THEBENCH Guide

Cem Bozşahin Ankara, Datça, Şile 2021–23

Version: 1.0 December 31, 2022 coloured text means changes from previous release

Home: `github.com/bozsahin/thebench` (all repos cited have the same URL prefix)

To download, do: `git clone https://github.com/bozsahin/thebench`

1 Introduction

THEBENCH is a tool to study monadic structures in natural languages, for writing grammars, exploring analyses, and training grammars for modeling. The monadic structures are binary combinations of elements of grammar using semantics of composition only, hence the name.

The theory regarding the limits of elementary (synthetic) vocabulary and analytic structure allowed in such grammars is described in Bozşahin (2023). In summary, it is a Polish School style categorial grammar that uses composition only, that is, application is also turned into composition. Unlike generative grammars, there are no top-down universals such as EPP, MLC or universal lexical categories such as N, V, A, P, or universal argument types such as those in X-bar theory. All of that is predicted from the ground up. Segments arise because we analyze by categorized reference, we do not assume analysis arises from (or rules 'generate') Harris-style distributional segments.

It is meant to study configurationality and referentiality in one system. The configurational approach to surface structure is adopted from Montague (1973), Steedman (2020), and the referential approach to constructing elements is adopted from Sapir (1924/1949), Swadesh (1938), Montague (1973), Schmerling (2018), with the following major differences: Formally, unlike Steedman (2000), all analytic structures handle one dependency at a time (i.e. there is no **S**-style analytic structure), and unlike Montague grammar, predicate-argument structures are not post-analytic, and syntactic categories of second-order case functions require a linguistic rather than logical theory. Also, distinct function-argument and argument-function surface configurations are the causes of categorial-structural asymmetries, an idea which goes back to Schönfinkel (1920/1924). Empirically, morphology and syntax do not compete for surface bracketing, morphology is not confined to the 'lexicon' or to some other component, or to leaves of a tree, and referential differences in all kinds of arguments cause categorial differences including those in idioms and phrasal verbs (Bozşahin, 2022). Typologically, any language-particular difference in elementary (i.e. synthetic) vocabulary can make its way into invariant analytic structure. It is a very verb-centric bottom-up view of grammar.

Technically, the description and interface functions of THEBENCH are written in Python. Processor functions are written in Common Lisp. The command interface has its own syntax combining the two aspects, which you can use online and offline; no need to know either language to use it. The installer takes care of the requirements to work with all popular computer platforms (well, almost all).

2 Basics

Slash modalities are implemented (see Baldridge 2002, Steedman and Baldridge 2011). The examples below show raw input to THEBENCH.[1] The first three are lexical items, and the last one is a unary rule.

[1] The mapping of modalities to those in papers is (. , .), ($\hat{}$,$\diamond$), ($*$,$\star$), (+,$\times$), from THEBENCH notation to CCG notation.

```
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 ;
```

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.

Structural unification plays no role in THEBENCH because we want to see how far grammatical inference can be carried out by combinators. Only atomic categories can carry features. Features are checked for atomic category consistency only, and no feature is passed non-locally. All features are simple, feature variables (prefixed by `?`) are for values only, they are local to a category, and they are atomic-valued. For example, if we have the category sequence (a) below, we get (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]`

Meta-categories such as *(X\X)/X*, written in THEBENCH as `(@X\@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 that you see in CCG papers have not been incorporated. They would require structured unification and jeopardize procedural neutralism. What that means for THEBENCH is that we can write `(@X\@X)/@X` kind of category, as above, or say `@Y/@Y`, but we cannot write `@X/(@X/NP)` or `@X/(@X/@Y)`, because matching `(@X/NP)` or `(@X/@Y)` to yield `@X` would require structured unification.

No basic category is built-in to THEBENCH. Singleton categories are supported; 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 THEBENCH notation:

(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, 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 (there is no such thing in THEBENCH). The underscore is just an ad hoc reminder that the first thing that follows is not an argument of the predicate. It is included here to show that LF set-up is all up to you.

