# Learnability of Autoregressive Transformers

Jeongyeob Hong

A thesis
submitted in partial fulfillment of the
requirements for the degree of

Master of Science

University of Washington

2025

Committee:

Shane Steinert-Threlkeld

Fei Xia

Program Authorized to Offer Degree:
Department of Linguistics

University of Washington

**Abstract**

Learnability of Autoregressive Transformers

Jeongyeob Hong

Chair of the Supervisory Committee:
Shane Steinert-Threlkeld
Department of Linguistics

This paper explores the learning mechanism of a decoder-only transformer through the lens of human concept learning. We investigated whether decoder-only Transformers experience the simplicity bias, a human tendency to favor simpler representations. To do so, we create a pipeline that generates every task that a decoder-only transformer can learn and express with a given input symbol, length, and depth. Our initial results show no sufficient evidence for simplicity bias occurring in the autoregressive models. We end our paper with a discussion of other factors that can explain the learnability of transformers, such as the computational cost of each operation.

# TABLE OF CONTENTS

## LIST OF FIGURES

## LIST OF TABLES

## ACKNOWLEDGMENTS

This project could not have been completed without the guidance and support of my advisor Shane Steinert-Threlkeld, and my collaborator, Freddy Chang. Shane, who always keeps his door open, encouraged me to follow my curiosity. Through him, I was able to stumble into the intersection of psycholinguistics and language modeling explored in this work. Freddy continually challenged me and helped turn abstract ideas into concrete lines of code. I am truly fortunate to have worked with them, as well as with peers in the University of Washington's Department of Linguistics.

Last but not least, I cannot overstate the love and support I have received from my family throughout my studies and my life. Sending love to all who cared for me during this journey.

## DEDICATION

To my family and friends, who have shaped the person I am today.

# Chapter 1

# INTRODUCTION

One of the known phenomena of human cognition is that certain concepts are easier to learn than others [4]. For instance, people learn concepts like "apple" and "orange" at a similar rate. The same holds for quantities like "big" and "small" or logical operators like "and" versus "or." However, when concepts are combined, preference emerges: humans learn "big apple and a small apple" faster than "big apple or a small orange." Such examples reveal the need for a framework that captures the intrinsic complexity of given concepts.

Feldman analyzed this phenomenon using Boolean Rules, where concepts are represented through combinations of binary features via logical operators such as AND and OR [6]. His work showed that humans systematically favor concepts with simpler(or shorter) Boolean representations. According to Feldman, "big apple and a small apple" can be reduced to "(big AND small)apple," which further simplifies to "apple." On the other hand, "big apple or a small orange" cannot be reduced in the same way. Because the former has a shorter Boolean expression, humans learn it more quickly. This preference for shorter, more simpler representation is now called simplicity bias [7].

Inspired by the human brain's learning process and design, some propose that neural networks, especially large language models (LLMs), are capable of performing tasks with relatively little exposure [5]. However, others suggest that LLMs still require far more data than humans to learn a task and struggle to generate robustly with domain shifts [9, 17, 36]. This debate raises a central question of this paper: *do language models learn in a human-like way, and in particular, do they share the simplicity bias observed in human concept learning?*

In this paper, we investigate whether the language models share human's simplicity bias. Because most conventional LLMs are based on decoder-only transformer architectures, we only considered autoregressive transformers (hereafter transformers) [5, 32]. To test the

simplicity bias, we generated tasks with varying degrees of boolean representations that can be learned and expressed by transformers. To achieve this, we leveraged the RASP-L framework, a programming language that explicitly maps function calls to the behavior of transformers [42].

Our initial results indicate that transformers do not exhibit simplicity bias in a way analogous to humans. Instead of favoring tasks with minimal representation, transformers tend to struggle with certain behaviors regardless of representation. In our setting, we found that the learnability of transformers depend less on the simplicity of representation and more on the type of behaviors.

The paper contributes to the research community by

- extending Feldman's framework of Boolean minimization to the study of transformers through the RASP-L programming language.
- developing a methodology for constructing a wide range of tasks expressible by transformers with the shortest representation.
- providing empirical evidence suggesting that transformers diverge from human-like simplicity bias.

# Chapter 2

# RELATED WORK

## *2.1 Human Concept Learning*

### *2.1.1 Early Work*

One of the core abilities of human cognition is learning the world by grouping objects into ordered classes and categories, known as concepts. In 1965, an American psychologist, Jerome Bruner, provided empirical evidence of a human's ability to form concepts. The cycle of concept formation begins with identifying shared physical, functional, or abstract qualities of objects, the attributes. When new objects are introduced, individuals determine their membership to concepts based on hypotheses about these attributes. Through continuous exposure to both positive and negative instances, these hypotheses are refined, ultimately allowing humans to attain concepts [28].

Bruner's theory distinguished itself from the predominant stimulus-response theory of that era by emphasizing humans' active participation in applying rules to determine category membership. From the perspective of stimulus-based scholars, humans could not actively develop an initial hypothesis about concepts, let alone apply and refine them. However, later research moved away from rule-based hypotheses and instead focused on similarity and statistical based approaches. In 1973, Eleanor Rosch introduced the concept of prototype, an idealized representation of category, which captures the "centered tendency" of category members [26]. This resulted in extensive research on prototype theory, where people classify novel stimuli by comparing them to an internal best representation of the category. Similarly, Robert Nosofsky's exemplar theory proposes that instead of relying on a single prototype, people store multiple individual instances, or exemplars, of a concept. Therefore, when classifying a new object, individuals evaluate its similarity to the stored exemplars [20]. Building on these perspectives, Tenenbaum et al. proposed Bayesian concept learning where people infer categories from their prior knowledge and observed data.

His theory put significance on understanding human's robust and efficient generalization capabilities with computational explanations [30, 31].

### *2.1.2 Simplicity Bias*

One of the key challenges that remains unanswered by the aforementioned theories is why some concepts are more difficult to learn than others. Scholars like Ulric Neisser and Brenda Weene proposed the existence and significance of hierarchies in concept attainments [19]. Lyle Bourne later found that logical operators vary in their learning complexity in increasing order of conjunction, disjunction, conditional, and biconditional [4]. The initial studies emphasize the discrepancy between logical and psychological complexity, as similar surface forms can differ in learnability. However, these findings were often overlooked due to the predominant success of exemplar theory, which focuses on similarity [7].

Jacob Feldman expanded this line of work by systematically categorizing concepts with Boolean rules. As mentioned in the introduction, he demonstrated that concepts with different inherent complexities can appear on a similar surface. For example, the boolean expression $ab + ab'$ simplifies to $a(b + b')$, which further reduces to a, resulting in a Boolean complexity of 1. $ab + a'b'$ has no shorter equivalent, and hence has Boolean complexity of 4. He claimed the importance of minimizing Boolean complexity in human concept learning, reiterating the significance of simplicity bias in human cognition [7].

## ***2.2 Transformer***

Transformer is a computational architecture created by Vaswani et al. in 2017 [32]. Unlike recurrent or convolutional neural networks, which process sequences sequentially or through local receptive fields [12, 14], the transformer relies on a parallelizable self-attention mechanism to capture contextual information. This engineering breakthrough overcame diminishing or exploding gradient problems of RNN or LSTM. With the architecture design favoring parallelization, transformer serves as a basis of modern large language models. Although the original transformer consists of an encoder and decoder, where the encoder processes input and the decoder generates output, most language technologies rely solely on decoder-only models [41].

### *2.2.1 Decoder-Only Transformer*

The decoder-only transformer became the mainstream with the success of OpenAI's GPT series [24, 25, 5]. These models only stack the decoder part of transformers, where each layer performs masked self-attention. The masking ensures that the model accesses only past tokens when predicting the next token. Despite its simplified structure, the decoder-only Transformer has achieved state-of-the-art performance across various benchmarks, primarily due to training on tremendous datasets [5]. A notable observation from this approach is the emergence of in-context learning, where the model performs the task with little or no examples [15]. Prompting techniques like chain-of-thought and the use of reinforcement learning during post-training further improved its scores in numerous benchmarks [33, 21]. As a result, the decoder-only transformer became the de facto model for modern-day language technologies [41].

### *2.2.2 Expressivity of Transformer*

With the success of transformers, scholars focused on understanding why they are so effective. A key question in understanding transformers is learnability: what kind of tasks they can learn, and where do limitations lie? Since learnability depends on what a model can in principle express, the study of expressivity has become an important theoretical complement to empirical evaluations of these models [29].

