# Representing Rule-based Chatbots with Transformers

**Dan Friedman** [1] **Abhishek Panigrahi** [1] **Danqi Chen** [1]

## Abstract

Transformer-based chatbots can conduct fluent, natural-sounding conversations, but we have limited understanding of the mechanisms underlying their behavior. Prior work has taken a bottom-up approach to understanding Transformers by constructing Transformers for various synthetic and formal language tasks, such as regular expressions and Dyck languages. However, it is not obvious how to extend this approach to understand more naturalistic conversational agents. In this work, we take a step in this direction by constructing a Transformer that implements the ELIZA program, a classic, rule-based chatbot. ELIZA illustrates some of the distinctive challenges of the conversational setting, including both local pattern matching and long-term dialog state tracking. We build on constructions from prior work—in particular, for simulating finite-state automata—showing how simpler constructions can be composed and extended to give rise to more sophisticated behavior. Next, we train Transformers on a dataset of synthetically generated ELIZA conversations and investigate the mechanisms the models learn. Our analysis illustrates the kinds of mechanisms these models tend to prefer—for example, models favor an induction head mechanism over a more precise, position based copying mechanism; and using intermediate generations to simulate recurrent data structures, like ELIZA's memory mechanisms. Overall, by drawing an explicit connection between neural chatbots and interpretable, symbolic mechanisms, our results offer a new setting for mechanistic analysis of conversational agents.[1]

[1]Princeton Language and Intelligence (PLI), Princeton University. Correspondence to: Dan Friedman <dfriedman@princeton.edu>.

[1]Code and data for reproducing the experiments are available at https://github.com/princeton-nlp/ELIZA-Transformer.

## 1 Introduction

State-of-the-art Transformer-based chatbots such as ChatGPT have remarkable capability of conducting fluent, natural-sounding conversations, but we have a limited understanding of the underlying mechanisms. One approach to understanding Transformers is to use constructions: identifying explicit mechanisms that a Transformer could theoretically use to solve a particular task. Prior work has constructed Transformers for a variety of synthetic and formal language tasks, including regular languages (Bhattamishra et al., 2020a; Liu et al., 2023), Dyck languages (Yao et al., 2021), and PCFGs (Zhao et al., 2023). However, this line of work has focused mainly on single-sentence tasks, and how to extend these approaches to more naturalistic conversational settings remains as an open question. In this work, we propose to use *rule-based chatbots* for formal and mechanistic analysis of neural conversational agents. First, we construct a Transformer that implements a classic rule-based chatbot algorithm, and then we use this construction to inform a series of empirical investigations into how Transformers learn conversational tasks.

In particular, we focus on ELIZA (Weizenbaum, 1966), one of the first artificial chatbots. The ELIZA algorithm is simple but exhibits a number of sophisticated conversational behaviors (Fig. 1). The majority of ELIZA's behavior is based on local pattern/transformation rules: ELIZA compares the user's input to an inventory of templates, and responds by reassembling the input according to an associated transformation rule. However, ELIZA also employs several mechanisms that make use of the full conversational history, including a mechanism for varying its responses between successive turns, and a "memory queue" to refer to turns from the beginning of the conversation. The resulting conversations can be surprisingly naturalistic, with early users ascribing emotion and understanding to the program (Weizenbaum, 1976). ELIZA therefore offers a natural next step from simpler, sentence-level settings, comprising both local pattern matching and long-distance dialog state tracking.

In the first part of the paper (Sec. 3), we describe how to implement the ELIZA algorithm with a decoder-only Transformer (Vaswani et al., 2017) (Fig. 2). We start by showing how we can use constructions from prior work as

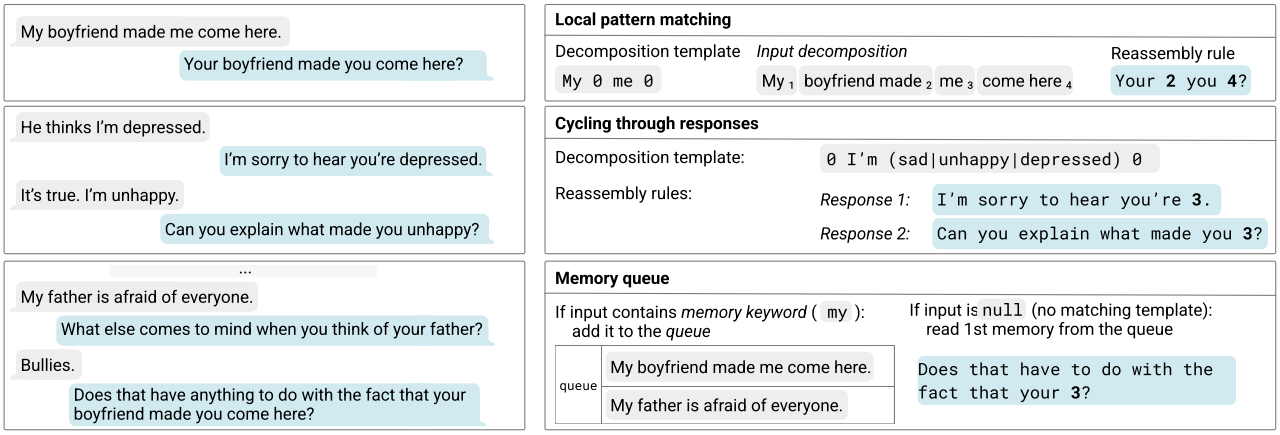



Figure 1: An example of an ELIZA conversation, adapted from (Weizenbaum, 1966). ELIZA uses both local pattern matching and two long-term memory mechanisms (cycling through responses, and a memory queue). At each turn, ELIZA compares the most recent input to an inventory of *decomposition templates* and applies one of the associated *reassembly rules*. If a template is matched more than once in a conversation, ELIZA cycles through a list of possible reassembly rules before repeating a response. If the input contains a special keyword ("my"), ELIZA stores it in a *memory queue*. Later, if an input does not match any of the templates, ELIZA reads the first memory from the queue.

modular building blocks—in particular, by decomposing the task into a cascade of finite state automata (Liu et al., 2023; Angluin et al., 2023), along with a copying mechanism for generating responses. This decomposition attests to the usefulness of algebraic automata as building blocks for characterizing complex behavior in Transformers. On the other hand, we also identify alternative constructions for key subtasks, including a more robust copying mechanism (Sec. 3.2) and memory mechanisms (Sec. 3.3) that make use of intermediate ELIZA outputs—akin to a scratchpad (Nye et al., 2021) or Chain-of-Thought (Wei et al., 2022b). These alternative constructions inform our empirical investigations later on. Incidentally, the ELIZA framework happens to be Turing complete (Hay & Millican, 2022); our results therefore lead to a simple, alternative construction for a Transformer that simulates a Turing machine, which we discuss in Appendix B.4.

In the second part of the paper, we generate a dataset of ELIZA transcripts and train Transformers to simulate the ELIZA algorithm (Sec. 4.1). First we investigate which aspects of the task are more difficult for the models to learn, finding that models struggle the most with precise copying and with the memory queue mechanism—which requires the composition of several distinct mechanisms (Sec. 4.2). Next, we investigate which of our hypothesized mechanisms better match what the models learn, and how the result varies according to the data distribution (Sec. 4.3). For copying, we find that models have a strong bias for an induction head mechanism (Olsson et al., 2022), leading to worse performance on sequences with a high degree of internal repetition. For the memory components, we find that models make use of intermediate outputs to simulate the relevant data structures, underscoring the importance of considering intermediate computation in understanding Transformers, even without an explicit scratchpad or Chain-of-Thought. Together, our results illustrate that ELIZA offers a rich setting for mechanistic analysis of learning dynamics, allowing us to decompose the task into subtasks, conduct fine-grained behavioral analysis, and connect this analysis to predictions about the model's mechanisms.

Overall, by drawing an explicit connection between neural chatbots and interpretable, symbolic mechanisms, our results offer a new setting for algorithm-level understanding of conversational agents. We conclude by discussing the broader implications of our results for future work on interpretability and the science of large language models.

## 2 Background: ELIZA

We start by describing the ELIZA algorithm (Weizenbaum, 1966), following the presentation of Jurafsky & Martin (2020). The ELIZA algorithm can be decomposed into two types of behavior: local pattern matching and long-term memory, illustrated in Fig. 1. We discuss ELIZA in more detail in Appendix A.

### 2.1 Local Pattern Matching

First, ELIZA compares the most recent user input to an inventory of pattern/transformation rules, such as the following:

`0 YOU 0 ME` → What makes you think I 3 you?

The left-hand side of the rule is called a *decomposition template* and corresponds to a simple regular expression, where the `0` symbol is a wildcard that matches 0 or more occurrences of any word. If an input matches a template, it is partitioned into a set of *decomposition groups* corresponding to the wildcards. For example, the input "It seems like you hate me" would be decomposed into four groups: (1) It seems like (2) you (3) hate (4) me. The right-hand side of the rule is called a *reassembly rule*, and a response is generated by replacing any number in the reassembly rule with the content of the corresponding decomposition group. In this case, ELIZA will respond, "What makes you think I hate you?" An ELIZA chatbot is defined by an inventory of these rules, which are organized into a configuration file known as the *script*. Each decomposition template is assigned a rank and associated with one or more reassembly rules. Given an input, ELIZA finds the highest ranked template that matches the sentence and applies one of the associated reassembly rules. The script also must assign some reassembly rules to a *null template*, which is used when none of the other templates matches.

### 2.2 Long-Term Memory

While most responses consider only the previous utterance, ELIZA also includes two mechanisms for referring to information from earlier in the conversation.

**Cycling through reassembly rules** First, each template in a script can be associated with a list of reassembly rules. If the template is matched multiple times in a conversation, ELIZA will cycle through all of the reassembly rules in the list before returning to the first item. For example, in Weizenbaum's ELIZA script, if the input contains the word "sorry," ELIZA will initially respond with "Please don't apologize." If the user says "sorry" a second time, ELIZA will say "Apologies aren't necessary." If the user contains to say "sorry", ELIZA will eventually say "I've told you that apologies are not required," and then cycle back to the first rule in the list.

**Memory queue** Second, if an utterance contains a particular keyword (by default, the word "my"), ELIZA stores it in a queue, referred to as the *memory queue*. Later in the conversation, if the user's input does not match any of the templates, ELIZA will output the first item in the queue, applying one of a set of memory reassembly rules. For example, at the beginning of the conversation in Fig. 1, the user states "My boyfriend made me come here." Many turns later, the user enters a sentence that does not match any of the patterns, and ELIZA replies, "Does that have anything to do with the fact that your boyfriend made you come here?"

## 3 Constructions

Now we present our constructions for implementing the ELIZA program with a Transformer decoder. We divide the constructions into four subtasks, illustrated in Fig. 2. We describe the constructions at a high-level in this section and defer the details to Appendix B.

**Setup** We consider a decoder-only Transformer with softmax attention. At each turn in the conversation, the input will be the concatenation of the conversation so far, with each user input and each ELIZA response preceded by a special delimiter character, either `u:` or `e:`, respectively. The constructions use no positional encodings, as we can use the self-attention mask to infer positional information (Haviv et al., 2022; Kazemnejad et al., 2023), and to segment the input into turns, in order to restrict attention to a particular utterance. See Appendix B.1 for more details.

### 3.1 Local Pattern Matching

We start by considering a single turn in the conversation, which involves first finding a template that matches the input, and then generating a response using the associated transformation rule.