Below is the output for the Latin phrase for *Balbus is building the wall*. Currently there is no separate morphology component, so we have morphological case syntactically combining in the example. The first column is the rule index. The rest of the line is about the current step: a lexical assumption, result of a unary rule, or one step of combination, in which the combining rule is shown. This is the output of the parser in response to `(p '(balb us mur um aedificat))`.

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

Final LF, normal-order evaluated:

    ((BUILD WALL) BALB) =
    (BUILD WALL BALB)
```

The parse ranking 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. 17 is the total here. The example is output in response to `(rank '(balb us mur um aedificat))`.

```
Most likely LF for the input: (BALB US MUR UM AEDIFICAT)

  ((BUILD WALL) BALB) =
  (BUILD WALL BALB)

Cumulative weight:  17.0

Most probable derivation for it: (5 1 1)
--------------------------------
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)
LEX      1.0  (MUR) := N
        : WALL
LEX      1.0  (UM) := ((S\NP)/((S\NP)\NP))\N
        : (LAM X (LAM P (P X)))
<        2.0  (MUR)(UM) := (S\NP)/((S\NP)\NP)
        : ((LAM X (LAM P (P X))) WALL)
LEX      1.0  (AEDIFICAT) := (S\NP)\NP
        : (LAM X (LAM Y ((BUILD X) Y)))
>        3.0  (MUR UM)(AEDIFICAT) := S\NP
        : (((LAM X (LAM P (P X))) WALL) (LAM X (LAM Y ((BUILD X) Y))))
>       17.0  (BALB US)(MUR UM AEDIFICAT) := S
        : (((LAM X (LAM P (P X))) BALB)
```

The parsing component computes all the constituents and their LFs which are derivable from lexical assumptions. The ranking 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 for the string (not shown above for brevity). The parsing component first generates them with unity weights (1.0) when the grammar is first compiled from `.ccg` textual form to `.ccg.lisp` compiled form. Your ranking (i.e. training) component can build on that to modify lexical parameters after parameter estimation.[2]

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

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

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 and parameters in PCCG (*ibid.*);[3] 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. If you want to count other substructures in D or L, as in Clark and Curran (2003), you need to write some code.[4]

THEBENCH is 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; (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.[5] 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.[6]

3 THEBENCH file types

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

By a grammar we mean a set of CCG assumptions like the one in §2 which you want to subject to CCG's universal combinatorics. By a model we mean a version of the grammar which you've subjected

[2] There is only one compiled grammar file type, with `.ccg.lisp` extension, because of common representation.

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

[4] A plug-in is provided, called `plugin-count-more-substructure`. That was one motivation for giving detailed specs. There is a growing body of literature on the topic, starting with Clark and Curran.

[5] This layer is post-parsing in the sense that although parsing builds an LF at every step, it does not reduce it till the very end. Therefore unevaluated LFs of THEBENCH are available for exploration.

[6] 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.

to empirical training and parameter estimation, results of which you want to subject to CCG's universal combinatorics. They have the same format.

A project say with name *P* will probably consist of the following files (we explain their format in §5):

- *P*.ccg : The CCG grammar as text file. You write this one.
- *P*.ccg.lisp : The Lisp translation of the *P*.ccg file. This file is generated when you call mlg function. This is the file used when you parse and rank expressions. It is the file whose parameters can be subjected to training. Unless you call mlg again or overwrite this file yourself, this file is untouched by the system. The trainer keeps the trained grammar in memory until you save the results by calling save-training.[7]
- *P*.sup : The supervision file in native format (optional). This is the training data for PCCG parameter estimation. It consists of expression-LF pairs in a list. Syntax and derivations are hidden variables in PCCG, and training is done solely on expression-LF pairs. These LFs are binarized in the native format. The lambda layer wants that. Here is an example native .sup file:

```
(
((JOHN PERSUADED MARY TO BUY THE CAR) ((((PERSUADE" (("BUY" ("DEF" "CAR")) "MARY")) "MARY") "JOHN")))
)
```

- *P*.supervision : A simpler way to create the .sup file (optional). Content is semicolon-separated sequence of training pairs such as below.

  words:lf;

  where words are space-separated, and lf is as parenthesis-free as it can be. Such LFs are much easier to write (and less error prone), unless you are a parenthesis monster. THEBENCH will do the conversion to curried LFs, i.e. put lots of parentheses. The trainer works on native LFs like those in .sup.

Here is the .supervision file you can write to get the .sup file above by calling make-supervision