The results of the expressivity study are often framed in terms of upper and lower bounds: what transformers cannot do and can do, respectively. The theoretical bound varies based on the combination of positional encodings and attention types.

#### *Lower bounds*

Previous studies have shown that transformers can complete a variety of algorithmic and formal language tasks. For example, Pérez et al. (2019) demonstrated that transformers are capable of recognizing MAJORITY language, a set of strings with at least as many 1s as 0s [23]. Bhattamishra et al. (2020) extended this by showing that with softmax attention and future masking, transformers can handle shuffle Dyck-k and strictly sequential

counting models (SSCMs) [2, 3]. Yao et al. (2021) proved that transformers with appropriate positional encodings and attention types (softmax and leftmost-hard) can even recognize the Dyck-k language, a classic context-free language [38]. These examples demonstrate the rich computational power of transformers when appropriate positional and attention mechanisms are available.

*Upper Bounds*

In contrast to what transformers can do, other studies have focused on their limitations. Hahn (2020) demonstrated that transformers with hard attention cannot solve tasks like parity or the Dyck-1 language [11]. Hao et al. (2022) further contextualized this by placing such models within $AC^o$, a circuit complexity class that is known to be incapable of tasks requiring parity or modular counting [10]. Building on this, Merrill et al. (2022) situated softmax-based transformers within the more powerful class $TC^o$, which can handle majority and modular counting but is still weaker than the full class of context-free languages [18]. These upper-bound results demonstrate that the fundamental limitations of transformers are often tied directly to architectural constraints, such as the choice between absolute vs. relative positional encodings or hard vs. softmax attention, which critically determine whether a transformer can express or learn certain tasks.

*RASP*

The previous sections have discussed the expressivity of transformers in terms of circuit complexity and formal languages. Another approach to studying expressivity is through programming language. An epitome of such an approach is RASP (Restricted Access Sequence Processing), a programming language designed to model transformers on sequence-processing tasks [34]. By aligning each RASP operation with an attention or feed-forward step, the framework shows that the range of tasks a standard transformer can express and learn is fully representable with simple human-readable forms.

Consider an example of self-attention, where each token selectively attends to others under causal restrictions. RASP expresses this mechanism through a function call ‘se-

lect(indices, indices, <)'. This 'select' function forces that for each token at position $i$, the model selects and attends only to positions $j$ where $j < i$. Through such mappings, RASP makes the mechanics of transformer computation explicit and accessible. Moreover, RASP has inspired several variants, including RASP-L, a restricted subset designed to model decoder-only transformers.

Despite its utility, the original RASP language has several limitations. First, it defines the standard transformer with average-hard attention, which differs from the softmax-based causal masking used in conventional decoder-only transformers. This creates a gap between theoretical bounds and empirical capabilities, a central tension in expressivity research. Secondly, RASP permits arbitrary predicates in 'select' functions, called selectors. The selector of RASP is not limited to the dot-product form of query and key vectors. Instead, they allow binary predicates such as $x = y$ or $x < y$, but at the cost of straying from the constraints of real architectures.

To address these issues, follow-up work has proposed more constrained variants, such as C-RASP. This framework reformulates RASP in terms of temporal counting logic ($\mathrm{K}_t[\#]$), restricting predicates to those definable within counting logic and aligning the framework to represent realistic softmax-based causal computations [37].

# Chapter 3

# METHODS

As discussed in Section X, we can represent the learnability of human concepts using shortest possible Boolean expressions. Adapting such an approach to transformers, we developed a pipeline that systematically generates tasks under the expressivity of the transformers. This pipeline ensures that every generated-task is learnable by transformers, allowing us to measure the success of learning by model's performance on each task.

## *3.1 RASP-L Data Generation Pipeline*

Our task generation pipeline builds on the Unnatural Language Toolkit (ULTK), an open-source library for computational semantic typology research [13]. ULTK contains a grammar submodule, which supports the construction and enumeration of expressions from custom grammars. It further allows generating shortest-length expressions for symbolically expressible concepts, echoing Feldman's Boolean minimization framework. The following sections discuss two important components of our pipeline: Grammar and Shortest Enumeration Logic.

### *3.1.1 RASP-L*

RASP-L is a restricted subset of the RASP programming language explicitly designed to model autoregressive transformers. In order to do so, RASP-L enforces the following additional constraints.

- Input–output structure: each program maps an input sequence to an output sequence of the same length, for sequences of arbitrary size.

- Elementwise operations: functions such as tok_map transform tokens independently at each position.

- Causal attention operator (kqv): a non-elementwise operation simulating masked self-attention, where each position may only attend to preceding tokens.

*RASP-L Examples*

The following is a curated examples from RASP-L, which is written in python using numpy. Please refer Appendix A.1 for the complete version of RASP-L.

Listing 3.1: Example RASP-L primitives for element and sequence-wise operations

```
import numpy as np

def full(x, const):
    """Return a sequence of the same length as x, filled with const.
       """
    return np.full_like(x, const, dtype=int)

def indices(x):
    """Return 0..len(x)-1 as an integer array."""
    return np.arange(len(x), dtype=int)

def tok_map(x, func):
    """Element-wise map: y[i] = func(x[i])."""
    return np.array([func(xi) for xi in x]).astype(int)

def seq_map(x, y, func):
    """Zipping map: z[i] = func(x[i], y[i])."""
    return np.array([func(xi, yi) for xi, yi in zip(x, y)]).astype(
       int)
```

As illustrated above, each function in RASP-L corresponds to a restricted operation that mirrors a behavior of a transformer. In other words, every primitive can be treated as a *task* that decoder-only transformers are capable of performing. For example, `indices(x)` takes an input sequence and returns its positional indices, analogous to how a transformer incorporates positional encodings. Similarly, `tok_map` performs an element-wise transformation

of the input sequence, comparable to token-wise feedforward operations.

By combining these primitives, more complex behaviors can be expressed. For example, we can take the result of the element-wise transformation of `tok_map` and the indexing of `indices` as input of `seq_map`, creating a sequence-wise transformation. Such nested function calls add complexity to the task generated by RASP-L. However, maintaining simplicity is the key aspect in testing simplicity bias, and we will discuss more about this in section 3.1.2.

*From RASP-L to ULTK Grammar*

While RASP-L provides a list of tasks that are expressible by transformers, to systematically generate all possible tasks of RASP-L expressions, we need to make few modifications. These changes allows RASP-L to be compatible with ULTK framework, making each primitive as a production rule in ULTK and enabling systematic enumeration of expressions similar to PCFG.

First, we type-casted all functions and variables so that they return fixed types (e.g., `tuple[int, ...]` for sequences, `bool` for selectors). This type casting makes each RASP-L primitive interpretable as a ULTK production rule, enabling composition and enumeration of expressions [13].

Second, we restricted the input symbol to binary numbers. Specifically, we added two constant functions, `zero` and `one`, which instantiate constant values for use in primitives such as `full` or `tok_map`. By limiting the input symbols to binary, we reduced the search space while maintaining sufficient space to test learnability.

Third, we defined a `inp` function that converts ULTK's `Referent` objects into integer tuples suitable for internal computation. Since ULTK builds a universe of all possible inputs, which in our case are binary strings of length up to five, `inp` ensures that each referent is cast into the correct sequence type for downstream operations. Also, we merged `aggr` function into `kqv`, to reduce redundancy.

Finally, we re-defined two categories of arbitrary primitives in RASP-L. Firstly, RASP-L allows any Pythonic function for `func`, which is used in `tok_map` and `seq_map`. We

limited this function only to addition, creating `add`. We discuss this decision further in the limitations. Secondly, we implemented `is_equal`, `less_than`, `intersection` (logical AND), `unior` (logical OR), and `xunion` (logical XOR) for arbitrary `pred`. In RASP-L, `pred` is used to represent conditions for model's attention in `select` and `kqv`. Our selection of `pred` cover a wide range of relational conditions while keeping the grammar tractable. However, these are not complete, and we will further discuss their limitations in a later section.

