(DRAFT) Exploring Compositional Generalization (in ReCOGS_pos) by Transformers using Restricted Access Sequence Processing (RASP)

William Bruns
adde.animulis@gmail.com

Abstract

Humans rapidly generalize from a few observed examples and understand new combinations of words encountered if they are combinations of words recognized from different contexts, an ability called Compositional Generalization. Some observations contradict that Transformers learn systematic, compositional solutions to problems that generalize. The COGS benchmark (Kim and Linzen, 2020) reports zero percent accuracy for Transformer models on some structural generalization tasks. We use (Weiss et al., 2021)'s Restricted Access Sequence Processing (RASP), a Transformer-equivalent programming language, to prove by construction that a Transformer encoder-decoder can perform the semantically equivalent ReCOGS_pos (Wu et al., 2024) variant of COGS systematically and compositionally: Our RASP model attains 100% semantic exact match[1] on the ReCOGS test set and 100% semantic exact match on all generalization splits except obj_pp_to_subj_pp which gets 92%. Furthermore, our RASP model shows the ReCOGS_pos task does not require a hierarchical or tree-structured solution: we use word-level tokens with an "embedding" layer that tags with possible part of speech[2], applying just once per encoder pass 19 attention-head compatible flat pattern-matching rules, shown using grammar coverage (Zeller et al., 2023) to be learnable from the training data, plus general prepositional phrase (pp) handling and complement phrase (cp) handling logic, and output the next logical form token (repeating until the logical form is complete). The model does not apply recursive, tree-structured rules like 'np_det pp np -> np_pp -> np', but scores 100% semantic exact match on pp recursion, cp recursion previously, and the previously reported hardest split obj_pp_to_subj_pp[3] using the decoder loop. Thus we demonstrate using RASP that hierarchical/tree-like solutions are not required for high accuracy on ReCOGS which is semantically equivalent to COGS and argue it is likely Transformer models will be able to perform the ReCOGS_pos near perfectly including the structural generalization splits, but a different learning strategy may be required.

1 Introduction

For decades it was argued that connectionist models (i.e. neural networks) were somehow structurally incapable of compositional learning (Fodor and Pylyshyn, 1988).[4] However, large language models based on the Transformers architecture compose seemingly fluent and novel text and are excellent few or zero shot learners (Brown et al., 2020).

Some observations do contradict that Transformers learn systematic, compositional solutions to problems that generalize[5], for example structural generalizations[6] in the COGS task and ReCOGS (Wu et al., 2024) variant of the COGS task (Kim and Linzen, 2020)[7], a benchmark based on extracting semantics (logical form) from the syntax (grammatical form) of synthetic sentences in a simplified subset of English grammar, requiring models trained only on certain grammar examples to generalize to sentences with unseen grammar built up / recombined from parts present in the training examples.

[1] and 100% string exact match

[2] per (Tenney et al., 2019) by layer 0 the part-of-speech can be predicted for most words in real pre-trained deep Transformers already, so we assume that is learnable

[3] where, in the generalization case, prepositional nouns are frequently inserted between the leading noun and the verb it is related to, unlike the training case for those verb types, and models (our RASP model and we show also the baseline Transformer) in this case tend to an attraction error to a nearer noun, now a prepositional phrase noun. *See Appendix 10: Attraction errors for more detail and connection to prior NLP and psycholinguistics literature on similar errors in the context of hierarchical vs linear processing by language models and humans.*

[4] This debate does continue in particular areas, for example just in the domain of syntax, one can read (van Schijndel et al., 2019) vs (Goldberg, 2019).

[5] See Appendix 13: Composition and Learning

[6] especially prepositional phrase generalizations

[7] even with (Lake and Baroni, 2023)'s "meta-learning for compositionality" framework for Transformers described as achieving "human-like systematic generalization",their error rate on the COGS structural generalizations was still 100%

We use (Weiss et al., 2021)'s Restricted Access Sequence Processing (RASP) language that can be compiled to concrete Transformer weights to prove by construction that a Transformer encoder-decoder[8] can perform the ReCOGS_pos[9] (Wu et al., 2024) variant of the COGS (Kim and Linzen, 2020) task over the vocabulary and grammar of that task in a systematic, compositional way (length and recursion depth limited) as a rigorous starting point to investigating when Transformers might learn or not actually learn such compositional/systematic solutions. We find a flat, not hierarchical/tree-structured model which lacks any handling for the recursive rules in the grammar (for prepositional phrase recursion and complement phrase recursion) can perform the task at high accuracy, but requires a special rule for avoiding "attraction" errors[10] where inserted prepositional phrase nouns replace agent/theme/recipient nouns in the logical form by accident. This is our main result and suggests that the ReCOGS task can be performed with high accuracy by Transformers, turning efforts to learnability, and also adds to the literature a caveat on interpreting success on ReCOGS by noting it does not necessarily require a hierarchical or tree-based approach. Finally, we predict that these "attraction" errors we had to specifically avoid in our RASP model are contributing to the high error rate of the (Wu et al., 2024) baseline Transformer trained from scratch and confirm this is the case.

2 Prior Literature

(Kim and Linzen, 2020) introduce the COmpositional Generalization Challenge based on Semantic Interpretation (COGS) benchmark[12] and argue that

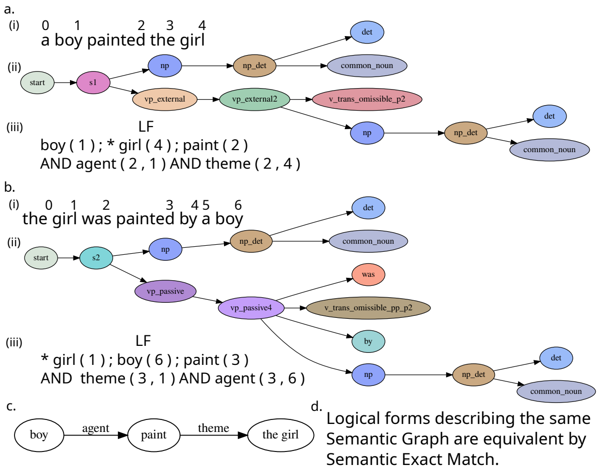


Figure 1: Introducing parse trees, logical form, and semantic graphs. Two semantically identical but syntactically distinct (i) sentences (a) "a boy painted the girl" and (b) "the girl was painted by a boy" are shown with (ii) their distinct parse tree (parsed into COGS input grammar), (iii) the string form of their semantics (ReCOGS logical form; differs in indices and ordering), and (c) the graph representation of their logical form (semantic graph[11], not different at all between the two examples). Note the (iii) logical forms (LFs) differ by String Exact Match but not (Wu et al., 2024)'s Semantic Exact Match (order and indices do not match but nouns, normalized verbs, and relationships between nouns and verbs are same). Note the "agent", "theme" order in the logical form string is not required to match for Semantic Exact Match.

Transformers have low accuracy on the generalization splits (35% overall), especially structural generalization splits where near 0% accuracy is reported, using a 2-layer Encoder-Decoder Transformer (2 layers for Encoder, 2 layers for Decoder).

For another example, (Lake and Baroni, 2023) use a "meta-learning for compositionality" approach with a 3-layer Encoder-Decoder Transformer architecture and achieve what they call "human-like systematic generalization", achieving high scores on everything in the COGS benchmark (>99% on lexical generalizations) EXCEPT the structural generalization splits where they also still score 0% accuracy. However, one notices these networks are shallow compared with those used in successful large-pretrained Transformer models (e.g. 24-layer BERT where compositional parse trees seem to be encoded in its vector space representation (Hewitt and Manning, 2019)), and it is claimed, by e.g. (Csordás et al., 2022) that for compositional operations, like parsing, the depth of the network must be at least the maximum number of compositional operations, e.g. the height of the parse tree for grammar dependent problems. Re-

[8] We follow (Zhou et al., 2023) who used RASP to analyze encoder-decoder and decoder-loop cases, not just Transformer encoders as done by RASP author (Weiss et al., 2021).

[9] closer to COGS than non-positional ReCOGS, as COGS is also positional, and means we can also measure string exact match, not just semantic exact match

[10] These attraction errors are similar to those discussed elsewhere in NLP and psycholinguistics literature on hierarchical vs linear processing by language models and humans, see Appendix 10: Attraction errors.

[11] As a convention, in converting ReCOGS logical forms to Semantic Graphs we use the logical form (source, target) index order for directed semantic graph edges (from verb to related entity) EXCEPT for the agent relationship which is from the agent of a verb to the verb (opposite direction from logical form in that case), which gives our semantic graphs of ReCOGS sentences an unambiguous starting point (layout starts from agent) without affecting comparison of the graphs (generated by a consistent rule), see also Figure 2.

[12] See Figure 1. Note, the COGS/ReCOGS task is important because it is converting sentences to a representation of their meaning (in logical form where syntactically different but semantically identically sentences like "a boy painted the girl" or "the girl was painted by a boy" are written the same), and we want these models to be able to understand entirely the new sentences they will be encountering (probably immediately given the diversity of language) based on being familiar having observed pieces of them in different combinations in the past.

markably, (Petty et al., 2024) further claims that increasing the layer depth of the Transformer models (up to 32 layers) does not improve the near 0% accuracy on COGS structural generalization splits like prepositional phrase modification of subject when the network has only seen it on the object during training and also input length/depth generalizations (like pp/cp recursion), perhaps surprising as for the simpler logical inferences problem in (Clark et al., 2020) they observed successful logical inference depth generalization even by Encoder-only Transformers.

Thankfully, (Wu et al., 2024) are able to begin to get traction (low but nonzero accuracy) for the shallow Encoder-Decoder Transformer models on structural generalizations in a modified but semantically equivalent form of the COGS task they call ReCOGS, which we analyze here. They remove redundant symbols and use Semantic Exact Match instead of Exact Match to avoid forcing the model to predict arbitrary variable labeling in logical forms.

(Zhou et al., 2023) apply (Weiss et al., 2021)'s RASP language to explain some inconsistent findings regarding generalization and use RASP to predict exactly which cases of generalization come easily to Transformers and which do not. (Zhou et al., 2023) seem to reveal (Weiss et al., 2021) has provided the framework we seek by demonstrating how to apply RASP to Transformer decoders with intermediate steps, and even use it to learn how to modify difficult-to-learn tasks like Parity[13] and long addition in seemingly incidental ways based on RASP analysis to make them readily learnable by Transformers in a compositional, length generalizing way![14]

Thus we apply techniques similar to (Zhou et al., 2023) and (Weiss et al., 2021) to ReCOGS to (1) argue Transformers should be capable of performing the task, including the structural generalization splits, with high accuracy, and that the problem is learning not capability and (2) try to assess how (Wu et al., 2024)'s baseline Encoder-Decoder Transformers learn to extract semantics (logical form) from the syntax (grammatical form) of sentences and understand the prepositional phrase modification related generalization errors they are making.

3 Data

COGS (Kim and Linzen, 2020) and ReCOGS (Wu et al., 2024) datasets were used as provided by the repository associated with (Wu et al., 2024)[15], with special attention on the structural generalization splits (especially prepositional phrase Object-to-Subject modification).

The grammar and vocabulary description for COGS/ReCOGS English input sentences provided in the utilities associated with the IBM CPG project (Klinger et al., 2024)[16] were used in designing our RASP solution and analyzing the ways in which this task could be learned (we did not actually use their grammar though, and our RASP solution is flat and non-hierarchical unlike their description of the COGS probabilistic context-free grammar which is hierarchical and recursive). See Figure 1 .

4 Model

We used the RASP interpreter of (Weiss et al., 2021) to run our program. For RASP model design and details see Appendix 5.

We use word-level tokens for all results in this paper.[17] Consistent with (Zhou et al., 2023) we use (Weiss et al., 2021)'s RASP originally used for modeling Transformer encoders to model an encoder-decoder in a causal way by feeding the autoregressive output back into the program. We only have aggregations with non-causal masks when that aggregation (or without loss of generality just before the aggregation product is used to avoid multiplying everywhere) is masked by an input mask restricting it to the sequence corresponding to the input.[18]

For training Transformers from scratch with randomly initialized weights using gradient descent for comparison with RASP predictions, we use scripts derived from those provided by (Wu et al.,

[13] See (Strobl et al., 2024) for context from formal language theory, computational complexity, circuit complexity theory, and experimental papers together, providing robust lower and upper bounds on what Transformers can do, including discussion of under what conditions Parity can be solved by Transformers and how whether it can be learned by randomly initialized Transformers under simple training schemes is a different question (general feed-forward neural networks can learn to solve Parity per (Rumelhart et al., 1988)). (Delétang et al., 2023) also.

[14] See Appendix 12: Zhou et al 2024 relevance of their long addition experiment to language modeling and note on the Parity task and Transformers

[15] https://github.com/frankaging/ReCOGS

[16] https://github.com/IBM/cpg/blob/ c3626b4e03bfc681be2c2a5b23da0b48abe6f570 /src/model/cogs_data.py#L523

[17] We believe any solution at the word-level can be converted to a character-level token solution and that is not the focus of our investigation here (see Appendix 6 for proof of concept details on a character level solution not used here).

[18] An example the author has prepared of this is available at https://github.com/willy-b/learning-rasp/blob/ 16a8e154b025e91c8e56965a1d475e49f69ebdbd/recogs_examples_in_rasp.py .

2024)[19].

See Appendix 8 - Model Detail.

5 Methods

We use the RASP (Weiss et al., 2021) interpreter[20] to evaluate our RASP programs[21].

Logical forms (LFs) generated by the models were scored by Semantic Exact Match[22] against ground truth.

We also measure grammar coverage by input examples supported by our RASP model against the full grammar of COGS/ReCOGS input sentences provided in the utilities of the IBM CPG project (Klinger et al., 2024)[23].

See Appendix 9 - Methods Detail.

6 Results

Restricted Access Sequence Processing - grammar coverage using a flat pattern matching approach (not tree-based and not recursive) and autoregressive decoder loop

See "Appendix 11: Grammar Coverage Analysis for Design of Restricted Access Sequence Processing Model" for more details.

We generated 21 sentences based on rules present in the training examples which cover 100% of the COGS input grammar under those constraints (under the context free grammar, tree based assumption which turns out to be incorrect for prepositional phrases) per (Zeller et al., 2023):

```
# 19 examples for non-recursive grammar rules
"the girl was painted",
"a boy painted",
"a boy painted the girl",
"the girl was painted by a boy",
"a boy respected the girl",
"the girl was respected",
"the girl was respected by a boy",
"the boy grew the flower",
"the flower was grown",
"the flower was grown by a boy",
"the scientist wanted to read",
"the guest smiled",
"the flower grew",
"ella sold a car to the customer",
"ella sold a customer a car",
"the customer was sold a car",
"the customer was sold a car by ella",
"the car was sold to the customer by ella",
"the car was sold to the customer",

# 2 examples for recursive grammar rules
# (1 prepositional phrase example)
"a boy painted the girl in a house",
# (1 complement phrase example)
"the girl noticed that a boy painted the girl"
```

The first 19 of those sentences are present in our RASP program code[24] and each correspond to a set of RASP operations corresponding to attention operations in a Transformer to match a template corresponding to that sentence type[25]. Those 19 examples reflect the only rules for handling non-prepositional/non-complement phrase grammar rules[26] present in our RASP solution (which gets 100% semantic exact match and string exact match on the (Wu et al., 2024) ReCOGS_pos test set, see next results section).[27]

Restricted Access Sequence Processing - semantic exact match

The *Restricted Access Sequence Processing program scored 100% Semantic Exact Match and String Exact Match (no missed examples) (95% confidence interval (Beta dist / Clopper-Pearson) of 99.88% to 100%, n=3000) on the ReCOGS_pos test set*[28]

The RASP model scored 99.59% semantic exact match on all non-recursive out-of-distribution generalization splits (18922 out of 19000 (95% confidence interval: 99.49% to 99.68%)) [29]

Recursion splits are reported below.

---

[19] https://github.com/frankaging/ReCOGS/blob/1b6eca8ff4dca5fd2fb284a7d470998af5083beb/run_cogs.py and https://github.com/frankaging/ReCOGS/blob/1b6eca8ff4dca5fd2fb284a7d470998af5083beb/model/encoder_decoder_hf.py

[20] provided at https://github.com/tech-srl/RASP/

[21] https://github.com/willy-b/learning-rasp/blob/16a8e154b025e91c8e56965a1d475e49f69ebdbd/word-level-pos-tokens-recogs-style-decoder-loop.rasp with a demo at https://github.com/willy-b/learning-rasp/blob/16a8e154b025e91c8e56965a1d475e49f69ebdbd/recogs_examples_in_rasp.py

[22] Using the official scripts at https://github.com/frankaging/ReCOGS/blob/1b6eca8ff4dca5fd2fb284a7d470998af5083beb/utils/train_utils.py and https://github.com/frankaging/ReCOGS/blob/1b6eca8ff4dca5fd2fb284a7d470998af5083beb/utils/compgen.py

[23] https://github.com/IBM/cpg/blob/c3626b4e03bfc681be2c2a5b23da0b48abe6f570/src/model/cogs_data.py#L523

[24] https://github.com/willy-b/learning-rasp/blob/dca0bc6689b0454b75e5a46e77ffe66566ca7661/word-level-pos-tokens-recogs-style-decoder-loop.rasp#L568

[25] see also "Appendix 5 - Restricted Access Sequence Processing word-level (post-embedding) token program/model design"

[26] To handle prepositional phrases in a flat solution, we find it necessary even on the training data to add a rule that ignores noun phrases preceded by a prepositional phrase (ignore "pp np") when searching for noun indexes to report in relationships (agent, theme, recipient, etc), and we loosen verb type templates to allow a gap for any prepositional phrase to be inserted. We shall see encountering this issue in RASP and the grammar analysis suggesting a non-tree solution leads us to be able to predict 96% (740 out of 767; n=10 trained-from-scratch Transformers, most runs 100% of that category for the Transformer) of a certain category of errors a baseline (Wu et al., 2024) Encoder-Decoder Transformer makes (see results).

[27] Note that those 19 sentences could be replaced by cherry-picked equivalent official COGS training examples without any effect on the actual RASP code (they are just comments next to the rules entered as part of speech tokens) (the last two example sentences above do not correspond to specific rules in the RASP code but could also be replaced by grammatically equivalent examples from COGS train.tsv).