```
JOHN PERSUADED MARY TO BUY THE CAR : !PERSUADE (!BUY (!DEF !CAR) !MARY) !MARY !JOHN ;
```

Only these files are checked by THEBENCH for their existence and well-formedness. If you are a grammarian, all you need is .ccg to create .ccg.lisp. If you are a modeler, you will also need .sup.

4 Workflows

There are basically two workflows. Model development cannot be fully worflowed, so it's harder.[8] In the cases below, the Lisp parts must be done after you launch THEBENCH. The README.md file in the repository explains how.

4.1 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:

[7] Legacy file types such as .ded and .ind are deprecated. If you have them, and spent a lot of effort developing, just rename them with .ccg.lisp extension for current use.

[8] Grammar development cannot be fully workflowed either, but that's another story. Beware of claims to the contrary. This is good for business, for us grammarians.

1. In your working directory, write your grammar and save it as plain text e.g. `example.ccg`
2. In THEBENCH do `(make-and-load-grammar "example")`
   This means Lisp tokens will be generated by `tokens` bash script from within Lisp.
   This step prepares the `example.ccg.lisp` file and loads it onto Lisp.
   If you update your `example.ccg` source you must re-make the grammar. Short form is `mlg`.
3. Do: `(p '(a sequence of words))` to run the parser.
   Replace `p` function with `rank` to run the parse-ranker.
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
   `(ders <bcat>)`, where `<bcat>` is a basic category. Calculations will be done
   with all available results, but only derivations onto `<bcat>` will be shown.
   Run `(cky-pprint)` to see the CKY table. It prints good detail for debugging.

If there are errors in your grammar file, step 2 will fail to make/load the grammar, and you'd need to go back to editing `example.ccg` to fix it. Partially-generated `example.ccg.lisp` will tell you where the *first error* was found, so multiple errors are caught one by one (sorry).

4.2 Model testing

At some point, you may turn to modeling. It usually means taking the `example.ccg.lisp` file and adjusting its parameters (because they were set to default in file creation) by parameter estimation, and playing with its categories. Keep in mind that parameters are weights, not probabilities.

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). THEBENCH helps with the basics (by having weights associated with every lexical element), to compute (1–2) and other formulae such as (3–6), to be explained soon.

In the end, you create an `example.ccg.lisp` 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 THEBENCH to change training. The model testing workflow is:

1. In THEBENCH, do: `(load-grammar "example")`
   This will load the grammar in `example.ccg.lisp` along with its parameter set.
2. Do: `(rank '(a sequence of words))` to parse then rank the parses.
3. Do: `(probs)` to see three results for the input: (i) most likely LF for it, (ii) its most likely derivation, and (iii) most likely derivation for any LF for the input. You can also do `(cky-pprint)` to see the CKY table. If you like, do `(ders)`, for display without ranking. 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 (see Table **??**), 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 THEBENCH for model debugging; in testing wide-coverage parsers might opt to switch it on (no promises).

4.3 Model development: parameter estimation

Parameters of a `.ccg.lisp` 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 to 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 (default) 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).

For example we can use the following workflow for model update:

1. `(make-supervision "example"))`
   The file `example.sup` will be generated from `example.supervision`.
2. `(lg "example")`
   `example.ccg.lisp` will be loaded. `lg` is same as `load-grammar`.
3. `(um "example" 10 1.0 1.0)` `um` is same as `update-model`
   updates the currently loaded grammar. The learning parameters are provided.
4. `(show-training)` shows training.
5. `(save-training "new-model")` saves training in `new-model.ccg.lisp`.
6. `(lg "new-model")`
   loads the file `new-model.ccg.lisp` just saved. Now you can parse and rank with it.

There is also a version of gradient update based on extrapolation. To use it, call `update-model-xp` and `show-training-xp` at steps 3 and 4, respectively. Eliminate iteration count (10 in step 3) from learning parameters. It runs fixed number of iterations (currently four) and extrapolates from there. The limits it finds usually fall between the kind of numbers you get with 6–10 iterations, which is quite handy for the development cycle.

4.4 A compiler for type-raising

You can manually set up your type-raising system. You can also use the compiler for that. For best results, remove the hand-written type-raising rules from your grammar first before using the compiler.

The compiler goes over all the argument-takers you have specified by identifying them from their POS tags. For each one, for its outermost argument, it generates the type-raised type. For example for the verbal category *(S\NP)/NP*, it will take the '*/NP*' and raise it to *(S\NP)\((S\NP)/NP)*, because of its directionality. For *S\NP*, it will take the '*\NP*' and generate *S/(S\NP)*. The semantics is always the same: $\lambda p.pa'$ for some argument a'.

It creates lots of rules, some of which may be redundant, because many verbs share the same argument structure if they happen to be in the same morpholexical class (valency, case, agreement, etc.). A rule subsumer then goes over these automatically-generated rules and finds a smaller set.

An example workflow is:

1. `(mlg "example")`
   The file `example.ccg.lisp` will be generated from `example.ccg`.
2. `(tr "example" '(iv tv))`
   loads `example.ccg.lisp` and applies type-raising to all functions with POS tags `iv` and `tv`.
3. `(mergesave-tr "example-tr")`
   Saves the compiled and subsumed type-raising rules along with current grammar.
4. `(lg "example-tr")` loads the file just saved, which is `example-tr.ccg.lisp`.

Now you can parse/rank with type-raising unary rules in the grammar. Step 2 creates a log of warnings. Warnings are about raising a non-basic type or having no argument to raise.

5 THEBENCH representations

The first subsection of this section is for everyone. Others may be relevant to developers and modelers.

5.1 `.ccg` file format

Here is an example textual input to THEBENCH, typed by me.