**Matching templates** For template matching, we make use of the fact that ELIZA templates are equivalent to star-free regular expressions; these can be recognized by simulating a corresponding finite-state automaton. We build on the constructions of Liu et al. (2023); Angluin et al. (2023). At a high level, we can recognize a template with $L$ symbols using a Transformer with $L$ layers. At each layer $\ell$ and position $i$, the Transformer determines whether the input matches the first $\ell$ symbols of the template at position $i$. The final output can be used to both (a) determine if an input matches a template, and (b) decompose the input according to the template's decomposition groups. Our constructions recognize multiple templates in parallel using two attention heads per layer—one attending uniformly to the full prefix, and one attending to the previous position. The depth of the Transformer therefore scales with the length of the longest template in the configuration script, and the width scales with the total number of templates in the script. See Appendix B.2 for more details.

**Generating a response** Now we assume that we have identified a matching template and that the embedding for each input token identifies the decomposition group to which that token belongs. The next step is now to apply the chosen reassembly rule to the input to generate a response. At each generation step, the model needs to either generate a constant word (defined by the reassembly rule), or copy a word from one of the decomposition groups of the user's input. We focus here on two high-level copying

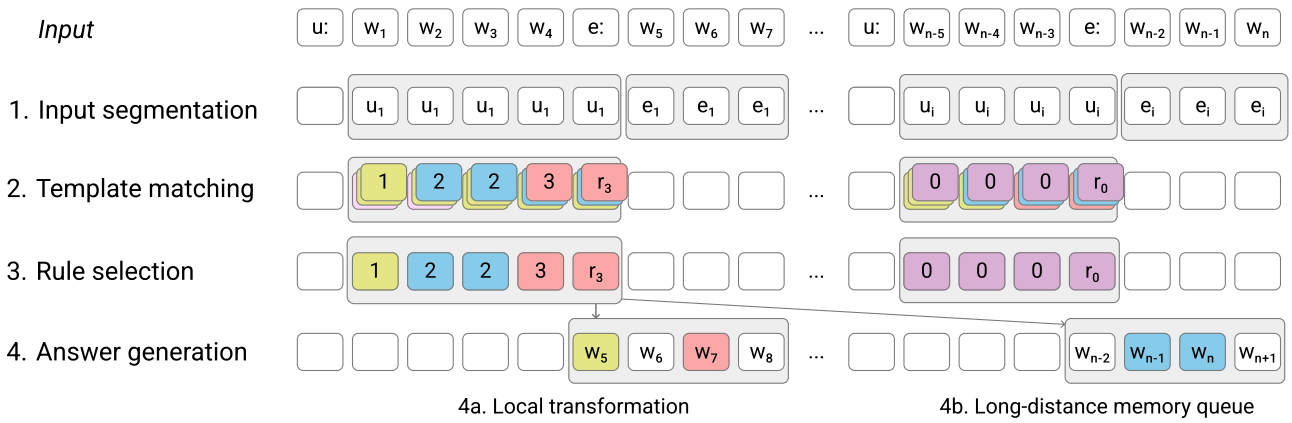



Figure 2: The input to the Transformer is the conversation history, consisting of user inputs (beginning with `u:`) followed by ELIZA's responses (`e:`). The constructions then have four parts. First, the input is divided into segments, each corresponding to a user input or ELIZA response. Second, the model attempts to match each user input to a *decomposition template*; this step is executed in parallel, with each input compared to every possible decomposition template. The model then identifies the highest scoring template and selects a *transformation rule*, taking into account the number of times this template has been matched earlier in the conversation. Finally, the model generates an answer, either by applying a transformation rule to the most recent user input (*4a*) or by transforming an input from earlier in the conversation, using the "memory queue" mechanism (*4b*).

mechanisms, deferring the precise details to Appendix B.3.

**Option 1: Content-based attention (induction head)** The first possible approach is based on the induction head (Olsson et al., 2022). This mechanism has been widely studied in prior work and is considered a key primitive in Transformers (e.g. Reddy, 2024; Singh et al., 2024; Akyürek et al., 2024; Edelman et al., 2024). In our setting, we define an induction head as follows: Given an input sequence $w$, at each output position $i$, an induction head attends to an input position $j$ such that $w_{i-n}, \ldots, w_i = w_{j-n-1}, \ldots, w_{j-1}$, and copies the token value $w_j$ (where $n$ is some context size). This mechanism has a key drawback: as noted by Zhou et al. (2023), this mechanism assumes that each word has a unique $n$-gram prefix, so it can fail if the same $n$-gram appears more than once in the input sequence.

**Option 2: Position-based attention** To avoid these shortcomings, we propose a second option that uses position rather than content to identify the next word to copy. Observe that, at each step, we can identify the position to copy next as a function of the reassembly rule; the number of tokens generated so far; and the number of tokens in each decomposition group. This can be accomplished using an attention layer to obtain the relevant counts, and a feedforward layer to calculate the target position. (See Appendix Fig. 10 for details.) Compared to the induction head, this mechanism works equally well regardless of the content of the copying segment. The drawback of this approach is that it relies on precise position arithmetic. This type of position arithmetic might not generalize to longer positions, which is why Zhou et al. (2023) do not allow it in RASP-L, their easily-learnable subset of RASP.

### 3.2 Cycling through Reassembly Rules

Now we turn to the first subtask that makes use of information from earlier in the conversation: cycling through reassembly rules. Specifically, we allow each template $t$ to be associated with a sequence of reassembly rules $r_1, \ldots, r_M$. When template $t$ appears in a conversation for the $i^{th}$ time, the model should respond with rule $r_{i\%M}$. We consider two mechanisms, illustrated in Fig. 11.

**Option 1: Modular prefix sum** One natural option is to use the modular prefix sum mechanism described by Liu et al. (2023): an attention head counts the number of times $t$ has been matched, and an MLP outputs the result modulo $M$. We anticipate that such a mechanism might perform worse as the sequence grows longer, as the model must attend over a longer sequence and process a larger count. Additionally, different templates can have a different numbers of reassembly rules, so the model must learn a separate modulus for each template.

**Option 2: Intermediate outputs** The model can avoid modular arithmetic by making use of its earlier outputs. Specifically, the model can reuse the template matching mechanism to identify outputs where it responded to template $t$ with any of $r_1, \ldots, r_M$. The model can then attend

to the most recent of these responses $r_i$, and respond with $r_{(i+1)\%M}$. This mechanism works regardless of the cycle number. However, it would fail if the same reassembly rule appears more than once in the list, or if the reassembly rules are difficult to identify.

### 3.3 Memory Queue

Finally, we incorporate the memory queue component. Recall that ELIZA adds a user input to the memory queue if it contains a special memory keyword (e.g. "my") and matches an associated template. ELIZA reads an item from the memory queue if (a) the most recent input does not match any templates and (b) the queue is not empty. Given the output of the template-matching stage, is simple to determine whether an input represents an `enqueue` event or a `no_match` event. The main challenge is to determine whether there are any items in the queue, and so whether a given `no_match` input should trigger a `dequeue`. Again, we present two mechanisms, illustrated in Fig. 11

**Option 1: Gridworld automaton** The first approach we consider is to use the construction from Liu et al. (2023) for simulating a one-dimensional "gridworld" automaton, which has $S$ numbered states and two actions: "increment the state if possible" and "decrement the state if possible." At each `enqueue` event, the automaton increments the state if possible, and at each `no_match` event, the model decrements the state if possible. If the state is decremented, we can conclude that this input should trigger a `dequeue`. We can then calculate the number of dequeues in the sequence, $d$, and read the $d^{th}$ memory in the queue. Liu et al. (2023) present a gridworld construction with two Transformer layers and $2S$ attention heads, which would allow us to implement a memory queue with a maximum size of $S$.

**Option 2: Intermediate outputs** Alternatively, as above, we can instead identify `dequeue` operations by examining earlier ELIZA outputs. By reusing the template matching mechanism, we can check whether an ELIZA response matches one of reassembly rules associated with the dequeue operation. Then, letting $d$ denote the number of dequeue operations, if $d$ is less than the number of enqueue operations, we read the $d^{th}$ memory from the queue. Compared to the gridworld approach, this construction uses fewer attention heads and does not limit the size of the memory queue, but it does impose a limit on the total number of enqueues (because we need to embed the number of enqueues to attend to the right memory).

## 4 Experiments

Now we investigate how Transformers learn this ELIZA program in practice when we train them on conversation transcripts. First, we study how well the models perform, with the goal of understanding which aspects of the task are more difficult. In the second part of the section, we examine the internal properties of the model to understand how the learned solutions compare to our construction.

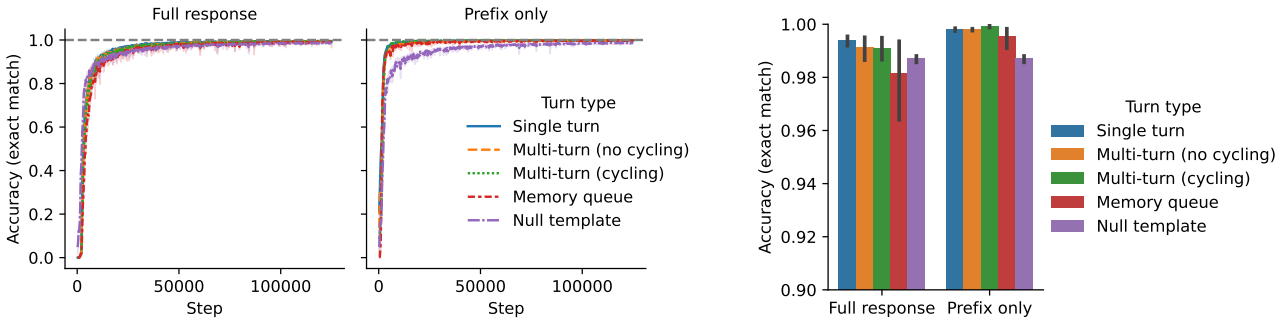
### 4.1 Experiment Setup

**Generating data** For these experiments, we generate synthetic ELIZA data. For our main experiments, we first sample a configuration script consisting of 32 templates, each containing 2-4 wildcard symbols, with up to five reassembly rules per template. We ensure that each reassembly rule begins with a unique two-letter prefix; this will provide a proxy for distinguishing rule recognition errors from copying errors. Given a script, we sample multi-turn conversations with up to 512 words. At each turn, we sample a template, and then sample a sentence that matches that template by replacing each wildcard with 0-10 words sampled uniformly from the vocabulary, and then generating a response according to the ELIZA rules. The vocabulary consists of the 26 lowercase letters. Details about data generation are provided in Appendix C.1.

**Model and training** We train Transformers with eight layers, twelve attention heads per-layer, and a hidden size of 768. We use the GPT-2 architecture but remove the position embeddings and train all models from scratch. The models are trained to predict the ELIZA responses (and not the user inputs). See Appendix C.2 for more details.

### 4.2 Which Aspects are Harder to Learn?

We start by training Transformers on ELIZA data and measuring how well they perform on the different subtasks. Here we fix the script parameters to the values described in Appendix C.1. In Figure 3, we plot the accuracy over the course of training and at the final checkpoint. The *Full response* accuracy is the per-turn exact match accuracy. The *Prefix only* accuracy is the accuracy on the two-word prefix of the response, which we ensure is unique for each reassembly rule. This metric provides a proxy for distinguishing whether errors are due to either (a) failure to identify the correct rule, or (b) failure to implement the rule correctly. We additionally break down the results by turn type, defined as follows: *Single-turn:* The first response in the conversation. *Multi-turn (no cycling):* The response for the first instance of a template in the conversation. *Multi-turn (cycling):* The response for a template that has already appeared at least once in the conversation. *Memory queue:* Responses that read from the memory queue. *Null template:* Responses to inputs that do not match any templates, when the memory queue is empty.




(a) Accuracy (training curve).

(b) Accuracy (end of training).

Figure 3: Turn-level accuracy of Transformers trained on ELIZA conversations over the course of training (Fig. 3a) and at the final checkpoint (Fig. 3b), for models trained with three random seeds. Transformers quickly learn to identify the correct reassembly rule (measured by *Prefix only* accuracy), and take longer to learn to implement the transformation correctly (*Full response*). Accuracy is slightly worse on multi-turn and memory queue examples; see §4.2.

