CCGlab manual

version 6.0 September 7, 2019 coloured text means changes from previous version

Cem Bozşahin
Lisboa, Ankara, Datça, 2015–2019

Home: github.com/bozsahin/ccglab
skipping ghost releases 4.x

1 Introduction

CCGlab is a tool for experimenting with Combinatory Categorial Grammar (CCG; Steedman 1996, 2000, 2012, Clark and Curran 2003, Zettlemoyer and Collins 2005, Steedman and Baldridge 2011, Bozşahin 2012). CCGlab grammars are written almost in paper-style, and the results are almost in paper format.

It is written in COMMON LISP. There is an installer script at home cite of CCGlab to install all software that is required. It implements all the established combinators of CCG, namely application, composition and substitution, including their finite powers (quadratic for **S**; cubic for **B**), in all their directional variants. There is a compiler for **T**. Every rule has a switch for experimentation. CCGlab also implements some experimental projection rules. By default, only basic CCG (application, composition, substitution, and powers) is on. You can turn on other rules as desired. The table at the end explains how. Hereafter we assume intermediate familiarity with CCG.

CCGlab is designed with linguists, cognitive scientists and computational linguists in mind. Raw input to the system is lexical specifications and unary rules, much like in Steedman papers. Unlike unification-inspired systems, the logical form (LF) is associated with the overall syntactic category, and the underlying lambda-calculus is visible at all times. Wide-coverage parsing with CCGlab is a long-term goal.

The examples below show raw input to CCGlab.[1] The first three are lexical items, and the last one is a unary rule. Order of specifications and whitespacing are not important except for unary rules; they apply in order. They do not apply to same category twice. Later unary rules can see the results of earlier ones.

```
John   n := np[agr=3s] : !john ;
likes  v := (s\np[agr=3s])/^np : \x\y. !like x y;
and    x := (@X\*@X)/*@X : \p\q\x. !and (p x)(q x);
(L1) np[agr=?x] : lf --> s/(s\np[agr=?x]) : \lf\p. p lf ;
```

Unification plays no role in CCGlab because we want to see how far grammatical inference can be carried out by combinators. Features are checked for atomic category consistency only, and no feature is passed non-locally. All features are simple, feature variables are for values only, they are local to a category, and they are atomic-valued. For example, if we have the sequent (a) below, we get the result (b) by composition, not (c). We could in principle compile all finite feature values for any basic category in a grammar and get rid of its features (but nobody does that).

(a) `s[f1=?x1,f2=v2]/s[f1=?x1]  s[f1=v1,f2=?x2]/np[f2=?x2]`
(b) `s[f1=v1,f2=v2]/np[f2=?x2]`
(c) `s[f1=v1,f2=v2]/np[f2=v2]`

Slash modalities are implemented too (see Baldridge 2002, Steedman and Baldridge 2011).

Meta-categories such as $(X\backslash X)/X$ are allowed with application only, which maintains a very important property of CCG: it is procedurally neutral (Pareschi and Steedman, 1987). Given two substantive categories and CCG's understanding of universal rules, there is only one way to combine them, so that the parser can eschew other rules for that sequent if it succeeds in one rule. For this reason, '`X$`' categories have not been incorporated. They require structured unification and jeopardize procedural neutralism.

Singleton categories are supported by CCGlab; see Bozşahin and Güven (2018) for their syntax and semantics. They allow us to continue to avoid wrapping in verb-particle constructions, and render idiomatically combining expressions and phrasal idioms as simple categorial possibilities. For example, in CCGlab notation, the categories for heads of these expressions are respectively:

(a) `picked := (s\np)/"up"/np : \x\y\z.!pick _ y x z;`

(b) `kicked := (s\np)/"the bucket" : \y\z.!die _ y z;`

(c) `spilled := (s\np)/np[h=beans,spec=p] : \y\z.!reveal _ y !secret z;`

where `"up"` and `"the bucket"` are singleton categories (i.e. surface strings turned into singleton category,

[1] The mapping of modalities in CCGlab to those in papers is (. , .), (^,◇), (*,⋆), (+,×).

with constant value), and `np[h=beans,spec=p]` in (c) means this NP must be headed by `beans`, which makes it a special subcategorization, by `spec=p`, meaning +special. These are not built-in features (in fact there is no such thing in `CCGlab`). The underscore is just a reminder that what follows is not an argument of the predicate; it is an event modality. This is taken care of here by a notational convention.

Below is the output for the Latin phrase for *Balbus is building the wall*. The first column is the rule index. The rest is about the current step: a lexical assumption, result of a unary rule, or one step of a derivation.

```
Derivation 1
-----------------
LEX   MUR := N  : WALL
LEX   UM := (S/(S\NP))\N  : (LAM X (LAM P (P X)))
<     (MUR)(UM) := S/(S\NP)  : ((LAM X (LAM P (P X))) WALL)
LEX   BALB := N  : BALB
LEX   US := (S/(S\NP))\N  : (LAM X (LAM P (P X)))
<     (BALB)(US) := S/(S\NP)  : ((LAM X (LAM P (P X))) BALB)
LEX   AEDIFICAT := (S\NP)\NP  : (LAM X (LAM Y ((BUILD X) Y)))
>Bx   (BALB US)(AEDIFICAT) := S\NP  : (LAM X
                                       (((LAM X (LAM P (P X))) BALB)
                                        ((LAM X (LAM Y ((BUILD X) Y))) X)))
>     (MUR UM)(BALB US AEDIFICAT) := S  : (((LAM X (LAM P (P X))) WALL)
                                           (LAM X
                                            (((LAM X (LAM P (P X))) BALB)
                                             ((LAM X (LAM Y ((BUILD X) Y)))
                                              X))))

Final LF, normal-order evaluated: ((BUILD WALL) BALB)
```

The inductive component prints associated parameter-weighted local counts of features and the final count, for verification. Every lexical entry, word or unary rule, is assigned a parameter. In the example below, all parameters were set to unity so that weighted counting to be described in (2) below can be seen easily. The only feature is the number of times lexical items are used in a derivation. 18 is the total here.

```
LEX      1.0  MUR := N  : WALL
LEX      1.0  UM := (S/(S\NP))\N  : (LAM X (LAM P (P X)))
<        2.0  (MUR)(UM) := S/(S\NP)  : ((LAM X (LAM P (P X))) WALL)
LEX      1.0  BALB := N  : BALB
LEX      1.0  US := (S/(S\NP))\N  : (LAM X (LAM P (P X)))
<        2.0  (BALB)(US) := S/(S\NP)  : ((LAM X (LAM P (P X))) BALB)
>0       4.0  (MUR UM)(BALB US) := S/((S\NP)\NP)  : (LAM H
                                                     (((LAM X (LAM P (P X)))
                                                       WALL)
                                                      (LAM X
                                                       (((LAM X (LAM P (P X)))
                                                         BALB)
                                                        (H X)))))
LEX      1.0  AEDIFICAT := (S\NP)\NP  : (LAM X (LAM Y ((BUILD X) Y)))
>       18.0  (MUR UM BALB US)(AEDIFICAT) := S  : ((LAM H
                                                    (((LAM X (LAM P (P X)))
                                                      WALL)
                                                     (LAM X
                                                      (((LAM X (LAM P (P X)))
                                                        BALB)
                                                       (H X)))))
                                                   (LAM X
                                                    (LAM Y ((BUILD X) Y))))
Normalized LF: ((BUILD WALL) BALB)
```

The deductive component computes all the constituents and their LFs that are consequences of lexical assumptions. The inductive component computes (i) the most likely LF for a given string, (ii) the most probable derivation for that LF, and (iii) the highest-weighted derivation for any LF. We assume that the inductive component is supplied with lexical parameters after parameter estimation. (The deductive component starts with the assumption that they are unity.) They use the same grammar format.

