Polarity Tagger

A quick guide for someone who has never used the tool, and has never seen CCG. Understand what it does, learn the grammar, drive the app, then read its output correctly.

`Type` → `Load` → `Build` → `Tag`

PART 1 · THE IDEA

You type a sentence. It labels every word.

Each label is a **polarity**: the direction of valid inference at that position. The whole job is to turn a sentence into a map of where you may reason **up**, **down**, or neither.

PART 1 · THE IDEA

The three tags

↑ up	Replace the word with something **more general**. `"a dog" ↑ gives "an animal"`
↓ down	Replace it with something **more specific**. `"no animal" ↓ gives "no dog"`
= equal	Neither. The position is **neutral**.

PART 1 · THE IDEA

"no" flips a position from up to down.

```
no animal moves
```

⇓ implies

```
no dog runs
```

Polarity comes from a deeper property called **monotonicity**, covered in Part 4.

02

The Grammar: CCG

Combinatory Categorial Grammar: how the words snap together into a sentence.

PART 2 · THE GRAMMAR

Almost everything lives in the dictionary.

Each word carries a **category**. A few universal rules combine those categories until the whole sentence becomes one **S**.

PART 2 · THE GRAMMAR

Slash notation

X/Y

"I need a **Y** on my **right**; give me one and I become an X."

X\Y

"I need a **Y** on my **left**; give me one and I become an X."

The slot is always written on the right of the slash; the result on the left.

PART 2 · THE GRAMMAR

Reading the lexicon

`dog : N`	a noun, complete already, needs nothing.
`Fido : NP`	a noun phrase, complete already.
`runs : S\NP`	needs an NP on its **left** (the subject) to become S.
`every : NP/N`	needs an N on its **right** to become NP.
`chase : (S\NP)/NP`	object on the right, then subject on the left.

PART 2 · COMBINATION RULES

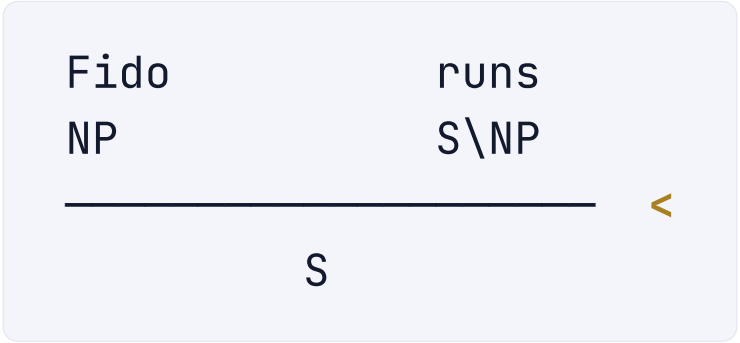
Forward application >

A functor **X/Y** consumes the **Y** on its **right**.

PART 2 · COMBINATION RULES

Backward application <

A functor `X\Y` consumes the `Y` on its **left**.



PART 2 · COMBINATION RULES

Type-raising >T <T

```
forward  >T  turns  X  into  S/(S\X)
backward <T  turns  X  into  S\(S/X)
```

Alone it changes nothing; its job is to **set up composition**. It lets an argument behave like a function, so it can glue to its neighbour before its head arrives.

PART 2 · COMBINATION RULES

Composition >B <B >B× <B×

$$X/Y \quad + \quad Y/Z \quad \rightarrow \quad X/Z$$

Glue two functors into one without applying yet; crossed variants let the slashes point different ways. Together with type-raising, this is what lets **"John saw"** act as a single constituent, used for relative clauses, extraction, and coordinating mismatched chunks.

PART 2 · COMBINATION RULES

Coordinate Φ

Join two phrases of the **same** category with "and / or / but".

03

Driving the App

The interface runs left to right: **Type** → **Load** → **Build** → **Tag.**

PART 3 · LOAD & BUILD

Load turns text into cards.

every	dog	runs
NP/N	N	S\NP

The tool already knows a fixed set of words. If you would rather not build a CCG derivation yourself, click a ready-made **Example** to load it complete. To see how that example was parsed, press **Undo** to step back through it move by move.

PART 3 · LOAD & BUILD

Word variations are handled for you.

`dogs → dog` `mice → mouse`

`ran → run` `saw → see` `chased → chase`

Plurals fold to singular; tensed verbs fold to their base. So **"no dogs ran"** loads the same cards as **"no dog run"**.