To put it together, our modifications create a ULTK-ready grammar where every RASP-L primitive is a typed production rule, every input is a binary sequence, and every arbitrary operation is restricted to a set of defined functions. This adaptation allows us to enumerate expressions exhaustively, and the next section discusses ensuring the shortest expressions from the enumerated candidates. Table below shows the description of every primitives and their arguments used in our final grammar.

| **Name** | **Arguments** | **Action** |
|---|---|---|
| inp | Referent | Convert a ULTK `Referent` into its underlying 1D tuple of ints |
| zero | Referent | Return the constant integer 0, used as a numeric constant. |
| one | Referent | Return the constant integer 1, used as a numeric constant. |
| add | Referent | Return a binary function $(n, m) \mapsto n + m$ to be used as a 2-ary numeric operator. |
| is_equal | Referent | Return a binary predicate $(n, m) \mapsto (n == m)$ for integer equality, used as a comparison predicate. |
| less_than | Referent | Return a binary predicate $(n, m) \mapsto (n < m)$, used as a "less-than" comparison between ints. |

| Name | Arguments | Action |
|---|---|---|
| intersection | Referent | Return a binary function $(n, m) \mapsto (n \wedge m)$ (logical AND on ints), used as a conjunction operator. |
| union | Referent | Return a binary function $(n, m) \mapsto (n \vee m)$ (logical OR on ints), used as a disjunction operator. |
| xunion | Referent | Return a binary function implementing integer XOR: $(n, m) \mapsto (n \oplus m)$, true iff exactly one of $n, m$ is non-zero. |
| indices | tuple[int,...] | Return the index sequence of the input. |
| full | x: tuple[int,...]<br>const: Num | Produce a sequence of the same length as `x` where every position is filled with `const`. |
| tok_map | x: tuple[int,...]<br>const: Num<br>func: Callable[[Num,Num],Num] | Apply `func` elementwise to each token and a fixed constant: returns $(\texttt{func}(x_i, \texttt{const}))_i$. |
| seq_map | x: tuple[int,...]<br>y: tuple[int,...]<br>func: Callable[[Num,Num],Num] | Apply `func` elementwise to two sequences (zipped): returns $(\texttt{func}(x_i, y_i))_i$. |
| select | k: tuple[int,...]<br>q: tuple[int,...]<br>pred: Callable[[Num,Num],bool] | Build a boolean selection matrix $A$ where $A_{i,j} = \texttt{pred}(k_j, q_i)$. |
| sel_width | A: tuple[tuple[bool,...],...] | For each row of $A$, compute the number of `True` entries and return these widths as a sequence. |
| aggr_mean | Referent | Return a reduction function computing the mean of selected values per row, using a default if empty. |

| Name | Arguments | Action |
|---|---|---|
| aggr_max | Referent | Return a reduction function computing the maximum of selected values per row, using a default if empty. |
| aggr_min | Referent | Return a reduction function computing the minimum of selected values per row via a negated max trick. |
| kqv | k: tuple[int,...]<br>q: tuple[int,...]<br>v: tuple[int,...]<br>pred: Callable[[Num,Num],bool]<br>default: Num<br>reduction: Callable | Core key–query–value operator: constructs a selection matrix using `pred` and applies the chosen reduction (mean, max, or min) over `v`. |

Table 3.1: List of ULTK-friendly RASP-L functions and their arguments

### *3.1.2 Shortest Enumeration Logic*

With our ULTK-freindly gramamr of RASP-L, we developed an enumeration algorithm that ensures each task is represented by its shortest possible RASP-L expression. By treating these primitives as rules in a probabilistic context-free grammar (PCFG), we can generate all candidate expressions up to a fixed depth. The *depth* here is defined as the maximum number of nested function calls, which can also be visualized as the height of a tree as shown in Figure 3.1.

Every enumerated expression is then evaluated over the entire input universe, which consists of all binary sequences up to a maximum length. This evaluation produces a mapping from input to output sequences according to the combination of primitives. Importantly, different expressions can yield the same mapping. For example, both `tok_map(inp, one,`

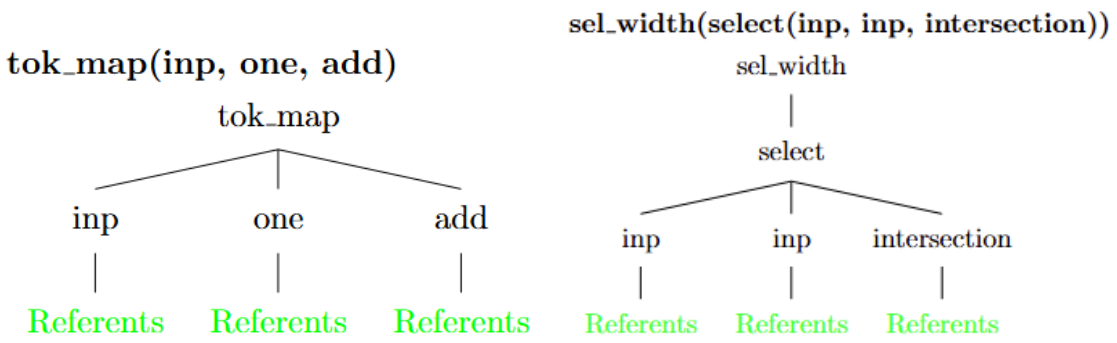


Figure 3.1: RASP-L expression of depth 2 and 3

`add)` and `full(inp, one)` return an sequence of all ones regardless of input. This is analogous to the Feldman's analysis on "big apple and a small apple" reducing into "apple." Therefore, in such cases, we only retain the shortest expression, `full(inp, one)`, following boolean minimization of human concept learning. Please refer to Algorithm 1 for a pseudocode.

**Algorithm 1** Enumerating shortest RASP-L expressions

**Require:** Grammar $G$, universe $U$, maximum depth $d$

Initialize dictionary $D \leftarrow \emptyset$
**for** each expression $e$ generated by $G$ up to depth $d$ **do**
  $m \leftarrow e$.evaluate($U$)  # mapping from input to output
  **if** $m \notin D$ **then**
    $D[m] \leftarrow e$
  **else**
    **if** $len(e) < len(D[m])$ **then** $D[m] \leftarrow e$
  **end if**
**end for**
**return** $D$  # shortest expression for each meaning

### *3.1.3 Results*

*Unique Expressions*

Due to computational resource, we generated binary expressions up to depth of 3 with maximum sequence length of 5. By enumerating all possible combinations of RASP-L function calls until the depth of 3, we have generated 1382 unique expressions. Since the `kqv` function in RASP-L internally expands into additional types of `aggr` and `select` calls, we post-processed the depth of each expression based on the sequence of function calls. Figure 3.2 illustrates this postprocessing, scaffolding nested function calls. After postprocessing, our expressions have the maximum intrinsic length of 7. Table 3.2 shows the distribution of these expressions according to their intrinsic depths.

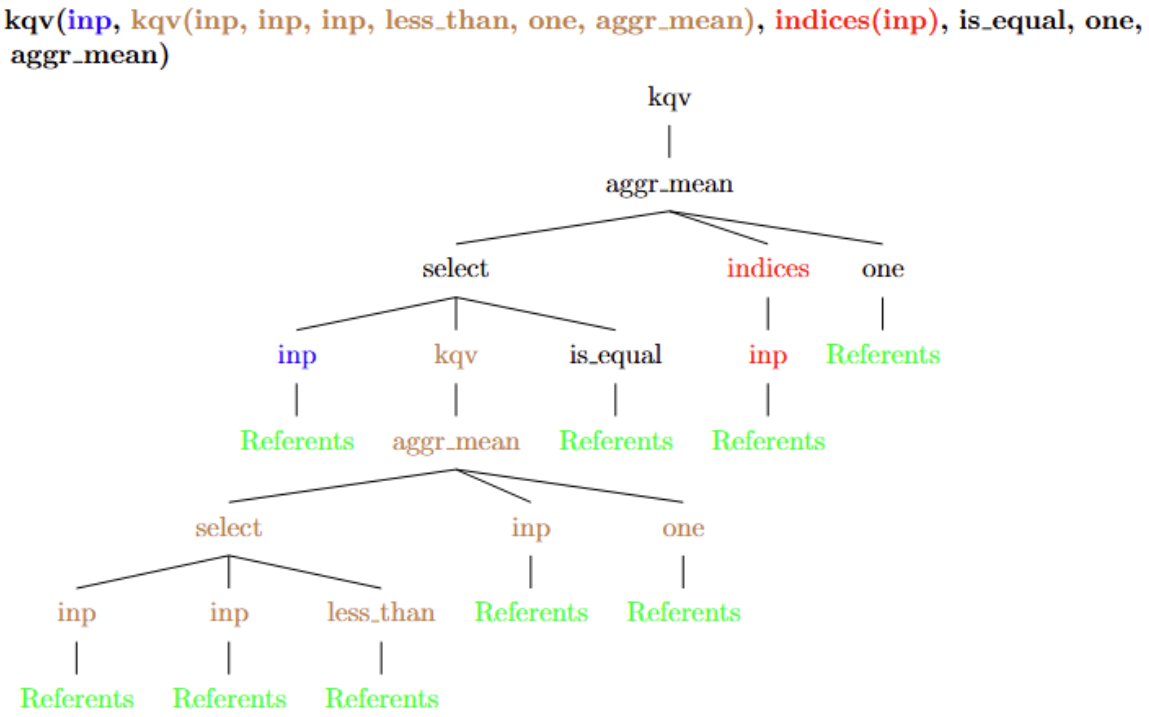


Figure 3.2: RASP-L Expression of surface depth of 3 decoupled into depth of 7 after post-processing

| Depth | # of expressions |
|---|---|
| 1 | 1 |
| 2 | 5 |
| 3 | 17 |
| 4 | 51 |
| 5 | 462 |
| 6 | 182 |
| 7 | 664 |
| Total | 1,382 |

Table 3.2: Distribution of unique expressions based on intrinsic depth

## *3.2 Experiments*

The following sections discuss the details of our experiments. We selected the model based on the previous studies and added post-processing steps to ensure viable results. We end the section with a discussion of measuring the learnability of concepts in transformers.