```
% a mini CCG grammar in CCGlab

mur n := N : !wall;
um aff := ((s\np[case=nom])/((s\np[case=nom])\np[case=acc]))\n : \x\p.p x;
balb n := N : !balb;
us aff := (s/(s\np[case=nom]))\n : \x\p.p x;   % manually value-raising the input N
aedificat v := s\np[case=nom]\np[case=acc] : \x\y.!build x y;
```

A file of this type defines lexical items and unary rules in Steedman style, with the following amendments (stylistic ones are marked with '–', not-so-stylistic ones with '⋆'):

- – '`;`' is the terminator of a lexical specification. It is required. Each spec must start on a new line. It can span more than one line.
- – '`-->`' is the unary rule marker. Keep in mind that unary rules of CCG are not necessarily lexical rules.
- ⋆ 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 (manually 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;`
- ⋆ 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 `@X`s, which are imposed in term match. We therefore avoid—once again—the need for structured unification.
- ⋆ If you use an `@X` category in a unary rule, it will be treated as an ordinary category (so don't).
- – Non-variables in LF must be Lisp strings, or they must be prefixed by `!` to avoid evaluation by Lisp. Write hit' as `!hit`. It will be converted to the Lisp string constant `"hit"` by a Lisp reader macro. Using non-alphabetic symbols in these LF constants produces unpredictable string matching performance.
- ⋆ 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.[9] The '`tokens`' script converts your `&i` to `(LAM x x)`.
- ⋆ The double slash has been 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. The tokenizer does not transform a string constant if its starts at the beginning of a line, and skips

[9] An example of this formal need can be given as follows. Suppose that we want to subcategorize for a type-raised NP, e.g. $f := S/(S/(S\backslash NP))$: $\lambda p.f'(p(\lambda x.x))$. Type-raised arguments are universally $\lambda p.pa'$, so an argument could be e.g. $a := S/(S\backslash NP)$: $\lambda p.pa'$. 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 $f'(\lambda p.pa')$.

the rest of the line for case conversion etc. We recommend writing string-type phonological values in a line of their own. Spacing and spanning multiple lines for an entry aren't important in THEBENCH, so it should not be a problem.

- 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 capitalization, not THEBENCH.
- Comments start with '`%`'. The rest of the line is ignored.
- ⋆ Because THEBENCH 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 THEBENCH 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`. 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 THEBENCH variables. Doubling the name as `xx` is my convention, e.g. `\p\q.(!forall xx)(!implies (p xx)(q` 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 **??**. They are used by the LALR parser, which converts the system-generated `.lisptokens` version of a textual grammar to `.ccg.lisp`. You will 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.

5.2 `.ccg.lisp` file format

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.[10] Table **??** describes the format. For THEBENCH, the order is not important in these lists because the names are unique.

5.3 `.sup` and `.supervision` file formats

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

[10] 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. In Lisp, `rest` returns the value of an association pair, whereas `second` returns the value of a

data : lf ;
where each `lf` has the same format as in `.ccg` file. The repository `ccglab-database` has examples.

5.4 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 THEBENCH's LFs is:[11]

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 **??**'s non-terminal called '*lf*' that THEBENCH'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.

5.5 Parser's representations

All parser objects are converted to hash tables during parsing and training. 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 search because beta-normalized LF is the key for that formula's table. When you have Catalan number of potential 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 THEBENCH. Table **??** shows example snapshots during parsing. Their keys and structure are explained in Table **??**.

5.6 How THEBENCH's term unification works

