syntax-phonology interface. *Linguistic Inquiry*, 31(4):649–689, 2000.

[Ste00b] Mark Steedman. *The Syntactic Process*. MIT Press, 2000.

[Sto02] Andreas Stolcke. SRILM — An extensible language modeling toolkit. In *Proc. ICSLP-02*, 2002.

[WB03] Michael White and Jason Baldridge. Adapting Chart Realization to CCG. In *Proc. EWNLG-03*, 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 *Proc. INLG-04*, 2004.

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