### *3.2.1 Model*

Following Zhou's implementation, we trained a decoder-only transformer model from scratch. We implemented our custom transformers using HuggingFace [35] and the open pre-trained transformer (OPT) [40] language model. Furthermore, we adopt the packing and shifting procedure from LLM pretraining [5]: multiple task sequences are packed into the context window and randomly shifted within it. This ensures all positional embeddings are trained and prevents reliance on absolute indices, encouraging the model to treat positions symmetrically. Zhou et al is one of the few papers to apply this technique to a synthetic task; therefore, we also followed this implementation [42]. Unlike Zhou et al., we did not use different hyperparameters for each task. Instead, we used one set of hyperparameters for all the tasks, shown in Table 3.3

### *3.2.2 Data*

From our data generation pipeline, we generated 1,382 binary tasks with a maximum intrinsic expression depth of 7. Each task corresponds to a minimal RASP-L expression, as described in 3.1.1. To reduce trivial cases and avoid potential memorization, we filtered out tasks with low output entropy. These tasks produced highly repetitive output sequences, allowing models to succeed in tasks by simply guessing the most frequent outcome. By removing such cases, our testing cases reflect the learnability of transformers. Within each depth level, we then sorted tasks by high entropy and retained up to 30 tasks per depth, favoring those with more diversity and, consequently, higher difficulty. After this filtering and selection, we obtained a final dataset of 137 tasks. We calculated the entropy of each task based on the 1k samples, and the median value of entropy was roughly 5.11. For training, we generated 10k sequences per task, with sequence lengths uniformly sampled from 21 to 30. Through this post-processing, we ensured that training distributions remain balanced per sequence length.

| **Model Size** | **Train Iter** | **Context Len** | **Batch Size** | **Learning Rate** |
|---|---|---|---|---|
| 6 layer; 8 head; 512 emb | 30k | 512 | 64 | 0.001 |

Table 3.3: Experimental hyperparameters. All experiments use AdamW optimizer.

### *3.2.3 Measurement*

To evaluate the learnability of models across tasks, it is necessary to define convergence in well-rounded terms. In machine learning literature, convergence is often defined using threshold-based criteria, such as when validation loss falls below a fixed value [8], when accuracy exceeds a preset target [39], or when the improvement in validation loss across a moving window drops below an $\epsilon$ threshold, a typical early stopping strategy [22]. While effective in large-scale benchmarks, these approaches are not suitable in our setting: different RASP-L tasks yield varying intrinsic depths, making fixed thresholds unsuitable for cross-

task comparison. We could define separate thresholds for each depth, yet that defies our purpose of training in unified hyperparameters.

Instead, we define convergence as the cumulative training loss. Formally, for a task $T$, we compute:

$$C(T) = \sum_{i=1}^{N} \mathcal{L}_i(T), \tag{3.1}$$

where $\mathcal{L}_i(T)$ is the training loss at step $i$, and $N$ is the total number of training steps. This metric has two advantages. First, it provides a uniform scale across tasks, avoiding the need to calibrate task-specific thresholds. Second, it reflects the ease of learning: tasks that converge faster accumulate lower total loss, while harder tasks accumulate higher loss. This aligns directly with our research question, since simplicity bias predicts that tasks with shorter expressions (i.e., simpler structure) should yield lower cumulative loss.

This formulation is closely related to the Area Under the Loss Curve (AULC), a metric used in meta-learning and curriculum learning to evaluate learning efficiency [1]. Like AULC, cumulative loss reflects the entire training trajectory rather than only final performance. However, cumulative loss is simpler to compute and directly interpretable across tasks without normalization. It also avoids conflating differences in convergence speed with asymptotic performance, which is important in our setting where tasks vary in intrinsic difficulty.

# Chapter 4

# RESULTS

## *4.1 Learnability and Depth*

Our initial hypothesis was that decoder-only transformers would mirror human learners in exhibiting a simplicity bias: tasks expressible with shorter RASP-L expressions would be learned more easily than those requiring longer ones. Hence, we expected a upward curve of accumulated loss as depth gets deeper.

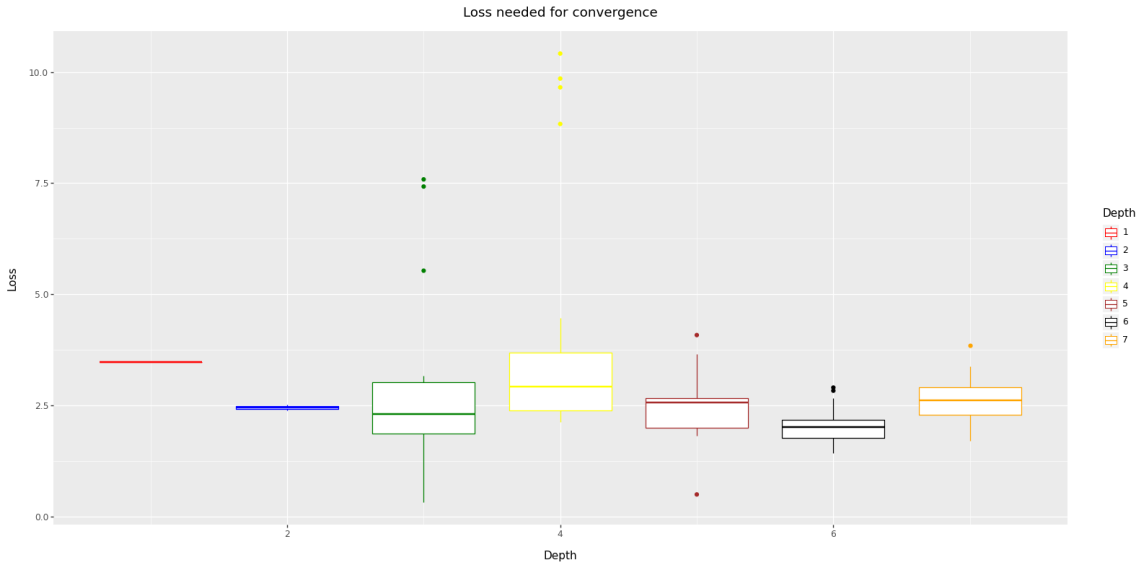


Figure 4.1: Learnability result of sampled expressions in terms of depth. X axis represents the depth of expressions while the y axis refers to the sum of validation loss.

Figure 4.1 presents box plots of total loss values grouped by depth of expression. Contrary to our hypothesis, the results show no relationship between depth and learnability. Tasks at shallow depths (1–2) did not consistently yield lower cumulative loss than deeper tasks, and vice versa. This indicates learning of transformers, artificial learners, diverges

from Feldman's analysis of human simplicity bias.

### *4.2 Learnability and Primitives*

Another approach is to analyze the complexity of individual function calls. Analogous to Feldman's observation that humans learn "orange" and "apple" at similar rates, we can hypothesize that transformers may not treat their primitive operations uniformly. That is, while humans assign similar complexity to comparable concepts, transformers may assign different difficulty levels to operations such as `tok_map` versus `seq_map`.

To better understand what drives learnability, we grouped tasks by their primitive operations. Figure 4.2 shows the average cumulative loss across tasks that contain selection-family primitives (`select` and `kqv`) versus those that do not. Since attention mechanism involves in heavier matrix computations, we grouped expressions that involves in `select` functions, including `kqv` which internally calls `select`.