**Accuracy by subtask** In Figure 3a, we see that the models quickly learn to identify the correct action (as measured by prefix accuracy), achieving near-perfect accuracy on almost all categories. Interestingly, the exception is the null template, which is used when the input does not match any other pattern and the memory queue is empty. Looking at the final checkpoint (Fig. 3b), we see that accuracy is high, but still imperfect, with slightly worse performance in the multi-turn setting. In the remainder of the section, we examine these errors in more detail to better understand which aspects of the task are more difficult to learn.

**Error analysis** In Figure 4, we test whether the model's errors are correlated with various properties of the input. We identify two main issues. First, the models seem to struggle with precise copying. In Fig. 4a, we see that accuracy is strongly correlated with the total number of tokens the model has to copy, and only slightly correlated with the complexity of the decomposition rule (defined as the number of distinct copying segments in the transformation). Similarly, Fig. 4b (*left*) shows that memory queue accuracy decreases with the distance between the current turn and the target memory, perhaps indicating issues with long-distance copying. Second, some errors seem to be related to tracking the state of the memory queue. Fig. 4b (*right*) shows that accuracy is negatively correlated with the total number of enqueue and dequeue operations in the sequence. Fig. 4c shows that the model performs perfectly on null inputs, provided that there have been no memory turns; accuracy decreases with the number of enqueues, indicating that the models struggle when the queue has been used but is now empty.

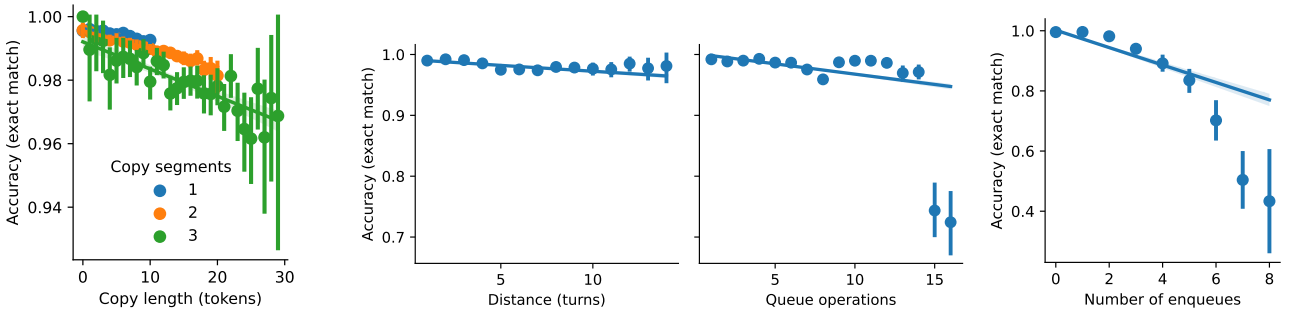
### 4.3 Which Mechanisms Do Transformers Learn?

Now we turn to the internal properties of the model to try to understand what mechanisms they learn and how they compare to our construction.

**Comparing copying mechanisms** In Section 3.1, we identified two possible mechanisms for copying: an induction head, which attends based on the content of the input, and a counting-based mechanisms that attends based on position. We predicted that the induction head will fail when the same $n$-gram appears more than once in the input, while the counting mechanism will generalize. To explore which mechanism the models seem to learn, we generate (single-turn) datasets that vary in how likely it is for the same $n$-gram to appear multiple times in a sequence. This property is controlled by a parameter $\alpha$, with $\alpha < 1$ corresponding to more repetition of $n$-grams and $\alpha > 1$ making it more likely that most $n$-grams are unique.[2] See Fig. 5a for examples.

We start by training models on the four different datasets and evaluating how well they generalize to datasets with more or less repetition. This result is plotted in Figure 5b. The model trained with the least amount of repetition ($\alpha = 100$) performs well in-domain but suffers severe degradation on data with more repetition; this provides preliminary evidence that, in our default setting, models learn an induction

[2]Specifically, given a template, we generate a sentence as follows: For each wildcard in the sentence, we sample a vector $\mathbf{p} \sim \text{Dirichlet}(\alpha\mathbf{1})$, where $\mathbf{1}$ is a 26-dimensional vector of all 1's and $\alpha$ is the *concentration* parameter. With $\alpha < 1$, $\mathbf{p}$ is more likely to concentrate most probability on a small number of items, meaning each segment is more likely to contain repeated $n$-grams. With $\alpha > 1$, $\mathbf{p}$ is more likely to be close to the uniform distribution (corresponding to our setting in the previous section). See Appendix C.1 for more details.

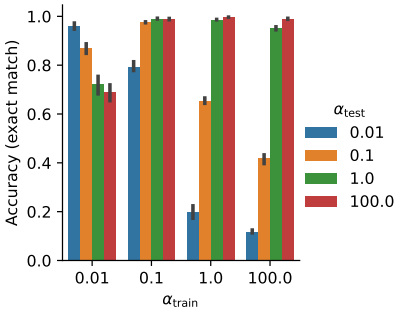



(a) Copying. (b) Dequeues. (c) *Null* inputs.

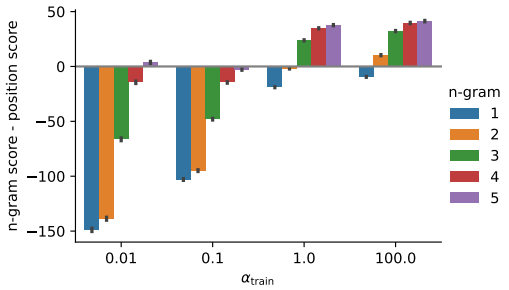
Figure 4: Which aspects of the task are most difficult for Transformers to learn? **Copying** (Fig. 4a): Accuracy decreases considerably with the total number of tokens to copy, and decreases slightly with the number of distinct copying segments. **Memory queue** (Fig. 4b): The dequeue accuracy decreases when there is a greater distance to the target memory and when there have been more total queue operations earlier in the sequence. **Null template** (Fig. 4c): The models do perfectly on *null* inputs provided there have been no memory turns in the sequence; accuracy decreases with the number of enqueues, indicating that the models struggle when the queue has been used but is now empty.

| $\alpha$ | Example sentence |
|---|---|
| 0.01 | c e e e e b p p f z f z z p w |
| 0.1 | c p x p p b r r r d r n r w |
| 1 | c y k g p f x j b c j d h c w |
| 100 | c r o k s r b y u j s a p g w |

(a) Examples.




(b) Comparing generalization.




(c) Comparing mechanisms.

Figure 5: We train and test models on datasets that vary in whether copying segments are more or less likely to contain the same $n$-gram multiple times (Fig. 5a). Models generalize poorly to data with significantly more or less repetition compared to the training distribution (Fig. 5b). Fig. 5c suggests that models trained on less repetitive data assign higher attention scores to tokens with matching contexts, rather than calculating the correct target position. See §4.3.

head mechanism that does not generalize when $n$-grams can repeat. On the other hand, models trained on the most repetitive data ($\alpha = 0.01$) generalize poorly to higher values of $\alpha$. The best-generalizing model is trained with a $\alpha = 0.1$, suggesting that some moderate amount of repetition is needed to learn a robust mechanism. In Appendix Fig. 13, we plot these results over the course of training, indicating that the most repetitive data also takes longer to learn.

To get a sense of what mechanism these models actually learn, we examine the final layer attention heads. Specifically, given an ELIZA response, for each output position $i$, we calculate the position $j$ of the input token that should be copied next. Then we calculate the average pre-softmax attention score between the query embedding at position $i$ and key embeddings drawn from other validation examples that satisfy one of two conditions: either the key has same $n$-gram prefix as the query $i$, but appears at a position $k \neq j$; or the key appears at position $j$ but has a different $n$-gram prefix ($w_{i-n:i} \neq w_{j-n-1:j-1}$). In Figure 5c, we plot the difference between these scores for different $n$-gram windows, averaging over attention heads, with positive values indicating that the model assigns higher scores to content than position. (We plot the results for each attention head in Appendix Fig. 14.) When $\alpha \geq 1$, the models prefer content to position once there is a prefix match of at least three tokens in length. For all models, the content score increases with the length of the matching $n$-gram, with a steeper increase when $\alpha < 1$. The model trained with a moderate amount of repetition ($\alpha = 0.1$) generalizes the best and is also the only model that prefers position to content even at the longest context window. While all models are sensitive to content to some extent, the results illustrate how chang-

ing the data distribution can influence which mechanism the model uses, and how well they generalize as a result.

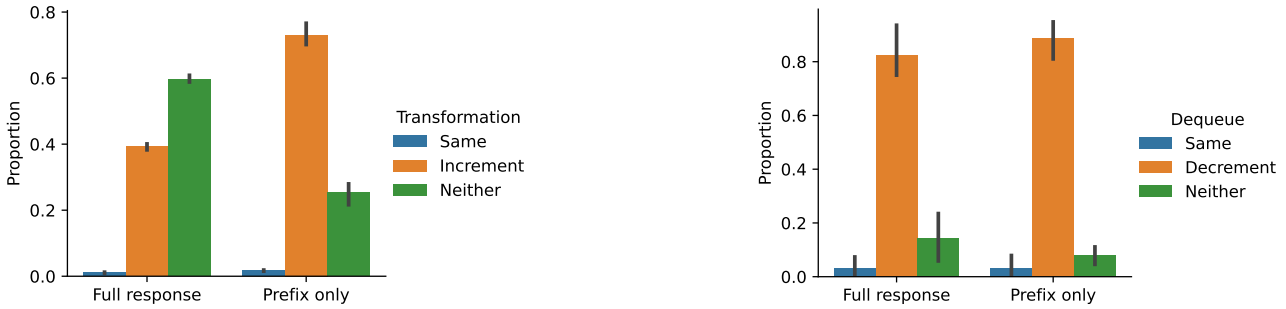
**Comparing memory mechanisms** Finally, we examine which mechanism the models learn for the two subtasks that rely on information from earlier in the conversation: cycling through reassembly rules, and the memory queue. In Section 3.2 and 3.3, we offered two possible constructions for each subtask: one construction based on simulating an automaton and one based on processing previously generated outputs. Here, we designed counter-factual experiments to test whether the model is sensitive to previous intermediate responses. For each mechanism, we edited the model's response to an intermediate turn in the sequence and then tested the model's response at a subsequent turn. (See Appendix C.3 for details.) In Figure 6, we test whether the response is consistent with the automaton construction, which predicts that the reponse will be unchanged (*Same*); the intermediate-output construction, which predicts that the response will change in a specific way—either incrementing the cycle counter or reading a memory from earlier in the clue; or whether it matches neither prediction. In both cases, the model's behavior is most consistent with the intermediate-output hypothesis, either incrementing the cycle counter or decrementing the memory queue counter as predicted. This result illustrates the importance of considering intermediate outputs in understanding Transformer behavior, even without an explicit scratchpad or chain-of-thought.

## 5 Discussion and Related Work

**Expressivity with formal languages** Numerous works have formalized the expressive power of Transformers on formal languages. Pérez et al. (2021); Pérez et al. (2019); Bhattamishra et al. (2020b) show that Transformers with hard attention are Turing complete, and Wei et al. (2022a) study their statistical learnability. Merrill et al. (2022); Merrill & Sabharwal (2023); Hao et al. (2022); Hahn (2020) further distinguish the expressivity of transformers with different hard attention patterns. Other works have investigated encoding specific algorithms in smaller simulators, e.g. bounded-depth Dyck languages (Yao et al., 2021), modular prefix sums (Anil et al., 2022), adders (Nanda et al., 2023), regular languages (Bhattamishra et al., 2020a), and sparse logical predicates (Edelman et al., 2022). Liu et al. (2023) propose a unified theory for expressivity of different automata with transformers. We refer the readers to Strobl et al. (2024) for a more comprehensive survey. However, the relation between the constructions and the performance of Transformers on real world datasets has been largely unclear. Our framework is the first to show that these constructions can be non-trivially extended to show capabilities of language models as general conversational agents. We hope that ELIZA inspires future works to connect existing constructions to the emergent abilities Transformers show at scale.