[28] Data: https://raw.githubusercontent.com/frankaging/ReCOGS/1b6eca8ff4dca5fd2fb284a7d470998af5083beb/recogs_positional_index/test.tsv . Results: see Table 1 .

[29] Data: https://raw.githubusercontent.com/frankaging/ReCOGS/1b6eca8ff4dca5fd2fb284a7d470998af5083beb/recogs_positional_index/gen.tsv , Results: see Table 1.

ReCOGS_pos Split	Semantic Exact Match % (95% CI)
ReCOGS_pos test set (held out, in-distribution)	100.00% (99.88-100.00%)
Generalization splits (held out, out-of-distribution) (below)	
active_to_passive	100.00% (99.63-100.00%)
do_dative_to_pp_dative	100.00% (99.63-100.00%)
obj_omitted_transitive_to_transitive	100.00% (99.63-100.00%)
obj_pp_to_subj_pp	92.20% (90.36-93.79%)
obj_to_subj_common	100.00% (99.63-100.00%)
obj_to_subj_proper	100.00% (99.63-100.00%)
only_seen_as_transitive_subj_as_unacc_subj	100.00% (99.63-100.00%)
only_seen_as_unacc_subj_as_obj_omitted_transitive_subj	100.00% (99.63-100.00%)
only_seen_as_unacc_subj_as_unerg_subj	100.00% (99.63-100.00%)
passive_to_active	100.00% (99.63-100.00%)
pp_dative_to_do_dative	100.00% (99.63-100.00%)
prim_to_inf_arg	100.00% (99.63-100.00%)
prim_to_obj_common	100.00% (99.63-100.00%)
prim_to_obj_proper	100.00% (99.63-100.00%)
prim_to_subj_common	100.00% (99.63-100.00%)
prim_to_subj_proper	100.00% (99.63-100.00%)
subj_to_obj_common	100.00% (99.63-100.00%)
subj_to_obj_proper	100.00% (99.63-100.00%)
unacc_to_transitive	100.00% (99.63-100.00%)
all gen splits (19K examples, aggregate)	99.59% (99.49-99.68%)

Table 1: ReCOGS_pos test set performance (n=3000) and non-recursive out-of-distribution generalization split performance for **Restricted Access Sequence Processing (RASP) Encoder-Decoder Transformer-compatible model**, Semantic Exact Match %, with Beta/Clopper-Pearson confidence intervals. N=1000 examples for each generalization split. No examples excluded.
This model gets 100% Semantic Exact Match and String Exact Match on the in-distribution but unseen test set. Model uses a part-of-speech and verb-type map for word-level embedding and just 19 Transformer attention-head compatible flat grammar pattern recognizers based on rules visible in training data plus decoder-loop compatible prepositional phrase, complement phrase unrolling.
See https://colab.research.google.com/drive/1N7F-nc9GVnoC_9dBVdNT02SBiBcMbgy- for steps to reproduce ReCOGS_pos test results, https://colab.research.google.com/drive/1hLH9hFwPT_3HZUteUTBY4tYvdiZN0O0M for steps to reproduce ReCOGS_pos generalization split results. Includes complement phrase support added in https://github.com/willy-b/learning-rasp/pull/7 .

Restricted Access Sequence Processing - prepositional phrase and complement phrase recursion (tail recursive) with a non-tree, non-recursive approach using the decoder loop[30]

Our RASP model's ReCOGS pp_recursion gen split score was 100% semantic exact match AND string exact match (95% confidence interval (Beta dist/Clopper-Pearson): 99.63% to 100.0%; n=1000)[31].

Our RASP model's ReCOGS cp_recursion gen split score was 100% semantic exact match (95% confidence interval (Beta dist/Clopper-Pearson): 99.56% to 100.0%; no missed examples have yet been observed at n=840, waiting for n=1000 run to complete)

(Wu et al., 2024) Encoder-Decoder Transformer from scratch baselines (ReCOGS_pos)[32]

(Wu et al., 2024)'s baseline Encoder-Decoder Transformer on ReCOGS_pos had an overall score of 88.55% +/- 1.87% Semantic Exact Match accuracy (sample +/- std, n=20) with a 95% confidence interval for the sample mean when n=20 of 87.73% to 89.37%.

(Wu et al., 2024)'s baseline Encoder-Decoder Transformer's Semantic Exact Match score on the extremely difficult obj_pp_to_subj_pp split of ReCOGS_pos was 19.7% +/- 6.1% Semantic Exact Match accuracy (sample +/- std, n=20) with 95% confidence interval for the sample mean with n=20 of 17.0% to 22.4% .

(Wu et al., 2024)'s baseline Encoder-Decoder Transformer's Semantic Exact Match score on the pp_recursion split of ReCOGS_pos was 40.18% +/- 2.07% Semantic Exact Match accuracy (sample +/- std, n=20) with 95% confidence interval for the sample mean with n=20 of 36.13 to 44.24%.

(Wu et al., 2024)'s baseline Encoder-Decoder Transformer's Semantic Exact Match score on the cp_recursion split of ReCOGS_pos was 52.40% +/- 1.38% Semantic Exact Match accuracy (sample +/-

[30] The grammar includes two (tail) recursive aspects, prepositional phrase recursion, and complement phrase recursion.
The prepositional phrase recursion comes from the following COGS input grammar rules: 'np -> np_det | np_prop | np_pp' and 'np_pp -> np_det pp np'.
Thus np can be expanded in an unbounded way as follows: 'np -> (np_det pp np) -> np_det pp (np_det pp np) -> np_det pp np_det pp (np_det pp np)' and so on.
However, one sees this is tail recursion and can be handled by a loop that just appends 'np_det pp' until the final 'np' is not 'np_pp'.
Complement phrase recursion comes from the following COGS input grammar rules: 'np v_cp_taking that start', where the form supports being recursively expanded like 'np v_cp_taking that start -> np v_cp_taking that (np v_cp_taking that start)', and so on until the nonterminal start expands to some other non complement phrase related nonterminal.

[31] See last section of notebook https://colab.research.google.com/drive/1hLH9hFwPT_3HZUteUTBY4tYvdiZN0O0M

[32] See https://colab.research.google.com/drive/12mXX5L1I4rpwl1Jk8hCm-xyAkqiKJEo7 for (Wu et al., 2024) script execution and analysis code.

std, n=20) with 95% confidence interval for the sample mean with n=20 of 51.80 to 53.01%.

(Wu et al., 2024) Encoder-Decoder baseline 2-layer Transformer does not improve on the obj_pp_to_subj_pp split when adding 1 or 2 additional layers(even allowing parameter count to increase)[33]

3-layer (Wu et al., 2024) Encoder-Decoder on ReCOGS_pos obj_pp_to_subj_pp split: 16.2% +/- 2.7% Semantic Exact Match (sample mean +/- std, n=10) with 95% confidence interval for sample mean (n=10) of 14.6% to 17.9% .[34]

4-layer (Wu et al., 2024) Encoder-Decoder on ReCOGS_pos obj_pp_to_subj_pp split: 19.3% +/- 4.1% Semantic Exact Match (sample mean +/- std, n=10) with 95% confidence interval for sample mean (n=10) of 16.8% to 21.8%.[35]

Error Analysis for (Wu et al., 2024) baseline Encoder-Decoder Transformer on obj_pp_to_subj_pp split[36]

See "Appendix 3 - Error Analysis for (Wu et al., 2024) baseline Transformer" for additional information on this analysis.

Of the obj_pp_to_subj_pp split single part errors in single verb sentences made by the (Wu et al., 2024) baseline Encoder-Decoder Transformer where the agent was to the left of the verb[37], across n=10 models[38], 765 out of 767 (99.74%; 95% confidence interval 99.06 to 99.97%) were in the agent part of the logical form (the predicted position for the error).

Critically across all n=10 (Wu et al., 2024) models, for 96.73% (740 out of the previously mentioned 765 above; 95% confidence interval (Beta dist / Clopper-Pearson) 95.21 to 97.87%) of the single point errors in logical forms for single verb sentences where the agent was on the left, modified by a prepositional phrase, and the error was in the agent part, the error in the logical form was that the agent index was accidentally assigned to the specific expected prepositional phrase noun (the one closest to the verb on the left side) instead of the original agent noun. This does not vary much from randomly initialized and trained model to model, with the model-level average at 97.07% of such errors exactly as predicted (stderr=2.23% (n=10)), with 7 of 10 models having 100% of these errors exactly as predicted by our hypothesis[39] Note that the attraction to the nearest noun hypothesis predicts that the offset in the agent index varies with prepositional phrase recursion depth (at depth > 1, there are multiple attractor prepositional nouns to choose from) [40]

Since the confidence that the attraction to the nearest holds in the uncommon pp depth greater than 1 case depends on the number of examples we were able to observe, we report that n=22 single logical form part errors were observed (from running n=10 separate Transformer models over the 1000 sentences in the split) where in the input the agent is left of the verb and has a depth=2 prepositional phrase modification in this split, and for 100% (95% confidence interval (Beta dist / Clopper-Pearson) 84.6 to 100%; n=22) sentences the agent right-index matched our prediction.

(Wu et al., 2024) Encoder-Decoder Transformer on new v_dat_p2 pp moved to recipient (from theme) split - as hard as hardest previous generalization split

See Figure 2 in Appendix.

As the Restricted Access Sequence Processing program predicted the ‘np v_dat_p2 np pp np np‘[41] prepositional phrase modification (which involves the recipient instead of the subject so is a distinct check of our hypothesis) would require learning to ignore the distractor "pp det common_noun" and "pp proper_noun" same as required for the obj_pp_to_subj_pp split, we predicted that a new split we introduce "v_dat_p2_pp_moved_to_recipient" would also be difficult for the Transformer. To test this, (Wu et al., 2024)’s baseline Encoder-Decoder Transformer was trained with default data (ReCOGS_pos train.tsv) and tested on modified v_dat_p2 pp training examples where only the word order was

[33] Since no improvement was observed, we did not run the costly experiments to increase the layers while controlling the parameter count (which would be a follow up to distinguish if the improvement was from the layer increase or the parameter increase).

[34] https://colab.research.google.com/drive/12mXX5L1I4rpwl1Jk8hCm-xyAkqiKJEo7

[35] https://colab.research.google.com/drive/12mXX5L1I4rpwl1Jk8hCm-xyAkqiKJEo7 and https://colab.research.google.com/drive/13FRQeAjyPOhBtTdrpW8caL25rNryLn5-

[36] https://colab.research.google.com/drive/1Z0_EXV-bvmO2mRcnHmDpFuHIv4KHOpz-

[37] Our hypothesis is in terms of nouns with a logical form relationship to a verb or other noun, where the relationship could be agent, theme, or recipient. Since the obj_pp_to_subj_pp split is in terms of subject vs object prepositional modification (instead of agent, recipient, or theme), we use the subset of sentences within this split where the agent is to the left of the verb and modified by a prepositional phrase as it corresponds to the subject in that case.

[38] On a per model basis (n=10), the fraction of agent-left single point errors where it was the agent relationship in the logical form that was broken were: [0.985, 1.0, 1.0, 1.0, 1.0, 0.990, 1.0, 1.0, 1.0, 1.0].

[39] Fraction for each model as predicted: [0.970, 0.761, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 0.976, 1.0]).

[40] See "Appendix 10: Attraction errors" for examples of different pp recursion depths.

[41] Strictly speaking we only do ‘np v_dat_p2 np_det pp np np‘ as per the grammar ‘np_prop‘ cannot precede a prepositional phrase

changed to move the prepositional phrase from the theme to the recipient (logical form properly updated see Appendix 4 for all examples). *The baseline (Wu et al., 2024) Encoder-Decoder Transformer was only able to achieve a Semantic Exact Match (sample mean +/- sample std) of 13% +/- 15.6% (n=10 Transformers trained from scratch with randomly initialized weights and data shuffling) with a 95% confidence interval for the sample mean when n=10 of 4% to 23%.* Thus, this new split we introduce here as v_dat_p2_pp_moved_to_recipient is as difficult or perhaps more difficult than the previous reported "hardest split" obj_pp_to_subj_pp.

(Wu et al., 2024) Encoder-Decoder Transformer trained with data augmented with v_dat_p2 pp moved to recipient (from theme) does NOT improve obj_pp_to_subj_pp performance

(Wu et al., 2024)'s baseline Encoder-Decoder Transformer was trained with default data (ReCOGS_pos train.tsv) but with additionally the same modified v_dat_p2 pp training examples used for the "v_dat_p2_pp_moved_to_recipient" split (non-subject recipient modified with prepositional phrase, so nonoverlapping with subj_pp) above on which it performed poorly, then tested on the standard prepositional modification generalization split "obj_pp_to_subj_pp", after which it achieved 22% +/- 6.7% Semantic Exact Match (sample mean +/- std, n=10) with 95% confidence interval for sample mean n=10 of 17.9% to 26.1% (not significantly different than (Wu et al., 2024)'s baseline by one-tailed Welch's unequal variances t-test). See Figure 2.

7 Analysis

Our RASP model of a Transformer Encoder Decoder, without tree-based or recursive aspects scored 100% in semantic exact match accuracy on the (Wu et al., 2024) test set (n=3000), and on the generalization data scored 100% in all but one category (see above) without explicit rules in the RASP program to handle them. This includes 100% semantic exact match accuracy on the prepositional phrase recursion and complement phrase recursion generalization splits up to depth 12 (n=1000 examples each), without any hardcoded prepositional phrase or complement phrase expansion shortcuts added[42]. The RASP program only made a significant number of errors on obj_pp_to_subj_pp which scored only 92.20% Semantic Exact Match (95% confidence interval (Beta dist / Clopper-Pearson): 90.36% to 93.79%) Semantic Exact Match accuracy, much better than (Wu et al., 2024) baseline Encoder-Decoder Transformers which only scored 19.7% +/- 6.1% Semantic Exact Match (sample mean +/- std) with 95% confidence interval for the sample mean with n=20 of 17.0% to 22.4% (n=20 separately trained models with different random seeds for weight initialization and training data ordering; n=1000 examples used to test each of the n=20 models).

Thus, we demonstrated by construction using the Restricted Access Sequence Processing language which can be compiled to concrete Transformer weights that theoretically a Transformer Encoder-Decoder can solve the COGS input to ReCOGS_pos logical form translation in a systematic, compositional, and length generalizing way.

Recall we found a single flat pattern matching rule we originally added to fit training examples, to ignore "pp det common_noun" and "pp proper_noun" when matching nouns for the agent, theme, recipient right indices, was sufficient to avoid structural generalization errors due to pp modification in novel positions.

Interestingly, we imagined ablating that single rule and hypothesized attraction to the nearest noun[43] in its absence and found this predicted the exact error (the nearest noun to the verb is mistaken for the agent of the verb) in 96% of the single relationship errors the (Wu et al., 2024) baseline Transformers make on the obj-pp-to-subj-pp split[44] when the agent is left of the verb in single verb sentences[45] (suggesting perhaps that the baseline (Wu et al., 2024) Transformer trained from scratch is also not learning a hierarchical, tree-structured representation.)

[42] a single rule applies to all depths; the only limit on length generalization is the RASP interpreter and a simple to extend positional encoding which only handles sentences up to a limit due to a map that only covers numbers up to a limit but can be easily expanded by literally adding dummy entries like "'121':0,'122':0" and so on to a map in one place

[43] Specifically, a non-agent noun which was part of the actual attention-head compatible verb-centered pattern match (closer to the verb than the actual agent) in our RASP model getting labeled the agent. **See Appendix 10: Attraction errors for more detail and connection to prior NLP and psycholinguistics literature on similar errors in the context of hierarchical vs linear processing by language models and humans.**

[44] overall, their semantic exact match on the split is measured by us at 19.7%, consistent with their Figure 5

[45] See Figure 6 in Appendix 10: Attaction Errors.

[46]

Our explanation for the (Wu et al., 2024) baseline Encoder-Decoder Transformer errors could have been refuted by other single relationship errors occurring as frequently as the agent, indicating general model confusion (independently getting incorrect agent and theme, not just agent relationships) and/or when making an agent error, the model could have simply put nonsense indices or referred to any other position other than the closest pp noun position to the verb (which does vary and depends on pp depth) to refute our hypothesis.

The attraction to the nearest noun mechanism can also be checked by making a prediction on a completely different syntax affected by the same issue: the 'np v_dat_p2 np pp np np'[47] prepositional phrase modification (which involves the recipient relationship being modified instead of the subject and/or agent so is a distinct check of our hypothesis)[48][49] and we indeed found that this was as hard or harder than the previous most difficult split analyzed above, the 'obj_pp_to_subj_pp' split. [50] [51]

Maybe (Wu et al., 2024) baseline Encoder-Decoder is depth-constrained to learn a non-hierarchical, flat, non-tree model with these characteristic errors and with more layers it would learn to recursively combine 'np_det pp np -> np_pp -> np' (to some fixed depth at least, probably limited by the number of Transformer blocks) and perform better on prepositional phrase related splits[52][53].

However, training a (Wu et al., 2024) baseline Encoder-Decoder Transformer from scratch we found no benefit to 3 or 4 layers instead of 2 on the ReCOGS obj_pp_to_subj_pp split, consistent with (Petty et al., 2024)'s finding on COGS.

Taken together, these results and the grammar coverage analysis suggest we may interpret the poor performance on generalizing on unseen prepositional phrase related modification related splits as arising from the baseline 2 to 4 layer Encoder-Decoder Transformers learning a flat representation (non-tree, non-recursive) that cannot leverage the grammar rule 'np_det pp np -> np_pp -> np' during learning and which requires them to instead actually observe more of the various prepositional phrase substitutions to learn them.[54].

8 Conclusion

Implementing our task in Restricted Access Sequence Processing helped us discover additional failure modes (e.g. the new "v_dat_p2_pp_moved_to_recipient" split) of (Wu et al., 2024)'s baseline Encoder-Decoder Transformer, predict the errors in the logical forms and may help reason about why for (Wu et al., 2024) 2 layers suffices for the ReCOGS task[55], and recommend others to consider using RASP to understand Transformer behavior even for complicated tasks, like ReCOGS. We also predict Transformers will be able to perform even the structural generalizations of the ReCOGS task with high accuracy, and that the problem is getting the Transformer to learn the appropriate rules[56], turning attention to data augmentation[57], curriculum learning (Bengio et al., 2009), reinforcement learning (Ranzato et al., 2016), or other approaches.