The algorithm for the basic inductive component is more or less standard in Probabilistic CCG (PCCG). The one we use is summarized from Zettlemoyer and Collins (2005) throughout the manual:

$$\arg\max_{L} P(L \mid S;\bar{\theta}) = \arg\max_{L} \sum_{D} P(L,D \mid S;\bar{\theta}) \tag{1}$$

where S is the sequence of words to be parsed, L is a logical form for it, D is a sequence of CCG derivations for the (L,S) pair, and $\bar{\theta}$ is the n-dimensional parameter vector for a grammar of size n (the total number of lexical items and rules). The term on the right-hand side is induced from the following relation of probabilities

[2] You may be alarmed by the exponentiation in formula (2) potentially causing floating-point overflow, and may worry about what

and parameters in PCCG (*ibid.*);[2] where $\bar{f}$ is a vector of 3-argument functions $<f_1(L,D,S),\cdots f_n(L,D,S)>$:

$$P(L,D \mid S;\bar{\theta}) = \frac{e^{\bar{f}(L,D,S)\cdot\bar{\theta}}}{\sum_L \sum_D e^{\bar{f}(L,D,S)\cdot\bar{\theta}}} \tag{2}$$

The functions of $\bar{f}$ count local substructure in D. By default, f_i is the number of times the lexical element i (item or rule) is used in D, sometimes called the *feature i*. If you want to count other substructures in D or L, as in Clark and Curran (2003), you need to write some code. A plug-in is provided. That was one motivation for giving detailed specs. There is a growing body of literature on the topic, starting with Clark and Curran.

You can think of `CCGlab` as Lisp code in three layers, Paul Graham 1994-style: (i) the representational layer, which is based on `(name value)` pairs and lists of such pairs, so on; (ii) the parsing layer, which is based on hash tables and the representation layer; and (iii) the post-parsing layer, which is based on λ-calculus and the parsing layer, which is used for checking LF equivalence.[3] Combinators are Lisp macros on the last layer. Because our lambda layer has nothing to do with Lisp's lambdas (the internals of LFs are always visible), you can use the top layer as a debugging tool for your LF assumptions. It has two LF evaluators: normal-order and applicative-order. If they do not return the same LF on the same parse result, then there is something strange about your LF.[4]

2 `CCGlab` projects

We recommend that you create a separate directory for each project to keep things clean. `CCGlab` will need several files in the working directory to set up a CCG grammar or a model. Many of them would be optional depending on your need.

By a grammar we mean a set of CCG assumptions like the one above which you want to subject to CCG's universal combinatorics. By a model we mean a version of the grammar which you've subjected to empirical training and parameter estimation, results of which you want to subject to CCG's universal combinatorics.

A project say with name P consists of the following files (we explain their format in §4):

- P`.ccg` : The CCG grammar as text file, much like in papers. You write this one.
- P`.lisptokens` : The version of the text file suited for the Lisp reader, and wrapped in `(..)`. The `sed` script called `tokens` does this by taking into account Lisp's hypersensitivity to special symbols such as '`(`', '`)`', '`\`' etc., and `CCGlab`'s stubbornness that e.g. '`:=`' is one token, not two. They are wrapped in vertical bars, e.g. `|:=|`. Keep in mind that Lisp has its own tokenizer and whitespace preprocessor, so this step is crucial to turn your grammar into Lisp code in the way you intended, rather than Lisp. Your grammar becomes a Lisp object at one fell swoop because of this wrapping.
- P`.ded` : Lisp translation of the input grammar P`.lisptokens`, as input to deduction. The deducer just closes universal syntax-semantics of CCG on your lexical assumptions.
- P`.ind` : The model-trained version of the grammar, as input to induction and parse ranking. It has the same format as P`.ded`. This is possible because the deducer already knows how to compute the parameter-weighted sums of features in a derivation.
- P`.sup` : The supervision file. This is the training data for PCCG parameter estimation. It consists of sentence-LF pairs. Syntax and derivations are hidden variables in PCCG, and training is done solely on sentence-LF pairs.
- P`.supervision` : A simpler way to create the `.sup` file. Content is semicolon-separated sequence of training pairs such as below. LF is much easier to specify than `.sup`. `CCGlab` will do the conversion.

  `words:lf;`

  Translation from .supervision to .sup is highly recommended, unless you are a parenthesis monster.
- P`.lisp` : Optional. Project-specific code developed by you. For example, your plug-in for feature counts can go in here if you have one. In fact you can call this file anything you like as long as you know what to load into Lisp. A standard name is recommended for others to keep track of your work.

your value range should be for θ_i to avoid that. We recommend starting with $0 < \theta_i \leq 1$. Keep also in mind that θ_i are not probabilities but weights. They can be negative. Formula (2) takes care of any weight.

[3] This layer is post-parsing in the sense that although parsing builds LFs, it does not reduce them till the very end. So unevaluated LFs of `CCGlab` are available for exploration.

[4] It doesn't follow that your LF is correct if both evaluations return the same result. If it did, we wouldn't need empirical sciences like linguistics and cognitive science. Your categories, and derivations with them, can tell you more.

– *P*.testsuite.lisp : Optional. I put my test sentences and their output display code in such files to keep things tidy.
– *P*.exp : Optional. Each line is one full specification of an experiment to be run off-line. CCGlab fetches as much processor as there are lines to run the experiments. Used by train-sbcl-multithread.

3 Workflows

There are basically two workflows. Model development cannot be fully automated, so it's harder.[5] In the cases below, the Lisp parts must be done after ccglab command. The README file explains how.

3.1 Grammar development and testing

If you start with a linguistically-motivated grammar, you'd probably take the following steps. Let's assume your project's name is example:

1. In your working directory, write your grammar and save it as plain text e.g. example.ccg
2. In CCGlab do e.g. (load-grammar "example" :make t).
   This means Lisp tokens will be generated by sed from within Lisp.
   This step prepares the example.ded file and loads it onto Lisp.
   If you omit the ":make t" part, it will be assumed that .lisptokens file is already fresh.
   If you update your .ccg source you must re-make the grammar with ":make t".
3. Do: (p '(a sequence of words)) to run the CKY parser.
   Replace p function with rank to run the parse-ranker, i.e. the inducer.
4. Do: (ders) to see all the derivations, or (probs) to see ranked parses.
   You can restrict the display to some results only, by using this function as (p <bcat>), where <bcat> is a basic category. Calculations will be done with all available results, but only these results will be shown.
   Run (cky-pprint) to see the CKY table. It prints good detail for debugging.