**Challenges for mechanistic interpretability** One direction for future work is to consider our ELIZA construction as a test bed for automatic interpretability methods—for example, compiling the construction into Transformer weights using Tracr (Lindner et al., 2023). Specifically, given a compiled Transformer corresponding to an ELIZA chatbot, to what extent could we recover the program using existing interpretability techniques, such as circuit finding (Conmy et al., 2023; Syed et al., 2023) and dictionary learning (Cunningham et al., 2023; Gurnee et al., 2024)? Possible difficulties include sharing of attention heads across different ELIZA operations like parsing and copying, and sharing of mechanisms for different ELIZA operations like cycling and memory queues. As such, our framework might encourage more sophisticated interpretable techniques in the future.

**Mechanistic dependence on data** Recent works have tried to understand the behavior of attention models when trained with synthetic datasets. Nanda et al. (2023) study feature formation in 1-layer transformer models on adders dataset, with Zhong et al. (2023) studying the dependence on model hyperparameters and initialization. Akyürek et al. (2024); Quirke et al. (2023) study formation of $n$-gram induction heads in language models. Allen-Zhu & Li (2023a); Zhao et al. (2023) study the behavior of language models when trained with different context-free grammars. Allen-Zhu & Li (2023b; 2024) further study knowledge manipulation and storage in language models trained on synthetic datasets. Zhang et al. (2022) propose LEGO synthetic reasoning dataset to understand generalization of transformers with simple boolean circuits. Finally, Zhang et al. (2023); Edelman et al. (2024); Nichani et al. (2024) give end-to-end convergence analysis of self-attention models when trained under simplistic data assumptions. However, such studies have been generally restricted to settings where the number of possible mechanisms and/or the number of features to learn are restricted. ELIZA provides a general framework that allows diverse mechanisms and features. To successfully implement ELIZA, a model has to perform local pattern matching, cycling through reassembly rules, and memory queues well. And for each feature, there are multiple mechansisms that can emerge, with each mechanism having different generalization abilities. As we show in section 4.3, different data distribution properties can lead to different mechanisms. With increasing interests to formalize relation between data and training behavior Chan et al. (2022); Hahn & Goyal (2023); Reddy (2024); Xie et al. (2021), we believe ELIZA can be a useful test bed for future studies.




(a) Response cycling.

(b) Memory queue.

Figure 6: We design counter-factual experiments to test whether models make use of intermediate generations to keep track of the response cycle (Fig. 6a) or memory queue (Fig. 6b), or rely only on the user inputs. Error bars show 95% confidence interval over models trained with three random seeds. Both experiments indicate that the models use their own outputs from earlier in the sequence. When we edit the model's earlier output, we can reliably influence it to increment the response cycle or read a memory from earlier in the queue.

## 6 Conclusion

In this work, we constructed a Transformer that implements the classic ELIZA chatbot algorithm. We then trained Transformers on ELIZA conversation transcripts and examined which aspects of the task were empirically more difficult to learn, and to what extent to the models matched our construction. Our constructions and dataset raise a number of possibilities for future research, including as a benchmark for automated interpretability methods, and as a setting for mechanistic analysis of learning dynamics.

**Limitations** Our constructions illustrate one way that Transformers can implement ELIZA, but they might not correspond to the solutions that Transformers actually learn. Characterizing the mechanisms that models learn empirically is a key challenge for future work on in interpretability. Second, we conduct some analysis of the mechanisms that models learn, but we do not conduct an exhaustive mechanistic analysis; future work could conduct further analysis using other interpretability techniques, such as causal methods (e.g. Vig et al., 2020; Feder et al., 2021; Geiger et al., 2021). Finally, while ELIZA offers a setting for investigating a number of aspects of conversations, real-world chatbots exhibit a number of behaviors that fall outside of the ELIZA framework. For example, ELIZA is a deterministic program, whereas most real-world chatbots are trained on data with more stochasticity.

**Acknowledgments** We thank Adithya Bhaskar, Alexander Wettig, Howard Yen, and the members of the Princeton NLP group for helpful comments and discussion. This research is funded by the National Science Foundation (IIS-2211779) and a Sloan Research Fellowship.

## References

Akyürek, E., Wang, B., Kim, Y., and Andreas, J. In-context language learning: Architectures and algorithms. *arXiv preprint arXiv:2401.12973*, 2024.

Allen-Zhu, Z. and Li, Y. Physics of language models: Part 1, Context-free grammar. *arXiv preprint arXiv:2305.13673*, 2023a.

Allen-Zhu, Z. and Li, Y. Physics of language models: Part 3.2, Knowledge manipulation. *arXiv preprint arXiv:2309.14402*, 2023b.

Allen-Zhu, Z. and Li, Y. Physics of language models: Part 3.3, Knowledge capacity scaling laws. *arXiv preprint arXiv:2404.05405*, 2024.

Angluin, D., Chiang, D., and Yang, A. Masked hard-attention Transformers and Boolean RASP recognize exactly the star-free languages. *arXiv preprint arXiv:2310.13897*, 2023.

Anil, C., Wu, Y., Andreassen, A., Lewkowycz, A., Misra, V., Ramasesh, V., Slone, A., Gur-Ari, G., Dyer, E., and Neyshabur, B. Exploring length generalization in large language models. *Advances in Neural Information Processing Systems (NeurIPS)*, 35:38546–38556, 2022.

Beltagy, I., Peters, M. E., and Cohan, A. Longformer: The long-document transformer. *arXiv:2004.05150*, 2020.

Bhattamishra, S., Ahuja, K., and Goyal, N. On the ability and limitations of transformers to recognize formal languages. In *Empirical Methods in Natural Language Processing (EMNLP)*, pp. 7096–7116, 2020a.

Bhattamishra, S., Patel, A., and Goyal, N. On the computational power of Transformers and its implications in

sequence modeling. In *Computational Natural Language Learning (CoNLL)*, pp. 455–475, 2020b.

Chan, S., Santoro, A., Lampinen, A., Wang, J., Singh, A., Richemond, P., McClelland, J., and Hill, F. Data distributional properties drive emergent in-context learning in transformers. *Advances in Neural Information Processing Systems (NeurIPS)*, 35:18878–18891, 2022.

Conmy, A., Mavor-Parker, A. N., Lynch, A., Heimersheim, S., and Garriga-Alonso, A. Towards automated circuit discovery for mechanistic interpretability. *arXiv preprint arXiv:2304.14997*, 2023.

Cunningham, H., Ewart, A., Riggs, L., Huben, R., and Sharkey, L. Sparse autoencoders find highly interpretable features in language models. *arXiv preprint arXiv:2309.08600*, 2023.

Edelman, B. L., Goel, S., Kakade, S., and Zhang, C. Inductive biases and variable creation in self-attention mechanisms. In *International Conference on Machine Learning (ICML)*, pp. 5793–5831. PMLR, 2022.

Edelman, B. L., Edelman, E., Goel, S., Malach, E., and Tsilivis, N. The evolution of statistical induction heads: In-context learning Markov chains. *arXiv preprint arXiv:2402.11004*, 2024.

Feder, A., Oved, N., Shalit, U., and Reichart, R. CausaLM: Causal model explanation through counterfactual language models. *Computational Linguistics*, 47(2):333–386, 2021.

Geiger, A., Lu, H., Icard, T., and Potts, C. Causal abstractions of neural networks. *Advances in Neural Information Processing Systems (NeurIPS)*, 34:9574–9586, 2021.

Gurnee, W., Horsley, T., Guo, Z. C., Kheirkhah, T. R., Sun, Q., Hathaway, W., Nanda, N., and Bertsimas, D. Universal neurons in GPT2 language models. *arXiv preprint arXiv:2401.12181*, 2024.

Hahn, M. Theoretical limitations of self-attention in neural sequence models. *Transactions of the Association of Computational Linguistics (TACL)*, 8:156–171, 2020.

Hahn, M. and Goyal, N. A theory of emergent in-context learning as implicit structure induction. *arXiv preprint arXiv:2303.07971*, 2023.

Hao, Y., Angluin, D., and Frank, R. Formal language recognition by hard attention transformers: Perspectives from circuit complexity. *Transactions of the Association of Computational Linguistics (TACL)*, 10:800–810, 2022.

Harris, C. R., Millman, K. J., Van Der Walt, S. J., Gommers, R., Virtanen, P., Cournapeau, D., Wieser, E., Taylor, J., Berg, S., Smith, N. J., et al. Array programming with NumPy. *Nature*, 585(7825):357–362, 2020.

Haviv, A., Ram, O., Press, O., Izsak, P., and Levy, O. Transformer language models without positional encodings still learn positional information. In *Findings of Empirical Methods in Natural Language Processing (EMNLP)*, pp. 1382–1390, 2022.

Hay, A. and Millican, P. ELIZA is Turing complete. https://sites.google.com/view/elizagen-org/blog/eliza-is-turing-complete, 2022. Accessed: 2024-01-09.

Jurafsky, D. and Martin, J. H. Chatbots and dialogue systems. *Speech and Language Processing*, 2020.

Kazemnejad, A., Padhi, I., Natesan, K., Das, P., and Reddy, S. The impact of positional encoding on length generalization in Transformers. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.

Kingma, D. P. and Ba, J. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

Lindner, D., Kramár, J., Farquhar, S., Rahtz, M., McGrath, T., and Mikulik, V. Tracr: Compiled transformers as a laboratory for interpretability. *Advances in Neural Information Processing Systems (NeurIPS)*, 36, 2023.

Liu, B., Ash, J. T., Goel, S., Krishnamurthy, A., and Zhang, C. Transformers learn shortcuts to automata. In *International Conference on Learning Representations (ICLR)*, 2023.

Merrill, W. and Sabharwal, A. A logic for expressing log-precision transformers. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.

Merrill, W. and Sabharwal, A. The expressive power of Transformers with chain of thought. In *International Conference on Learning Representations (ICLR)*, 2024.

Merrill, W., Sabharwal, A., and Smith, N. A. Saturated Transformers are constant-depth threshold circuits. *Transactions of the Association of Computational Linguistics (TACL)*, 10:843–856, 2022.

Nanda, N., Chan, L., Lieberum, T., Smith, J., and Steinhardt, J. Progress measures for grokking via mechanistic interpretability. In *International Conference on Learning Representations (ICLR)*, 2023.

Nichani, E., Damian, A., and Lee, J. D. How transformers learn causal structure with gradient descent. *arXiv preprint arXiv:2402.14735*, 2024.

Nye, M., Andreassen, A. J., Gur-Ari, G., Michalewski, H., Austin, J., Bieber, D., Dohan, D., Lewkowycz, A., Bosma, M., Luan, D., et al. Show your work: Scratchpads for intermediate computation with language models. *arXiv preprint arXiv:2112.00114*, 2021.

Olsson, C., Elhage, N., Nanda, N., Joseph, N., DasSarma, N., Henighan, T., Mann, B., Askell, A., Bai, Y., Chen, A., et al. In-context learning and induction heads. *arXiv preprint arXiv:2209.11895*, 2022.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., et al. PyTorch: An imperative style, high-performance deep learning library. *Advances in Neural Information Processing Systems (NeurIPS)*, 32, 2019.

Pérez, J., Barceló, P., and Marinkovic, J. Attention is Turing-complete. *The Journal of Machine Learning Research (JMLR)*, 22(75):1–35, 2021.

Pérez, J., Marinković, J., and Barceló, P. On the Turing completeness of modern neural network architectures. In *International Conference on Learning Representations (ICLR)*, 2019.