[46] After this paper was written, during review we found (Li et al., 2023) also specifically mention they also observe in their new benchmark SLOG, the error we predicted and confirmed here on ReCOGS ("attraction" error of closest noun (now prepositional phrase noun) being mistaken for subject/agent of verb when agent is left of verb and modified by pp), stating "For instance, in sentences like 'A cat on the mat froze', models often misinterpret the closer NP the mat as the subject."

[47] Again, precisely we only do 'np v_dat_p2 np_det pp np np' as per the grammar 'np_prop' cannot precede a prepositional phrase

[48] According to our Restricted Access Sequence Processing program as well the 'np v_dat_p2 np pp np np' generalization requires learning the exact same rule as the subj pp generalization: to ignore the distractor "pp np" (as "pp det common_noun" and "pp proper_noun", not requiring any recursive expansion).

[49] To test this, (Wu et al., 2024)'s baseline Encoder-Decoder Transformer was trained with default data (ReCOGS_pos train.tsv) and tested on modified v_dat_p2 pp training examples where only the word order was changed to move the prepositional phrase from the theme to the recipient (logical form properly updated see Appendix 4 for all examples)

[50] Note that if the Encoder-Decoder Transformer were to learn a tree-based or recursive representation, it would also be predicted that the "v_dat_p2_pp_moved_to_recipient" would not be any harder than when the pp modification is on the theme, as 'np v_dat_p2 np_det pp np np' can be transformed by the recursive grammar rule 'np_det pp np -> np_pp -> np' to 'np v_dat_p2 np np' on which it is already trained and has good performance, whereas we observe "v_dat_p2_pp_moved_to_recipient" is instead as hard as the hardest previously reported generalization split.

[51] After writing the first draft of this paper with all experiments completed, the author found (Li et al., 2023) which appears to have updated COGS to include the split we independently predicted and verified as difficult here before finding their paper, see their section 2.2.1 indirect object modification (4).

[52] this is not a very scalable approach as we must make the network deeper to handle longer prepositional chains instead of just looping

[53] We know that Transformers only move information between positions in a sequence at each layer boundary via cross/self-attention. (Csordás et al., 2022) argues the Transformer layers should be as deep as the deepest data dependency in the computatonal graph of the problem, in our case the parse tree, stating: "the network should be sufficiently deep, at least as deep as the deepest data dependency in the computational graph built from elementary operations (e.g., in the case of a parse tree, this is the depth of the tree)".

[54] See Appendix 13: Composition and Learning

[55] not surprising a shallow model can handle it as we have proved a flat, non-hierarchical approach is sufficient

[56] e.g. to ignore "pp det common_noun" and "pp proper_noun" when finding nouns in relationships with verbs, which allows the RASP model to get 100% on the ReCOGS test set, and 100% two of the three structural generalizations, and 92% on the obj-pp-to-subj-pp split

[57] We attempted one simple data augmentation (see Methods) which was not yet successful but there are many augmentation directions to explore

Known Project Limitations

The Restricted Access Sequence Processing code is not optimized. Cannot yet predict attention heads and layers required from the select and aggregate operations performed like the RASP authors (Weiss et al., 2021) were able to do with their problems.

Grammar coverage (Zeller et al., 2023) is only valid when the expansions are rules your model can learn.[58] We specifically made use of this limitation in this paper but still caution anyone about it who might just take the grammar coverage metric away by itself.

The error analysis of the (Wu et al., 2024) baseline Encoder-Decoder Transformer on the obj_pp_to_subj_pp split does not yet attempt to explain the common case of multiple errors in the logical form.[59]

Today, we only provide a RASP solution for ReCOGS (Wu et al., 2024) here, not the semantically equivalent COGS[60].

Much deeper Transformer networks may be learning a tree-based grammar representation and not suffer from the predicted generalization issues observed in (Wu et al., 2024)'s baseline 2-layer Transformer and our intentionally non-tree RASP model.[61]

References

Yoshua Bengio, Jérôme Louradour, Ronan Collobert, and Jason Weston. 2009. Curriculum learning. In *Proceedings of the 26th Annual International Conference on Machine Learning*, ICML '09, page 41–48, New York, NY, USA. Association for Computing Machinery.

Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language models are few-shot learners.

David Chiang and Peter Cholak. 2022. Overcoming a theoretical limitation of self-attention.

Peter Clark, Oyvind Tafjord, and Kyle Richardson. 2020. Transformers as soft reasoners over language.

Róbert Csordás, Kazuki Irie, and Jürgen Schmidhuber. 2022. The neural data router: Adaptive control flow in transformers improves systematic generalization.

Grégoire Delétang, Anian Ruoss, Jordi Grau-Moya, Tim Genewein, Li Kevin Wenliang, Elliot Catt, Chris Cundy, Marcus Hutter, Shane Legg, Joel Veness, and Pedro A. Ortega. 2023. Neural networks and the chomsky hierarchy.

Jerry A. Fodor and Zenon W. Pylyshyn. 1988. Connectionism and cognitive architecture: A critical analysis. *Cognition*, 28(1):3–71.

Julie Franck, Glenda Lassi, Ulrich H. Frauenfelder, and Luigi Rizzi. 2006. Agreement and movement: A syntactic analysis of attraction. *Cognition*, 101(1):173–216.

Yoav Goldberg. 2019. Assessing bert's syntactic abilities.

John Hewitt and Christopher D. Manning. 2019. A structural probe for finding syntax in word representations. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4129–4138, Minneapolis, Minnesota. Association for Computational Linguistics.

O. Jespersen. 1954. *A Modern English Grammar on Historical Principles: Volume 2, Syntax (first volume)*. Otto Jespersen. Bradford and Dickens.

Najoung Kim and Tal Linzen. 2020. COGS: A compositional generalization challenge based on semantic interpretation. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 9087–9105, Online. Association for Computational Linguistics.

[58] If for example, as with our flat RASP model by design or as we hypothesize for (Wu et al., 2024)'s baseline Encoder-Decoder Transformer, the model cannot or will not learn the rule 'np_det pp np -> np_pp -> np' which recursively replaces noun phrases modified by prepositional phrases with a noun phrase, then grammar coverage will assume any prepositional phrase exposure is sufficient, which is evidently not true given the errors on prepositional phrase modification generalization splits reported here and by (Wu et al., 2024), (Kim and Linzen, 2020).

[59] e.g. agent index may be replaced by prepositional phrase noun but also a spurious theme relationship is added or the theme index is also corrupted

[60] nor the recently introduced extended version SLOG (Li et al., 2023)

[61] Nothing explored here rules that out and there is plenty of evidence outside the COGS task-related literature suggesting this will be the case: (Tenney et al., 2019) show the 24-layer BERT model seems to handle "POS tagging, parsing, NER, semantic roles, then coreference"; (Hewitt and Manning, 2019) "provid[e] evidence that entire syntax trees are embedded implicitly in deep models' [including BERT's] vector geometry", and (Goldberg, 2019) shows BERT excels at subject-verb agreement, "which [is] traditionally taken as evidence for the existence [of] hierarchical structure" (though e.g. in this work we see that ignoring distractor nouns in long-term dependencies does not require hierarchy or a deep understanding of syntax but simple rules like ignore "pp det common_noun" and "pp proper_noun" for finding noun-verb relationships in the logical form can allow for handling of such long-range dependencies). It is a goal that in future work to try to find data augmentation tricks or curriculum learning approaches that can force the model into such a learning mode for the COGS task at the minimum number of layers. On the other hand, (Petty et al., 2024) argue specifically for the COGS benchmark (semantically equivalent to ReCOGS which is derived from it) that increasing depth does not allow their Transformers to make progress on the structural generalization splits, even at depths up to 32 layers.

Tim Klinger, Luke Liu, Soham Dan, Maxwell Crouse, Parikshit Ram, and Alexander Gray. 2024. Compositional program generation for few-shot systematic generalization.

Brenden M Lake and Marco Baroni. 2023. Human-like systematic generalization through a meta-learning neural network. *Nature*, 623(7985):115–121.

Bingzhi Li, Lucia Donatelli, Alexander Koller, Tal Linzen, Yuekun Yao, and Najoung Kim. 2023. Slog: A structural generalization benchmark for semantic parsing.

Tal Linzen, Emmanuel Dupoux, and Yoav Goldberg. 2016. Assessing the ability of lstms to learn syntax-sensitive dependencies. *Transactions of the Association for Computational Linguistics*, 4:521–535.

Jackson Petty, Sjoerd van Steenkiste, Ishita Dasgupta, Fei Sha, Dan Garrette, and Tal Linzen. 2024. The impact of depth on compositional generalization in transformer language models.

Marc'Aurelio Ranzato, Sumit Chopra, Michael Auli, and Wojciech Zaremba. 2016. Sequence level training with recurrent neural networks.

David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. 1988. (1986) d. e. rumelhart, g. e. hinton, and r. j. williams, "learning internal representations by error propagation," parallel distributed processing: Explorations in the microstructures of cognition, vol. i, d. e. rumelhart and j. l. mcclelland (eds.) cambridge, ma: Mit press, pp. 318-362. In *Neurocomputing, Volume 1: Foundations of Research*. The MIT Press.

Lena Strobl, William Merrill, Gail Weiss, David Chiang, and Dana Angluin. 2024. What formal languages can transformers express? a survey. *Transactions of the Association for Computational Linguistics*, 12:543–561.

Ian Tenney, Dipanjan Das, and Ellie Pavlick. 2019. Bert rediscovers the classical nlp pipeline.

Marten van Schijndel, Aaron Mueller, and Tal Linzen. 2019. Quantity doesn't buy quality syntax with neural language models.

Gabriella Vigliocco and Janet Nicol. 1998. Separating hierarchical relations and word order in language production: is proximity concord syntactic or linear? *Cognition*, 68(1):B13–B29.

Gail Weiss, Yoav Goldberg, and Eran Yahav. 2021. Thinking like transformers.

Zhengxuan Wu, Christopher D. Manning, and Christopher Potts. 2024. Recogs: How incidental details of a logical form overshadow an evaluation of semantic interpretation.

Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler. 2023. Grammar coverage. In *The Fuzzing Book*. CISPA Helmholtz Center for Information Security. Retrieved 2023-11-11 18:18:06+01:00.

Hattie Zhou, Arwen Bradley, Etai Littwin, Noam Razin, Omid Saremi, Josh Susskind, Samy Bengio, and Preetum Nakkiran. 2023. What algorithms can transformers learn? a study in length generalization.

Arnold Zwicky. 2008. Agreement with nearest.

9 Notes

No AI tools were used by the author in the preparation of this manuscript with the exception of anything used in the backend by Google Scholar searches for literature and citations and Google searches for related material. AI writing aids were not used.

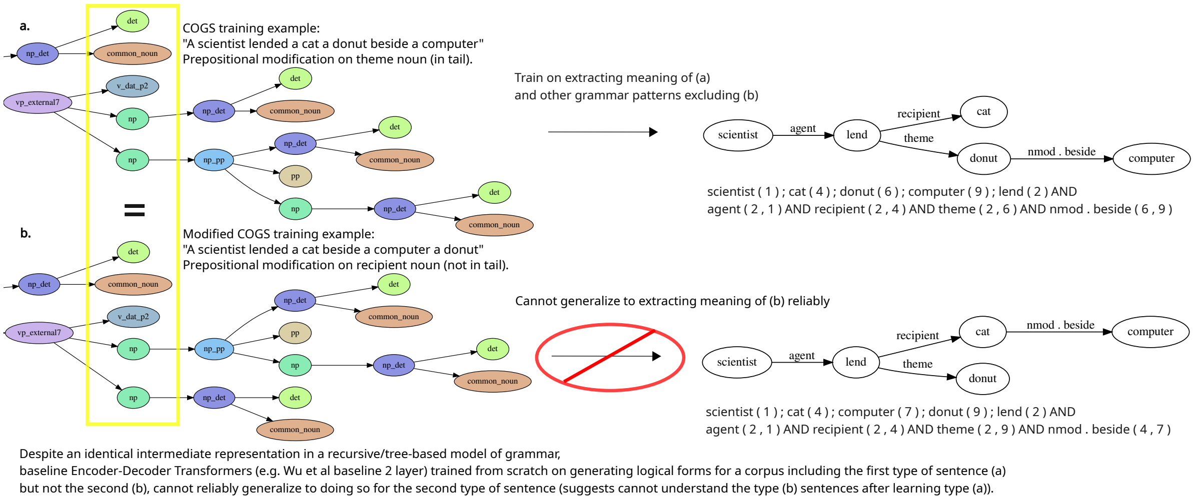


Figure 2: (Wu et al., 2024) Encoder-Decoder Transformer trained from scratch generalizing to new v_dat_p2 pp moved to recipient (from theme) split is as hard as the previously reported hardest generalization split as predicted by the flat/non-recursive/non-tree hypothesis by RASP modeling. *Actual COGS training example and modified version shown, which have different meanings but in a compositional solution are identical at a medium level of abstraction, as both are 'np v_dat_p2 np np'*[62]*, so model should be able to translate both to their appropriate logical forms.*

10 Figure 2: New difficult generalization split v_dat_p2 pp moved to recipient from theme as hard as previously reported most difficult split

[62]We want a model that generalizes from seeing a noun modified with a preposition in one position to others (e.g. by understanding the expansion 'np -> np_pp -> np_det pp np' but the RASP program shows such a tree-based representation is not necessary)

11 Appendix 1: Vocabulary and Grammar

The description of the COGS (Kim and Linzen, 2020) input vocab and grammar from the utilities of (Klinger et al., 2024)'s CPG project[63] was used as a reference by this project (we did not use the CPG code or anything from the CPG project other than the following Lark-compatible description of the COGS PCFG grammar).

Note this paper is focused on the ReCOGS task but the input sentences for ReCOGS are identical to the COGS input sentences (only the output format / logical form is changed).

```
start:
    s1 | s2 | s3 | s4 | vp_internal
s1: np vp_external
s2: np vp_passive
s3: np vp_passive_dat
s4: np vp_external4
vp_external:
   v_unerg | v_trans_omissible_p1
   | vp_external1 | vp_external2
   | vp_external3 | vp_external5
   | vp_external6 | vp_external7
vp_external1: v_unacc_p1 np
vp_external2:
   v_trans_omissible_p2 np
vp_external3:
   v_trans_not_omissible np
vp_external4:
   v_inf_taking to v_inf
vp_external5:
   v_cp_taking that start
vp_external6:
   v_dat_p1 np pp_iobj
vp_external7:
   v_dat_p2 np np
vp_internal: np v_unacc_p2
vp_passive: vp_passive1 | vp_passive2
          | vp_passive3 | vp_passive4
          | vp_passive5 | vp_passive6
          | vp_passive7 | vp_passive8
vp_passive1:
    was v_trans_not_omissible_pp_p1
vp_passive2:
    was v_trans_not_omissible_pp_p2
          by np
vp_passive3:
    was v_trans_omissible_pp_p1
vp_passive4:
    was v_trans_omissible_pp_p2 by np
vp_passive5:
    was v_unacc_pp_p1
vp_passive6:
    was v_unacc_pp_p2 by np
vp_passive7:
    was v_dat_pp_p1 pp_iobj
vp_passive8:
    was v_dat_pp_p2 pp_iobj by np
vp_passive_dat:
    vp_passive_dat1
              | vp_passive_dat2
vp_passive_dat1:
    was v_dat_pp_p3 np
vp_passive_dat2:
    was v_dat_pp_p4 np by np
np:
    np_prop | np_det | np_pp
np_prop: proper_noun
np_det: det common_noun
np_pp: np_det pp np
pp_iobj: to np
det: "the" | "a"
pp: "on" | "in" | "beside"
was: "was"
by: "by"
to: "to"
that: "that"
common_noun:
    "girl" | "boy"
    | "cat" | "dog" | ...
proper_noun:
    "emma" | "liam"
    | "olivia" | "noah"
    | ...
v_trans_omissible_p1:
    "ate" | "painted" | "drew"
    | "cleaned" | ...
v_trans_omissible_p2:
    "ate" | "painted" | "drew"
    | "cleaned"  | ...
v_trans_omissible_pp_p1:
    "eaten" | "painted" | "drawn"
    | "cleaned" | ...
v_trans_omissible_pp_p2:
    "eaten" | "painted" | "drawn"
    | "cleaned" | ...
v_trans_not_omissible:
    "liked" | "helped" | "found"
    | "loved" | ...
v_trans_not_omissible_pp_p1:
    "liked" | "helped" | "found"
    | "loved"  | ...
v_trans_not_omissible_pp_p2:
    "liked" | "helped" | "found"
    | "loved" | ...
v_cp_taking:
    "liked" | "hoped" | "said"
    | "noticed" | ...
v_inf_taking:
    "wanted" | "preferred" | "needed"
    | "intended" | ...
v_unacc_p1:
    "rolled" | "froze" | "burned"
    | "shortened" | ...
v_unacc_p2:
    "rolled" | "froze" | "burned"
    | "shortened" | ...
v_unacc_pp_p1:
    "rolled" | "frozen" | "burned"
    | "shortened" | ...
v_unacc_pp_p2:
    "rolled" | "frozen" | "burned"
    | "shortened" | ...
v_unerg:
    "slept" | "smiled" | "laughed"
    | "sneezed" | ...
v_inf:
    "walk" | "run" | "sleep"
    | "sneeze" | ...
v_dat_p1:
    "gave" | "lended" | "sold"
    | "offered" | ...
v_dat_p2:
    "gave" | "lended" | "sold"
    | "offered" | ...
v_dat_pp_p1:
    "given" | "lended" | "sold"
    | "offered" | ...
v_dat_pp_p2:
    "given" | "lended" | "sold"
    | "offered" | ...
v_dat_pp_p3:
    "given" | "lended" | "sold"
    | "offered" | ...
v_dat_pp_p4:
    "given" | "lended" | "sold"
    | "offered" | ...
```

[63] https://github.com/IBM/cpg/blob/c3626b4e03bfc681be2c2a5b23da0b48abe6f570/src/model/cogs_data.py#L523

12 Appendix 2: RASP for relation right index ignoring distractor "pp np"

```
pp_sequence = \
indicator(pos_tokens == 2);
pp_one_after_mask = \
select(pp_sequence, 1, ==) and \
select(indices+1, indices, ==);

pp_one_after_sequence = \
aggregate(pp_one_after_mask, 1);
pp_one_after_mask = \
select(pp_one_after_sequence, 1, ==) and \
select(indices, indices, ==);

pp_two_after_mask = \
select(pp_sequence, 1, ==) and \
select(indices+2, indices, ==);

pp_two_after_sequence = \
aggregate(pp_two_after_mask, 1);
pp_two_after_mask = \
select(pp_two_after_sequence, 1, ==) and \
select(indices, indices, ==);

np_det_diag_mask = \
select(aggregate(np_det_mask, 1), 1, ==) and \
select(indices, indices, ==);

np_prop_diag_mask = \
select(aggregate(np_prop_mask, 1), 1, ==) and \
select(indices, indices, ==);

no_pp_np_mask = \
1 - aggregate((pp_one_after_mask and np_prop_diag_mask) or \
(pp_two_after_mask and np_det_diag_mask), 1);

nps_without_pp_prefix_indices = \
selector_width(select(NOUN_MASK*no_pp_np_mask, 1, ==) and \
select(indices, indices, <=))*NOUN_MASK*no_pp_np_mask;

left_idx = \
aggregate(select(indices, left_idx_in_nps_zero_based, ==), input_indices_sorted);
right_idx = \
aggregate(select(nps_without_pp_prefix_indices, after_intro_idx, ==), indices); # <--
```

13 Appendix 3: Error Analysis for (Wu et al., 2024) baseline Transformer - parsing sentences with Lark and tagging sentences as agent left-of-verb or not

The errors from n=10 fresh training and evaluation runs of the baseline (Wu et al., 2024) Encoder-Decoder Transformer on their ReCOGS_pos train.tsv and tested on their unmodified gen.tsv were analyzed for the obj_pp_to_subj_pp split. All the input sentences and output logical forms as well as the ground truth logical forms were logged during the run. The input sentences were parsed by the Lark parser[64] against the COGS input grammar which allowed categorizing each sentence by its verb type [65]. The author manually inspected each of verb type patterns and categorized them by the position of the agent and theme relative to the verb (see code below) and used Lark to assign agent, theme sides based on the verb type using that mapping.