PART 3 · LOAD & BUILD

Some cards switch ⇄

A dashed underline means more than one reading; click to cycle.

who / whom / that	object extraction (default) or subject extraction
if	clause-initial "If A, B" or clause-final "B if A"

PART 3 · LOAD & BUILD

Build by combining the cards.

Select **two adjacent** cards, then click **>**, **<**, or a composition.

Select **one** card, then click **>T**, **<T** to type-raise.

Select **three adjacent** cards, then click **Φ** to coordinate.

You are done when a single S card remains, ready to Tag.

04

The Semantics: Monotonicity

The tree was pure syntax. Polarity rests on one idea.

PART 4 · MONOTONICITY

Each category gets a semantic type.

These types form a **preordered family** that records monotonicity: every function is increasing, decreasing, or neutral in each argument.

+	**increasing** function: a bigger input gives a bigger output; inference flows the same way.
-	**decreasing** function: a bigger input gives a smaller output; inference reverses.
•	**neutral** function: no guaranteed direction either way.

PART 4 · MONOTONICITY

The marks live on the arrows.

some	(e→t) →+ ((e→t) →+ t)
no	(e→t) →- ((e→t) →- t)
every	(e→t) →- ((e→t) →+ t)
most	(e→t) →• ((e→t) →+ t)

Read the marks straight off. "no" reverses inference in both arguments, so both arrows are **−**. "most" says nothing about its noun, so that arrow is **•**.

05

The Typing Rules

Every rule carries a polarity d on its terms and a mark on each arrow.

PART 5 · THE RULES

Polarity red, markings blue

$$\frac{u^{d} : \sigma \to^{m} \tau \quad v^{md} : \sigma}{(uv)^{d} : \tau}\ {>} \qquad \frac{u^{d} : \tau \to^{n} \rho \quad v^{nd} : \sigma \to^{m} \tau}{(u \circ v)^{d} : \sigma \to^{mn} \rho}\ \mathrm{B} \qquad \frac{u^{md} : \sigma}{u^{d} : (\sigma \to^{m} \tau) \to^{+} \tau}\ \mathrm{T}$$

$$\frac{u^{md} : e \to \tau}{u^{d} : (n \to^{m} t) \to^{+} \tau}\ \mathrm{H} \qquad \frac{u^{d} : \sigma \qquad \sigma \preccurlyeq \sigma'}{u^{d} : \sigma'}\ \mathrm{W} \qquad \frac{u^{=} : e}{u^{d} : np^{+}}\ \mathrm{ML}$$

$$\frac{x^{d} : \sigma \quad and^{d} : \sigma \to^{+} (\sigma \to^{+} \sigma) \quad y^{d} : \sigma}{(x \ and \ y)^{d} : \sigma}\ \Phi$$

$n = e \to t,\ np^{+} = (n \to^{+} t),\ np^{-} = (n \to^{-} t),\ np^{\bullet} = (n \to^{\bullet} t)$

06

What "Tag" Does

Pressing Tag runs two automatic passes over the finished tree.

PART 6 · TAG

Two passes over the tree

PASS 1 ↑

Fit the types

Bottom-up. Each leaf takes its marked type from the lexicon; the tool walks up, checking each step against the rules and computing the result type.

PASS 2 ↓

Push the polarity

Top-down. Start at the root S with **up**; apply the rule at each node until every word is tagged.

PART 6 · TAG

A nuance: type-raising is resolved against its neighbour.

A type-raised card carries no fixed type on its own, and it must feed a composition. So in Pass 1 the tool does not raise it in the abstract: it looks at the **other premise**, the functor it is composing with, and picks one of two behaviours.

CASE 1 real raise

The card genuinely lifts.

If the card's type matches what the functor expects deep inside, it is raised to a higher type so the two can compose, exactly like the T rule.

CASE 2 identity

Raising is a no-op.

If the card is already shaped to absorb the functor directly, it passes through **unchanged** and the two compose as they are.

Which case applies is a consequence of the **lexicon** you tagged with, itself an input, as the next slide shows. If neither fits, Pass 1 reports that the card cannot be raised against its neighbour.

PART 6 · TAG

The lexicon is an input too.

The marked types are **editable**, so the result depends on two things you control: the **derivation** you built and the **lexicon** you tagged it with.

Tag is best-effort with fallbacks. When none applies, Pass 1 **fails** and reports where. The practical goal: get the types to fit.

PART 6 · TAG

The derivation is an input too.