Quirke, L., Heindrich, L., Gurnee, W., and Nanda, N. Training dynamics of contextual n-grams in language models. *arXiv preprint arXiv:2311.00863*, 2023.

Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., Sutskever, I., et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

Reddy, G. The mechanistic basis of data dependence and abrupt learning in an in-context classification task. In *International Conference on Learning Representations (ICLR)*, 2024.

Singh, A. K., Moskovitz, T., Hill, F., Chan, S. C., and Saxe, A. M. What needs to go right for an induction head? A mechanistic study of in-context learning circuits and their formation. *arXiv preprint arXiv:2404.07129*, 2024.

Strobl, L., Merrill, W., Weiss, G., Chiang, D., and Angluin, D. What formal languages can Transformers express? A survey. *Transactions of the Association of Computational Linguistics (TACL)*, 12:543–561, 2024.

Syed, A., Rager, C., and Conmy, A. Attribution patching outperforms automated circuit discovery. *arXiv preprint arXiv:2310.10348*, 2023.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. Attention is all you need. *Advances in Neural Information Processing Systems (NeurIPS)*, 30, 2017.

Vig, J., Gehrmann, S., Belinkov, Y., Qian, S., Nevo, D., Sakenis, S., Huang, J., Singer, Y., and Shieber, S. Causal mediation analysis for interpreting neural NLP: The case of gender bias. *arXiv preprint arXiv:2004.12265*, 2020.

Waskom, M. L. Seaborn: Statistical data visualization. *Journal of Open Source Software*, 6(60):3021, 2021.

Wei, C., Chen, Y., and Ma, T. Statistically meaningful approximation: A case study on approximating Turing Machines with Transformers. *Advances in Neural Information Processing Systems (NeurIPS)*, 35:12071–12083, 2022a.

Wei, J., Wang, X., Schuurmans, D., Bosma, M., Xia, F., Chi, E., Le, Q. V., Zhou, D., et al. Chain-of-Thought prompting elicits reasoning in large language models. *Advances in Neural Information Processing Systems (NeurIPS)*, 35: 24824–24837, 2022b.

Weiss, G., Goldberg, Y., and Yahav, E. Thinking like Transformers. In *International Conference on Machine Learning (ICML)*, pp. 11080–11090. PMLR, 2021.

Weizenbaum, J. ELIZA—A computer program for the study of natural language communication between man and machine. *Communications of the ACM*, 9(1):36–45, 1966.

Weizenbaum, J. *Computer power and human reason: From judgment to calculation.* WH Freeman & Co, 1976.

Wolf, T., Debut, L., Sanh, V., Chaumond, J., Delangue, C., Moi, A., Cistac, P., Rault, T., Louf, R., Funtowicz, M., et al. Transformers: State-of-the-art natural language processing. In *Empirical Methods in Natural Language Processing (EMNLP): System Demonstrations*, pp. 38–45, 2020.

Xie, S. M., Raghunathan, A., Liang, P., and Ma, T. An explanation of in-context learning as implicit Bayesian inference. *arXiv preprint arXiv:2111.02080*, 2021.

Yao, S., Peng, B., Papadimitriou, C., and Narasimhan, K. Self-attention networks can process bounded hierarchical languages. In *Association for Computational Linguistics and International Joint Conference on Natural Language Processing (ACL-IJCNLP)*, pp. 3770–3785, 2021.

Zhang, R., Frei, S., and Bartlett, P. L. Trained transformers learn linear models in-context. *arXiv preprint arXiv:2306.09927*, 2023.

Zhang, Y., Backurs, A., Bubeck, S., Eldan, R., Gunasekar, S., and Wagner, T. Unveiling Transformers with LEGO: A synthetic reasoning task. *arXiv preprint arXiv:2206.04301*, 2022.

Zhao, H., Panigrahi, A., Ge, R., and Arora, S. Do transformers parse while predicting the masked word? In Bouamor, H., Pino, J., and Bali, K. (eds.), *Empirical Methods in Natural Language Processing (EMNLP)*, pp. 16513–16542, 2023.

Zhong, Z., Liu, Z., Tegmark, M., and Andreas, J. The clock and the pizza: Two stories in mechanistic explanation of neural networks. *Advances in Neural Information Processing Systems (NeurIPS)*, 36, 2023.

Zhou, H., Bradley, A., Littwin, E., Razin, N., Saremi, O., Susskind, J., Bengio, S., and Nakkiran, P. What algorithms can Transformers learn? A study in length generalization. *arXiv preprint arXiv:2310.16028*, 2023.

| |
|---|
| Men are all alike. |
| IN WHAT WAY |
| They're always bugging us about something or other. |
| CAN YOU THINK OF A SPECIFIC EXAMPLE |
| Well, my boyfriend made me come here. |
| YOUR BOYFRIEND MADE YOU COME HERE |
| He says I'm depressed much of the time. |
| I AM SORRY TO HEAR YOU ARE DEPRESSED |
| It's true. I am unhappy. |
| DO YOU THINK COMING HERE WILL HELP YOU NOT TO BE UNHAPPY |
| I need some help, that much seems certain. |
| WHAT WOULD IT MEAN TO YOU IF YOU GOT SOME HELP |
| Perhaps I could learn to get along with my mother. |
| TELL ME MORE ABOUT YOUR FAMILY |
| My mother takes care of me. |
| WHO ELSE IN YOUR FAMILY TAKES CARE OF YOU |
| My father. |
| YOUR FATHER |
| You are like my father in some ways. |
| WHAT RESEMBLANCE DO YOU SEE |
| You are not very aggressive, but I think you don't want me to notice that. |
| WHAT MAKES YOU THINK I AM NOT VERY AGGRESSIVE |
| You don't argue with me. |
| WHY DO YOU THINK I DON'T ARGUE WITH YOU |
| You are afraid of me. |
| DOES IT PLEASE YOU TO BELIEVE I AM AFRAID OF YOU |
| My father is afraid of everybody. |
| WHAT ELSE COMES TO MIND WHEN YOU THINK OF YOUR FATHER |
| Bullies. |
| DOES THAT HAVE ANYTHING TO DO WITH THE FACT THAT YOUR BOYFRIEND MADE YOU COME HERE |

Table 1: An example ELIZA conversation, reproduced from Weizenbaum (1966). Lines with all capital letters are generated by ELIZA.

## A ELIZA Algorithm Details

Here we provide some additional details about the ELIZA algorithm. Our presentation of the ELIZA algorithm in Section 2 omits some details of the original ELIZA algorithm, to improve clarity, so we describe these details here.[3]

**Word-level translation** An ELIZA script can include word-level translation rules—for example, `I = YOU`, `YOU = I`, and `ME = YOU`. These translations are applied to all of the words in the input before trying to match the input to a pattern. Therefore, in the original ELIZA script, the patterns are written to match inputs after the word-level translations have been applied. So, for example, the rule

`(0 ARE I 0)` $\rightarrow$ Would you prefer if I weren't 4?

would match the input "Are you laughing at me?" and transform it to "Would you prefer if I weren't laughing at you?" In this paper, we write rules to match the input prior to word-level translations—so, for example, we would present the pattern above as `(0 ARE YOU 0)`. Word-level translation is trivial to incorporate into the Transformer construction, by using the final linear layer to map each word to its translation.

**Keywords** Each entry in an ELIZA script consists of a ranked keyword. Each keyword is associated with a list of decomposition templates, and each decomposition template is associated with one or more transformation rules. See Figure 7 for an example. To select a decomposition template, ELIZA finds the highest ranked keyword that appears in the input, and then finds the first decomposition template in the associated list that matches the input. If none of the templates matched, ELIZA checks the next highest-ranked keyword. In this paper, we ignore the role of keywords, and instead define an ELIZA script by a set of ranked decomposition templates and associated transformation rules.

[3]For an annotated explanation of an ELIZA script, see https://github.com/jeffshrager/elizagen.org/blob/master/1965_Weizenbaum_MAD-SLIP/1966_01_CACM_article_Eliza_script.txt.

```
(REMEMBER 5
    ((0 I REMEMBER 0)
         (DO YOU OFTEN THINK OF 4)
         (DOES THINKING OF 4 BRING ANYTHING ELSE TO MIND)
         (WHAT ELSE DO YOU REMEMBER)
         (WHY DO YOU REMEMBER 4 JUST NOW)
         (WHAT IN THE PRESENT SITUATION REMINDS YOU OF 4)
         (WHAT IS THE CONNECTION BETWEEN ME AND 4))
    ((0 DO YOU REMEMBER 0)
         (DID YOU THINK I WOULD FORGET 5)
         (WHY DO YOU THINK I SHOULD RECALL 5 NOW)
         (WHAT ABOUT 5)
         (YOU MENTIONED 5)))

(IF 3
    ((0 IF 0)
        (DO YOU THINK ITS LIKELY THAT 3)
        (DO YOU WISH THAT 3)
        (WHAT DO YOU THINK ABOUT 3)
        (REALLY, 2 3)))
```

Figure 7: Part of an ELIZA script, from Weizenbaum (1966). Each entry in the script consists of a ranked keyword and a list of patterns, with each pattern associated with multiple transformation rules.

**Pre-transformation rules** The pre-transformation rule is a special rule that applies a transformation to the input, and then "passes control" to another keyword in the script. There is one use of the pre-transformation rule in Weizenbaum's ELIZA script: if the input matches the pattern `(0 I'm 0)`, it is reassembled as "I am 3," and then matched against templates with the keyword "am," such as `(0 I am 0)`. However, the pre-transformation rule is critical to the construction of Hay & Millican (2022) for embedding a Turing machine in an ELIZA script, which we will discuss in more detail below (App B.4). In this construction, the input at each step represents the tape of the Turing machine, and keywords in the script correspond to states. Each pre-transformation rule transforms the input by applying one update to the tape, and then passes control to a new keyword corresponding to the next state.

## B Construction Details

In this section, we provide additional details about our ELIZA constructions, including sample implementations in RASP (Weiss et al., 2021). The input to a RASP program is a sequence of `tokens`. The program then consists of a series of operations that output new sequences of equal length to `tokens`, corresponding to intermediate embeddings in the Transformer. The `select` and `aggregate` operations correspond to the attention mechanism in the Transformer; these are the only operations that can combine information from different positions in the sequence. All other operations must operate independently at each position, corresponding to feedforward layers. Like Weiss et al. (2021), we allow feedforward layers to implement arbitrary element-wise transformations. We do not provide explicit constructions for these element-wise transformations; we leave this for future work. Figure 8 shows the RASP (Weiss et al., 2021) attention primitives we use in our construction, implemented in NumPy (Harris et al., 2020).

### B.1 Input Segmentation and Position Encoding

Our first step is to divide the input into segments, corresponding to the turns in the conversation. This is accomplished by using the special delimiter tokens to count the number of utterances seen so far:

```
segment_ids = selector_width(select(tokens, tokens, lambda q, k: k in ("u:", "e:"), max_width=max_segments)
```

We will use these `segment_ids` throughout the construction to restrict attention to a particular utterance. The `segment_ids` are also used to generate local positional encodings:

```
segment_positions = selector_width(select(segment_ids, segment_ids, ==), max_width=max_segment_length)
```

This value encodes the position relative to the start of the current segment.

```python
def select(keys, queries, predicate):
    # Calculate a (binary) attention pattern.
    selector = np.array([[predicate(q, k) for k in keys] for q in queries])
    return np.tril(selector)

def selector_width(selector, max_width=None):
    # Count the number of keys attended by each query, up to `max_width`.
    width = selector.sum(-1)
    if max_width:
        return np.minimum(width, max_width)
    return width

def aggregate(selector, values, one_hot=False):
    # Aggregate either a single value vector or a batch of value vectors
    # stored in a dictionary.
    if type(values) == dict:
        return {k: aggregate(selector, v, one_hot) for k, v in values.items()}
    if one_hot:
        return values[selector.argmax(-1)]
    attn = selector / np.maximum(selector.sum(-1, keepdims=True), 1e-9)
    return values @ attn.T
```

Figure 8: Code for the primitive RASP operations (Weiss et al., 2021) we use in our construction, using NumPy (Harris et al., 2020). Each attention head can implement one pair of `select` and `aggregate` operations. The `selector_width` function corresponds to an attention head followed by a feed-forward layer, which maps the scalar attention output to an embedding that can be used in subsequent attention layers. Because `selector_width` maps each possible width to a unique, orthogonal embedding, the program must specify in advance the maximum width it will handle.