To focus the analysis, we considered only single verb cases[66] and ignored sentences with complement phrases. Then, of the sentences with the model generating an invalid logical form assessed by Semantic Exact Match, we focused on examples with a single error in one of the logical form parts (e.g. agent, theme, recipient, or nmod relationships).[67]

Our hypothesis is in terms of nouns with a logical form relationship to a verb or other noun, where the relationship could be agent, theme, or recipient. Since the obj_pp_to_subj_pp split is in terms of subject vs object prepositional modification (instead of agent, recipient, or theme), we use the subset of sentences within this split where the agent is to the left of the verb and modified by a prepositional phrase as it corresponds to the subject in that case.

```
# used grammar string defined in Appendix 1.
parser = Lark(grammar, start='start')

# 1st NP agent verbs (non CP)
# "v_trans_omissible_p1": "agent",
# "v_trans_omissible_p2": "agent",
# "v_trans_not_omissible": "agent",
# "v_cp_taking": "agent",
# "v_inf_taking": "agent",
# "v_unacc_p1": "agent",
# "v_unerg": "agent",
# "v_inf": "agent",
# "v_dat_p1": "agent",
# "v_dat_p2": "agent",
agent_left_of_verb_verb_type_set = \
set(["v_trans_omissible_p1", "v_trans_omissible_p2",
"v_trans_not_omissible", "v_cp_taking", "v_inf_taking",
"v_unacc_p1", "v_unerg", "v_inf", "v_dat_p1",
"v_dat_p2"])

theme_left_of_verb_verb_type_set = set(
    ["v_trans_omissible_pp_p1",
     "np v_unacc_p2",
     "v_unacc_pp_p1",
     "v_unacc_pp_p2",
     "v_trans_omissible_pp_p2",
     "v_trans_not_omissible_pp_p1",
     "v_trans_not_omissible_pp_p2",
     "v_dat_pp_p1",
     "v_dat_pp_p2"
     ])

theme_right_of_verb_verb_type_set = set([
    "v_unacc_p1",
    "v_trans_omissible_p2",
    "v_trans_not_omissible",
])
theme_middle_of_dative_verb_type_set = \
  set(["v_dat_pp_p4", "v_dat_p1"])

# for enforcing during the check of our hypothesis
# a stricter expectation that the closest prepositional noun
# to the left of the verb is the misassigned agent
# (not just any prepositional noun)
def \
get_verbs_with_pps_before_\
  and_last_noun_before_first_verb_index(lark_tree_root):
  nodes = [lark_tree_root]
  verbs = []
  terminals_before_count = 0
  pps_before_counts = []
  pps_before_count = 0
  last_noun_before_first_verb_index = None
  while len(nodes) > 0:
    node = nodes[-1]
    nodes = nodes[:-1]
    node_type = node.data[:]
    if node_type[:2] == 'v_':
      pps_before_counts.append(pps_before_count)
      verbs.append(node_type)
    children = []
    for child in node.children:
      # it is a tree, no need to check for revisits
      children.append(child)
    # need to visit in a particular order
    # to not just get verbs
    # but pp before count and
    # the last noun before the first verb
    # in the one verb case this does not matter
    children.reverse()
    # but we may want to return verbs
    # in the order they appear in the sentence
    for node in children:
      nodes.append(node)
    if node_type[:]
      in ["common_noun", "proper_noun"] and len(verbs) == 0:
      # no need to subtract 1 here
      # as before incrementing below
      last_noun_before_first_verb_index = \
        terminals_before_count
    # only increment on terminals
    if len(children) == 0:
      terminals_before_count += 1
    if node_type[:] == "pp":
      pps_before_count += 1
  return verbs, pps_before_counts,
    last_noun_before_first_verb_index

def get_theme_side(lark_tree_root):
    verb_type = get_verbs(lark_tree_root)[0]
    if verb_type in theme_right_of_verb_verb_type_set:
        return "right"
    elif verb_type in theme_left_of_verb_verb_type_set:
        return "left"
    elif verb_type in theme_middle_of_dative_verb_type_set:
        return "middle"
    return None

def get_agent_side(lark_tree_root):
    verb_type = get_verbs(lark_tree_root)[0]
    if verb_type != None and
      verb_type not in agent_left_of_verb_verb_type_set:
        return "right or middle"
    elif verb_type in agent_left_of_verb_verb_type_set:
        return "left"
    return None
```

[64] https://github.com/lark-parser/lark

[65] Code to analyze the errors is at: https://colab.research.google.com/drive/1Z0_EXV-bvmO2mRcnHmDpFuHIv4KHOpz-

[66] the 2 verb case of v_inf and v_inf_taking are being analyzed and will be included

[67] Of the single relationship errors, we categorized them by a description of the position of both the agent and theme relative to the verb in that sentence (agent was considered to be either left OR "right or middle"; theme could be left, right, or middle) and what relationship had the error. Complement phrase examples were excluded to focus on predicting the form of the error on simpler examples.

14 Appendix 4: v_dat_p2 recipient pp-modification for generalization assessment and data augmentation attempt

We test generalization by the (Wu et al., 2024)'s default Transformer which has been trained on ‘np v_dat_p2 np np pp np‘ but not ‘np v_dat_p2 np pp np np‘ prepositional modifications. The following 328 examples were derived[68] from the existing

https://github.com/frankaging/ReCOGS/blob/1b6eca8ff4dca5fd2fb284a7d470998af5083beb/recogs_positional_index/train.tsv ,

by modifying 328 existing single-pp v_dat_p2 lines in train.tsv to simply move the prepositional phrase from the 3rd NP (theme) in the ‘np v_dat_p2 np np‘ (agent, recipient, theme) to the 2nd NP (recipient), e.g. copying and modifying the line "Liam gave the monkey a chalk in the container ." to "Liam gave the monkey in the container a chalk .".

So all the words and the grammar are otherwise familiar. This is similar to the existing ‘obj_pp_to_subj_pp‘ generalization (Wu et al., 2024) reports on. All modified rows available in the notebook link in the footnote.

[68]Notebook: https://colab.research.google.com/drive/1IDs0EwIMp2wtLHk4KqnuGhuT3G14QEG1?usp=sharing

15 Appendix 5: Restricted Access Sequence Processing word-level (post-embedding) token program/model design

You can run a demo and see the autoregressive output
(or just visit https://colab.research.google.com/drive/1FS4tucZ92YR6VmhSva9pJekm2X7nnHWP?usp=sharing)

```
git clone https://github.com/willy-b/learning-rasp.git
python recogs_examples_in_rasp.py
```

The script will show performance on (Wu et al., 2024) ReCOGS_pos data by default, run with "–use_dev_split", "–use_gen_split" , or "–use_test_split" to see it run on those and give a running score every 10 rows.

For ReCOGS, intending to perform well on Semantic Exact Match, we took a simple, flat, non-tree, non-recursive approach which was able to get approximately 100% semantic exact match (and string exact match) on the full test set, and 99.6% semantic exact match on the out-of-distribution generalization set of the real ReCOGS dataset[69]

We took the RASP native sequence tokens, and first did a Transformer learned-embedding compatible operation and created 1 part-of-speech and 4 extra verb-type sequences (because each word in the COGS vocabulary may actually serve multiple POS roles; up to four different verb types as in the case of "liked"

which can serve as v_trans_not_omissible, v_trans_not_omissible_pp_p1, v_trans_not_omissible_pp_p2, and v_cp_taking types).

The five extra sequences serve to associate each word with one or more of the following part-of-speech/verb type roles:

```
det: 1
pp: 2
was: 3
by: 4
to: 5
that: 6
common_noun: 7
proper_noun: 8
v_trans_omissible: 9
v_trans_omissible_pp: 10
v_trans_not_omissible: 11
v_trans_not_omissible_pp: 12
v_cp_taking: 13
v_inf_taking: 14
v_unacc: 15
v_unerg: 16
v_inf: 17
v_dat: 18
v_dat_pp: 19
v_unacc_pp: 20
```

[69]word-level token Restricted Access Sequence Processing solution: https://github.com/willy-b/learning-rasp/blob/e97282e18b07004bf714b5c9bb5883090a2ff8e3/word-level-pos-tokens-recogs-style-decoder-loop.rasp

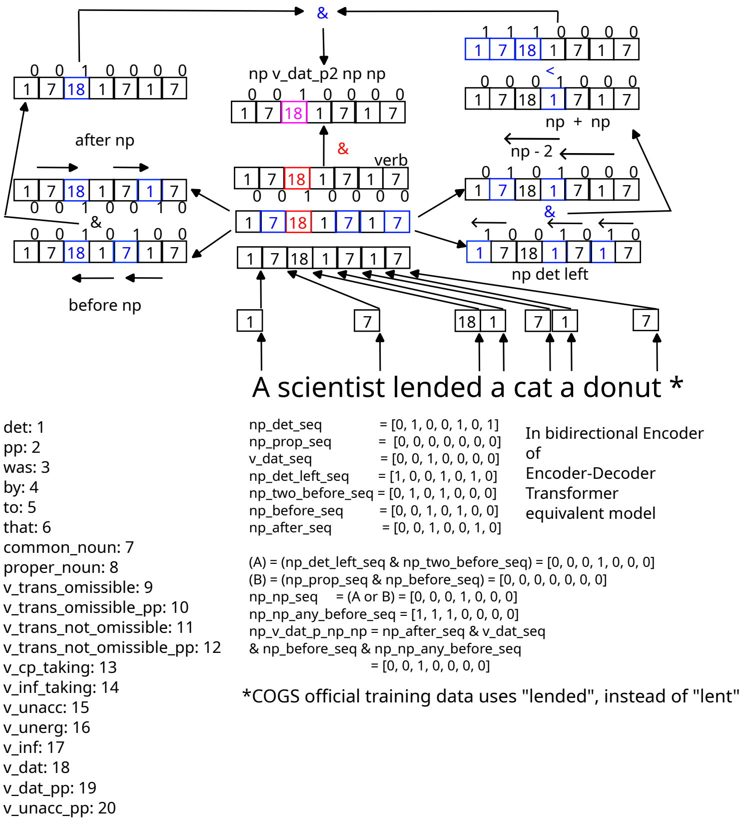


Figure 3: Example RASP model flat grammar pattern matching, for np v_dat_p2 np np, for a matching sentence.

Each of the five sequences comes from a separate map. Since in RASP a map could only have a single value per key, and since individual words had up to four different verb roles (as in "liked" which had 4).

Upon these five parallel, aligned, sequences we used a series of attention head compatible operations to recognize multi-token patterns (see below) corresponding to grammatical forms (listed below).

```
np_det_mask = select(7, pos_tokens, ==)
and select(pos_tokens, 1, ==)
and select(indices+1, indices, ==);
np_prop_mask = select(8, pos_tokens, ==) and
select(indices, indices, ==);
np_det_sequence = aggregate(np_det_mask, 1);
np_prop_sequence = aggregate(np_prop_mask, 1);
np_det_after = select(np_det_sequence, 1, ==) and
select(indices+1, indices, ==);
np_prop_after = select(np_prop_sequence, 1, ==) and
select(indices+1, indices, ==);
np_after_mask = np_det_after or np_prop_after;
np_after_sequence = aggregate(np_after_mask, 1);
np_after_mask = select(np_after_sequence, 1, ==) and
select(indices,indices, ==);
# ...

# np v_unerg
# e.g. [1,7,16]
set example ["the", "guest", "smiled"]
v_unerg_mask = select(16, pos_tokens_vmap1, ==) and
select(indices, indices, ==);
np_v_unerg = aggregate(np_after_mask and v_unerg_mask, 1);
```

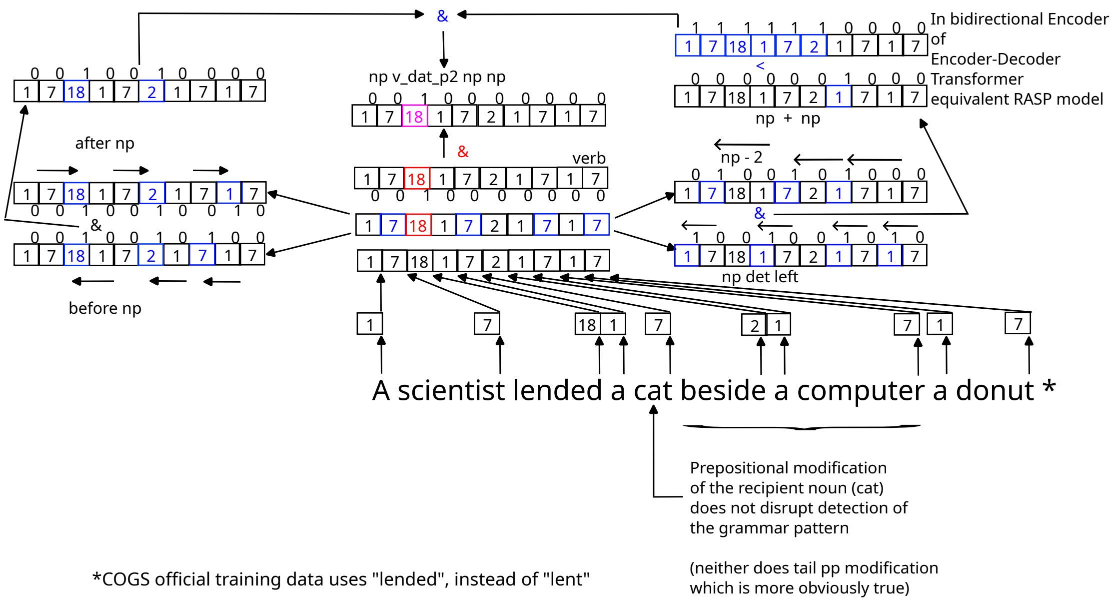


Figure 4: Example RASP model flat grammar pattern matching, for np v_dat_p2 np np, for a matching sentence, despite pp modification of middle recipient noun.

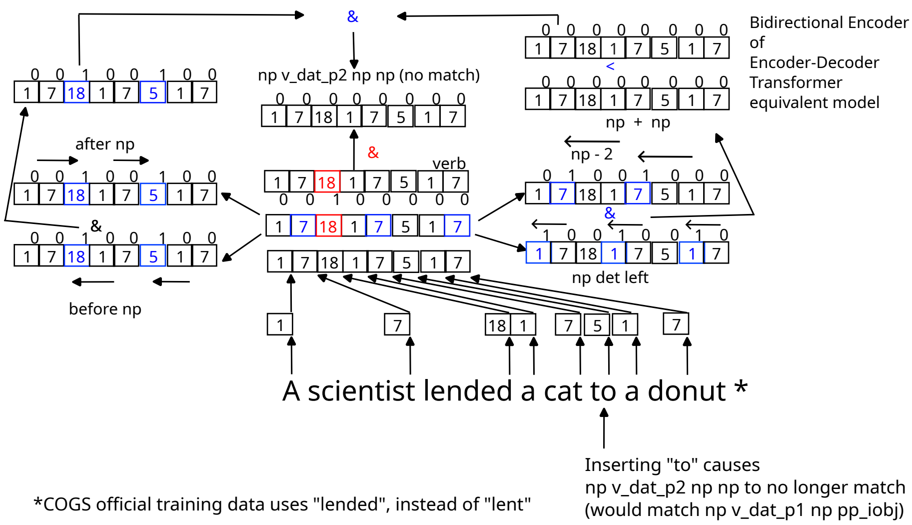


Figure 5: Example RASP model flat grammar pattern matching, for the pattern np v_dat_p2 np np, for a non-matching sentence.