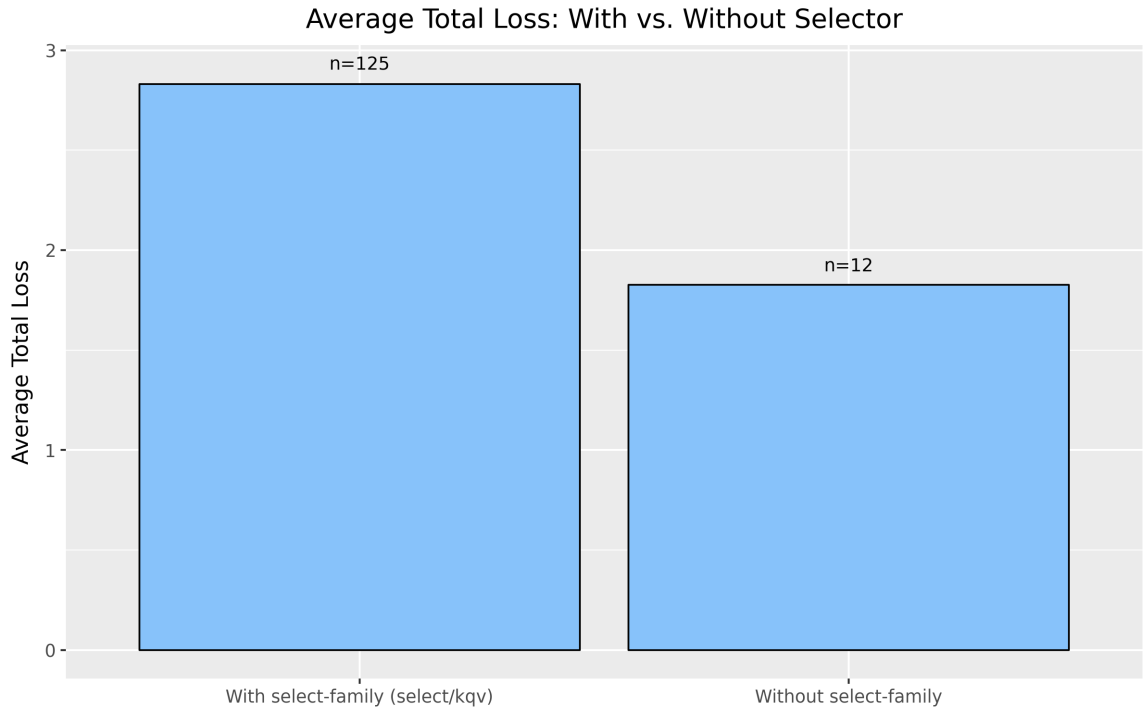


Figure 4.2: Comparison of accumulated total loss between expressions with and without select-related operations

The results reveal a promising room for further investigation. Tasks involving the selection family are substantially harder, with higher average loss. Tasks with select-family primitives had an average cumulative loss of 2.83 (n = 125), compared to 1.83 for tasks without them (n = 12). This difference is statistically significant (Welch's $t = 3.41$, $p = 0.003$; Mann–Whitney $U = 1048$, $p = 0.024$).

### *4.3 Learnability and Width*

Another perspective on transformer learnability is to examine the *width* of expressions, defined as the total number of arguments across the tree. Note that we are not counting only the leaf nodes. Intuitively, wider expressions require combining more intermediate computations and could therefore impose greater computational resource.

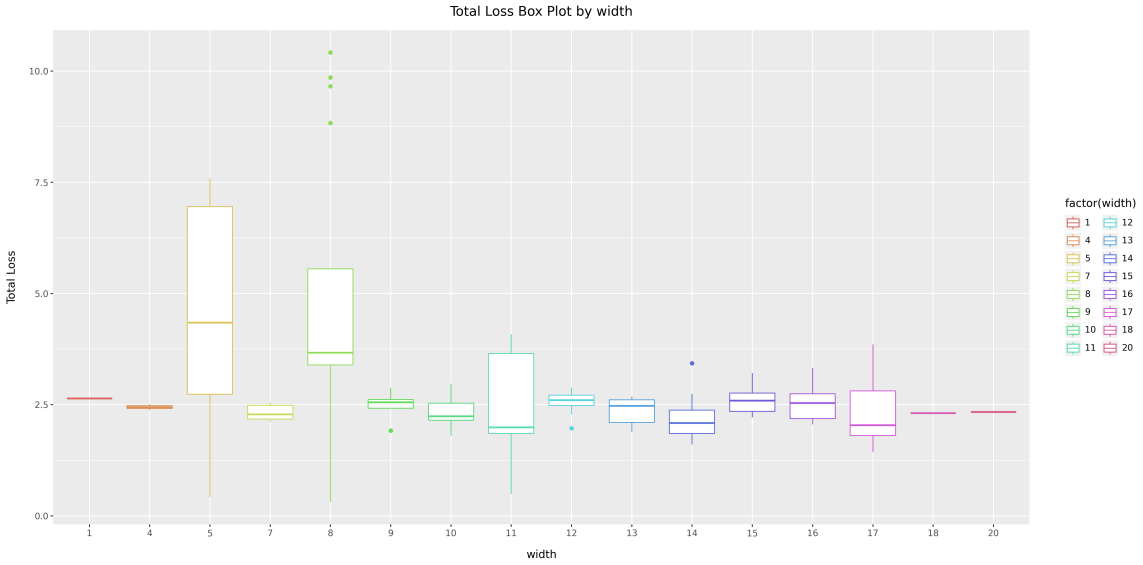


Figure 4.3: Learnability result of sampled expressions in terms of width. X axis represents the width of expressions while the y axis refers to the sum of validation loss.

In our data, expression widths ranged up to 20. However, as shown in Figure 4.3, there is no clear evidence that width correlates with learnability. Tasks with large widths did not consistently yield higher cumulative loss, suggesting that breadth of expression is not a deciding factor for the learnability of transformers.

# Chapter 5

# LIMITATIONS AND FUTUER WORKS

## *5.1 Limitations*

While our study provides new insights into the learnability of transformer primitives, it also faces several limitations. First, the coverage of our grammar is constrained. In particular, the inclusion of arbitrary functions such as `func` and `pred` threatens the "sealed" nature of RASP-L as a formal language. Our definition of ULTK-friendly RASP Grammar is also limited, as we allowed only additions for internal computation.

Second, the attention mechanism we adopt follows the "average-hard" formulation of RASP rather than the softmax-based attention used in most industrial transformer architectures. This creates a gap between theoretical analysis and empirical behavior, which is essentially what learning literature aims to reduce. Third, our enumeration of expressions was limited due to computational resources. We generated a surface depth of three and an intrinsic depth of seven. While this range was sufficient for a set of experiments, it leaves open the question of whether deeper or more complex tasks would reveal different patterns of learnability.

Finally, we didn't modify the hyperparameter for each task. OPT transformers did not struggle with any of our tasks, scoring near perfect accuracy. This potentially suggests that the task we created may not be complex enough for the transformer to show simplicity bias. In other words, our tasks could be too simple to show simplicity bias. Either developing more complex tasks or tuning the hyperparameters for each task can be the next steps in this direction.

## *5.2 Future Work*

Future work should extend our methodology to C-RASP, a variant that provides a well-defined framework with softmax attention [37]. C-RASP is more closely aligned with

industry-standard implementations of decoder-only transformers and restricts the primitives more tightly than RASP-L. This reduction of primitives could make it feasible to generate a broader range of tasks across varying depths, while ensuring that the operations remain interpretable and theoretically grounded. Such an extension would offer a more precise account of transformer expressivity and allow for stronger claims about the alignment (or misalignment) between human and model biases in concept learning.

# Chapter 6

# CONCLUSION

We analyzed whether the decoder-only transformer model shares the humans' simplicity bias. To achieve this, we compiled the shortest yet unique mappings of input–output behaviors expressible in transformers through RASP-L and evaluated transformer performance across 137 systematically generated tasks. Contrary to our hypothesis, in our experiment settings, we found no evidence that shorter expression correlates with learnability, suggesting that transformers may not carry simplicity bias. Our analysis further revealed that specific primitives, rather than expression length, could be a more reliable metric for understanding the learnability of transformers. Tasks involving selection-family operations (`select` and `kqv`) were significantly harder to learn than those without. These findings highlight that the computational cost of particular operations, especially attention-like mechanisms, could play a more significant role in transformer learning dynamics than structural minimality.

Overall, this study provides a methodological contribution of generating a large amount of algorithmic tasks learnable by transformers while demonstrating differences in learning mechanisms between artificial and human learners.