**Remark on length generalization** While not the focus of our investigation here, our approach to segment and position encodings has implications for length generalization, similar to the cases studied by Zhou et al. (2023). In particular, we must specify in advance the maximum number of segments per conversation, as well as the length of each segment. This is because the `selector_width` operator is implemented using one attention layer followed by one feed-forward layer. At each position $i$, the attention layer outputs $1/c$, where $c$ is the number of key positions attended to from position $i$. The feed-forward layer then maps each value of $1/c$ to an orthogonal embedding. In our construction, we implement this second step as a look-up table, meaning that we must decide in advance on the maximum possible value of $c$. This means that our construction sets a limit on the number of segments per conversation, as well as the length within each segment. If a model learned this mechanism, we would expect it to fail to generalize if the number of segments or the length of a segment increases beyond the training set. (On the other hand, the construction does not place a direct limit on the total conversation length.)

### B.2 Template Matching

The next step in the construction is to compare the most recent input to the inventory of decomposition templates. Template matching involves two things: finding a template that matches the input, and decomposing the input according to that template's decomposition groups. Our construction makes use of the fact that ELIZA templates are equivalent to star-free regular expressions. As a result, we can recognize these by simulating the corresponding finite-state automaton, building on the constructions of Liu et al. (2023) and Angluin et al. (2023), adapted to recognize multiple templates in parallel.

**Decomposition templates** Given a vocabulary $\mathcal{V}$, a decomposition template is a sequence $t = t_1, \ldots, t_L$, where each $t_i$ is either a word from $\mathcal{V}$; the wildcard character `0`, which matches a sequence of zero or more words from $\mathcal{V}$; or a positive integer $n$, which matches a sequence of exactly $n$ words from $\mathcal{V}$.[4] We assume that the vocabulary contains two special beginning- and end-of-sequence delimiters, `^` and `$` respectively, and for every input $w_1, \ldots, w_N$ and template $t_1, \ldots, t_L$, $w_1 = t_1 =$ `^` and $w_N = t_L =$ `$`. We will use $t_{:i}$ to denote the template prefix $t_1, \ldots, t_i$. As a working example, consider the vocabulary $\mathcal{V} = \{\texttt{a}, \texttt{b}\}$ and the template $t =$ `^a0bb0$`. This template matches the input `^aaabbaa$` and decomposes it into five groups: (1) `a` (2) `aa` (3) `b` (4) `b` (5) `aa`. Note that each decomposition group corresponds to a prefix of the template: word $w_i$ is in group $\ell$ if $w_{:i}$ matches the template prefix $t_{:\ell}$.

[4] A template can also include an equivalence class $W \subset \mathcal{V}$, which matches one instance of any word in $W$. For example, the template `1(a|b)1` matches both `cab` and `cbb`. This can be addressed at the embedding layer by assigning one dimension to the value of the indicator $\mathbf{1}\{w \in \mathcal{W}\}$ for each word $w$.

```
def match_templates(tokens, segment_ids, segment_positions, templates):
  L = max(len(t) for t in templates)
  prefixes = [{("u:",): tokens == "u:"}]

  # Each layer l checks if the input matches t[:l+1]
  for l in range(1, L):
    just_matched = select_prev(prefixes[-1], segment_ids, segment_positions)
    ever_matched = frac_prev(prefixes[-1], segment_ids, segment_positions)
    new_matches = {}
    for t in templates:
      if len(t) <= l: continue
      if t[l] == "0":
        new_matches[t[:l+1]] = ever_matched[t[:l]] > 0
      elif t[l-1] == "0" and t[l] == "1":
        new_matches[t[:l+1]] = prefixes[-1][t[:l]]
      elif t[l-1] == "0":
        new_matches[t[:l+1]] = prefixes[-1][t[:l]] & (tokens == t[l])
      elif t[l] == "1":
        new_matches[t[:l+1]] = just_matched[t[:l]]
      else:
        new_matches[t[:l+1]] = just_matched[t[:l]] & (tokens == t[l])
    prefixes.append(new_matches)

  # For each template, identify the longest matching prefix at each position.
  states = {}
  for t in templates:
    s = np.stack([m[t[:l+1]] for l, m in zip(range(len(t)), prefixes)])
    ind = np.arange(s.shape[0])
    states[t] = (ind[:, None] * s).max(0)

  return states
```

Figure 9: Code for matching an input sequence `tokens` to a set of decomposition templates.

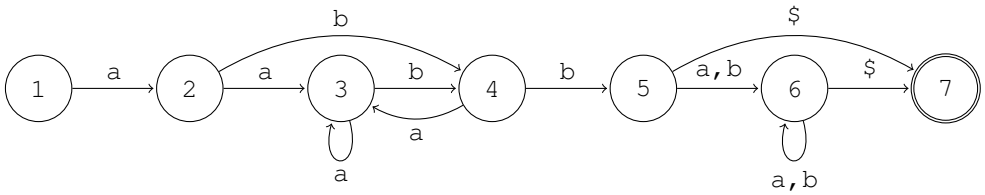
**Matching templates** Our construction uses $L$ Transformer layers, where $L$ is the maximum number of states in any template. At each layer $\ell$, we calculate whether the input matches the template prefix $t_{:\ell}$ for each template $t$ and at each position $i$. If $t_\ell$ is the wildcard character `0`, then $w_{:i}$ matches $t_{:\ell}$ if $t_{:\ell-1}$ has been matched at any position $j < i$. If $t_\ell$ is a vocabulary item $w$, then $w_{:i}$ matches $t_\ell$ if $w_i = w$ and $w_{:i-1}$ matches $t_{:\ell-1}$ (or, if $t_{\ell-1}$ is `0`, if $w_{:i}$ matches $t_{:\ell-1}$, to account for the possibility that `0` matches zero words). We check these conditions using two attention heads per layer:

```
def frac_prev(values, segment_ids, segment_pos):
  return aggregate(
    (select(segment_ids, segment_ids, eq) &
     select(segment_pos, segment_pos, not_eq)),
    values)
```

```
def select_prev(values, segment_ids, segment_pos):
  return aggregate(
    (select(segment_ids, segment_ids, eq) &
     select(segment_pos, segment_pos, is_prev)),
    values)
```

These attention heads restrict attention to the most recent utterance by taking the logical AND between two selectors; see Lindner et al., 2023, Appendix F for a discussion of mechanisms for combining selectors. Note that each layer uses two attention heads, with each attention head calculating `frac_prev` or `select_prev` for all templates in parallel.

**Templates as finite-state automata** While our construction is presented in terms of ELIZA templates, we note that the ELIZA template language defines a subset of star-free regular languages. As a result, we can formulate this construction as an approach to simulating a finite-state automaton, building on the constructions of Liu et al. (2023) and Angluin et al. (2023). In particular, consider again our example template $t =$ `^a0bb0$`. We can recognize this template by simulating the following finite-state automaton:




Each state in the automaton corresponds to a prefix of the template: if the automaton is in state $\ell$ after processing words $w_1, \ldots, w_i$, then the sequence $w_{:i}$ matches the template prefix $t_{:\ell}$. Given a template $t_1, \ldots, t_L$, we will therefore refer to the

states of the corresponding automaton using the template prefixes $t_{:1}, \ldots, t_{:L}$. Note that some special handling is required because the automaton states are assigned from left to right with no ability to look ahead in the input. For example, consider the template `0ab` and input `bacaab`, which should be decomposed as (1) `baca` (2) `a` (3) `b`. Without looking ahead in the input, we have no way of knowing that the first two `a` tokens belong in group 1 rather than 2. Our template matching procedure would assign this sequence the states `121223`. A similar issue arises if we have a template such as `01ab`, which should decompose input `bacaab` as (1) `bac` (2) `a` (3) `a` (4) `b`. These issues can be addressed by taking some additional care in the generation stage, discussed in more detail below (App. B.3).

**Comparison to existing constructions** Our construction differs in some ways from prior work for simulating finite state automata with Transformers. In particular, the construction of Angluin et al. (2023) uses hard (one-hot) attention to recognize star-free regular expressions. Our construction uses a `frac_prev` attention head, which attends uniformly to all positions in the sequence; this allows us to match multiple templates using one attention head. While the number of attention heads is constant with respect to the number of templates, the embedding dimension increases linearly with the number of templates, in order to encode the automaton state for each template in parallel.

**Reducing the number of layers** For ease of presentation, we described a template matching construction that uses one Transformer layer for each symbol in the template. Here, we describe two modifications that reduce the number of layers to the total number of wildcard symbols in the template.

*Combining wildcards:* First, we can use one layer to match both a wildcard symbol and the symbol that immediately follows. For example, consider the template `a0b0` and input `accbabc`, which we aim to decompose as (1) `a` (2) `cc` (3) `b` (4) `abc`. The computations are as follows:

| Input | a | c | c | b | a | b | c |
|---|---|---|---|---|---|---|---|
| Attention 1 | a | a0 | a0 | a0 | a0 | a0 | a0 |
| MLP 1 | - | - | - | a0b | - | a0b | - |
| Attention 2 | - | - | - | - | a0b0 | a0b0 | a0b0 |
| Output | 1 | 2 | 2 | 3 | 4 | 4 | 4 |

Here, each entry in the table illustrates a value calculated at that layer, corresponding to a template prefix that has been matched at that point. For example, the first-layer MLP identifies that the prefix `a0b` has been matched at two positions. We distinguish between the first and second matches of this prefix by assigning each position to the longest prefix that matches at that point.

*Handling $n$-gram literals:* The second modification pertains to $n$-gram literals in the template. For example, consider the template `a0bc0`. As presented above, our construction uses one layer to match the prefix `a0b` and another to match the prefix `a0bc`. Instead, we can combine these operations into a single layer by using two attention heads. At position $i$, one attention head checks whether the previous word $w_{i-1}$ is `b`. The second attention head checks whether the prefix `a0` has been matched anywhere to the left of $w_{i-1}$, attending to all tokens at positions less than $i-1$. We can use this approach for any $n$-gram up to some maximum $n$, defined by the number of attention heads per layer.

### B.3 Generating a Transformation

Now we assume that we have identified a matching template and that the embedding for each input token identifies the decomposition group to which that token belongs. The next step is now to apply the chosen *reassembly rule* to the input to generate a response.