One sentence can have **two builds**. Each build gets its own tags, and each set of tags is correct **for that build**.

BUILD 1: APPLY ONLY

the subject takes wide scope

BUILD 2: RAISE THE SUBJECT, THEN COMPOSE

the object takes wide scope

Only **every** and **dog** change. This is not a bug: each build carries its own **reading**, and the tags are sound for that reading.

07

The Adapt Function

When a typing step does not fit, Adapt rewrites the two types so the derivation can be typed.

PART 7 · ADAPT

What Adapt does

Adapt is a short list of type adaptations, tried in order. Given a rule and the pair of types it is combining, it returns a new pair of types for that step. While the box is **empty** it behaves as the identity: it returns the input pair unchanged, so the raw algorithm fails exactly where help is needed. A line you add affects only the pairs it matches; every untouched pair passes through as before.

We claim that two adaptations cover almost everything: H and W.

PART 7 · ADAPT

"no dog runs" gets stuck.

runs has the simple type e→t, so it expects a plain **entity** as its subject.

But **no dog** is a **quantifier**, a function over properties, not a plain entity.

The types do not fit, so Pass 1 stops here. This is where H comes in.

PART 7 · ADAPT

H: verb-lifting

H enlarges the verb's argument so it can take a quantifier instead of a bare entity.

RULE

$$\frac{u^{md} : e \to \tau}{u^{d} : (n \to^{m} t) \to^{+} \tau}\ \mathrm{H}$$

EXAMPLE: NO DOG RUNS

$$\frac{\mathit{no\ dog}^{\uparrow} : n \to^{-} t \qquad \dfrac{\mathit{runs}^{\downarrow} : e \to t}{\mathit{runs}^{\uparrow} : (n \to^{-} t) \to^{+} t}\ \mathrm{H}}{\mathit{no\ dog\ runs}^{\uparrow} : t}\ <$$

PART 7 · ADAPT

W: weakening for coordination

When two conjuncts have different types, W sends both to their **join**: keep **+** where both agree, **−** where both agree, **•** elsewhere.

RULE

$$\frac{u^{d} : \sigma \qquad \sigma \preccurlyeq \sigma'}{u^{d} : \sigma'}\ \mathrm{W}$$

EXAMPLE: SOME DOG AND NO CAT

$$\frac{\dfrac{\textit{some dog} : np^{+}}{\textit{some dog} : np^{\bullet}}\ \mathrm{W} \qquad \textit{and} : np^{\bullet} \to^{+} (np^{\bullet} \to^{+} np^{\bullet}) \qquad \dfrac{\textit{no cat} : np^{-}}{\textit{no cat} : np^{\bullet}}\ \mathrm{W}}{\textit{some dog and no cat} : np^{\bullet}}\ \Phi$$

PART 7 · THE PRACTICAL LOOP

`Build the tree` → `Tag` → `does not fit? add H or W` → `Tag again`

Usually H and W are enough. If they are not, and you find yourself adding a new condition to the Adapt function, please share it with us so we can improve the program.

PART 7 · ADAPT

Writing your own Adapt line

A new line brings three things: the **pair of types** it rewrites, a **marking** for the child, and the **operation** it performs on the meaning. Then there is **one check**:

marking	the check on the operation
+	it must be **increasing**: larger inputs give larger outputs
-	it must be **decreasing**: larger inputs give smaller outputs
•	nothing to check

08

Reading the Output

Every word carries a final tag. Read each as a substitution license.

PART 8 · READING THE OUTPUT

Each tag is a substitution license.

↑ up	replace with something **more general**.
↓ down	replace with something **more specific**.
= equal	neither substitution is licensed here.

PART 8 · READING THE OUTPUT

"no dog runs", tagged

Under "no", both "dog" and "runs" come out **down**, so you may only specialise them. That is exactly why **"no animal moves" implies "no dog runs"**, and **"no dog runs" implies "no pug sprints"**.

PART 8 · READING THE OUTPUT

Change the determiner, change the polarity

Sentence	dog	runs	because the determiner is…
some dog runs	↑ up	↑ up	increasing in both arguments
no dog runs	↓ down	↓ down	decreasing in both arguments
every dog runs	↓ down	↑ up	decreasing noun, increasing verb
most dogs run	= equal	↑ up	neutral noun, increasing verb

Now try it.

`Type a sentence` → `Build to one S` → `Tag` → `H / W if needed` → `Read ↑ ↓ =`

Or load an Example and Undo through it to see how a finished derivation was built.