First we re-iterate that there is no re-entrant unification in THEBENCH. 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.

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

There are two Y's to match in the rule, and two Z's, to project by this rule. Notice that Z is passed on, and Y match is needed to see if rule's conditions are satisfied, so that we can project its unique semantics, viz. $\lambda x.fx(gx)$ where f is semantics of the first element, and g is that of second.

In each one of these matches 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). 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

name-value pair.

[11] 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.

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 above 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 §2 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 for speed. All projected valuations can be unique to a particular rule use, hence need a fresh copy of results.

6 Top-level functions, and names to watch out for

The basic parsing functions were explained in §4. Others are quite useful for grammar development and testing. A more comprehensive list is provided in Table **??**. The code includes commentary to make use of them and others. All THEBENCH functions are callable if you know what you're doing.

The names of all the features and hash table keys listed in Tables **??** through **??** are considered reserved names by the system. You will get a warning message from `make-and-load-grammar` if you use them in your `.ccg` file. Using them as a basic category feature results in unpredictable behavior. For example, if you use ARG as a feature, the system might crash because it expects such features in right places to be hash-valued at parse time.

7 THEBENCH 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 parse 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. For data space we comment later about using multiple processors.

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, and it is a commonly used function. 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. Non-ANSI Common Lisps are not compatible with THEBENCH.

7.1 The 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. Number of CCG derivations can grow up to Catalan numbers on input size if left unconstrained. Beam is one way to constrain it. By default it is off.

Sort is essential to the beam. We use the formula 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.

7.2 Heap and garbage collection

One easy way to get the best of both worlds of fast sort and big heap is to adjust the `CCGLAB_LISP` variable after install. For example, if you do the following before running THEBENCH, it will set SBCL's heap to 6GB.

```
export CCGLAB_LISP='/usr/bin/sbcl --noinform --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. 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.[12] This is the number of maps, not memory size.

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

7.3 Normal-form parsing

Normal Form (NF) parsing is 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 once turned on 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 parsing and ranking modes. There is one switch to control is behavior, listed in Table **??**.

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 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, THEBENCH'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 Eisner paper) for the output of the rule.

7.4 Training on large datasets

Parsing after training is fast. Type-raising can change that if there are too many rules to apply.[13]

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

[13] How many is too many? something around 1,000 seems somewhat noticeable, half of that hardly noticeable, more than

In any case you need all the help you can get in training. A bash script named `train-sbcl-multithread` is available in the repository to run various experiments simultaneously if you have multicore 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`; that is, they are immune to hangups and logouts (but not to control-D; let the terminal process die rather than control-D it).

Here is one example of an experiment file, `g1.exp`:

```
7000 4000 g1 t xp 1.2 1.0 train-basic-xp basic-ccg
4000 2000 g1 t 10 0.5 1.0 train-10
```

You would run it as: `train-sbcl-multithread g1.exp`.