**Reassembly rules** Given a template $t_1, \ldots, t_L$ and vocabulary $\mathcal{V}$, a reassembly rule is a sequence $r = r_1, \ldots, r_M$, where each $r_i$ is either a word $w \in \mathcal{V}$ or an integer $n \in [M]$ such that $t_n \in \{0, 1\}$. Given an input $w_1, \ldots, w_N$, let $s_1, \ldots, s_N \in [L]$ denote the lengths of the longest matching template prefix at each position—that is, $t_{:s_i}$ is the longest prefix matching $w_{:i}$. We refer to each $s_i$ as a *decomposition group*. For each $r_i$, if $r_i \in \mathcal{V}$, the model outputs $r_i$. If $r_i \in [L]$, the model outputs the subsequence of $w$ such that, for each $w_j$, $s_j = r_i$. For example, consider the template $t =$ `a0bb0` and example input `aaabbab`, with automaton states `1223455`. The reassembly rule $r =$ `c2d5` would generate the response `caadab`. We can divide this process into two stages. First, at each step, we need to determine the reassembly state—that is, which symbol of the reassembly rule are we currently processing. In Fig. 10, we illustrate how we can determine the state as

```python
def get_reassembly_action(group_count, template, rule, step):
  # For each template t, group_count[t][l]  is the number of input tokens with group t[l]
  counts = group_count[template]

  # The position in the input sequence at the start of each group
  group_start_positions = np.concatenate([np.array([0]), np.cumsum(counts[:-1])])

  # The number of tokens in each part of the reassembly rule
  rule_part_sizes = np.array([counts[int(r)] if r.isnumeric() else 1 for r in rule])

  # The length the output will be after applying each part of the reassembly rule
  rule_part_end_positions = np.cumsum(rule_part_sizes)

  # Return to the user if we're done generating.
  if step == rule_part_sizes.sum():
    return "u:"

  # Which part of the rule are we in?
  i = np.argmax(rule_part_end_positions > step)
  r = rule[i]

  # Return the position of the token to copy:
  if r.isnumeric():
    num_already_copied = step - (rule_part_end_positions[i - 1] if i > 0 else 0)
    target_position = int(group_start_positions[int(r)] + num_already_copied + 1)
    return "copy", target_position

  # Return a constant token to output.
  return "print", r
```

Figure 10: Code for generating an output token at step `i` given a user input `x`, the corresponding sequence of automaton states, and a reassembly rule.

a function of the number of tokens that have been generated so far and the number of tokens in each decomposition group. Second, if the next token should be copied from the input, we need to identify the exact token in the input that should be copied. We present two mechanisms for copying, one using content-based attention and one using position-based attention.

**Option 1: Content-based attention (induction head)** The first possible approach uses content-based attention, akin to an $n$-gram level induction head (Olsson et al., 2022; Akyürek et al., 2024). First, at each input position $j$, the key embedding encodes the decomposition group to which the token belongs as well as the identity of the previous $n$ tokens, where $n$ is the maximum context window. Second, at each output position $i$, the query embedding encodes the decomposition group $s_i$ from which we should copy at this step, as well as the identity of the current token and any previous output tokens associated with this decomposition group. An attention head can then attend to the earliest input position $j$ such that $s_j = s_i$ and, for all $k$ from 0 to $n$, if $s_{i-k} = s_i$ then $w_{j-k-1} = w_{i-k}$. Note that we must specify a maximum context window, $n$, which is constrained by the embedding size. If $n$ is less than the length of a decomposition group, this mechanism can fail if the same $n$-gram appears more than once in the decomposition group, as noted by Zhou et al. (2023).

For example, consider the template $t =$ `a0b0` and reassembly rule $r =$ `h2`. For an input `acdecdfbg` that matches this template, the output under the reassembly rule is given by `hcdecdf`. If the model uses a 2-gram induction head, the behavior of the model for the same input is given in Tab. 2

| Input | a | c | d | e | c | d | f | b | g | E | h | c | d | e | c | d |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Previous 2-gram | 00 | 0a | ac | cd | de | ec | cd | df | fb | | | | | | | |
| Decomposition group | 1 | 2 | 2 | 2 | 2 | 2 | 2 | 3 | 4 | | | | | | | |
| Current 2-gram | | | | | | | | | | | 00 | 0c | cd | de | ec | cd |
| Output | | | | | | | | | | | c | d | e | c | d | e(×) |

Table 2: Behavior of a model that uses 2-gram induction head on input `acdecdfbg` to match to a template $t =$ `a0b0` and respond with reassembly rule $r =$ `h2`. The response by a 2-gram induction head is to copy the first token from input that matches the current 2-gram and the previous 2-gram (we break ties by the character that appears earlier in the input segment). For example, after the model sees $\cdots$ `Ehcde`, it outputs `c`. However, as we follow the same rule, the model mistakes at the second occurrence of `cd`.

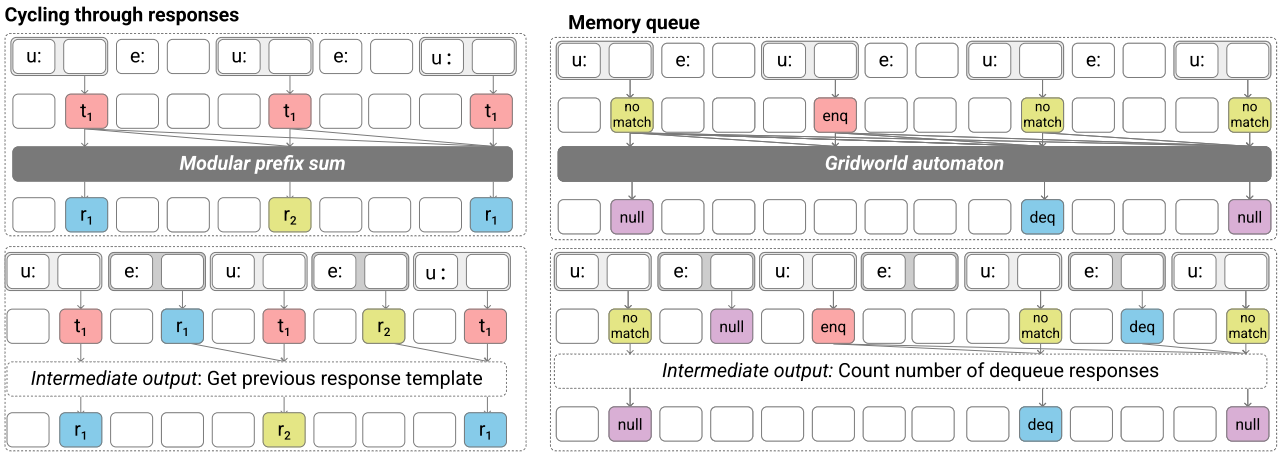



Figure 11: ELIZA includes two components that make use of the long-term conversation history: cycling through response templates (*left*), and the memory queue (*right*). We identify two mechanisms for these components. *Top:* First, after parsing the user's input, we can use existing automaton constructions (Liu et al., 2023) as black box components to simulate the relevant data structures. *Bottom:* Alternatively, we can re-use the template matching mechanism to also parse intermediate ELIZA outputs, resulting in simpler constructions with different generalization tradeoffs.

**Option 2: Position-based attention** Our second possible approach uses position-based attention and is described in Fig. 10. Specifically, we can use an attention head to count the number of tokens in each decomposition group, as well as the position in the input sequence at which that decomposition group begins. A feedforward layer can then calculate the position of the input token that should be copied at a given generation step. As discussed by Zhou et al. (2023), this form of position arithmetic might be more difficult for the model to learn. However, if this mechanism is learned correctly, we predict that it might generalize better than content-based attention in settings where the same $n$-gram appears multiple times in the sequence. The behavior of the model for an input is outlined in Tab. 2.

| Input | a | c | d | e | c | d | f | b | g | E | h | c | d | e | c | d |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Position | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | | | | | | | |
| Decomposition group | 1 | 2 | 2 | 2 | 2 | 2 | 2 | 3 | 4 | | | | | | | |
| Position to copy | | | | | | | | | | | 2 | 3 | 4 | 5 | 6 | 7 |
| Output | | | | | | | | | | | c | d | e | c | d | f |

Table 3: Behavior of a model uses position-based attention input `acdecdfbg` to match to a template $t =$ `a0b0` and respond with reassembly rule $r =$ `h2`. The position-based attention counts number of tokens in each copy group and an MLP to calculate target position based on current step and number of tokens per group. Finally, an attention is used to copy the token from the target position.

### B.4 Pre-transformation Rules and an ELIZA Transformer Turing Machine

In this section we discuss how to incorporate the special pre-transformation rule into our construction. This rule is used by Hay & Millican (2022) to prove that ELIZA is Turing-complete, which will allow us to immediately derive a Turing machine construction for the ELIZA Transformer.

**Pre-transformations with the ELIZA Transformer** As discussed in Appendix A, a pre-transformation rule consists of a decomposition template, a transformation rule, and a reference to another keyword in the script. If an input $w$ matches the template, ELIZA reassembles it according to the transformation rule to get a new input $w'$, and then reprocesses $w'$ according to the specified keyword. Pre-transformation rules can trigger an arbitrary number of computational steps (for example, we can write a script corresponding to a Turing machine that never halts). Therefore, given a Transformer with a finite number of layers, the only way to incorporate arbitrary pre-transformation rules into our construction is to enable the Transformer to perform variable computation depending on the input. The most natural way to do this is using a

Chain-of-Thought-style approach (Wei et al., 2022b): if the input matches a pre-transformation rule, the ELIZA Transformer will output the transformed input (along with some indicator of the new state), and then reprocess the newly generated output. This approach also follows from Merrill & Sabharwal (2024), who demonstrate that intermediate-decoding steps are necessary for simulating arbitrary Turing machines with decoder-only Transformers.

**ELIZA Transformer Turing Machine** Having incorporated pre-transformation rules into the ELIZA Transformer, we can now use the ELIZA construction from Hay & Millican (2022) to immediately get a new construction for simulating a Turing machine with an auto-regressive Transformer. In this construction, each action in the Turing machine is expressed as a pre-transformation rule, and the input at each timestep encodes the tape. Given a Turing machine (TM) that runs in $T(n)$ steps (where $n$ is the length of the input), this construction uses $T(n)^2$ generation steps: at each step, it finds the pattern that matches the most recent input, regenerates the tape according to the associated transformation rule, and then reprocesses the new version of the tape. This resembles existing constructions, but with some differences. For example, Wei et al. (2022a) give a construction that uses $T(n)$ generation steps: at each step, the model generates one new token, which encodes the state and action taken at that step. (On the other hand, Wei et al. (2022a) assumes the TM uses a single-directional tape, so will take $T(n)^2$ steps to simulate a TM with a bi-directional tape running in $T(n)$ steps.) Note that the ELIZA construction does not use either of the long-term memory mechanisms (response cyling or the memory queue). At each step, the model needs to attend only to the most recent version of the tape—which has a length of $T(n)$—rather than the full conversation history, which has a final length of $T(n)^2$. The construction could therefore use a sliding window attention scheme (e.g. Beltagy et al., 2020) to reduce the number of attention comparisons at each step.

# C Experimental Details

Here we provide more details about how we generate the data and conduct the experiments. Code and data for reproducing the experiments are available at `https://github.com/princeton-nlp/ELIZA-Transformer`.

## C.1 Data Generation

To generate an ELIZA dataset, we first generate a set of decomposition templates and reassembly rules, and then generate conversations by generating sentences that match the different decomposition templates and applying the corresponding rules. For all templates and sentences are drawn from a vocabulary $\mathcal{V}$ consisting of the 26 lower-case English letters. Each turn begins with a special delimiter character—`U` for user inputs and `E` for ELIZA inputs—and ends with a period, and each conversation begins with a special beginning-of-sequence token.

**Decomposition templates** Our distribution over decomposition templates is defined by the following parameters: the minimum and maximum number of wildcard symbols per template; and the maximum $n$-gram length, meaning the maximum number of contiguous non-wildcard symbols. For example, the template `0a0bc0` has two wildcards and a maximum $n$-gram length of two (`bc`). To generate a template, we first pick the number of wildcards by sampling a number $\ell$ uniformly from between the minimum and maximum, and then form a template by interleaving $\ell$ wildcard symbols with $\ell + 1$ $n$-grams. Each $n$-grams is sampled by first sampling a length $m$ uniformly from between 0 and the maximum length (for the first and last $n$-gram) or between 1 and the maximum length (for any $n$-gram between two wildcard symbols), and sampling $m$ words uniformly from $\mathcal{V}$. For our first set of experiments (Section 4.2), we sample 31 templates with between two and four wildcards and a maximum $n$-gram length of three. For our second set of experiments (comparing copying mechanisms in Section 4.3), we sample 15 templates, each with exactly two wildcard characters and a maximum $n$-gram length of 1. For all experiments, the final template is the null template. The only wildcard symbol we use is `0`, corresponding to zero or more words, although ELIZA templates can also include symbols that match exactly $n$ wildcard words.