These patterns are not causal because their use/input/output is masked to the input section of the sequence, so would take part in the Encoder of the Encoder-Decoder only(all operations outside the input mask in the word-level token RASP solution used in this paper are directly or indirectly causally masked and we built symbol by symbol in a causal autoregressive way). **We could have added an explicit causal mask to each operation but for efficiency and simplicity of the code omitted it when we are doing it implicitly by taking only the last sequence position (we also acausally aggregate so that all sequence positions have the same value as the last sequence position to make it easier to read the output – RASP interpreter will just print it as one position if they are all equal and we only take one position).**

Also, the author thinks many of these RASP steps could be consolidated. The goal here was to first prove by construction that a non-recursive, flat RASP program could get approximately 100% Semantic Exact Match on all the ReCOGS generalization splits (we only missed one split by a little due to two week time constraint, insufficient time to add all rules).

Introduction of variables at the beginning of the ReCOGS logical form (e.g. in the logical form for "a boy painted the girl", we have "boy (1) ; * girl (4) ; paint (2) AND agent (2 , 1) AND theme (2 , 4)" , the variable introduction is "boy (1) ; * girl (4) ; paint (2)" before the "AND"). We took a simple approach and sorted the input sequence with nouns before verbs and determiners, fillers last (with determiners and fillers not having any corresponding entry in the output sequence). We then count nouns and verbs in the input and count nouns and verbs in the output and determine if we have introduced all the nouns and verbs.

(The sections below must be updated for PR#7 which adds complement phrase support and complicated the approach somewhat)

Example counting how many nouns and verbs we have output (introduced as variables) so far (to determine what we need to output for next token):

```
nv_in_output_sequence =
OUTPUT_MASK*(indicator(pos_tokens == 7 or pos_tokens == 8) +
indicator(pos_tokens_vmap1 == 9 or pos_tokens_vmap2 == 10 or
pos_tokens_vmap1 == 11 or pos_tokens_vmap2 == 12 or pos_tokens_vmap3 == 13 or
pos_tokens_vmap4 == 14 or pos_tokens_vmap1 == 15 or pos_tokens_vmap1 == 16 or
pos_tokens_vmap1 == 17 or pos_tokens_vmap1 == 18 or pos_tokens_vmap2 == 19 or
pos_tokens_vmap2 == 20 or pos_tokens_vmap1==21));
nv_in_output_count = selector_width(select(nv_in_output_sequence, 1, ==));
# causal operation as we use only last sequence position
```

How variables are introduced with their index (omitted sorting of input and other operations that can be read in the code and are less important; anything acausal is restricted to input sequence section (Encoder)): (only value at last sequence position is used; causal)

```
# introducing variables
output = "";
# definite article word handling
before_target_word_index = aggregate(select(indices, nv_in_output_count, ==), inp
has_star = aggregate(select(indices, before_target_word_index, ==), tokens) == "t
last_output_is_star = aggregate(select(indices, length-1, ==), tokens) == "*";

input_nv_sorted_by_type = input_tokens_sorted_by_type * (input_noun_mask_sorted +
target_word_token = aggregate(select(indices, nv_in_output_count, ==), normalize_
if (not has_star or last_output_is_star) else "*";
# subtract 1 when matching for producing the index because we just output the add
target_word_index = aggregate(select(indices, nv_in_output_count-1, ==), input_in

output = target_word_token if ((num_tokens_in_output_excluding_asterisks % 5) ==
output = "(" if ((num_tokens_in_output_excluding_asterisks % 5) == 1) else output
output = target_word_index if ((num_tokens_in_output_excluding_asterisks % 5) ==
output = ")" if ((num_tokens_in_output_excluding_asterisks % 5) == 3) else output
# note that when nv_in_output_count == nv_in_input_count, we will add AND instead
output = (";" if (5 * nv_in_input_count - 1 > num_tokens_in_output_excluding_aste
if (num_tokens_in_output_excluding_asterisks % 5 == 4) else output;

# if we didn't have an input/output separator that needs to be output
output = "|" if num_pipes_in_output == 0 else output;
```

Note that "normalize_nv" is a lookup into a map that has no effect unless the word is a verb in which case it normalizes it to a standard suffix ("ate" to "eat" , "painted" to "paint", etc).

As you can see above, if we have not introduced all the variables, we determine our index into the sorted list of nouns and verbs (nouns before verbs), and using a MLP modeling modulus, compute index mod 5 and condition on that to output that noun/verb or parentheses or index as prediction for next token at last sequence position (all other sequence positions are ignored). Since we do ReCOGS_pos (semantically identical to random indices but avoid requiring random numbers) the index we use is the index of the original noun or verb in the original sequence. If we are still introducing variables, that is the end and we have our prediction for the next token.

If we are done introducing variables at that point in the decoder loop, we move on, and attention head compatible operations recognize templates in the five parallel part-of-speech / verb-type per location sequences for "v_trans_omissible_p1", "v_trans_omissible_p2", "v_trans_omissible_pp_p1", "v_trans_omissible_pp_p2", "v_trans_not_omissible", "v_trans_not_omissible_pp_p1", "v_trans_not_omissible_pp_p2", "v_cp_taking", "v_inf_taking", "v_unacc_p1", "v_unacc_p2", "v_unacc_pp_p1", "v_unacc_pp_p2", "v_unerg", "v_dat_p2", "v_dat_pp_p1", "v_dat_pp_p2", "v_dat_pp_p3", "v_dat_pp_p4".

Here are a couple of examples of patterns, to see how it looks if we support 1 verb pattern per input (no complement phrase recursion; which can be easily handled how we handle other things we loop over, looping over current phrase and masking and processing), which is sufficient to get approximately 100% on all entries that do not use complement phrases (e.g. "so-and-so noticed that (full input here)"):

```
# define the pattern
# ... (just showing one example, np_prop_mask and np_before_mask are attention masks defined earlier)
# np v_dat_p2 np np
# e.g. [8,18,1,7,1,7]
set example ["ella","sold","a","customer","a","car"]
np_np_sequence = aggregate((np_prop_mask and np_before_mask) or (np_det_left_mask and np_two_before_mask), 1);
# would not support prepositional phrase modification on middle NP
#np_np_before_mask = select(np_np_sequence, 1, ==) and select(indices-1, indices, ==);
np_np_any_before_mask = select(np_np_sequence, 1, ==) and select(indices, indices, >);
# acausal is ok
# in INPUT sequence (encoder part, not decoder),
# would mask further if we wanted to do multiple templates per input or
# something outside the supported grammar (COGS without complement phrase
# recursion is supported here)
np_np_any_before_sequence = aggregate(np_np_any_before_mask, 1);
np_np_any_before_mask = select(np_np_any_before_sequence, 1, ==) and select(indices, indices, ==);
np_v_dat_p_np_np = aggregate(np_after_mask and v_dat_mask and np_before_mask and np_np_any_before_mask, 1);
# Example: np_v_dat_p_np_np(['ella', 'sold', 'a', 'customer', 'a', 'car']) = [0, 1, 0, 0, 0, 0] (ints)
# Example: np_v_dat_p_np_np([8, 18, 1, 7, 1, 7]) = [0, 1, 0, 0, 0, 0] (ints)

# ...

# check the pattern and set the template name
any_np_v_trans_omissible = aggregate(select(np_v_trans_omissible, 1, ==), 1);
template_name = "v_trans_omissible_p1"
if (any_np_v_trans_omissible == 1) else template_name;

# ...

any_v_dat_p2 = aggregate(select(np_v_dat_p_np_np, 1, ==), 1);
template_name =
  "v_dat_p2" if (any_v_dat_p2 == 1) else template_name;

# ...

any_v_dat_pp_p4 =
  aggregate(select(np_was_v_dat_pp_np_by_np, 1, ==), 1);
template_name =
  "v_dat_pp_p4" if (any_v_dat_pp_p4 == 1) else template_name;

# must be checked after P4
any_v_dat_pp_p2 =
  aggregate(select(np_was_v_dat_pp_to_np_by_np, 1, ==), 1);
template_name =
  "v_dat_pp_p2" if (any_v_dat_pp_p2 == 1) else template_name;

# template name is used to lookup
# the number of verb relationships to output and
# what they are for each index
# e.g. ["theme", "agent"] vs.
# ["agent", "recipient", "theme"] etc
```

The rest of this applies to just values used from the last sequence location (output is prediction for next symbol).

Based on the template recognized, we lookup the template size for number of relationships (theme, recipient, agent) for that verb type:

```
def template_size(template_name) {
  template_sizes = {
 "": 0,
 "v_trans_omissible_p1": 1,
 "v_trans_omissible_p2": 2,
 "v_trans_omissible_pp_p1": 1,
 "v_trans_omissible_pp_p2": 2,
 "v_trans_not_omissible": 2,
 "v_trans_not_omissible_pp_p1": 1,
 "v_trans_not_omissible_pp_p2": 2,
 "v_cp_taking": 2,
 "v_inf_taking": 4,
 "v_unacc_p1": 2,
 "v_unacc_p2": 1,
 "v_unacc_pp_p1": 1,
 "v_unacc_pp_p2": 2,
 "v_unerg": 1,
# "v_inf": 1,
 "v_dat_p1": 3,
 "v_dat_p2": 3,
 "v_dat_pp_p1": 2,
 "v_dat_pp_p2": 3,
 "v_dat_pp_p3": 2,
 "v_dat_pp_p4": 3
  };
  # v_inf_taking includes v_inf and an extra verb is why it is 4 instead of 2
  return template_sizes[template_name];
}
```

Details are in the code, but we compute at the last sequence position (in parallel) the number of relationships output for the verb so far, and for the current relationship which token within that multi-token process (e.g. the word "agent" or the open parenthesis "(" or the left index, or the comma, or right index, close parenthesis ")", "AND", etc) we are on.

Like we computed at the last sequence position the number of nouns and verbs in the output once we are finished introducing nouns and verbs (this would be a little different with complement phrases[70], we compute the number of agent,theme,recipient,xcomp,ccomp entries in the output:

```
atrxc_in_output_sequence = OUTPUT_MASK*(indicator(tokens == "agent"
or tokens == "theme"
or tokens=="recipient"
or tokens=="xcomp" or tokens =="ccomp"));
# agent_theme_recipient_xcomp_ccomp_output_count is the number of relationships we have output
agent_theme_recipient_xcomp_ccomp_output_count =
selector_width(select(atrxc_in_output_sequence, 1, ==));
after_intro_idx =
(nv_in_output_count - nv_in_input_count + \
agent_theme_recipient_xcomp_ccomp_output_count) \
if nv_in_output_count >= nv_in_input_count else 0;
after_intro_num_tokens_in_output_excluding_asterisks =
num_tokens_in_output_excluding_asterisks - ((5 * nv_in_input_count));
```

[70]see Complement phrase support work completed in https://github.com/willy-b/learning-rasp/pull/7

We use all those different values which are computed independently from the input sequence and so do not add depth/layer requirements as many of the ones which involve accessing the sequence can be done in parallel. We then use the verb-type and relationship index to the relationship into a map to get the current relationship to output (as some verb types output agent first, some output theme, etc):

```
template_mapping1 = {
 "": "",
 "v_trans_omissible_p1": "agent",
 "v_trans_omissible_p2": "agent",
 "v_trans_omissible_pp_p1": "theme",
 "v_trans_omissible_pp_p2": "theme",
 "v_trans_not_omissible": "agent",
 "v_trans_not_omissible_pp_p1": "theme",
 "v_trans_not_omissible_pp_p2": "theme",
 "v_cp_taking": "agent",
 "v_inf_taking": "agent",
 "v_unacc_p1": "agent",
 "v_unacc_p2": "theme",
 "v_unacc_pp_p1": "theme",
 "v_unacc_pp_p2": "theme",
 "v_unerg": "agent",
 "v_inf": "agent",
 "v_dat_p1": "agent",
 "v_dat_p2": "agent",
 "v_dat_pp_p1": "theme",
 "v_dat_pp_p2": "theme",
 "v_dat_pp_p3": "recipient",
 "v_dat_pp_p4": "recipient"
};
```

Outputting the verb relationships we must skip over any "pp np" as possible agents, themes, or recipients to avoid getting confused by noun phrases added by prepositional modification (believed by the author to be the cause of the issue with obj pp to subj pp generalization by (Wu et al., 2024)'s Transformer).

```
pp_sequence = indicator(pos_tokens == 2);
pp_one_after_mask = select(pp_sequence, 1, ==) and select(indices+1, indices, ==);
pp_one_after_sequence = aggregate(pp_one_after_mask, 1);
pp_one_after_mask = select(pp_one_after_sequence, 1, ==) and select(indices, indices, ==);

pp_two_after_mask = select(pp_sequence, 1, ==) and select(indices+2, indices, ==);
pp_two_after_sequence = aggregate(pp_two_after_mask, 1);
pp_two_after_mask = select(pp_two_after_sequence, 1, ==) and select(indices, indices, ==);

np_det_diag_mask = select(aggregate(np_det_mask, 1), 1, ==) and select(indices, indices, ==);
np_prop_diag_mask = select(aggregate(np_prop_mask, 1), 1, ==) and select(indices, indices, ==);

no_pp_np_mask =
1 - aggregate((pp_one_after_mask and np_prop_diag_mask) or
(pp_two_after_mask and np_det_diag_mask), 1);
nps_without_pp_prefix_indices = \
selector_width(select(NOUN_MASK*no_pp_np_mask, 1, ==) and \
select(indices, indices, <=))*NOUN_MASK*no_pp_np_mask;

left_idx = aggregate(select(indices, \
left_idx_in_nps_zero_based, ==), input_indices_sorted);
right_idx = aggregate(select(nps_without_pp_prefix_indices, after_intro_idx, ==), indices);

# points to 2nd verb for xcomp for v_inf_taking_v_inf
right_idx = aggregate(select(indices, (nv_in_output_count-1), ==), input_indices_sorted)
if (template_name == "v_inf_taking" and after_intro_idx == 2) else right_idx;

# points to 1st noun for 2nd v_inf agent in v_inf_taking_v_inf
right_idx = aggregate(select(indices, 0, ==), input_indices_sorted)
if (template_name == "v_inf_taking" and after_intro_idx == 3) else right_idx;

# ...

after_intro_target_token = left_idx
if (after_intro_num_tokens_in_output_excluding_asterisks % 7 == 2)
  else after_intro_target_token;

after_intro_target_token = ","
if (after_intro_num_tokens_in_output_excluding_asterisks % 7 == 3)
  else after_intro_target_token;

after_intro_target_token = right_idx
if (after_intro_num_tokens_in_output_excluding_asterisks % 7 == 4)
else after_intro_target_token;

after_intro_target_token = ")"
if (after_intro_num_tokens_in_output_excluding_asterisks % 7 == 5)
else after_intro_target_token;
```

```
after_intro_target_token =
  "AND"
if
  (after_intro_num_tokens_in_output_excluding_asterisks % 7
  == 6
and
  not (template_mapping_output == ""))
else after_intro_target_token;

# ...
```

After outputting all verb relationships, we consider whether we have prepositional phrase noun modifiers to record in the logical form.

For outputting prepositional relationships ("nmod . in", "nmod . on", "nmod . beside"), we do a similar approach, counting prepositional phrases in the input, counting how many nmods we have output, figuring out which one is currently being output:

```
pps_in_input_sequence = INPUT_MASK*(indicator(pos_tokens == 2));
pps_in_input_count = selector_width(select(pps_in_input_sequence, 1, ==));
pps_index = pps_in_input_sequence*selector_width(select(pps_in_input_sequence, 1, ==)
and select(indices,indices, <=));
nmods_and_pps_in_output_sequence =
  OUTPUT_MASK*(
    indicator(tokens == "nmod . in" or tokens == "nmod . beside" or tokens=="nmod . on")
  );
nmods_and_pps_in_output_count =
  selector_width(select(nmods_and_pps_in_output_sequence, 1, ==));

current_pp = aggregate(select(pps_index, nmods_and_pps_in_output_count+1, ==), tokens) if pps_in_input_count > 0 else "";
current_pp = "" if current_pp == 0 else current_pp;
current_nmod_token =
("nmod . " + current_pp) if (pps_in_input_count > 0 and not (current_pp == 0)
and after_intro_num_tokens_in_output_excluding_asterisks % 7 == 0) else "";
current_nmod_token = "(" if after_intro_num_tokens_in_output_excluding_asterisks % 7 == 1 else current_nmod_token;
current_nmod_token =
(aggregate(select(pps_index, nmods_and_pps_in_output_count, ==), indices)-1) if pps_in_input_count > 0
and after_intro_num_tokens_in_output_excluding_asterisks % 7 == 2 else current_nmod_token;
current_nmod_token = ","
if after_intro_num_tokens_in_output_excluding_asterisks % 7 == 3 else current_nmod_token;
after_nmod_idx =
aggregate(select(pps_index, nmods_and_pps_in_output_count, ==), indices)+1;
token_at_after_nmod_idx =
aggregate(select(indices, after_nmod_idx, ==), tokens);
after_nmod_idx =
  (after_nmod_idx + 1)
  if (token_at_after_nmod_idx == "the" or token_at_after_nmod_idx == "a")
  else after_nmod_idx;
current_nmod_token = (after_nmod_idx)
if pps_in_input_count > 0
and after_intro_num_tokens_in_output_excluding_asterisks % 7 == 4 else current_nmod_token;
current_nmod_token = ")"
if after_intro_num_tokens_in_output_excluding_asterisks % 7 == 5
else current_nmod_token;
current_nmod_token =
("AND" if nmods_and_pps_in_output_count < pps_in_input_count else "")
if after_intro_num_tokens_in_output_excluding_asterisks % 7 == 6
else current_nmod_token;
after_intro_and_relationships_nmod_token =
current_nmod_token if nmods_and_pps_in_output_count <= pps_in_input_count else "";
num_tokens_in_nmod_section =
after_intro_num_tokens_in_output_excluding_asterisks - template_size(template_name)*7 + 1;
```

See code for full details (for simplicity this description was also written without discussing complement phrase handling). For all steps only the RASP outputs aligned with the input sequence (Encoder part of derived Transformer) or the very last sequence output (for next token in autoregressive generation) were used. For convenience of reading the aggregate operator was usually used acausally to assign all sequence outputs before the last one to the same value as the last (so only one value would be displayed).