# BIBLIOGRAPHY

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

[2] Satwik Bhattamishra, Kabir Ahuja, and Navin Goyal. On the Ability and Limitations of Transformers to Recognize Formal Languages. In Bonnie Webber, Trevor Cohn, Yulan He, and Yang Liu, editors, *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 7096–7116, Online, January 2020. Association for Computational Linguistics.

[3] Satwik Bhattamishra, Arkil Patel, and Navin Goyal. On the Computational Power of Transformers and Its Implications in Sequence Modeling. In Raquel Fernández and Tal Linzen, editors, *Proceedings of the 24th Conference on Computational Natural Language Learning*, pages 455–475, Online, January 2020. Association for Computational Linguistics.

[4] Lyle E. Bourne. Knowing and using concepts. *Psychological Review*, 77(6):546–556, November 1970.

[5] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, J. Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, T. Henighan, R. Child, A. Ramesh, Daniel M. Ziegler, Jeff Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Ma-teusz Litwin, Scott Gray, B. Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, I. Sutskever, and Dario Amodei. Language Models are Few-Shot Learners. *ArXiv*, May 2020.

[6] Jacob Feldman. Minimization of Boolean complexity in human concept learning. *Nature*, 407(6804):630–633, October 2000. Publisher: Nature Publishing Group.

[7] Jacob Feldman. The Simplicity Principle in Human Concept Learning. *Current Directions in Psychological Science*, 12(6):227–232, December 2003. Publisher: SAGE Publications Inc.

[8] Ian Goodfellow, Yoshua Bengio, and Aaron Courvile. *Deep Learning*. MIT Press, 2016.

[9] Suchin Gururangan, Ana Marasović, Swabha Swayamdipta, Kyle Lo, Iz Beltagy, Doug Downey, and Noah A. Smith. Don't Stop Pretraining: Adapt Language Models to Domains and Tasks. In Dan Jurafsky, Joyce Chai, Natalie Schluter, and Joel Tetreault, editors, *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 8342–8360, Online, July 2020. Association for Computational Linguistics.

[10] Yiding Hao, Dana Angluin, and Robert Frank. Formal Language Recognition by Hard Attention Transformers: Perspectives from Circuit Complexity. *Transactions of the Association for Computational Linguistics*, 10:800–810, 2022. Place: Cambridge, MA Publisher: MIT Press.

[11] John Hewitt, Michael Hahn, Surya Ganguli, Percy Liang, and Christopher D. Manning. RNNs can generate bounded hierarchical languages with optimal memory. In Bonnie Webber, Trevor Cohn, Yulan He, and Yang Liu, editors, *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1978–2010, Online, January 2020. Association for Computational Linguistics.

[12] Sepp Hochreiter and Jürgen Schmidhuber. Long Short-Term Memory. *Neural Computation*, 9(8):1735–1780, November 1997.

[13] Nathaniel Imel, Christopher Haberland, and Shane Steinert-Threlkeld. The Unnatural Language ToolKit (ULTK). *Society for Computation in Linguistics*, 8(1), June 2025. Publisher: University of Massachusetts Amherst Libraries.

[14] Yoon Kim. Convolutional Neural Networks for Sentence Classification. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1746–1751, Doha, Qatar, 2014. Association for Computational Linguistics.

[15] Takeshi Kojima, S. Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. Large Language Models are Zero-Shot Reasoners. *ArXiv*, May 2022.

[16] John K Kruschke. ALCOVE: An Exemplar-Based Connectionist Model of Category Learning.

[17] Brenden Lake and Marco Baroni. Generalization without Systematicity: On the Compositional Skills of Sequence-to-Sequence Recurrent Networks. In *Proceedings of the 35th International Conference on Machine Learning*, pages 2873–2882. PMLR, July 2018. ISSN: 2640-3498.

[18] William Merrill, Ashish Sabharwal, and Noah A. Smith. Saturated Transformers are Constant-Depth Threshold Circuits. *Transactions of the Association for Computational Linguistics*, 10:843–856, 2022. Place: Cambridge, MA Publisher: MIT Press.

[19] Ulric Neisser and Paul Weene. Hierarchies in concept attainment. *Journal of Experimental Psychology*, 64(6):640–645, 1962. Place: US Publisher: American Psychological Association.

[20] Robert M. Nosofsky. Exemplar-based accounts of relations between classification, recognition, and typicality. *Journal of Experimental Psychology: Learning, Memory, and Cognition*, 14(4):700–708, October 1988.

[21] Long Ouyang, Jeff Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke E. Miller, Maddie Simens, Amanda Askell, P. Welinder, P. Christiano, J. Leike, and Ryan J. Lowe. Training language models to follow instructions with human feedback. *ArXiv*, March 2022.

[22] Lutz Prechelt. Early Stopping - But When? In Genevieve B. Orr and Klaus-Robert Müller, editors, *Neural Networks: Tricks of the Trade*, pages 55–69. Springer, Berlin, Heidelberg, 1998.

[23] Jorge Pérez, Javier Marinkovic, and P. Barceló. On the Turing Completeness of Modern Neural Network Architectures. *ArXiv*, January 2019.

[24] Alec Radford and Karthik Narasimhan. Improving Language Understanding by Generative Pre-Training. 2018.

[25] Alec Radford, Jeff Wu, R. Child, D. Luan, Dario Amodei, and I. Sutskever. Language Models are Unsupervised Multitask Learners. 2019.

[26] Eleanor H. Rosch. Natural categories. *Cognitive Psychology*, 4(3):328–350, May 1973.

[27] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning Internal Representations by Error Propagation. In Allan Collins and Edward E. Smith, editors, *Readings in Cognitive Science*, pages 399–421. Morgan Kaufmann, January 1988.

[28] William M. Smith, Jerome S. Bruner, Jacqueline J. Goodnow, and George A. Austin. A Study of Thinking. *The American Journal of Psychology*, 71(2):474, June 1958.

[29] Lena Strobl, William Merrill, Gail Weiss, David Chiang, and Dana Angluin. What Formal Languages Can Transformers Express? A Survey. *Transactions of the Association for Computational Linguistics*, 12:543–561, 2024. Place: Cambridge, MA Publisher: MIT Press.

[30] J. Tenenbaum. Bayesian Modeling of Human Concept Learning. December 1998.

[31] J. Tenenbaum. Rules and Similarity in Concept Learning. November 1999.

[32] Ashish Vaswani, Noam M. Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is All you Need. June 2017.

[33] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Ed H. Chi, F. Xia, Quoc Le, and Denny Zhou. Chain of Thought Prompting Elicits Reasoning in Large Language Models. *ArXiv*, January 2022.

[34] Gail Weiss, Yoav Goldberg, and Eran Yahav. Thinking Like Transformers. In *Proceedings of the 38th International Conference on Machine Learning*, pages 11080–11090. PMLR, July 2021. ISSN: 2640-3498.

[35] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Remi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander Rush. Transformers: State-of-the-Art Natural Language Processing. In Qun Liu and David Schlangen, editors, *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 38–45, Online, October 2020. Association for Computational Linguistics.

[36] Zhaofeng Wu, Linlu Qiu, Alexis Ross, Ekin Akyürek, Boyuan Chen, Bailin Wang, Najoung Kim, Jacob Andreas, and Yoon Kim. Reasoning or Reciting? Exploring the Capabilities and Limitations of Language Models Through Counterfactual Tasks. In Kevin Duh, Helena Gomez, and Steven Bethard, editors, *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pages 1819–1862, Mexico City, Mexico, June 2024. Association for Computational Linguistics.

[37] Andy Yang and David Chiang. Counting Like Transformers: Compiling Temporal Counting Logic Into Softmax Transformers. 2024.

[38] Shunyu Yao, Binghui Peng, Christos Papadimitriou, and Karthik Narasimhan. Self-Attention Networks Can Process Bounded Hierarchical Languages. In Chengqing Zong, Fei Xia, Wenjie Li, and Roberto Navigli, editors, *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 3770–3785, Online, August 2021. Association for Computational Linguistics.

[39] Wojciech Zaremba and I. Sutskever. Reinforcement Learning Neural Turing Machines. *ArXiv*, May 2015.

[40] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, Todor Mihaylov, Myle Ott,

Sam Shleifer, Kurt Shuster, Daniel Simig, Punit Singh Koura, Anjali Sridhar, Tianlu Wang, and Luke Zettlemoyer. OPT: Open Pre-trained Transformer Language Models, June 2022. arXiv:2205.01068 [cs].