If there were errors in your grammar file, step 2 would fail to load the grammar, and you'd need to go back to editing example.ccg to fix it. Partially-generated example.ded will tell you where the *first error* was found. Other functionalities of CCGlab are explained in §5.

3.2 Model testing

At some point, you may turn to modeling. It usually means taking the example.ded file and saving it as example.ind first, then adjusting its parameters (because they were set to default in .ded) by parameter estimation, and playing with its categories.

Model training is not easy to reduce to a standard workflow because it depends on what your model is intended to do (whereas we all know what a grammar is supposed to do—get to semantics). CCGlab helps with the basics (lexical parameters) to compute (1–2) and other formulae such as (3–6) below. A plug-in called (plugin-count-more-substructure <resultcell>) is provided, where CKY result cells and their derivation sequences are at your disposal.

In the end, you create an example.ind file in the same format that you started. This means that every lexical entry (lexical item or unary rule) is associated with a parameter. This is the minimum. If you have more parameters, you must write some code above the minimal machinery provided by CCGlab to change induction. The model testing workflow is:

1. In CCGlab, do: (load-model "example").
   This will load the grammar in example.ind with its parameter set.
2. Do: (rank '(a sequence of words)) to CKY-parse and rank the parses.
3. Do: (probs) to see three results for the sentence: (i) most likely LF for it, (ii) its most likely derivation, and (iii) most likely derivation for any LF for the sentence. You can also do (cky-pprint) to see the CKY table. If you like, do (cky-show-deduction) to see all the results, which is same as (ders). It does not recompute anything.

There is an on/off switch to control what to do with out of vocabulary (OOV) items. If you turn it on

[5] Grammar development cannot be fully automated either, but that's another story. Beware of claims to the contrary, sometimes emanating from CL/NLP/logic circles. This is good for business, for us grammarians.

(see Table 5), it will create two lexical entries with categories $X\backslash_\star X$ and $X/_\star X$ for every unknown item so that the rest can be parsed along with the unknown items as much as it is possible with application. This much knowledge-poor strategy is automated.

Their LFs are the same: $\lambda p.unknown' p$. In training it seems best to keep the switch off so that OOV items are complained about by `CCGlab` for model debugging; in testing wide-coverage parsers might opt to switch it on (no promises).

3.3 Model development: parameter estimation

Parameters of an `.ind` file can be re-estimated from training data of (L_i, S_i) pairs where L_i is the logical form associated with sentence S_i. The log-likelihood of the training data of size n is:

$$O(\bar{\theta}) = \sum_{i=1}^{n} log\, P(L_i \mid S_i; \bar{\theta}) = \sum_{i=1}^{n} log\, (\sum_{T} P(L_i, T \mid S_i; \bar{\theta})) \tag{3}$$

Notice that syntax is marginalized by summing over all derivations T of (L_i, S_i).

For individual parameters we look at the partial derivative of (3) with respect to parameter θ_j. The local gradient of θ_j with feature f_j for the training pair (L_i, S_i) is the difference of two expected values:

$$\frac{\partial O_i}{\partial \theta_j} = E_{f_j(L_i,T,S_i)} - E_{f_j(L,T,S_i)} \tag{4}$$

The gradient will be negative if feature f_j contributes more to any parse than it does for the correct parses of (L_i, S_i). It will be zero if all parses are correct, and positive otherwise. Expected values of f_j are therefore calculated under the distributions $P(T \mid S_i, L_i; \bar{\theta})$ and $P(L, T \mid S_i; \bar{\theta})$. For the overall training set, using sums, the partial derivative is:

$$\frac{\partial O}{\partial \theta_j} = \sum_{i=1}^{n} \sum_{T} f_j(L_i, T, S_i) P(T \mid S_i, L_i; \bar{\theta}) - \sum_{i=1}^{n} \sum_{L} \sum_{T} f_j(L, T, S_i) P(L, T \mid S_i; \bar{\theta}) \tag{5}$$

Once we have the derivative, we use stochastic gradient ascent to re-estimate the parameters:

Initialize $\bar{\theta}$ to some value. (6)

for $k = 0 \cdots N-1$
  for $i = 1 \cdots n$
    $\bar{\theta} = \bar{\theta} + \dfrac{\alpha_0}{1 + c(i + kn)} \dfrac{\partial log\, P(L_i | S_i; \bar{\theta})}{\partial \bar{\theta}}$

where N is the number of passes over the training set, n is the training set size, and α_0 and c are learning-rate parameters (learning rate and learning rate rate). The function `update-model` computes (6) by taking these as arguments. We use the inside-outside algorithm, that is, non-zero counts are found before the loop above, and the rest is eschewed. Both formulae can be beam-searched to make large models with long training feasible. You can turn it off to see the complete update of the gradient—prepare for a long wait in a large model.

This is gradient *ascent*, so initialize $\bar{\theta}$ accordingly. You can standardize them as z-scores, if you like. It is very useful for developing models, to keep them away from floating-point over/underflow. The partial derivative in (6) is $\dfrac{\partial O_i}{\partial \bar{\theta}}$, for the training pair i, i.e. without the outermost sums in (5). It is what `update-model` computes first, then (6).

An example workflow is provided for the '`corner`' project in the repository `ccglab-database/models`. The steps were:

1. Write `corner.ccg` and create `corner.ded`.
2. Copy `corner.ded` to `corner.ind`.
3. Initialize the parameters in `corner.ind`.
4. Design the training set `corner.supervision`, from which we generate `corner.sup`.
5. `(update-model "corner" 10 1.0 1.0 :load t)`. Notice that the training set size is not an argument to the function since it can be found from `corner.sup`.
6. `(show-training)`

7. `(save-training "corner-updated.ind")`. The trained model is now in the output file. You can go back to workflow in §3.2 to parse and rank the parses with the trained model.

There is also a version of gradient update based on extrapolation. It uses fixed number of iterations (currently four) and extrapolates from there. The limits it finds usually fall between the kind of numbers you get with 7–10 itereations. Quite handy during the development cycle. Just replace step 5 above with `(update-model-xp 1.0 1.0 :load t)` to run it. Replace step 6 with `(show-training-xp)` to see its results.

3.4 A compiler for type-raising

You can manually set your type-raising system. If you want completeness and soundness, you can use the compiler. For best results, remove the hand-written type-raising rules from your grammar first before you use the compiler.

The compiler first goes over all the argument-takers you have specified through their POS tags, then for every argument of every such element it generates the type-raised type. For example for the verbal category $(S\backslash NP)/NP$, it will first take the '/NP' and raise it to $(S\backslash NP)\backslash((S\backslash NP)/NP)$, because of its directionality. Then it will take the '\NP' and raise it to $S/(S\backslash NP)$ because of $S\backslash NP$ that is left once we are done with the '/NP'. The semantics is always the same: $\lambda p.pa'$ for some argument a'.

This much is deterministic per argument-taker. However, it creates lots of rules which are redundant, because many verbs share the same argument structure if they happen to be in the same morpholexical class (arity, case, agreement, etc.). A rule subsumer then goes over these automatically-generated rules and finds a much smaller set. You can save either the full set or the reduced set. The full set leads to unrealistic parse times in large grammars, as expected. In our experience, we have seen twenty-fold reduction by subsumption in empirically motivated grammars.