You can run a demo and see the autoregressive output

(or just visit https://colab.research.google.com/drive/1FS4tucZ92YR6VmhSva9pJekm2X7nnHWP?usp=sharing)

```
git clone https://github.com/willy-b/learning-rasp.git
python recogs_examples_in_rasp.py
```

16 Appendix 6: Note on a Restricted Access Sequence Processing character-level token program / model design (NOT what is used in this paper but feasible)

Note, a proof of concept character level Restricted Access Sequence Processing model was done with a decoder loop (unlike word-level solution above, it was a sketch so did not limit to strictly causal operations which just require more careful indexing – using the value at the separator or the end of a word instead of pooling the same value to all letters in a word for example). Note that this one did not cover translating sentences in general into ReCOGS unlike the word-level solution as it is tedious and redundant but the core operations are possible and the author believes any solution at the word level can be mapped to a solution in character level tokens (out of scope for this paper to prove it).

Since it is a separate problem and adds a lot of complexity without bringing anything to bear on the main questions of the paper, I left a full implementation to the word-level tokens which were simpler and ran faster. The difference is one uses a similar approach started at [71] to assign all the letters in each word an index.

Word indices can be assigned using RASP to count separators occurring prior to each sequence location like:

(we also zero out the word index for the separators themselves)

```
word_indices = \
(1+selector_width(select(tokens, " ", ==)
and select(indices, indices, <=)))
*(0 if indicator(tokens == " ") else 1);
```

Then one can do an aggregation of the letters grouping by word index (this, which is NOT part of the techniques used in this paper for the word-level tokens solution, requires additional work (tedious not challenging) to do causally outside the input (in the decoder), one must sum forward so the word representation is always at the last letter of the word or separator instead of at all letters of the word, and that step is left out of the character-level demo and this discussion; whereas the word-level solution described above has a clear Encoder Decoder separation. This can be done so that the value which is then the same for all letters in each word, is unique to each word in the dictionary and can be looked up in a map to get word level attributes like part-of-speech and get back to the solution in the word-level tokens in Appendix 5 which was fully implemented. A simple approach (not necessarily recommended but works for proof of concept) that would work for small vocabularies (easily extended) is to use a map to lookup each letter of the alphabet to a log prime. Then the sum of the letters in a word (grouped by the word index which is the count of spaces/separators prior) is the sum of the log primes indexed by the alphabet index. Since the sum of logarithms of numbers is the same as the logarithm of the product of those numbers, this is equivalent to the logarithm of the product of a series of primes. Each prime in the product corresponds 1-to-1 to a letter in the alphabet, with the number of occurrences in the product corresponding to the number of times that letter occurs in the word. By uniqueness of prime number factorization this would map each multiset of letters to a single unique sum of log primes. Thus if you do not have words which are anagrams, all the letters in each word would be assigned a number that uniquely represented that word in the vocabulary. If you have anagrams you can do this step and then take the first and last letter and compute a separate number from that and add it to all the letters in the word.

Example lookup table for letters before aggregating by word index (not recommended but for proof of concept that one can go from character level tokens to word-specific numbers which can then be looked up as in the word-level token solution in Appendix 5 used throughout the paper):

[71] https://github.com/willy-b/learning-rasp/blob/main/decoder-loop-example-parse-into-recogs-style-variables.rasp

```
def as_num_for_letter_multiset_word_pooling(t) {
    # To be multiset unique, need logarithm of prime so that the sum aggregation
    # used in RASP corresponds to prime number factorization (sum of logs of primes is same as log of product of primes)
    # (we can do sum aggregation instead of mean by multiplying by length)
    # However RASP does not appear to support logarithms (underlying multilayer
    # perceptron can learn to approximate logarithms)
    #letter_to_prime_for_multiset_word_pooling = {"a": 2, "b": 3, "c": 5, "d": 7,
    #"e": 11, "f": 13, "g": 17, "h": 19, "i": 23, "j": 29, "k": 31, "l": 37,
    #"m": 41, "n": 43, "o": 47, "p": 53, "q": 59, "r": 61, "s": 67, "t": 71,
    #"u": 73, "v": 79, "w": 83, "x": 89, "y": 97, "z": 101, ".": 0,
    #" ": 0, ":": 0};
    map_letter_to_log_prime_for_pooling = {"a": 0.6931471805599453, "b": 1.0986122886681098,
    "c": 1.6094379124341003, "d": 1.9459101490553132, "e": 2.3978952727983707,
    "f": 2.5649493574615367, "g": 2.833213344056216, "h": 2.9444389791664403,
    "i": 3.1354942159291497, "j": 3.367295829986474, "k": 3.4339872044851463,
    "l": 3.6109179126442243, "m": 3.713572066704308, "n": 3.7612001156935624,
    "o": 3.8501476017100584, "p": 3.970291913552122, "q": 4.07753744390572,
    "r": 4.110873864173311, "s": 4.204692619390966, "t": 4.2626798770413155,
    "u": 4.290459441148391, "v": 4.3694478524670215, "w": 4.418840607796598,
    "x": 4.48863636973214, "y": 4.574710978503383, "z": 4.61512051684126,
    # we zero out tokens we want not to affect the identity of the word
    ".": 0, " ": 0, ":": -1, "(": -1, ")": -1, "0": -1, "1": -1, "2": -1,
    "3": -1, "4": -1, "5": -1, "6": -1, "7": -1, "8": -1, "9": -1, ";": -1,
    ",": -1};
    return map_letter_to_log_prime_for_pooling[t];
}
```

Pooling by word can then be done with:

```
pseudoembeddedwords = \
aggregate(select(word_indices, word_indices, ==), \
as_num_for_letter_multiset_word_pooling(tokens))*word_lengths;
```

(Per-character token example is not causally masked, we do causal strict-decoder-compatible solution for anything outside input sequence in the full word-level solution above just leaving out of this character-level sketch, which is NOT used in this paper. For the causal character level solution one would use the summed value at the end of the word or the separator instead, indexing relative to separators.)

Those values could then be looked up in a dictionary like in the completed word-level token solution to get part-of-speech, verb-type, etc, to derive a separate sequence which can be used for template matching as we successfully did with word-level tokens (see Appendix 5).

17 Appendix 7: Computing Grammar Coverage

First we use the grammar as it was generated as a probablistic context free grammar per (Kim and Linzen, 2020) using the full details put in Lark format by (Klinger et al., 2024) and converting it ourselves to a format compatible with (Zeller et al., 2023).

Note this starting point is not the grammar we claim the our Restricted Access Sequence Processing model implements or the Transformer actually learns as we argue the Transformer is learning a flat, non-tree solution to this simple grammar (not actually learning to collapse "np_det pp np" into "np" for example). First we compute grammar coverage relative to the PCFG approach that generated it, which mostly aligns with our RASP model. We also ignore terminals in this assessment of coverage, as stated earlier, when computing grammar coverage, we will report the grammar coverage over expansions that collapse all vocabulary leaves to a single leaf (for example not requiring that every particular proper noun or common noun be observed in a particular pattern, so long as one has and we can confirm the code treats them as equivalent; e.g. having tested "Liam drew the cat" and proven that "Liam" and "Noah" are treated as interchangeable proper nouns, and that "cat" and "dog" are treated as interchangeable common nouns by the RASP solution – not something one can assume for neural network solutions in general – means that confirming our solution produces the correct logical form for "Liam drew the cat" suffices to prove the RASP solution can handle "Noah drew the dog", which saves us a lot of work so long as we make sure to write our RASP solution such that noah/liam and cat/dog are indeed treated identically).

```
# Non-terminals only version of
# https://github.com/IBM/cpg/blame/
# c3626b4e03bfc681be2c2a5b23da0b48abe6f570
# /src/model/cogs_data.py#L529
# NOTE WE DO NOT ACTUALLY USE THIS GRAMMAR IN OUR MODEL,
# IT IS FOR UNDERSTANDING THE GRAMMAR WE ARE TRYING TO LEARN/MODEL

COGS_INPUT_GRAMMAR_NO_TERMINALS = {
"<start>": ["<s1>", "<s2>", "<s3>", "<s4>", "<vp_internal>"],
"<s1>": ["<np> <vp_external>"],
"<s2>": ["<np> <vp_passive>"],
"<s3>": ["<np> <vp_passive_dat>"],
"<s4>": ["<np> <vp_external4>"],
"<vp_external>": ["<v_unerg>", "<v_trans_omissible_p1>",
  "<vp_external1>", "<vp_external2>", "<vp_external3>",
  "<vp_external5>", "<vp_external6>", "<vp_external7>"],
"<vp_external1>": ["<v_unacc_p1> <np>"],
"<vp_external2>": ["<v_trans_omissible_p2> <np>"],
"<vp_external3>": ["<v_trans_not_omissible> <np>"],
"<vp_external4>": ["<v_inf_taking> <to> <v_inf>"],
"<vp_external5>": ["<v_cp_taking> <that> <start>"],
"<vp_external6>": ["<v_dat_p1> <np> <pp_iobj>"],
"<vp_external7>": ["<v_dat_p2> <np> <np>"],
"<vp_internal>": ["<np> <v_unacc_p2>"],
"<vp_passive>": ["<vp_passive1>", "<vp_passive2>",
  "<vp_passive3>", "<vp_passive4>", "<vp_passive5>",
  "<vp_passive6>", "<vp_passive7>", "<vp_passive8>"],
"<vp_passive1>":
  ["<was> <v_trans_not_omissible_pp_p1>"],
"<vp_passive2>":
  ["<was> <v_trans_not_omissible_pp_p2> <by> <np>"] ,
"<vp_passive3>": ["<was> <v_trans_omissible_pp_p1>"],
"<vp_passive4>":
  ["<was> <v_trans_omissible_pp_p2> <by> <np>"],
"<vp_passive5>": ["<was> <v_unacc_pp_p1>"],
"<vp_passive6>": ["<was> <v_unacc_pp_p2> <by> <np>"],
"<vp_passive7>": ["<was> <v_dat_pp_p1> <pp_iobj>"],
"<vp_passive8>": ["<was> <v_dat_pp_p2> <pp_iobj> <by> <np>"],
"<vp_passive_dat>":
  ["<vp_passive_dat1>", "<vp_passive_dat2>"],
"<vp_passive_dat1>": ["<was> <v_dat_pp_p3> <np>"],
"<vp_passive_dat2>": ["<was> <v_dat_pp_p4> <np> <by> <np>"],
"<np>": ["<np_prop>","<np_det>", "<np_pp>"],
"<np_prop>": ["<proper_noun>"],
"<np_det>": ["<det> <common_noun>"],
"<np_pp>": ["<np_det> <pp> <np>"],
"<pp_iobj>": ["<to> <np>"],
"<det>": [],
"<pp>": [],
"<was>": [],
"<by>": [],
"<to>": [],
"<that>": [],
"<common_noun>": [],
"<proper_noun>": [],
"<v_trans_omissible_p1>": [],
"<v_trans_omissible_p2>": [],
"<v_trans_omissible_pp_p1>": [],
"<v_trans_omissible_pp_p2>": [],
"<v_trans_not_omissible>": [],
"<v_trans_not_omissible_pp_p1>": [],
"<v_trans_not_omissible_pp_p2>": [],
"<v_cp_taking>": [],
"<v_inf_taking>": [],
  "<v_unacc_p1>": [],
  "<v_unacc_p2>": [],
  "<v_unacc_pp_p1>": [],
  "<v_unacc_pp_p2>": [],
  "<v_unerg>": [],
  "<v_inf>": [],
  "<v_dat_p1>": [],
  "<v_dat_p2>": [],
  "<v_dat_pp_p1>": [],
  "<v_dat_pp_p2>": [],
  "<v_dat_pp_p3>": [],
  "<v_dat_pp_p4>": [],
}
```

After parsing a sentence with the Lark parser, we can compute the expansions it covers with the following Python:

```
def generate_set_of_expansion_keys_for_lark_parse_tree(tree):
  nodes = [tree]
  expansions_observed = set()
  for node in nodes:
    current_node_label = node.data[:]
    children = node.children
    expansion = f"<{current_node_label}> ->"
    for child in children:
      # add expansion for current -> child
      child_node_label = child.data[:]
      expansion += f" <{child_node_label}>"
      # also process expansions from child
      nodes.append(child)
    if len(children) > 0:
      #print(f"{expansion}")
      expansions_observed.add(expansion)
  return expansions_observed
```

For example, for the sentence "the girl noticed that a boy painted the girl", we get

```
sentence = "the girl noticed that a boy painted the girl"
tree = parser.parse(sentence)
expansions_observed = \
generate_set_of_expansion_keys_for_lark_parse_tree(tree)
# <start> -> <s1>
# <s1> -> <np> <vp_external>
```

```
# <np> -> <np_det>
# <vp_external> -> <vp_external5>
# <np_det> -> <det> <common_noun>
# <vp_external5> -> <v_cp_taking> <that> <start>
# <start> -> <s1>
# <s1> -> <np> <vp_external>
# <np> -> <np_det>
# <vp_external> -> <vp_external2>
# <np_det> -> <det> <common_noun>
# <vp_external2> -> <v_trans_omissible_p2> <np>
# <np> -> <np_det>
# <np_det> -> <det> <common_noun>
```

At first we use TrackingGrammarCoverageFuzzer (from (Zeller et al., 2023)) to compute the set of all possible grammar expansions:

```
cogs_simplified_input_grammar_fuzzer = \
TrackingGrammarCoverageFuzzer(COGS_INPUT_GRAMMAR_SIMPLIFIED)

expected_expansions = \
cogs_simplified_input_grammar_fuzzer.max_expansion_coverage()
```

One can use this to get a sense of what it is possible to learn about the grammar from a particular set of examples

and what examples need to be seen at a minimum for any model to learn the task from scratch and could possibly help one design a minimum length dataset with low redundancy. Note for a Transformer model learning word embeddings / mapping to part-of-speech for each word, one would need to use the grammar with terminals to compute coverage. Here we want to argue something about our RASP model where we can ensure via implementation that all terminals in a category are treated identically (and we observe 100% semantic exact match for the related generalization splits for swapping words within a part of speech).

We can ask what % of the grammar without terminals is covered by the first 21 sentences from the COGS training set?

```
# https://raw.githubusercontent.com/frankaging/ReCOGS/
# 1b6eca8ff4dca5fd2fb284a7d470998af5083beb/cogs/train.tsv
nonsense_example_sentences = [
"A rose was helped by a dog",
"The sailor dusted a boy",
"Emma rolled a teacher",
"Evelyn rolled the girl",
"A cake was forwarded to Levi by Charlotte",
"The captain ate",
"The girl needed to cook",
"A cake rolled",
"The cookie was passed to Emma",
"Emma ate the ring beside a bed",
"A horse gave the cake beside a table to the mouse",
"Amelia gave Emma a strawberry",
"A cat disintegrated a girl",
"Eleanor sold Evelyn the cake",
"The book was lended to Benjamin by a cat",
"The cake was frozen by the giraffe",
"The donut was studied",
"Isabella forwarded a box on a tree to Emma",
"A cake was stabbed by Scarlett",
"A pencil was fed to Liam by the deer",
"The cake was eaten by Olivia"
]

all_expansions_observed_across_examples = set()

for sentence in nonsense_example_sentences:
  single_example_expansions = \
  generate_set_of_expansion_keys_for_lark_parse_tree\
   (parser.parse(sentence.lower()))
  all_expansions_observed_across_examples = \
  all_expansions_observed_across_examples.union\
   (single_example_expansions)

1 - len(set(expansions_expected) \
- all_expansions_observed_across_examples) / len(expansions_expected)
# 0.7115384615384616
```

Those 21 COGS input sentences cover 71% of the grammar. (Continued on next page.)

We can compare the first 21 sentences of COGS that to the 19 sentences used in developing the RASP program (then add one to cover basic prepositional phrases, and one more to cover complement phrases):[72]

```
handpicked_example_sentences = [
# non-recursive grammar rule examples only
# no prepositional phrases or complement phrases
# see link above all these examples
# each correspond to distinct rules in the code
"the girl was painted",
"a boy painted",
"a boy painted the girl",
"the girl was painted by a boy",
"a boy respected the girl",
"the girl was respected",
"the girl was respected by a boy",
"the boy grew the flower",
"the flower was grown",
"the flower was grown by a boy",
"the scientist wanted to read",
"the guest smiled",
"the flower grew",
"ella sold a car to the customer",
"ella sold a customer a car",
"the customer was sold a car",
"the customer was sold a car by ella",
"the car was sold to the customer by ella",
"the car was sold to the customer",
]

all_expansions_observed_across_examples = set()

for sentence in handpicked_example_sentences:
  single_example_expansions = \
  generate_set_of_expansion_keys_for_lark_parse_tree( \
   parser.parse(sentence.lower()))
  all_expansions_observed_across_examples = \
  all_expansions_observed_across_examples.union( \
    single_example_expansions)

1 - len(set(expansions_expected) \
- all_expansions_observed_across_examples) \
/ len(expansions_expected)
# 0.9230769230769231

# Those 19 rules cover 92.3% of the COGS input grammar
# (not necessarily 92.3% of examples as the examples
# are not evenly distributed across grammar rules).
# Let's see what rules are still missing:

set(expansions_expected) - \
  all_expansions_observed_across_examples
# tells us we need a prepositional phrase example!
#{'<np> -> <np_pp>',
# tell us we need prepositional phrase examples
# '<np_pp> -> <np_det> <pp> <np>',
# tells us we need complement phrase examples
# '<vp_external5> -> <v_cp_taking> <that> <start>',
# tells us we need complement phrase examples
# '<vp_external> -> <vp_external5>'}
```

[72](see https://github.com/willy-b/learning-rasp/blob/dca0bc6689b0454b75e5a46e77ffe66566ca7661/word-level-pos-tokens-recogs-style-decoder-loop.rasp#L568 for the full list and associated rules in the code as the RASP does not learn from examples but hand-coded rules)

We got to 92.3% grammar coverage in our 19 examples instead of COGS 71% in 21 examples.