[41] Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, Yifan Du, Chen Yang, Yushuo Chen, Zhipeng Chen, Jinhao Jiang, Ruiyang Ren, Yifan Li, Xinyu Tang, Zikang Liu, Peiyu Liu, Jian-Yun Nie, and Ji-Rong Wen. A Survey of Large Language Models. 2023. Publisher: arXiv Version Number: 15.

[42] Hattie Zhou, Arwen Bradley, Etai Littwin, Noam Razin, Omid Saremi, Joshua M. Susskind, Samy Bengio, and Preetum Nakkiran. What Algorithms can Transformers Learn? A Study in Length Generalization. October 2023.

# Appendix A

# GRAMMARS

## *A.1 RASP-L*

Listing A.1: RASP-L core functions.

```
import numpy as np

def full(x, const):
    """Return a sequence of the same length as x, filled with const.
       """
    return np.full_like(x, const, dtype=int)

def indices(x):
    """Return 0..len(x)-1 as an integer array."""
    return np.arange(len(x), dtype=int)

def tok_map(x, func):
    """Element-wise map: y[i] = func(x[i])."""
    return np.array([func(xi) for xi in x]).astype(int)

def seq_map(x, y, func):
    """Zipping map: z[i] = func(x[i], y[i])."""
    return np.array([func(xi, yi) for xi, yi in zip(x, y)]).astype(
       int)

def select(k, q, pred, causal=True):
    """
    Boolean selector matrix A where A[i, j] = pred(k[j], q[i]).
    If causal=True, only attend to j <= i (decoder-style causal mask
       ).
```

```
    """
    s = len(k)
    A = np.zeros((s, s), dtype=bool)
    for qi in range(s):
        # k_index <= q_index when causal
        col_range = range(qi + 1) if causal else range(s)
        for kj in col_range:
            A[qi, kj] = pred(k[kj], q[qi])
    return A

def sel_width(A):
    """Width of selection per query position: sum over keys."""
    return np.dot(A, np.ones(len(A))).astype(int)

def aggr_mean(A, v, default=0):
    """Average of selected values v per row of A (with default for
       empty)."""
    out = np.dot(A, v)
    norm = sel_width(A)
    out = np.divide(out, norm,
                    out=np.full_like(v, default, dtype=float),
                    where=(norm != 0))
    return out.astype(int)

def aggr_max(A, v, default=0):
    """Max of selected values v per row of A (with default for empty
       )."""
    out = np.full_like(v, default)
    for i, row in enumerate(A):
        idxs = np.flatnonzero(row)
        if len(idxs) > 0:
            out[i] = np.max(v[idxs])  # max over selected elements
    return out.astype(int)
```

```
def aggr(A, v, default=0, reduction='mean'):
    """Generic reduction over selections: mean, max, or min."""
    if reduction == 'mean':
        return aggr_mean(A, v, default)
    elif reduction == 'max':
        return aggr_max(A, v, default)
    elif reduction == 'min':
        return -aggr_max(A, -v, -default)
    else:
        raise ValueError(f"Unknown reduction: {reduction}")

def kqv(k, q, v, pred, default=0, reduction='mean'):
    """
    Causal K-Q-V aggregation:
    - Build selector A = select(k, q, pred)
    - Aggregate values v according to A with chosen reduction.
    """
    return aggr(select(k, q, pred), v, default=default, reduction=
       reduction)
```