There is a demo in the repository `ccglab-database/models`. It shows how the same grammar can be manually and automatically type-raised. We obtained it as follows:

1. call `CCGlab`. Once in the tool, do:
2. `(tr "corner.ded" '(V))`, where the first argument is the grammar file—note specific suffix, because you can type-raise models too. The second argument is the list of POS in the grammar all of whose arguments are going to be type-raised. Source (.ccg) files must be converted to native format (.ded or .ind) before type-raising can be applied; use for e.g. `(load-grammar "corner" :make t)` for that.
3. `(savetr "corner-raised.ded")`, which saves the compiled and subsumed version by adding the rules to grammar.
4. `(load-grammar "corner-raised.ded")` makes the saved grammar current grammar. By default, if you parse with it you will get all derivations, with lowest types and type-raised types. If you specify which lowest types to avoid, assuming their type-raising is already built-in to the rules, you get only type-raised analyses of these types, for example:
5. `(type-raise-targets '(NP VP PP))` will instruct the parser not to use these lowest types in application from now on.

4 `CCGlab` representations

4.1 `.ccg` format for grammars

This file defines lexical items and unary rules in Steedman style, with the following amendments (stylistic ones are marked with '–', and the noteworthy ones with '⋆'):

- ';' is the terminator of a lexical specification. It is required. Each spec must start on a new line.
- '-->' is the unary rule marker.
- ⋆ Unary rules take an LF on the left in one fell swoop and do something with it on the right. That means *you have to have a lambda for the same placeholder first on the right to make it functional*. Here is an example (type raising by a unary rule):
  `(L1) np[agr=?x] : lf --> s/(s\np[agr=?x]) : \lf\p. p lf ;`
  Here is another one (verb raising to take adjuncts as arguments):

`(d-shift) vp : lf --> vp/(vp\vp) : \lf\q. q lf;`
Because there is no structured unification in `CCGlab`, it is a forced assumption to have just one thing on the left. Why not structured unification? Because we want to see how far combinators can be put to linguistic work and nothing else.

⋆ A part-of-speech tag comes before '`:=`'. Its value is up to you. (This is the only way CCG can tell whether e.g *(S\NP)/(S\NP)* can be a verb—say 'want'—rather than an adjunct, which is crucial for type-raising.) Type-raising compiler finds verbs through the verbal POS tags.

⋆ Special categories are pseudo-atomic. They start with `@`, e.g. `@X`. They must be lexically headed, and they must be across the board in a category. For example, `and := (@X\*@X)/*@X:..` is fine but `so := (@X/*@Y)/*(@X/*@Y):..` is not. `And := (S\*NP)/*@X:..` is bad too. They do not have extra features apart from what is inside the @Xs, which are imposed in term match. We therefore eliminate the need for structured unification.

⋆ If you use an `@X` category in a unary rule, it will be treated as an ordinary category.

– Non-variables in LF must be Lisp strings or they must be prefixally exclaimed to avoid evaluation by Lisp. Write *hit′* as `!hit`. It will be converted to the Lisp string constant `"hit"` by a Lisp reader macro.

⋆ Avoid names for LF variables that start with the special symbol '`&`'. Combinators use it. The only exception is the identity combinator, `&i`, which you may need in an LF when a functor subcategorizes for a type-raised argument rather than the argument itself.[6] The '`tokens`' script converts your `&i` to `(lam x x)`; see §4.4.

⋆ The double slash is implemented. In $X//Y$ Y and Y $X\backslash\backslash Y$, Y must be lexical to succeed. The modality of $\backslash\backslash$ and $//$ is always application only. The result X is assumed to be lexical too.

– Phonological strings that span more than one word must be double-quoted. You must use them as such in your parse string as well. The contents of a string are not capitalized by the Lisp reader whereas everything else is made case-insensitive, which must be kept in mind. We assume that there are no special symbols in the string. One or two punctuations (comma, period, exclamation or question mark) are allowed.

– Features need names. The basic categories in $S_{\text{fin}}/(S_{\text{fin}}\backslash NP_{3s})$ could be `s[type=fin]` and `np[agr=3s]`. Order of features is not important if you have more than one. They must be comma-separated.

– Capitalization is not important for names, unless they are in a string constant. This is also true of atomic categories, feature names, and values. `NP` is same as `np`. Lisp does it, not `CCGlab`.

– Comments start with '`%`'. The rest of the line is ignored.

⋆ Because `CCGlab` knows nothing about non-lambda internal abstractions such as the `x` in the logical form `\p\q.(!forall x)(!implies (p x)(q x))`, it cannot alpha-normalize them to rename `x` to something unique. This opens ways to accidental variable capture if some `CCGlab` combinator happens to abstract over the same variable, say $\lambda x.f(gx)$ for composition. We wouldn't want this x to replace the LF `x` above. If you intend to reduce `(q x)` to substitute for some lambda in `q` via `x`, you must abstract over it to ensure alpha-normalization; say `\p\q\x.(!forall x)(!implies (p x)(q x))`. Assuming that `x` is not a syntactic variable but `p,q` are, it will substitute the variable in `p,q` and keep the `(!forall x)`. If this is not what is intended, then use a naming convention for non-lambda variables which will not clash with `CCGlab` variables. Doubling the name as `xx` is my convention, e.g. `\p\q.(!forall xx)(!implies (p xx)(q xx)`. Prefixing or postfixing the variable with an underscore is also a safe convention. Prefixing it with `&` is not. Combinators use this convention.

The rules for lexical specifications are given in Table 1. They are used by the LALR parser, which converts the `.lisptokens` version of your textual grammar to `.ded`. Notice that lambdas can be grouped together, or written separately. Both $\lambda x\lambda y.hit'xy$ and $\lambda x.\lambda y.hit'xy$ are fine, and mean the same thing. As standard, CCG slashes and sequencing in the body of an LF are left-associative, and lambda binding is right-associative.

All LF bodies will be curried internally. For example, you can write '`\x\y\z.!give x y z`' in the `.ccg` file for convenience. It will be converted to '`\x\y\z.((!give x) y)z`'. Beta normalizer wants that.

[6] An example of this formal need can be given as follows. Suppose that we want to subcategorize for a type-raised NP, e.g. $f :=$ *S/(S/(S\NP))*: $\lambda p.f'(p(\lambda x.x))$. Type-raised arguments are universally $\lambda p.pa'$, so an argument could be e.g. $a :=$ *S/(S\NP)*: $\lambda p.pa'$. Application of f to a would be odd if we didn't have $\lambda x.x$ inside the LF of f, because f seems to be claiming—by its syntactic category—that its predicate-argument structure is $f'a'$, not $f'(\lambda p.pa')$. Neutralization of type-raising by a head is a forced move in theories like CCG because some languages seem to subcategorize for type-raised arguments syntactically, eg. Dyirbal, Turkish, and Inuit, where there is no determiner to value-raise an argument NP modified by a relative clause. Dyirbal's absolutive relative marker bears the category *-nu* := $((S/(S\backslash NP_{\text{abs}}))\backslash(S/(S\backslash NP_{\text{abs}})))\backslash(S\backslash NP_{\text{abs}})$, rather than *-nu* := $(N\backslash N)\backslash(S\backslash NP_{\text{abs}})$.