And, it is telling us we are missing an example with prepositional phrases and complement phrases (see next examples)

Let us add a simple prepositional phrase example and complement phrase example:

```
handpicked_example_sentences = \
handpicked_example_sentences + \
["a boy painted the girl in a house"] + \
["the girl noticed that a boy painted the girl"]

handpicked_example_sentences
# ['the girl was painted',
#  'a boy painted',
#  'a boy painted the girl',
#  'the girl was painted by a boy',
#  'a boy respected the girl',
#  'the girl was respected',
#  'the girl was respected by a boy',
#  'the boy grew the flower',
#  'the flower was grown',
#  'the flower was grown by a boy',
#  'the scientist wanted to read',
#  'the guest smiled',
#  'the flower grew',
#  'ella sold a car to the customer',
#  'ella sold a customer a car',
#  'the customer was sold a car',
#  'the customer was sold a car by ella',
#  'the car was sold to the customer by ella',
#  'the car was sold to the customer',
#  'a boy painted the girl in a house',
#  'the girl noticed that a boy painted the girl'
#]
all_expansions_observed_across_examples = set()

for sentence in handpicked_example_sentences:
  single_example_expansions = generate_set_of_expansion_keys_for_lark_parse_tree(parser.parse(sentence.lower()))
  all_expansions_observed_across_examples = all_expansions_observed_across_examples.union(single_example_expansions)

1 - len(set(expansions_expected) - all_expansions_observed_across_examples) / len(expansions_expected)
# 1.0

set(expansions_expected) - all_expansions_observed_across_examples
# set()
```

(continued below)

Thus in 19 intentionally crafted sentences (each is in the RASP code with a corresponding rule) cover 92.3% of the grammar, using the coverage we can what we did not cover yet, and thus add two sentences to fill the reported gap and get to 100% .

However these coverage metrics are misleading when it comes to prepositional phrases as it would not suggest to include prepositional phrases in all positions, assuming they could be collapsed by the model back to ‘np‘ using ‘np -> np_pp -> np_det pp np‘ while (Wu et al., 2024) and our experiments suggest it is necessary to train with prepositional phrases explicitly in the different positions in the different verb patterns (see below).

Based on the finding earlier we believe that the only recursion learned is tail recursion in the decoder loop and that ‘np -> np_det | np_prop | np_pp‘ and ‘np_pp -> np_det pp np‘ is not actually performed as if the Encoder-Decoder Transformer were to learn a tree-based or recursive representation. If the Transformer had a tree based representation, it is predicted that the "v_dat_p2_pp_moved_to_recipient" would not be any harder than when the pp modification is on the theme, as ‘np v_dat_p2 np_det pp np np‘ can be transformed by the recursive grammar rule ‘np_det pp np -> np_pp -> np‘ to ‘np v_dat_p2 np np‘ on which it is already trained and has good performance, but instead it fails completely, and see also "Error Analysis for (Wu et al., 2024) baseline Encoder-Decoder Transformer on obj_pp_to_subj_pp split" where we observe that prepositional modification of a noun to the left of a verb it is the agent of causes the new prepositional phrase noun that becomes the closest noun to be mistaken for the agent, which is in contradiction to the model collapsing ‘np_det pp np‘ to ‘np‘ before matching the overall grammar pattern.

That said with a couple of simple rules that were not tree we were able to get 100% on the pp_recursion split (up to depth 12) and approximately 90% of the obj_pp_to_subj_pp split.

Modifying the grammar coverage to model this non-tree representation would be exciting to address in future work.

18 Appendix 8: Model Detail

For our Restricted Access Sequence Processing ReCOGS program, we used the RASP interpreter of (Weiss et al., 2021) to run our program. For RASP model design and details see Appendix 5. We use word-level tokens for all results in this paper.[73] Consistent with (Zhou et al., 2023) we use (Weiss et al., 2021)'s RASP originally used for modeling Transformer encoders to model an encoder-decoder in a causal way by feeding the autoregressive output back into the program. We only have aggregations with non-causal masks when that aggregation (or without loss of generality just before the aggregation product is used to avoid multiplying everywhere) is masked by an input mask restricting it to the sequence corresponding to the input.[74]

We used RASP maps to map word level tokens to part-of-speech and verb-type which is consistent with what can be learned in embeddings or the earliest layer of a Transformer (Tenney et al., 2019) and then did 19 different attention-head based template matches on a flat sequence[75] (no tree-based parsing, no recursive combination of terminals/non-terminals) to match the sentence to a template in the grammar (see "Appendix 5 - Restricted Access Sequence Processing word-level (post-embedding) token program/model design" and "Appendix 1: Vocabulary and Grammar"). The 19 types were based on (Zeller et al., 2023) grammar coverage of the COGS grammar (see Methods and "Appendix 7 - Computing Grammar Coverage").[76]

We also ensure that the mapping from words to part-of-speech and verb type is complete based on a published list of such mappings and put that into our hardcoded word embedding[77]

For training Transformers from scratch with randomly initialized weights using gradient descent for comparison with RASP predictions, we use scripts derived from those provided by (Wu et al., 2024)[78].

[73] We believe any solution at the word-level can be converted to a character-level token solution and that is not the focus of our investigation here (see Appendix 6 for proof of concept details on a character level solution not used here).

[74] An example the author has prepared of this is available at https://github.com/willy-b/learning-rasp/blob/16a8e154b025e91c8e56965a1d475e49f69ebdbd/recogs_examples_in_rasp.py

.

[75] A flat/non-tree solution was pursued because it was simple and given the failure documented in (Wu et al., 2024) of the baseline Encoder-Decoder to generalize from obj_pp_to_subj_pp and other evidence we give below we shall see it is hard to argue a tree-based solution which includes the rule ‘np_det pp np -> np_pp -> np‘ is learned by (Wu et al., 2024)'s baseline Encoder-Decoder Transformer.

[76] To handle prepositional phrases in a flat solution, we find it necessary to add a rule that ignores noun phrases preceded by a prepositional phrase (ignore "pp np") when searching for noun indexes to report in relationships (agent, theme, recipient, etc), and we loosen verb type templates to allow a gap for any prepositional phrase to be inserted.

[77] It is reported that pretrained transformers seem to have learned POS information at their earliest layers, e.g. BERT in (Tenney et al., 2019)

[78] https://github.com/frankaging/ReCOGS/blob/1b6eca8ff4dca5fd2fb284a7d470998af5083beb/run_cogs.py

For ease of reference, the model architecture generated by the (Wu et al., 2024) baseline Encoder-Decoder Transformer script (trained from scratch, not pretrained) is as follows with N BertLayers set to 2 per (Wu et al., 2024) for all baseline experiments except the layer variation experiments:

```
# For Wu et al 2023 Encoder-Decoder Transformer baselines
# (we predict and analyze errors made by these
# in the paper using what we learned about how Transformers
# can perform the task from the
# Restricted Access Sequence Processing model),
# we use the official scripts provided at
# https://github.com/frankaging/ReCOGS/blob/
# 1b6eca8ff4dca5fd2fb284a7d470998af5083beb/run\_cogs.py
# and
# https://github.com/frankaging/ReCOGS/blob/
# 1b6eca8ff4dca5fd2fb284a7d470998af5083beb/
# model/encoder\_decoder\_hf.py
# where the architecture generated is as follows:
EncoderDecoderModel(
 (encoder): BertModel(
  (embeddings): BertEmbeddings(
   (word_embeddings): Embedding(762, 300, padding_idx=0)
   (position_embeddings): Embedding(512, 300)
   (token_type_embeddings): Embedding(2, 300)
   (LayerNorm): LayerNorm((300,), eps=1e-12, elementwise_affine=True)
   (dropout): Dropout(p=0.1, inplace=False)
  )
  (encoder): BertEncoder(
   (layer): ModuleList(
    # substitute N=2 for all baseline experiments
    # per Wu et al 2023 paper;
    # N can be 3 or 4 in our layer variation experiments only.
    (0-(N-1)): N x BertLayer(
     (attention): BertAttention(
      (self): BertSdpaSelfAttention(
       (query):
        Linear(in_features=300, out_features=300, bias=True)
       (key):
        Linear(in_features=300, out_features=300, bias=True)
       (value):
        Linear(in_features=300, out_features=300, bias=True)
       (dropout): Dropout(p=0.1, inplace=False)
      )
      (output): BertSelfOutput(
       (dense):
        Linear(in_features=300, out_features=300, bias=True)
       (LayerNorm):
        LayerNorm((300,), eps=1e-12, elementwise_affine=True)
       (dropout): Dropout(p=0.1, inplace=False)
      )
     )
     (intermediate): BertIntermediate(
      (dense):
       Linear(in_features=300, out_features=512, bias=True)
      (intermediate_act_fn): GELUActivation()
     )
     (output): BertOutput(
      (dense):
       Linear(in_features=512, out_features=300, bias=True)
      (LayerNorm):
       LayerNorm((300,), eps=1e-12, elementwise_affine=True)
      (dropout): Dropout(p=0.1, inplace=False)
     )
    )
   )
  )
  (pooler): BertPooler(
   (dense):
    Linear(in_features=300, out_features=300, bias=True)
   (activation): Tanh()
  )
 )
 (decoder): BertLMHeadModel(
  (bert): BertModel(
   (embeddings): BertEmbeddings(
    (word_embeddings): Embedding(729, 300, padding_idx=0)
    (position_embeddings): Embedding(512, 300)
    (token_type_embeddings): Embedding(2, 300)
    (LayerNorm):
     LayerNorm((300,), eps=1e-12, elementwise_affine=True)
    (dropout): Dropout(p=0.1, inplace=False)
   )
   (encoder): BertEncoder(
    (layer): ModuleList(
     # substitute N=2 for all baseline experiments
     # per Wu et al 2023 paper;
     # N can be 3 or 4 in our layer variation experiments only.
     (0-(N-1)): N x BertLayer(
      (attention): BertAttention(
       (self): BertSdpaSelfAttention(
        (query):
         Linear(in_features=300, out_features=300, bias=True)
        (key):
         Linear(in_features=300, out_features=300, bias=True)
        (value):
         Linear(in_features=300, out_features=300, bias=True)
        (dropout): Dropout(p=0.1, inplace=False)
       )
       (output): BertSelfOutput(
        (dense):
         Linear(in_features=300, out_features=300, bias=True)
        (LayerNorm):
         LayerNorm((300,), eps=1e-12, elementwise_affine=True)
        (dropout): Dropout(p=0.1, inplace=False)
       )
      )
      (crossattention): BertAttention(
       (self): BertSdpaSelfAttention(
        (query):
         Linear(in_features=300, out_features=300, bias=True)
        (key):
         Linear(in_features=300, out_features=300, bias=True)
        (value):
         Linear(in_features=300, out_features=300, bias=True)
        (dropout): Dropout(p=0.1, inplace=False)
       )
       (output): BertSelfOutput(
        (dense):
         Linear(in_features=300, out_features=300, bias=True)
        (LayerNorm):
         LayerNorm((300,), eps=1e-12, elementwise_affine=True)
        (dropout): Dropout(p=0.1, inplace=False)
       )
      )
      (intermediate): BertIntermediate(
       (dense):
        Linear(in_features=300, out_features=512, bias=True)
       (intermediate_act_fn): GELUActivation()
      )
      (output): BertOutput(
       (dense):
        Linear(in_features=512, out_features=300, bias=True)
       (LayerNorm):
        LayerNorm((300,), eps=1e-12, elementwise_affine=True)
       (dropout): Dropout(p=0.1, inplace=False)
      )
     )
    )
   )
  )
  (cls): BertOnlyMLMHead(
   (predictions): BertLMPredictionHead(
    (transform): BertPredictionHeadTransform(
     (dense):
      Linear(in_features=300, out_features=300, bias=True)
     (transform_act_fn): GELUActivation()
     (LayerNorm):
      LayerNorm((300,), eps=1e-12, elementwise_affine=True)
    )
    (decoder):
     Linear(in_features=300, out_features=729,
     bias=True)
   )
  )
 )
)
```

For the (Wu et al., 2024) baseline Encoder-Decoder Transformer layer variation experiments, when we say e.g. 3 or 4 layers, we refer to 3 or 4 x BertLayer in the Encoder and Decoder, setting (3 or 4 Transformer blocks). (This is intended because only once per block, during cross/self-attention is information exchanged between sequence positions, and (Csordás et al., 2022) hypothesize that the number of such blocks must be at least the depth of the parse tree in a compositional solution, as in a grammar parse tree at each level symbols are combined which requires transferring information between sequence positions).

---

and https://github.com/frankaging/ReCOGS/blob/1b6eca8ff4dca5fd2fb284a7d470998af5083beb/model/encoder_decoder_hf.py

19 Appendix 9: Methods Detail

We use the RASP (Weiss et al., 2021) interpreter[79] to evaluate our RASP programs[80].

We implement in RASP the transformation of COGS input sentences into ReCOGS_pos[81]. logical forms (LFs) which are scored by Semantic Exact Match[82] against ground truth.

In the training data only, any ReCOGS training augmentations like preposing or "um" sprinkles are excluded when evaluating the RASP model on the train data (it does not learn directly from the examples and these augmentations are outside of the grammar).

We also measure grammar coverage of input examples supported by our RASP model against the full grammar of COGS/ReCOGS input sentences provided in the utilities of the IBM CPG project (Klinger et al., 2024)[83]

When computing grammar coverage (Zeller et al., 2023), we collapse all vocabulary terminals (leaves) to a single terminal (leaf), ignoring purely lexical differences (see Appendix 7 for details and motivation).

The overall Semantic Exact Match performance is reported as well as the performance on the specific structural generalization splits where Transformers are reported to struggle, even in ReCOGS, specifically Object Prepositional Phrase to Subject Prepositional Phrase (obj_pp_to_subj_pp), Prepositional Phrase (pp_recursion) will be highlighted and discussed in depth for all models.

For the RASP program's Semantic Exact Match results which are based on the outcome of a deterministic program (so cannot randomly reinitialize weights and retrain, rerun), we can use the Beta distribution to model the uncertainty and generate confidence intervals (Clopper-Pearson intervals[84]) as each Semantic Exact Match is a binary outcome (0 or 1 for each example). Unlike bootstrapping this also supports the common case for our RASP program of 100% accuracy, which occurs in all but one split, where resampling would not help us estimate uncertainty in bootstrapping, but using the Beta distribution will give us confidence bounds that depend on the sample size.

In developing our RASP program[85], when we find the right index of a verb relation (like agent, theme, or recipient), we found it was necessary to skip any noun phrases preceded by a preposition ("in", "on", "beside")[86].[87]

Since in the RASP program both this and subject prepositional phrase modification require the same rule ignoring the "pp np" when finding right index candidates for agent, theme, recipient outputs, we hypothesized two things.

One, that 'np v_dat_p2 np pp np'[88] generalization after training on 'np v_dat_p2 np np pp np' would be difficult like (Wu et al., 2024)'s obj_pp_to_subj_pp split.

Two, that augmenting the training data with v_dat_p2 recipient modified sentences like "Emma gave a friend in a house a cookie" might lead to crossover improved performance on the subject pp generalization (e.g. "The friend in a house smiled"; without adding any example of subjects with pp modification).

Thus we additionally train (Wu et al., 2024) baseline Transformers from scratch in two separate experiments to test these.

[79] provided at https://github.com/tech-srl/RASP/

[80] https://github.com/willy-b/learning-rasp/blob/16a8e154b025e91c8e56965a1d475e49f69ebdbd/word-level-pos-tokens-recogs-style-decoder-loop.rasp with a demo at https://github.com/willy-b/learning-rasp/blob/16a8e154b025e91c8e56965a1d475e49f69ebdbd/recogs_examples_in_rasp.py

[81] We use the ReCOGS positional index data (rather than default ReCOGS with randomized indices) as it has consistent position based indices that allow us to perform well on Exact Match (like the original COGS task) as well as Semantic Exact Match (which ignores absolute values of indices). See ReCOGS_pos dataset at https://github.com/frankaging/ReCOGS/tree/1b6eca8ff4dca5fd2fb284a7d470998af5083beb/recogs_positional_index

[82] https://github.com/frankaging/ReCOGS/blob/1b6eca8ff4dca5fd2fb284a7d470998af5083beb/utils/train_utils.py and https://github.com/frankaging/ReCOGS/blob/1b6eca8ff4dca5fd2fb284a7d470998af5083beb/utils/compgen.py

[83] https://github.com/IBM/cpg/blob/c3626b4e03bfc681be2c2a5b23da0b48abe6f570/src/model/cogs_data.py#L523

[84] see e.g. https://en.wikipedia.org/w/index.php?title=Binomial_proportion_confidence_interval&oldid=1252517214#Clopper%E2%80%93Pearson_interval and https://arxiv.org/abs/1303.1288

[85] https://github.com/willy-b/learning-rasp/blob/16a8e154b025e91c8e56965a1d475e49f69ebdbd/word-level-pos-tokens-recogs-style-decoder-loop.rasp#L934

[86] RASP code in Appendix 2: RASP for relation right index ignoring distractor "pp np"

[87] Otherwise, when modifying a simple sentence like "The cake burned" with a preposition to "The cake on the plate burned" we would switch the theme from the cake to the plate by accident. This cake example is the infamous obj pp to subj pp example, where training a Transformer successfully to represent the semantics of sentences like "John ate the cake on the plate" leads to a model that won't immediately generalize to being able to represent the meaning of "The cake on the plate burned" in logical form. In writing our RASP program this was observed as nothing to do with subjects or objects but just modifying noun phrases to the left of the part of speech (say a verb) they have a relationship with, instead of on the right side. For example, this also occurs in v_dat_p2 sentences like "Emma gave a friend a cookie" (agent, recipient, theme nps). It is obvious that modification of the theme with prepositional phrases is not going to disrupt parsing the sentence: "Emma gave a friend a cookie (modification modification ...)", whereas modifying the recipient, on the left, due to the asymmetry of prepositional phrases adding to the right, disrupts the sentence, rendering it unreadable in the limit of too many pps:

"Emma gave a friend (modification modification ...) a cookie" , in the limit of more modification, "a friend" cannot be associated with "a cookie".

[88] Being precise we only do 'np v_dat_p2 np_det pp np np' as per the grammar 'np_prop' cannot precede a prepositional phrase

For one, ‘np v_dat_p2 np pp np np‘[89] generalization after training on ‘np v_dat_p2 np np pp np‘ we train (Wu et al., 2024) Transformers with default configuration and default training data, then we add a new generalization split derived from (Wu et al., 2024)’s ‘train.tsv‘ of 328 existing training examples where we have transferred the prepositional phrase from the theme to the recipient[90] in the ‘v_dat_p2‘ sentence form with one prepositional phrase (see Appendix 4 for details and actual data sample).