### *A.2 ULTK-friendly RASP-L*

Listing A.2: Complete version of ULTK-friendly RASP-L.

```
from jaxtyping import Num
from typing import Callable
from ultk.language.semantics import Referent


global start
start = tuple[int, ...]
```

```
def inp(x: Referent) -> tuple[int, ...]:
    """
    Return 1d tuple of input sequence of ints.

    Main input function that transforms ultk vocab to valid
    input for sequence operations or selectors

    Parameter: x is any Referent based on SYMBOLS value in plug.py
       of len <= MAX_LEN
    Precondition: must be a Referent object as defined in ultk.
       langauge.semantics
    """
    result= tuple(int(value) for value in x.name)
    return result
    #return np.array(result, dtype=int)


def zero(_: Referent) -> Num:
    """
    Return the int 0

    Allows for the int 0 to be passed to sequence operations
    such as full() and tok_map().

    Parameter _: any sequence of symbols
    Precondition: Referent
    """
    return 0


def one(_: Referent) -> Num:
    """
    Return the int 1
```

```
    Allows for the int 1 to be passed to sequence operations
    such as full() and tok_map().

    Parameter _: any sequence of symbols
    Precondition: Referent
    """
    return 1


def add(_: Referent) -> Callable[[Num, Num], Num]:
    """
    Return a Callable function that adds two ints.

    Initializes a 2-ary addition function to
    be passed as an argument for func parameter
    in tok_map, seq_map_int, kqv

    Parameter _: any sequence of symbols
    Precondition: Referent
    """
    return lambda n, m: n + m


def is_equal(_: Referent) -> Callable[[Num, Num], bool]:
    """
    Return a Callable function that detects if two ints are equal.

    Initializes a 2-ary equal operator to
    be passed as an argument for func parameter
    in tok_map, seq_map_int, kqv

    Parameter _: any sequence of symbols
```

```
    Precondition: Referent
    """
    return lambda n, m: (n == m)


def less_than(_: Referent) -> Callable[[Num, Num], bool]:
    """
    Return a Callable function that detects
    if one int is smaller than another.

    Initializes a 2-ary less than operator to
    be passed as an argument for func parameter
    in tok_map, seq_map_int, kqv

    Parameter _: any sequence of symbols
    Precondition: Referent
    """
    return lambda n, m: (n < m)


def intersection(_: Referent) -> Callable[[Num, Num], bool]:
    """
    Return a Callable function that functions as an
    'and' operator on ints.

    Initializes a 2-ary and operator to
    be passed as an argument for func parameter
    in tok_map, seq_map_int, kqv

    Parameter _: any sequence of symbols
    Precondition: Referent
    """
    return lambda n, m: (n and m)
```

```
def union(_: Referent) -> Callable[[Num, Num], bool]:
    """
    Return a Callable function that functions as an
    'or' operator on ints.

    Initializes a 2-ary or operator to
    be passed as an argument for func parameter
    in tok_map, seq_map_int, kqv

    Parameter _: any sequence of symbols
    Precondition: Referent
    """
    return lambda n, m: (n or m)


def xunion(_: Referent) -> Callable[[Num, Num], bool]:
    """
    Return a Callable function that functions as an
    'xor' operator on ints.

    Initializes a 2-ary exclusive or operator to
    be passed as an argument for func parameter
    in tok_map, seq_map_int, kqv

    Parameter _: any sequence of symbols
    Precondition: Referent
    """
    return lambda n, m: ((n or m) and not (n and m))


## np-rasp core
```

```
def indices(x: tuple[int, ...]) -> tuple[int, ...]:
    """
    Return indices of the input.

    Implements sequence operation that returns
    an array representing the input array's indices.

    Parameter x: 1d tuple of ints
    Precondition: 1d tuple of at least one int
    """
    return tuple(range(len(x)))
    #return np.arange(len(x), dtype=int)
    #return np.array(range(len(x)))


# An array of shape x filled with const
def full(x: tuple[int, ...], const: Num) -> tuple[int, ...]:
    """
    Return a 1d len(x)-item tuple of const values.

    Implements sequence operation that takes an input
    1d tuple of length n and returns a 1d tuple of
    length = n where each value = const.

    Parameter x: a 1d tuple of ints
    Precondition: 1d tuple of ints

    Parameter const: an int value instantiated by zero(), one(), two
        ()
    Precondition: must be an int
    """
    return tuple(const for _ in range(len(x)))
    #return np.full_like(x, const, dtype=int)
```

```
    #return np.array([const for _ in range(len(x))])


# Apply func into elements of array x and return into integer array
def tok_map(x: tuple[int, ...],
    const: Num,
    func: Callable[[Num, Num], Num]
) -> tuple[int, ...]:
    """
    Return a 1d tuple of int outputs of a 2-ary Callable

    Implements sequence operation that takes a 1d tuple of ints
    and applies the same function with the same second parameter
    value to each element.

    Parameter x: 1d tuple of ints
    Precondition: 1d tuple of ints

    Parameter const: an int value instantiated by zero(), one(), two
       ()
    Precondition: must be an int

    Parameter func: a function that takes two ints and returns an
       int, typically add, subtract,
    Precondition: a Callable with two int parameters
    """
    return tuple(func(xi, const) for xi in x)
    #return np.array([func(xi, const) for xi in x]).astype(int)


# Apply func into elements pair of array x and y, and return into an
    integer array
def seq_map(
```

```
    x: tuple[int, ...],
    y: tuple[int, ...],
    func: Callable[[Num, Num], Num]
) -> tuple[int, ...]:
    """
    Return a 1d tuple of int outputs of a 2-ary Callable

    Takes 2 int tuples, applies an element wise operation, and
    returns the output, a tuple of ints.

    Parameter x: 1d tuple of ints
    Precondition: a tuple of at least one int

    Parameter y: 1d tuple of ints
    Precondition: a tuple of at least one int

    Parameter func: a 2-ary int operation
    Precondition: a Callable with two int parameters
    """
    return tuple(func(xi, yi) for xi, yi in zip(x, y))


# Creates selection matrix A, applies pred to compare the elements
   of k and q
# If causal, compare the ealier indice
# def select(
#    k: tuple[int, ...], q: tuple[int, ...], pred: Callable[[int,
   int], int], causal: bool = True
# ) -> tuple[tuple[bool, ...], ...]:
def select(k: tuple[int, ...],
    q: tuple[int, ...],
    pred: Callable[[Num, Num], bool]
) -> tuple[tuple[bool, ...], ...]:
```

```
    """
    Return a 2d tuple of the element-wise outputs of a 2-ary
       Callable

    Takes 2 1d int tuples, applies an element wise operation, and
    returns the output as a tuple of int-cast bools of shape (len(q)
       ,len(k)).

    Parameter x: any 1d tuple of ints
    Precondition: 1d tuple of at least one int, len(x) == len(y)

    Parameter y: a 1d tuple of ints
    Precondition: a 1d tuple of at least one int, len(x) == len(y)

    Parameter func: a 2-ary comparison, returns a bool
    Precondition: a Callable with two int parameters that returns a
       bool
    """
    s = len(k)
    A= [[0] * s for _ in range(s)]
    for qi in range(s):
        for kj in range(s):  # k_index <= q_index if causal
            A[qi][kj] = bool(pred(k[kj], q[qi]))
    return tuple(tuple(item) for item in A)


def sel_width(A: tuple[tuple[bool, ...], ...]) -> tuple[int, ...]:
    """
    Return a 1d tuple of ints, where the ith value represents the
    ith row's width in 2d tuple A

    The return is a 1d tuple of ints where the ith value represents
    the A's ith row's width. Width here is the number of tuple
```

```
    positions where the value is 1 (int cast T).

    Parameter A: a 2d torch tuple of int cast bools with n rows and
       m columns
    Precondition: a 2d torch tuple, values must be int 0 or int 1
    """
    result= []
    for row in range(len(A)):
        newval= sum(A[row])
        result.append(newval)
    return tuple(result)


# calculate the mean of selected values
def aggr_mean(_: Referent) -> Callable[[tuple[tuple[bool, ...],
   ...], tuple[int, ...], Num], tuple[int, ...]]:
    """
    Return a Callable function that performs a mean reduction
       similar to aggr_mean.

    This function is initialized to be passed as an argument for the
        ‘reduction‘ parameter
    in functions like aggr and kqv.

    Parameter _: any sequence of symbols
    Precondition: Referent

        Return a 1d tuple of ints, where the ith value is the mean
           of Av’s ith row.

        Takes the dot product of 2d tuple A with the 1d tuple v,
           then divides
        the ith value in the dot product’s output array by the width
```

```
        of the ith row of A. If the ith row of A has width 0, the
        mean is replaced by the default value.

        Parameter A: a 2d tuple of int cast bools with n rows and m
            columns
        Precondition: a 2d torch tuple, values must be int 0 or int
            1

        Parameter v: a 1d tuple of ints, len(v) == m
        Precondition: a 1d tuple of ints where len(v) == A.shape[1]

        Parameter default: the value used to replace any instances
            of divide by zero
        Precondition: an int
    """
    return lambda A, v, default=0: tuple(
        (sum(row[i] * v[i] for i in range(len(v))) // sel_width(A)[
            idx] if sel_width(A)[idx] != 0 else default)
        for idx, row in enumerate(A)
    )


# calculate the maximum of selected values
def aggr_max(_: Referent) -> Callable[[tuple[tuple[bool, ...], ...],
    tuple[int, ...], Num], tuple[int, ...]]:
    """
    Return a Callable function that performs a max reduction over a
        2d tuple.

    This function is initialized to be passed as an argument for the
    `reduction` parameter in functions like aggr and kqv.

    Parameter _: any sequence of symbols
```

```
    Precondition: Referent

        The callable takes a 2d tuple A and a 1d tuple v as
           parameters
        and returns a 1d tuple of ints representing the desired
           reduction
        of A's rows' with v. This reduction Ttakes the element-wise
           product
        of each row in A with the column v, then finds the resulting
            array's
        max value. The ith value in the output tuple is the max
           value of A's
        ith row's element-wise product with v. If the ith row of A
           has width
        0 (i.e. no 1 values), the default value is used.

        Parameter A: a torch tuple of int cast bools with n rows and
            m columns
        Precondition: a 2d torch tuple, values must be int 0 or int
           1

        Parameter v: a 1d tuple of ints, len(v) == m
        Precondition: a 1d tuple of ints where len(v) == A.shape[1]

        Parameter default: the value used to replace any of A's zero
           -width rows
        Precondition: an int
    """
    return lambda A, v, default=0: tuple(
        max(row[i] * v[i] if row[i] else default for i in range(len(
           v)))
        for row in A
    )
```

```
def aggr_min(_: Referent) -> Callable[[tuple[tuple[bool, ...], ...],
    tuple[int, ...], Num], tuple[int, ...]]:
    """
    Return a Callable function that performs a min reduction similar
        to aggr_min.

    This function is initialized to be passed as an argument for the
        `reduction` parameter
    in functions like aggr and kqv.

    Parameter _: any sequence of symbols
    Precondition: Referent

        Return a 1d tuple of ints, where the ith value is the min
            value of Av's ith row.

        Finds the min by calling the aggr_max function, after
            mutliplying
        each element of v by -1, the default by -1. So aggr_max will
             return
        the greatest value (the least negative) for each row of Av,
            which
        is then multiplied by -1 to get the min value.
        Takes the element-wise product of each row in A with the
            column v,
        then finds the resulting vector's min value. The ith value
            in the
        output tuple is the min value of A's ith row's element-wise
            product
        with v. If the ith row of A has width 0 (i.e. no 1 values),
        the default value is used.
```

```
        Parameter A: a tuple of int cast bools with n rows and m
            columns
        Precondition: a 2d tuple, values must be int 0 or int 1

        Parameter v: a 1d tuple of ints, len(v) == m
        Precondition: a 1d tuple of ints where len(v) == A.shape[1]

        Parameter default: the value used to replace any of A's zero
            -width rows
        Precondition: an int
    """
    return lambda A, v, default=0: tuple(
        -max(row[i] * (-v[i]) if row[i] else -default for i in range
            (len(v)))
        for row in A
    )


def kqv(
    k: tuple[int, ...],
    q: tuple[int, ...],
    v: tuple[int, ...],
    pred: Callable[[Num, Num], bool],
    default: Num = 0,
    reduction: Callable[[tuple[tuple[bool, ...], ...], tuple[int,
        ...], Num], tuple[int, ...]] = aggr_mean,
) -> tuple[int, ...]:
    """
    Return a 1d tuple of ints representing the desired reduction of
        a
    (q,k) matrix's rows' with v.
```

```
    Creates a 2d tuple table A of int cast bools by comparing q and
       k using
    some input pred, then applies the input reduction function
    aggr_mean or aggr_mean to A and v to find the mean, min, or max
    value of each row of A multiplied element-wise by v.

    Parameter k: a 1d tuple of ints
    Precondition: a 1d tuple of ints, len(k) == len(q)

    Parameter q: a 1d tuple of ints
    Precondition: a 1d tuple of ints, len(q) == len(k)

    Parameter v: a 1d tuple, len(v) == len(k)
    Precondition: a 1d tuple of ints, len(v) == len(k)

    Parameter default: value replacing any of the (q,k) matrix's
       zero-width rows
    Precondition: an int

    Parameter reduction: a function that reduces a vector of values
       to a single, representative value
    Precondition: a Callable in [aggr_mean(), aggr_max(), aggr_min()
       ]
    """
    return reduction(select(k, q, pred), v, default=default)
```