It will fetch two processors, one using 7GB of RAM in which 4GB is heap, the other with 4GB RAM and 2GB of heap, loads `g1.ccg.lisp` and `g1.sup` in both cases, saves the result of training because of `t` (specify `nil` if you don't want to save the trained grammar). The rest are training parameters: `xp` means use the extrapolator. The second experiment does not extrapolate; it iterates 10 times. Then comes learning rate, learning rate rate, prefix of the log file (full name is suffixation of training parameters), and the function to call before training starts. In the first experiment it is `basic-ccg`. In the second experiment, there is nothing to call.

The bash script `train-sbcl-1` is a 9-argument monster. It assumes that constraints which are not within control of SBCL are already handled, such as `/proc/sys/vm/max_map_count` above. Even if you have no multicore support, use `train-sbcl-multithread` to run this script. If you have only one line of experiment data, it will fetch one processor anyway, and you'd be free of hangups.

7.5 Hash tables and growth

The hash tables for the CKY parser can grow very big. To work with very long sentences or many automatically generated type-raising rules 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.

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 **??**. 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.

7.6 Z-scoring and floating-point overflow/underflow

Because of exponentiation in formula (2) you have to watch out for floating point overflow and underflow. If the parameters become too large or small, which may happen if you run many training sessions on the same grammar, you can z-score a grammar as explained in Table **??**. In a z-scored grammar all parameters are fraction of the standard deviation distant from the mean. Two methods are provided: (i) assuming that all elements of grammar come from the same distribution, (ii) finding sample means and standard deviation per lexical form, because the same form's different lexical entries are competing in parsing and ranking, respectively called `z` and `z2`. If you will merge grammars

later on, the second method would be unpredictable but first method might be fine if they come from the same distribution. You can set a threshold for cutoff and a comparison method; see Table **??**. 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.)

Z-scoring is also useful if you want to compare relative entropy of two probability distributions. The `klz` function in Table **??** computes Kullback-Leibler divergence of two probability distributions. It uses z-scores of parameters to compute probabilities.

Acknowledgments

I thank the Python and Lisp communities, `stackoverflow` and `stackexchange` for answering my questions before I ask them. The self-contained lambda-calculus processor is based on Alessandro Cimatti's abstract data structure implementation of lambda calculus. David Beazley's `sly` python library made grammar description a lot easier. Marco Heisig's `cl4py` library made Python-Lisp communication easy, which allowed me to recycle some old Lisp code. Thanks to all three gentlemen.

I blame good weather for remaining errors.

References

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

Bozşahin, Cem. 2022. Referentiality and configurationality in the idiom and the phrasal verb. *Journal of Logic, Language and Information*, 27:1–33.

Bozşahin, Cem. 2023. *Monadic Structures*. Leiden/Boston: Brill. Forthcoming.

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.

Montague, Richard. 1973. The proper treatment of quantification in ordinary English. In *Approaches to Natural Language*. eds.J. Hintikka and P. Suppes. Dordrecht: D. Reidel.

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.

Sapir, Edward. 1924/1949. The grammarian and his language. In *Selected Writings of Edward Sapir in Language, Culture, and Personality*. ed.David G. Mandelbaum. Berkeley: University of California Press. Originally published in *American Mercury 1: (1924) 149–155*.

Schmerling, Susan. 2018. *Sound and Grammar: a Neo-Sapirian Theory of Language*. Leiden/Boston: Brill.

Schönfinkel, Moses Ilyich. 1920/1924. On the building blocks of mathematical logic. In *From Frege to Gödel*. ed.Jan van Heijenoort. Harvard University Press, 1967. Prepared first for publication by

5,000 needs coffee break, or maybe taking over a processor farm.

H. Behmann in 1924.

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

Steedman, Mark. 2020. A formal universal of natural language grammar. *Language*, 96:618–660.

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

Swadesh, Morris. 1938. Nootka internal syntax. *International Journal of American Linguistics*, 9:77–102.

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.