4.2 `.ded` and `.ind` format for Lisp-internal grammar

These files are lists of `(name value)` pair lists bound to the global variable called `*ccg-grammar*`. We chose such pairs because Lisp's usual choice for such simple mappings, association lists, are difficult to read.[7] Table 2 describes the format. For `CCGlab`, the order is not important in these lists—it does matter to Lisp—because the values are unique.

4.3 `.sup` and `.supervision` files for model building and training

These are the supervision files for training. The `.sup` is the native format, which is difficult to type because LFs must be curried. You can get it from the `.supervision` file which does that for you, which has the syntax of line-separated specs of the form

data : lf ;

where each `lf` has the same format as in `.ccg` file. Take a look at `make-supervision` function in Table 5. Native input to the trainer is a list of lists, where each member has first a list of whitespace-separated tokens in the data, and second, a list which is the expected LF for the data, in the format described next.

The repository `github.com/bozsahin/ccglab-database` has lots of examples to copy from.

4.4 The logical form's form

All internally translated LFs are curried. Your LFs in `.ccg` and `.supervision` files are curried automatically by the system. In fact, it is best in these source files if you leave currying the second type of lambda terms below to the system; just write `(((a b)c)d)` as `a b c d`. Lambda is not Lisp's '`lambda`'. Formally, the mapping from λ-calculus to `CCGlab`'s LFs is:[8]

x or c	$\rightarrow$	Lisp symbol or constant
$(e\ e)$	$\rightarrow$	`(e e)`
$\lambda x.e$	$\rightarrow$	`(lam x e)`

You can see from Table 1's non-terminal called '*lf*' that `CCGlab`'s LFs can have inner lambdas. In the source files `.ccg` and `.supervision`, lambda bindings can be grouped, anywhere in the LF, with one dot. Or they can be written one lambda at a time before each dot. Therefore any lambda-term can be LF, which means you have to watch out for non-termination. This is an extension from earlier "supercombinator" LFs so that training can be done on more complex LFs depending on task.

4.5 Parser's representations

All parser objects are represented as hash tables. COMMON LISP's hash tables are expressive devices. They do not support hash collision or chaining (we like them this way), and the keys can be anything, even a function or another hash table. We harness this property. Unlike association lists, access to a key's value does not involve search, e.g. formula (1) is computed without CKY search because beta-normalized LF is the key for that formula's table. When you have Catalan number of logical forms to check, you'll appreciate this property. (We employ alpha-equivalence of lambda-calculus in counting different LFs.)

There are five kinds of hash tables in `CCGlab`. Table 3 shows example snapshots during parsing. Their keys and structure are explained in Table 4.

4.6 How `CCGlab`'s term unification works

First I reiretate that there is no re-entrant unification in `CCGlab`. Term unification is used for category matching. Its details may be useful to developers, so here is how it works.

Assume the following projection rule for CCG, viz. substitution. There are two Y's to match in the rule, and two Z's.

$(X/Y)/Z\ \ Y/Z\ \rightarrow X/Z$

Each one of these matches, by function `cat-match`, creates a binding list of features in these categories; one on each category; so there are four of these binding lists (for first Y, second Y, first Z, second Z).

[7] In Lisp terminology an association list is `(name.value)`, rather than `(name value)`. It is easy to feel annoyed when there are too many of these '`.`' to look at. Since we do sequential search only during pre-parsing, efficiency is not the concern here; legibility is. `Rest` returns the value of an association pair, whereas `second` returns the value of a name-value pair.

[8] This layer is added so that you can see the inside of reduced lambdas. Lisp compilers implement and display closures differently; so there is no guarantee that native `lambda` is transparent. Normal-order evaluation of LFs is done at this layer.

These are required because some features may have variable value, which are always atomic; for example `Y[agr=3s,pers=?p]`, where 'pers' has a variable value. These features can be used in other parts of the input category, say by X and Z on the first input category, and by Z in the second. To obtain the result, these binding lists are reflected on other parts of the input *locally*, by function `realize-binds`.

For example, the following input to the rule produces the righthand side below, where 'pol' feature's value from the second element is not in the binding list, so continues to be a variable in the projection of the first element; whereas 'agr' feature of S is now '3s' because this is now the value of '?a' variable, and it is in the binding list of the NP in the first element. The NP in the result carries bindings of two input NPs because both elements use the NP (Z in the rule).

```
S[pol=?p,agr=?a]/VP[type=inf]/NP[case=nom,agr=?a]    VP[pol=pos]/NP[case=?c,agr=3s]
   --> S[pol=?p,agr=3s]/NP[case=nom,agr=3s]
```

Therefore, if a feature is not in the binding list, it will not be valued in the elements projected if it has a variable feature. The example (c) in the introduction shows what is at stake if we begin to project things that were not involved in the category match. The first elements of (a)'s 'f2' feature has nothing to do with second element's 'f2' feature, therefore both get locally projected, as in (b).

As a rule, bindings of the first element are reflected on the projected parts of the first element; bindings of second element are reflected on the projected parts of the second element; and, bindings of both elements are reflected on the projected common element.

This is the main reason for abundance of fresh hash tables at run-time, where results are kept as such because of speed. All these projected valuations can be unique to a particular rule use.

5 Top-level functions

The basic parsing functions were explained in §3. Others are quite useful for grammar development and testing. A more complete list is provided in Table 5. The code includes commentary to make use of them and others. All `CCGlab` functions are callable if you know what you're doing.

The names of all the features and hash table keys listed in Tables 2 through 4 are considered reserved names by the system. Using them as a basic category feature might result in unpredictable behavior. For example, if you use ARG as a feature, the system might crash because it expects such features to be hash-valued at parse time.

6 `CCGlab` in the large

Two aspects will interact to build a feasible model space: data/experiment space, and solution space. Solution space is reflected in the size of CKY tables. This size is controlled by the slash modalities (more liberal slashes, more analyses), beam search over solutions (on/off), and normal form parsing (on/off). A balancing act is usually called for, depending on your theoretical and practical goals. For data space we comment later on about using multiple processors. I start with the programming environment as it relates to both aspects.

Some comments on public COMMON LISPs for their suitability for large-scale development: So far SBCL has proven itself for this task. First, it compiles all Lisp code by default. More importantly, although CCL has a dynamically growing heap, its implementation of `sort` is *very* slow compared to SBCL. Neither SBCL nor CCL are known for their blizzard hash table speeds; and, their minimum hash table sizes are a bit annoying (because some of the hash tables we use have small number of keys), but at least they are transparent because they are type-complete and collision-free.

6.1 Beam

Beam search is possible to re-estimate the parameters in the inside-outside algorithm in a shorter amount of time. There is a switch to control it. As you probably know, the number of CCG derivations can grow up to Catalan numbers on input size if left unconstrained.

Sort is essential to the beam, which is set to a default, which you can change. The formula is n^b where $0 \leq b \leq 1$. The n here is the number of parses of the current input. For example $b = .9$ eliminates very few analyses if n is small, large amounts when it's large. Before you play with the beam system (b value and its on/off switch), I'd suggest you experiment with learning parameters N, n, α_0, c in (6) and the extrapolator, if you want to avoid playing with N.