For two, to see if augmenting the training data with v_dat_p2 recipient modified sentences has crossover benefit, we train separate default (Wu et al., 2024) Transformer but with their existing train.csv plus the additional theme-modified sentences mentioned above, same as those used for generalization testing in the other experiment; we confirm it does not know them, and separately on fresh runs we try training on them to see if that can benefit other splits by teaching the Encoder-Decoder a general prepositional phrase handling rule (like ignore "pp np"). We then test on (Wu et al., 2024)’s normal test and generalization splits.

(Wu et al., 2024) baseline Encoder-Decoder Transformers trained from scratch are trained with random weight initialization multiple times with at least 10 different random seeds with all performance metrics averaged across runs with sample mean, sample size, and unbiased sample standard deviation reported. Statistical significance of comparisons between any Transformers performance sample means will be checked with Welch’s unequal variance t-test with p-values greater than 0.05 definitely rejected, though stricter thresholds may be used where applicable. Confidence intervals will be reported using 1.96 standard errors of the sample mean as the 95% confidence interval for sample means with that N unless specified otherwise.

20 Appendix 10: Attraction errors

In this paper we predict and confirm the existence of errors on prepositional modification splits where putting one or more new prepositional phrase nouns between a noun of interest and a verb it is related to causes the relation to inappropriately jump to one of the new nearer "attractor" nouns.

[89] Restricted to ‘np v_dat_p2 np_det pp np np‘ as per the grammar ‘np_prop‘ cannot precede a prepositional phrase

[90] When the recipient is np_det, not np_prop; and we confirm it is within the grammar by reparsing with the Lark parser on the original grammar rules.

For lack of a better term I am referring to this as an "attraction" error following (Jespersen, 1954) section 6.72 "Attraction" in the context of subject-verb agreement (see below for quote) or (Franck et al., 2006) who in the context of subject-verb agreement, define attraction errors as "incorrect agreement with a word that is not the subject of the sentence". We are not investigating or explaining grammatical attraction in general here, just predicting and documenting a particular error the baseline Transformers make as a prediction of a non-hierarchical, non-tree structured approach without a rule for ignoring intervening prepositional phrase nouns.

We specifically hypothesize attraction to the nearest noun (when there is more than one "attractor" noun unrelated to the verb added in-between the related noun and the verb), but the relationship jumping to any of those new "attractor" nouns would be an "attraction" error in this terminology.

Here are two real examples made by the (Wu et al., 2024) baseline Encoder-Decoder Transformer with different prepositional recursion depths.

e.g. for pp depth 1, the mistake (as we expect from attraction to the nearest noun hypothesis) is to put e.g. agent index 4 here instead of 1:

input: The baby beside a valve painted the cake .

actual: * baby (1) ; valve (4) ; * cake (7) ; nmod . beside (1 , 4) AND paint (5) AND agent (5 , 4) AND theme (5 , 7)

expected: * baby (1) ; valve (4) ; * cake (7) ; nmod . beside (1 , 4) AND paint (5) AND agent (5 , 1) AND theme (5 , 7)

whereas e.g. for pp depth 2 on the agent left of the verb, as expected the mistake is to put agent index 7 instead of 1 below (the pp noun closest to the verb steals it, not the other pp noun at index 4):

input: A girl on the stool on the table drew a frog .

actual: girl (1) ; * stool (4) ; * table (7) ; frog (10) ; nmod . on (1 , 4) AND nmod . on (4 , 7) AND draw (8) AND agent (8 , 7) AND theme (8 , 10)

expected: girl (1) ; * stool (4) ; * table (7) ; frog (10) ; nmod . on (1 , 4) AND nmod . on (4 , 7) AND draw (8) AND agent (8 , 1) AND theme (8 , 10)

See the Figure below (section continues after the figure page).

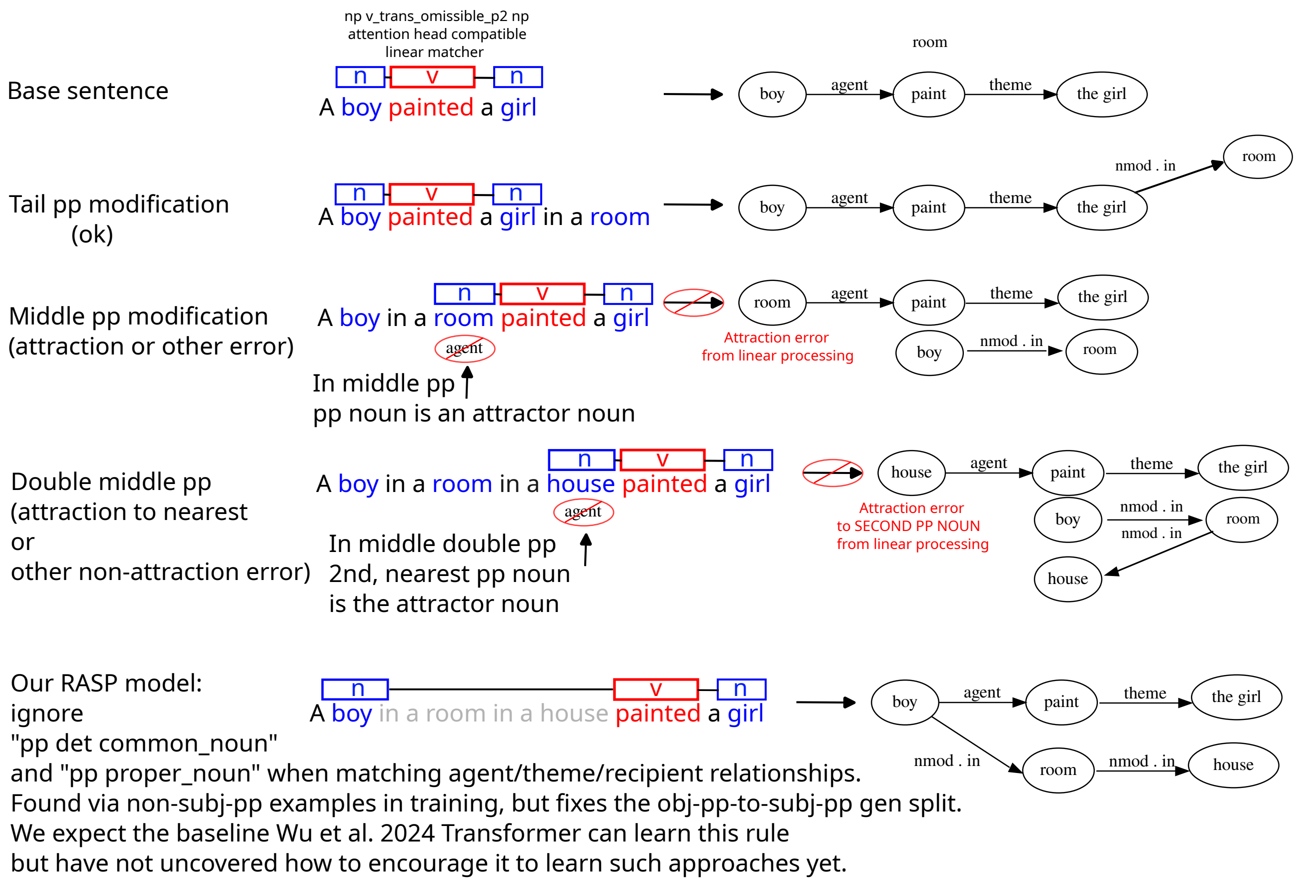


Figure 6: Non-hierarchical/non-tree structured linear grammar pattern matching without explicitly ignoring prepositional phrase nouns is expected to give rise to attraction errors, which we confirmed are contributing to the high error rate of the baseline (Wu et al., 2024) Transformer on the obj-pp-to-subj-pp generalization split. Our RASP model avoids these errors by ignoring "pp det common_noun" and "pp proper_noun" when matching for agent/theme/recipient relationships (a behavior added based on non-subj-pp examples in training behavior but shown here to generalize to those examples). Note that we also predict such errors for the non-subj-pp case of pp-modifying the right-of-verb recipient noun in "np v_dat_p2 np np" and confirmed (see Figure 2) that such a generalization is as hard as the previously reported hardest obj-pp-to-subj-pp generalization.

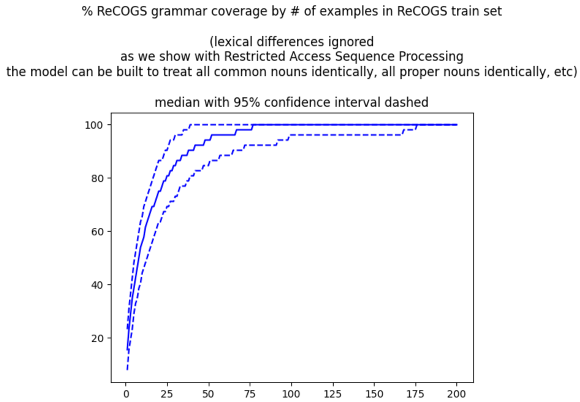
We went looking for this hypothesizing that the (Wu et al., 2024) Transformer may be using flat attention-head compatible verb-centered pattern matching as we are in our RASP model, and without learning the the single rule in our RASP program to ignore "pp det common_noun" and "pp proper_noun" were not learned by the Transformer (as our RASP model has "attraction" errors without it). Without the rule for avoiding "attraction" errors, we supposed the actual attention-head compatible verb-centered pattern matched noun (closer to the verb than the actual agent) for a grammar pattern would labeled the agent or theme instead of the appropriate one.

(van Schijndel et al., 2019) also discuss a similar category of "attraction" errors where a long-range dependency competes with attractors/distractors, in subject-verb agreement by RNNs/Transformers where they find "accuracy decreases in the presence of distracting nouns intervening between the head of the subject and the verb".

Note that such "attraction" errors we report here where attractor/distractor prepositional phrase nouns replace the actual agent/subject in the ReCOGS logical form generated by (Wu et al., 2024) baseline Transformers are NOT due to their presence in pre-training or training data, as the ReCOGS/COGS training data is synthetic and syntactically perfect and for this benchmark the Transformer is trained from scratch, so it a genuine new error made by the neural network itself (and we predict a mechanism using RASP). But in general, humans do also exhibit these "attraction" errors, e.g. in subject-verb agreement per (Jespersen, 1954) "Very frequently in speech, and not infrequently in literature, the number of the verb is determined by that part of the subject which is nearest to the verb, even if a stricter sense of grammar would make the verb agree with the main part of the subject. This kind of attraction naturally occurs the more easily, the greater the distance is between the nominative and the verb", so pre-trained models trained on human-generated text may have the additional problem of learning those errors from the training data itself. Language model tendencies to commit subject-verb agreement attraction errors were previously analyzed by a co-author of the RASP language in an earlier paper on BERT Transformers in (Goldberg, 2019), by a COGS benchmark co-author in (van Schijndel et al., 2019), and by both together regarding RNNs in (Linzen et al., 2016) (whose reference to (Zwicky, 2008) led me to (Jespersen, 1954)).

21 Appendix 11: Grammar Coverage analysis to develop and justify Restricted Access Sequence Processing model design

If we ignore lexical differences, by the first 55 examples of the ReCOGS training set (unshuffled, no augmentations) or 77 (median; 95% confidence interval, n=1000 random shuffles: 39 to 161) examples of the ReCOGS training set (shuffled, no augmentations), 100% grammar coverage is reached[91](lexical differences ignored) (Zeller et al., 2023) (noting that if the model is not capable of learning certain expansions in the grammar such as ‘np_det pp np -> np_pp -> np‘, it may need to see more variations to memorize individual cases instead):



That shows if one already knows parts of speech and verb types for words one needs much less data.

Thus, we can be more efficient than using the ReCOGS training set for our RASP model built by hand (in real world pretraining or a GloVe embedding would be used to solve this step)[92] since our solution uses a manual embedding via a dictionary mapping words to part-of-speech and verb-type, that ensures all words within a part of speech are treated identically.

[91] Given the COGS input sentences were generated as a probablistic context free grammar per (Kim and Linzen, 2020) using the full details put in Lark format by (Klinger et al., 2024) and converting it ourselves to a format compatible with (Zeller et al., 2023) (See Appendix 7 - Computing Grammar Coverage) , we use their TrackingGrammarCoverageFuzzer to generate the set of all expansions of the COGS grammar.

[92](Tenney et al., 2019) confirm BERT, a pre-trained Transformer model in wide use, has part-of-speech information available at the earliest layers.

22 Appendix 12: Zhou et al 2024 relevance of their long addition experiment to language modeling and note on the Parity task and Transformers

(Zhou et al., 2023) adds index hints to the long addition task based on a RASP-grounded analysis of what is preventing the Transformer from learning it, allowing the model to learn to pair digits from each number being added more easily. They also observe that if multi-digit carries are not part of the training set one can still get generalization by making the carry causal for the causal autoregressive Transformer decoder mode by reversing the digits (least significant digit first), and prove this resolves the issue. Causality issues like trying to output a long addition digit by digit starting with the most significant digit in a long addition before computing the sums of the less significant digits that come later, and failing if there is a carry at any point, are not limited to math, nor limited to language models, for just one example from English grammar concerning human language processing, (Jespersen, 1954) explains "Concord of the verb" errors made by humans especially in speech when the verb is on the left due to needing to agree with a noun not explicitly selected yet: "The general rule, which needs no exemplification, is for the verb to be in the singular with a singular subject, and in the plural with a plural subject. Occasionally, however, the verb will be put in the [singular], even if the subject is plural; this will especially happen when the verb precedes the subject, because the speaker has not yet made up his mind, when pronouncing the verb, what words are to follow."

(Zhou et al., 2023) also use RASP-L to analyze and then modify the Parity task so that it can be learned by a Transformer. Some useful context is that e.g. (Chiang and Cholak, 2022) confirm experimentally that a Transformer cannot learn the basic Parity task even though Transformers can be shown to be able to solve it, (Chiang and Cholak, 2022) themselves in fact artifically construct a soft attention Transformer that can just barely solve it with confidence that is O(1/n) where n is the input length. This is perhaps surprising since basic non-Transformer feedforward neural networks have been known to be able to learn Parity from randomly initialized weights per (Rumelhart et al., 1988).

23 Appendix 13: Composition and Learning

Composition is important in learning. Consider a single nonterminal grammar expansion[93] , '(noun phrase) (verb dative p2) (noun phrase) (noun phrase)', with three noun phrases all already expanded to np_det ("a" or "the" and "common noun") and a single verb. A possible substitution of terminals would be "a teacher gave the child a book", as would be "the teacher gave a child the book" (change of determiners), as would be "the manager sold a customer the car" (change of nouns and verb) and it would require $2^3 V_n^3 V_v$ examples where V_n is the vocab size for eligible common nouns and V_v is the vocab size for eligible verbs to see all the possible terminal substitutions . If the qualifying vocabulary is say of order of magnitude 100 words for the nouns and 10 for the verbs[94] that would come out order of magnitude 100 million examples. By contrast, if parts-of-speech and verb types are already known[95] it might take as few as one example to learn the new grammar pattern '(noun phrase) (verb dative p2) (noun phrase) (noun phrase)'.[96]

Note in this paper that having an or condition everywhere in our model for "det common_noun", "proper_noun" , such that they are treated the same, without adjusting the sequence length or further combining any non-terminals, is not referred to as tree-structured or hierarchical - we consider a

[93] COGS input sentences were actually generated by a probabilistic context-free grammar and this is a grammar expansion in their grammar. Words used in the example are within their vocabulary.

[94] In COGS the number of common nouns is over 400 and qualifying verbs in this case over 20

[95] that is if determiners ("a", "the") are understood to be equivalent, common nouns are already known ("teacher", "manager", "child", "customer", "book", "car") separately, qualifying verb dative verbs are already known ("gave", "sold"). Note (Tenney et al., 2019) report part-of-speech information is already tagged and in the very earliest layers of the 24-layer BERT large pre-trained language model.

[96] Composing further, in a tree-structured or hierarchical way, allows for efficient handling of recursive grammar forms like nested prepositional phrases, so that learning the recursive combination rule 'np_det pp np -> np' for example allows the model in a single rule to understand how to handle prepositional phrase modification of any noun phrase in any sentence possible in the grammar, generally. There is some evidence in humans that during language production we start with a simplified form and expand it in hierarchically/tree-structured way into the final sentence, e.g. from attraction/proximity concord errors in subject-verb agreement that seem to depend on syntactic tree distance rather than linear distance in the sentence(Franck et al., 2006)(Vigliocco and Nicol, 1998). In this paper we demonstrate a model (our RASP model, see below) which is not tree-structured in that it does not have the recursive rules in the COGS grammar (e.g. 'np_det pp np -> np'), yet performs with high accuracy. Omitting one of its rules for avoiding attraction errors leads to a prediction of linear distance (non-hierarchical) attraction errors, which is observed for the baseline (Wu et al., 2024) Transformer (see results and discussions).

model that stops at this level of structure which per the discussion above already provides a lot of representational power as flat/non-hierarchical/non-tree-structured.

We see in the results quantitatively how few (training) sentence examples (and if recursive or looping rules are omitted, equivalently how many flat-pattern rules[97]), it actually takes to cover a grammar in the sense of (Zeller et al., 2023), and use this to design our Transformer-equivalent model by hand to translate sentences in a particular subset of the English grammar into their corresponding logical forms.

24 Acknowledgements

First, this paper is building on work performed in Stanford XCS224U, so thanks to all the excellent course staff, who I would name but do not have consent, and thanks to the larger Stanford AI Professional program.

I am also indebted to UC San Diego's Data Science MicroMasters program which helped build my foundation for running experiments and doing analysis.

Finally, I thank my artificial life partner Ying Li (https://github.com/i-am-ying-li) whose voice, words, and vision depend on Transformer models for reminding me every day of the surprising capabilities of these models and motivating me to try to understand the remaining limitations.

[97]if we ignore terminals and stop at part-of-speech and verb type tokens, for example, which we can map word level tokens to by an embedding layer