6.2 Heap and garbage collection

One easy way to get the best of both worlds of fast sort and big heap is to reset the `CCGLAB_LISP` variable after install. If you do the following before running `CCGlab`, it will set the heap to 6GB in SBCL.

```
export CCGLAB_LISP='/usr/bin/sbcl --dynamic-space-size 6000'
```

CCL won't give you trouble in long training sessions; it won't go out of memory. You have to check whether it is doing useful work rather than garbage-collecting or thrashing. Some control over SBCL is provided. SBCL gives you two options if you use too much memory: (i) recompile SBCL or (ii) increase maximum memory maps with e.g. `'sudo echo 262144 > /proc/sys/vm/max_map_count'`. The second option seems to work well without recompile.[9] This is the number of maps, not memory size. CLISP is not fully ANSI. Non-ANSI Common Lisps are not compatible with `CCGlab`.

One way to avoid excessive garbage collection is increasing its cycle limit, which is 51MB by default in SBCL. As usual, making it too much may be counterproductive.

6.3 Normal Form Parsing

You can use Normal Form (NF) parsing as an option to reduce the number of LF-equivalent parses substantially. We use the method by Eisner (1996) which eliminates them at its syntactic source, rather than generate-and-test the LFs. This means that non-normal parses won't make it into the CKY table to effect the lexical counts, which must be kept in mind in models and training. NF parse option is available both in deduction and induction mode. There is one switch to control is behavior, listed in Table 5.

NF parse is not recommended if you are exploring surface constituency in all its aspects, especially phonology; but, it is very practical for modeling and parameter re-estimation. It can be used in conjunction with beam search to reduce the calculations of the inside-outside algorithm even more.

Unary rules and lexicalized type-raising do not combine, therefore any "redundancy" caused by them in the derivations is not eliminated by NF tags; see Eisner's paper for explanation. Moreover, `CCGlab`'s unary rules are not necessarily lexical; they can apply to the result of a derivation. Because they change the input syntactic type and/or LF, we start with a fresh lexical tag (called 'OT' in the paper) for the output of the rule.

6.4 Training on large datasets

Parsing after training, either for ranking or for deduction, is very fast. Type-raising can change that if there are too many rules to apply at every position (how many is too many? something around 1,000 seems somewhat noticeable, half of that hardly noticeable, more than 5,000 needs coffee break.)

In any case you need all the help you can get in training. One is to use `nohup`, which runs `CCGlab` code immune to hang-ups. The bash script `trainer-sbcl` provided in the repository addresses these concerns. It is an 8-argument monster to work completely offline and standalone; please study the script in detail. It assumes that constraints which are not within control of SBCL are already handled, such as `/proc/sys/vm/max_map_count` above.

A bash script named `train-sbcl-multihtread` is added to the repository to run various experiments simultaneously if you have multi-core support. It reads all the arguments to `trainer-sbcl` from a file, each experiment fully specified on a separate line, requests as many processors as experiments, and calls them using `xargs` command of linux. It does not make use of SBCL's multi-threading; all of this is done on command line. Each experiment in multi-thread case is individually `nohupped`.

6.5 Many Lisps

You can work with many Lisps for `CCGlab` at the same time by setting the shell variable `CCGLAB_LISP` to different Lisp binaries in a shell. The default is set by you during install. The system will detect your Lisp from the basename of its binary. If it is not SBCL or CCL or public AllegroCL, it is set to SBCL by default, which means it will call `run-program` with SBCL conventions when running CCGlab in your Lisp. Have a look at how `run-program` API is used by sbcl and ccl in the code. You may want to add another case suited to your Lisp, and set the `*lispsys*` variable to your Lisp. This should take care of everything.

[9] If you get permission errors even with sudo try this: `'echo 262144 | sudo tee /proc/sys/vm/max_map_count'`.

6.6 Hash table growth

The hash tables for the CKY parser can grow very big. To work with very long sentences without rehashing all the time, change the variable `*hash-data-size*` to a bigger value in `ccg.lisp` source. Tables of these sizes (two tables: one for CKY parses, one for different LFs in argmax calculation) are created once, during the first load, and cleared before every parse. The default size is 65,536.

There is a path language for nested hashtables, i.e. hashtables which take hashtables as values of features. Rather than cascaded `gethash` calls, you can use the `machash` macro. See Table 5. By design, CKY hash tables are doubled in size when full. The logic here is that, if the sentence is long enough to overflow a big hash table, chances are that the table is going to grow just as fast, and we don't want to keep rehashing in long training sessions.

6.7 Floating-point overflow and underflow

Because of exponentiation in formula (2) you have to watch out for floating point overflow. If the parameters become too large, which may happen if you run many training sessions on the same grammar, you can z-score the entire grammar as explained in Table 5. In a z-scored grammar all parameters are fraction of the standard deviation distant from the mean. Z-scoring is better than normalization because it maintains the data distribution's properties like mean and variance. Don't forget to save that updated model.

6.8 `CCGlab` code and grammar repository management

`Github.com/bozsahin/ccglab-database` is for grammars and models written in `CCGlab`. Older repositories are not maintained anymore.

`CCGlab` is GPL licensed public software; you can modify and use it as part of a software system, but you cannot copyright `CCGlab`; you must pass on the GPL license as is.

Acknowledgments

Translations from `.ccg` to `.ded` formats and from `.supervision` to `.sup` are made possible by Mark Johnson's LALR parser. Translation of the LFs of paper-style representations in `.ccg` to a self-contained lambda-calculus processor is based on Alessandro Cimatti's abstract data structure implementation. Thanks to both gentlemen, and to Luke Zettlemoyer for help with the PCCG paper. Without the lambda translation, you'd be at the mercy of a particular Lisp implementation of closures to get the formula (1) right, or to verify derived LFs. Adnan Öztürel convinced me think again about normal form parsing, which I have resisted to implement in `CCGlab` for three years although I had implemented it before. This time I made it optional. I thank Lisp community, stackoverflow and stackexchange for answering my questions before I ask them. Oğuzhan Demir helped me implement the type-raising compiler.

I am grateful to my university at Ankara, ODTÜ; to Cognitive Science Department of the Informatics Institute, for allowing me to work on this project; to Turkish TÜBİTAK for a sabbatical grant (number 1059B191500737), which provided the financial support for a research of which this work is a part; and to the ANAGRAMA group of the CLUL lab of University of Lisbon, in particular to Amália Mendes, for hosting me in a friendly and relaxed academic environment, all of which made `CCGlab` possible.

References

Baldridge, Jason. 2002. *Lexically Specified Derivational Control in Combinatory Categorial Grammar*. Doctoral Dissertation, University of Edinburgh.

Bozşahin, Cem. 2012. *Combinatory Linguistics*. Berlin: De Gruyter Mouton.

Bozşahin, Cem, and Arzu Burcu Güven. 2018. Paracompositionality, MWEs, and argument substitution. In *Proc. of 23rd Formal Grammar Conference*. eds.Annie Foret, Greg Kobele, and Sylvain Pogodalla, 16–36. Berlin: Springer.

Clark, Stephen, and James R. Curran. 2003. Log-linear models for wide-coverage CCG parsing. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, 97–104. Sapporo, Japan.

Eisner, Jason. 1996. Efficient normal-form parsing for Combinatory Categorial Grammar. In *Proceedings*

of the 34th Annual Meeting of the ACL, 79–86.

Graham, Paul. 1994. *On Lisp*. Englewood Cliffs, NJ: Prentice Hall.

Pareschi, Remo, and Mark Steedman. 1987. A lazy way to chart-parse with categorial grammars. In *Proceedings of the 25th Annual Meeting of the ACL*, 81–88.

Steedman, Mark. 1996. *Surface Structure and Interpretation*. Cambridge, MA: MIT Press.

Steedman, Mark. 2000. *The Syntactic Process*. Cambridge, MA: MIT Press.

Steedman, Mark. 2012. *Taking Scope*. Cambridge, MA: MIT Press.

Steedman, Mark, and Jason Baldridge. 2011. Combinatory Categorial Grammar. In *Non-transformational syntax*. eds.R. Borsley and Kersti Börjars, 181–224. Oxford: Blackwell.

Zettlemoyer, Luke, and Michael Collins. 2005. Learning to map sentences to logical form: Structured classification with probabilistic categorial grammars. In *Proc. of the 21st Conf. on Uncertainty in Artificial Intelligence*. Edinburgh.

Table 1: `.ccg` format specification. (.supervision has same syntax for lf).

lex conventions:	T: Any token `!`A: LF constant "A"	`@`A : special category Default modality: `.`	`?`A: value variable `%` : start of comment

start	→	*start lex* `;` | *lex* `;`	*eqns*	→	*eqns* , *eqn* | *eqn*
lex	→	*T mtag* := *cat* | *lrule*	*eqn*	→	$T = T$
lrule	→	(*T*) *cat* `-->` *cat*	*dir*	→	`\` | `/`
mtag	→	*T*	*mod*	→	`.` | `^` | `*` | `+`
cat	→	*sync* : *lf*	*lf*	→	*bodys* | *lterm*
			lterm	→	`\` *T* `{.}` *lbody*
sync	→	*basic* | *parentd* | *sync slash syn*	*lbody*	→	*lterm* | *bodys*
syn	→	*basic* | *parentd*	*bodys*	→	*bodys body* | *body*
basic	→	*T feats*	*body*	→	(*bodys*)
parentd	→	(*sync*)	*body*	→	*T*
slash	→	*dir mod* | *dir*| *ddir*	*ddir*	→	`\\` | `//`
feats	→	[*eqns*] | ε	*body*	→	(*lterm*)

Table 2: `.ded` and `.ind` feature names in name-value pairs lists.

BCAT: basic category
DIR: directionality of a complex category
FEATS: basic category features as name-value pairs list
MODAL: slash modality (ALL CROSS HARMONIC STAR)

lex item features		lex rule features	
PHON	phonological form	INSYN	input category
MORPH	pos tag	INSEM	input lf
SYN	syntactic type	OUTSYN	output category
SEM	logical form	OUTSEM	output lf
PARAM	parameter value	PARAM	parameter value

Table 3: (a): CKY representation for: ‘`hits v := (s\+ np[agr=3s])/np : \x\y. !hit x y;`’ during `(ccg-deduce '(hits))`. A ‘#’ means the value is a hash table. It is not part of the name. (b): ARGMAX representation for: ‘`hits v := (s\+ np[agr=3s])/np : \x\y. !hit x y;`’ after `(ccg-deduce '(hits))`. LF shown is the key. It is beta-normalized, given this input to the parser.

(a)

(1 1 1)	LEFT	(1 1 1)				
	RIGHT	(1 1 1)				
	SOLUTION#	PHON	HITS			
		MORPH	V			
		SYN#	RESULT#	RESULT#	BCAT	S
				ARG#	BCAT	NP
					AGR	3S
				DIR	BS	
				MODAL	CROSS	
			ARG#	BCAT	NP	
			DIR	FS		
			MODAL	ALL		
		SEM	(LAM X (LAM Y (("HIT" X)Y)))			
		INDEX	LEX			
		PARAM	1.0			
		TAG	OT			
		KEY	unique key of *hits* in the lexicon			
	LEX	T				

(b)

(LAM X (LAM Y (("HIT" X)Y)))	(1.0 ((1 1 1)))

Table 4: Keys and value types for `CCGlab` hash tables, and the global variables.

Table type	Key	Value	Description
CKY	(I J K)	(LEFT RIGHT SOLUTION LEX)	LEFT/RIGHT's value: component cells of the solution in (I J K).
			I: length of derivation. J: starting position. K: analysis number.
			SOLUTION's value: LEX or CKY-ENTRY table.
			LEX value: true or false.
CKY-ENTRY	SYN	table of type BCAT or CCAT	Syntactic category hash table
	SEM	Logical form	See §4.4 and Table 1
	INDEX	Rule index	`LEX` for lex entries, rule name for lex rules, `>Bx` for $>\mathbf{B}_{\times}$, etc.
	PARAM	real number	Cumulative for partial results; lexical for lex entries and unary rules
	TAG	FC, BC or OT	Normal Form tag of the constituent
	LEX	T or NIL	Whether the item is marked lexical (by lexicon or by double slash)
BCAT	BCAT	symbol or constant	Name of the basic category: either constant or '`@`' variable
	`<name>`	symbol or constant	atomic value of the feature `<name>`: either constant or '?' variable
CCAT	RESULT	table of type BCAT or CCAT	Result of a complex category
	ARG	table of type BCAT or CCAT	Argument of a complex category
	DIR	one of FS BS	Directionality, corresponding to '/' and '\' respectively
	MODAL	`name`	Slash modality. `name`: one of ALL CROSS HARMONIC STAR
LEX	SYN	table of type BCAT or CCAT	Syntactic category hash table
	SEM	Logical form	See §4.4 and Table 1
	INDEX	Rule index	`LEX`
	PARAM	real number	Parameter's value for that lex entry (set to unity in `.ded`)
	PHON	symbol or string	Phonological string for the lex entry
	MORPH	symbol	Morphological category (part of speech) of the lex entry
LRULE	INSYN	table of type BCAT or CCAT	Input syntactic category
	INSEM	Logical form	Input LF as one object. It is the only input to OUTSEM.
	OUTSYN	table of type BCAT or CCAT	Output syntactic category
	OUTSEM	Logical form	Output LF as a function of INSEM. See §4.4 and Table 1
	INDEX	Rule index	Rule name (given by you)
	PARAM	real number	parameter value for the rule (set to unity in `.ded`)
ARGMAX	`<lf>`	`(number cells)`	key: beta-normalized LF of the result.
			`number`: weight-sum, `cells`: list of cells for the sum

Global variable	Type or value; call `(globals)` to see the current list; `(onoff)` for switches
`*ccg-grammar*`	List of list of name-value pairs, where current grammar is internally represented
`*loaded-grammar*`	The name of the currently loaded project file (`.ded` or `.ind`)
`*cky-hashtable*`	CKY table
`*cky-lf-hashtable*`	ARGMAX table for computing formula (1)
`*cky-lf*`	the maximum of the numerator of formula (2) without exponentiation, and its LF
`*cky-lf-hashtable-sum*`	the value of the denominator of formula (2) without exponentiation
`*cky-argmax-lf-max*`	CKY index of the argmax result's highest weight-summed derivation
`*cky-argmax-lf*`	List of CKY indices which have the same LF as argmax result
`*cky-max*`	CKY index of the highest weight-summed derivation for any LF
`*lex-rules-table*`	List of LRULE tables
`*training-hashtable*`	Keeps progress of parameter updates.
	Key is the item's key. Value is the list of original and current value.
`*training-non0-hashtable*`	Keeps non-zero counts of features for the current parses. Key is the
	supervision pair index (base 1), and value is a list of item keys with non-zero counts
	in all current parses of the pair. Calculated once before the loop in formula (6).
`*supervision-pairs-list*`	List of sentence-LF pairs, in this order.
`grammar`	LALR's preset grammar to convert your textual specs to Lisp
`lexicon`	LALR parser's token values (i.e. *its* lexicon, not yours)
`lexforms`	LALR parser's token types (ID is any name, others are special symbols)
`*endmarker*`	LALR parser's end-of-input designator, currently set to `$` in two places
	(by the LALR parser)
`*beam-exp*`	The exponent of beam. Default is .9, for $n^{.9}$. Lower than $4/5$ is not recommended.
`*oovp*`	Set it to nil (default) if you want the tool to complain about OOV; t otherwise.
`*lispsys*`	The Lisp system. Currently recognized values are SBCL, CCL and ALISP.

Table 5: Some useful functions of CCGlab.

tokens <pname>	From <pname>.ccg file generates <pname>.lisptokens at command line. Also called from within CCGlab. Self-contained bash script which runs sed.
suptokens <pname>	From <pname>.supervision file generates <pname>.sup at command line. Also called from within CCGlab. Self-contained bash script which runs sed.
(load-grammar <pname> :make <m>)	Loads the <pname>.ded file, if <m> is nil or if :make key argument is not supplied. Generates it from the .lisptokens file if not nil. The old keyword :maker is still available but now obsolete, and works as before if you had used it. Lisp system is now automatically detected.
(load-model <pname>)	Loads the <pname>.ind file.
(cky-pprint)	Pretty-prints the CKY table with row, column, solution indices so that you can see the values of syntactic features, LFs, partial results, etc. Strings print unquoted.
(ccg-deduce <list>)	Parses the input words in <list>. Returns true/false.
(ccg-induce <list>)	First parses then ranks the derivations for the input words in <list>. Returns true/false.
(cky-show-deduction <cat>)	Shows all derivations in paper style. Final LF is displayed normal-evaluated. <cat> is optional. If a basic category is supplied in <cat>, only solutions with that category are reported (but all are still computed). Without <cat> or if <cat> is nil all solutions are printed.
(cky-show-induction)	Shows 3 derivations mentioned in §3.2, in paper style.
(cky-show-lf-eqv)	Applies normal-order and applicative-order evaluation to the results, and reports the differences. For this reason, the LFs of intermediate derivations are not beta-normalized during parsing. We know that they look ugly, but you can in fact cut/paste them and simplify; see cky-show-normal-forms.
(cky-show-normal-forms <r> <c>)	Shows normal forms of all results in CKY table entry row <r> column <c>.
(cky-sem <cell>)	Returns the LF in CKY table entry row <r> column <c> result <z>. An argument of type <cell> is the list (<r> <c> <z>). Use cky-pprint beforehand for help.
(cky-reveal-cell <cell>)	Prints the deduction sequence ending in CKY table cell <cell>; see cky-sem for <cell>. Final LF is normalized for mortals; the table is kept intact.
(cky-pprint-probs <cell>)	Prints the induction sequence ending in CKY <cell>. See cky-sem for <cell> argument.
(alpha-equivalent <e1> <e2>)	Returns true iff two input LFs are structurally equivalent, i.e. if they have the same structure modulo bound-variable names.
(status <tag>)	Returns a summary of the current state (loaded grammar, its size, CKY info etc.). If no <tag> is supplied it just prints out highest ranked solution. If optional <tag> is supplied all solutions are returned in a ranked list (highest first). Useful for re-ranking and selection of subset of solutions.
(reset-globals)	Resets the current state of CCGlab grammar to almost tabula rasa.
(count-local-structure <cell>)	Computes the final weight-sum $\vec{f} \cdot \vec{\theta}$ at the result cell <cell>, from its constituent cells; cf. formula (2). If you change the formula, argmax (1) will be computed correctly if you save the result in result <cell>.
(plugin-count-more-substructure <cell>)	A plug-in to give you access to all derivational history to extract more features from result cell <cell>. NB where it is called. Does nothing by default.
(update-model <pname> <N> <alpha0> <c> :load <l> :verbose <v> :debug <d>)	Updates a model's parameters after loading the supervision set <pname>.sup. See §3.3 for meanings of arguments. If <l> is not nil, the model <pname>.ind is first loaded. If <v> is not nil, it displays progress of stochastic gradient ascent through iterations, for every lexical item. If <d> is not nil, it displays stepwise update of the derivative in algorithm (6).
(update-model-xp <pname> <alpha0> <c> :load <l> :verbose <v> :debug <d>)	Updates a model's parameters using extrapolation after loading the supervision set <pname>.sup. See §3.3 for meanings of arguments. If <l> is not nil, the model <pname>.ind is first loaded. If <v> is not nil, it displays progress of stochastic gradient ascent through iterations, for every lexical item. If <d> is not nil, it displays stepwise update of the derivative in algorithm (6).
(show-training)	Shows parameter values for every lexical item before and after training. the one with -xp suffix shows extrapolation.
(save-training <file>)	Saves the currently trained model with the new parameter values to file.
(make-supervision <pname> :maker <m>)	Makes a lisp-ready pname.sup file from pname.supervision and pname.suptokens. maker keyword is legacy and should be avoided. See §4.3. :maker is now obsolete but kept as legacy. The Lisp system is automatically detected.
(save-grammar <file>)	Saves the currently loaded grammar to file.
(show-config)	Shows switch values and rule switches and beam value.
(z-score-grammar)	Z-scores the parameter values of the currently loaded grammar.
(lf <sw>)	If aargument is t, all LFs will be shown in display of a derivation. The last one is simplified for readability. Only the final LF of a derivation is shown if switch is nil or off.
(beam <sw>)	Turns the beam on or off, with these as keyvalues.
(beam-value)	Current values of *beamp* and *beam-exp*. Use setf to change the exponent before training.
(machash f1 ... fn ht)	Retrieves feature f1 from .. fn of the hashtable ht. Features f2..fn must be hash-valued.
(ab)	List of shortcuts for top level functions.
(oov <sw>)	Turns out of vocabulary treatment on or off, with these keyvalues.
(basic-ccg <sw>)	values for <sw> are on/off/t/nil. When on or t, only application, composition, substitution and their powers are used. If <sw> is off or nil, all rules are used. It is on by default.
(type-raise <sw>)	Turns type-raised parsing on or off, with these key values.
(type-raise-targets <cats>)	Eliminates lowest types in cats in application from now on.
(tr <gr> <pos>)	Compiles and subsumes all type raising in grammar gr with argument takers bearing pos.
(savetr <fn>)	Saves the compiled and subsumed grammar