**Reassembly rules** Given a decomposition templates, a reassembly rule consists of a sequence of words from $\mathcal{V}$ and integers indexing wildcards in the template. We refer to these wildcards as *copying segments*. Our distribution over reassembly rules is defined by the minimum and maximum number of copying segments and the maximum $n$-gram length. Given the set of integers corresponding to the available copying segments in the template, we generate a transformation rule by sampling up to $\ell$ of these numbers without replacement (where $\ell$ is sampled uniformly for each rule), and then form a rule by interleaving numbers with randomly sampled $n$-grams as above. We additionally prepend each reassembly rule with a unique, constant two-word prefix. For our first set of experiments (Section 4.2), we sample up to five reassembly rules per templates, each with between one and four copying segments. For our second set of experiments (comparing

copying mechanisms in Section 4.3), we sample one reassembly rule per template, each with exactly two copying segments characters.

**Single turn** To generate a single turn of a conversation, we sample a decomposition template and then sample a sentence that matches that template. For each wildcard in the template, we pick a segment length $m$ uniformly from between 0 and the maximum segment length, and then sample $m$ words from the vocabulary. For our first set of experiments, the maximum segment length is 10 and we sample the $m$ words uniformly for each segment. In our second set of experiments, the maximum segment length is 20, and, for each segment, we first sample a unigram distribution $\mathbf{p} \sim \text{Dirichlet}(\alpha\mathbf{1})$, and then sampling $m$ words from $\text{Categorical}(\mathbf{p})$, as described in Section 4.3).

**Conversations** For our experiments in Section 4.2, we generate conversations by sampling a sequence of turns until we reach the maximum input length (512 tokens). (For our experiments with copying mechanisms in Section 4.3, each conversation consists of a single turn.) We take some additional considerations to ensure that the data demonstrates the cycling behavior—that is, to ensure that each template occasionally appears enough times in a conversation to cycle through all of the associated reassembly rules. In particular, for each conversation, we sample a distribution over templates $\mathbf{p} \sim \text{Dirichlet}(\boldsymbol{\alpha})$, and then for each turn sample a template $t \sim \text{Categorical}(\mathbf{p})$. Here, $\boldsymbol{\alpha}$ is a 32-dimensional vector, corresponding to the 32 templates (including the null template); setting the entries of $\boldsymbol{\alpha}$ to be less than one makes it more likely that $\mathbf{p}$ assigns most probability to a small number of templates. We set the entries to be 1/32, with the exception of the memory template, which is set to 1/4 (to increase the proportion of examples that demonstrate the memory queue). Additionally, after sampling $\mathbf{p}$, we ensure that the likelihood assigned to the null template is at least half the likelihood assigned to the memory template; this is to increase the proportion of examples that contain both enqueue operations and dequeue operations (which are triggered by the null template). For our first set of experiments, we sample 100,000 conversations for training and 20,000 for testing. For our second set of experiments, we sample 32,000 and 16,000 conversations for training and evaluation, respectively.

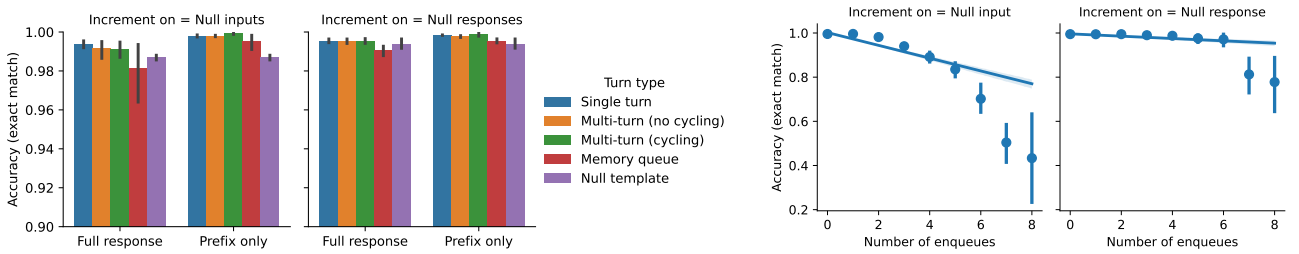
**Memory queue** To incorporate the memory queue mechanism, we select one of the 32 templates to serve as the memory template. This template is associated with two lists of reassembly rules: the first list is used to respond to inputs that match the template (enqueue reassembly rules), and the second list is used later in the conversation when the memory is read from the queue (dequeue reassembly rules). In Weizenbaum's ELIZA program (Weizenbaum, 1966), for each memory, a dequeue reassembly rule is selected at random from the list. In our experiments, we instead use the cycling mechanism, to ensure that the behavior is deterministic. That is, given dequeue reassembly rules $r_1, \ldots, r_M$, at the $n^{th}$ dequeue in the conversation we use the reassembly rule $r_{n\%M}$. In our dataset, there are four dequeue reassembly rules. We also limit the size of the queue: when sampling conversations, we ensure that the queue contains at most four memories at any time.

### C.2 Models and Training

For all of our experiments, we train 8-layer decoder-only Transformers with 12 attention heads per layer, a hidden dimension of 768. The models have no position embeddings but are otherwise based on the GPT-2 architecture (Radford et al., 2019) and are implemented using PyTorch (Paszke et al., 2019) and HuggingFace (Wolf et al., 2020). We use the Adam optimizer (Kingma & Ba, 2014) with a learning rate of 1e-4. For multi-turn experiments (Sec. 4.2), we use a batch size of 8 and train for 10 epochs. For single-turn experiments (Sec. 4.3), we use a batch size of 64 and train for 100 epochs. For each setting, we train models with three random seeds; plots are generated with Seaborn (Waskom, 2021) and show the 95% confidence intervals.

### C.3 Additional Details: Mechanism Analysis

**Cycling through responses** Given a template $t$ with reassembly rules $r_1, \ldots, r_M$, we select conversations in which $t$ appears $n > 1$ times. For some $i < n$, we identify the turn at which $t$ is matched for the $i^{th}$ time in the conversation, and replace the response with $r_j$ for some $j \neq i$. Then we evaluate the model's response at the next occurrence of template $t$. If the model used the modular sum, we would expect it to give the *Same* response as before the intervention (responding with $r_{i+1\%M}$); if it uses the intermediate output, we would expect it to instead reply with $r_{j+1\%M}$ (*Increment*). Figure 6a indicates that the model almost always increments its response, indicating that the model relies on previous responses to




(a) Accuracy (end of training).

(b) *Null* inputs.

Figure 12: We recreate our experiments from Sec. 4 using a different version of the cycling mechanism for null templates. In our original experiments, we incremented the cycle number every time the null input is matched, even if the subsequent response is to read from the memory queue. Here, we instead increment the cycle number only when the null input is followed by a null response. While the overall trends are similar, models trained on the second version of the data perform better overall (Fig. 12a); and accuracy on null inputs does not decrease as dramatically as a function of the number of enqueues in the conversation (Fig. 12b). This suggests that the task is easier for models to learn when they can keep track of the cycle number using their previous responses, rather than having to count the number of null inputs. See App D.1 for more details.

update the response cycle.[5]

**Memory queue** We conduct a similar experiment to test the memory queue mechanism. We select conversations containing $n > 1$ two dequeue turns. For some $i < n$, we identify the $i^{th}$ dequeue turn and replace the response with a constant string, corresponding to a null response, and evaluate the model's response at dequeue $i + 1$. If the model used the gridworld automaton, we would expect it to give the *Same* response as before, replying with memory $i + 1$. If the model relied on intermediate outputs, we would expect it to instead reply with memory $i$ (*Decrement*). Figure 6b shows that the model almost always decrements the memory counter, indicating that it examines its own earlier responses to identify the state of the memory queue.

# D Additional Results

## D.1 Errors on null inputs

In Sec. 4, we found that models perform worse on inputs that do not match any of the templates, in situations where the memory queue is empty. We refer to inputs that do not match any templates as *null inputs*, and say that they match the *null template*. Note that, like the other templates, the null template is associated with multiple reassembly rules, and the model should cycle through these rules when the null template is matched multiple times. (In our experiment, there are five rules associated with the null template.) We conjecture that the lower performance on null inputs could be related to difficulty tracking the cycle number for null templates.

In particular, there is some ambiguity in how to track the cycle number for the null template, because a null input does not always lead to a null response: if the memory queue is non-empty, the model should respond by reading from the memory queue. In our experiments, we increment the cycle number every time the null input is matched, even if the subsequent response is to read from the memory queue. However, we could instead increment the cycle number only when the null input is followed by a null response. For example, consider a case where the null template is associated with three reassembly rules ("Response 1", "Response 2", "Response 3"). The difference between these two mechanisms is illustrated in the following conversation:

[5]The difference between *Full response* and *Prefix only accuracy* indicates that the model generally selects the reassembly rule as predicted by the *Increment* hypothesis, but does not implementing the copying step correctly, perhaps because different reassembly rules can use different decomposition groups.

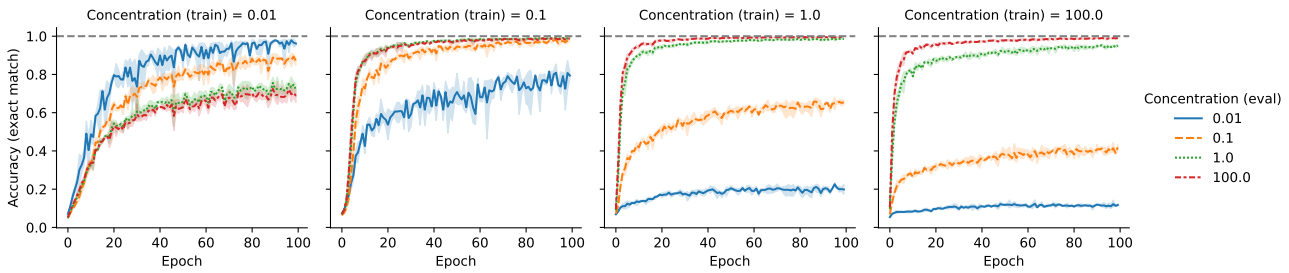



Figure 13: We train and evaluate models on datasets that vary in how likely it is for an n-gram to appear multiple times in a sequence. These training curves correspond to the experiments discussed in §4.3. Lower values of the concentration parameter, $\alpha$, correspond to higher amounts of repetition. For each setting, we train models with three random seeds and plot the accuracy (mean and 95% CI) on each of the four test distributions over the course of training. The biggest performance drop occurs when models trained with $\alpha_{\text{train}} > 0.01$ are evaluated on the setting with the most repetition ($\alpha_{\text{test}} = 0.01$); accuracy on this data also improves more slowly compared to the other settings, even when $\alpha_{\text{train}} = 0.01$.

| *User* | *Cycling on null inputs* | *Cycling on null responses* |
|---|---|---|
| U: Null. | E: Response 1. | E: Response 1. |
| U: Memory A. | E: Enqueue. | E: Enqueue. |
| U: Null. | E: Dequeue A. | E: Dequeue A. |
| U: Null. | E: Response **3**. | E: Response **2**. |

We hypothesize that the first mechanism (*Cycling on null inputs*) is more difficult for the model to learn; for example, the model cannot determine the cycle number by using the intermediate output mechanism described in Sec. 3.2. To test whether this is the case, we create new conversation dataset using the same script as in our original experiments, but using the second approach to determining the cycle number for null inputs (*Cycling on null responses*). All other training details are unchanged. The results of this experiment are plotted in Fig. 12. While the error patterns are broadly similar in both cases, models trained on this second version of the data perform better overall, and do not suffer as much performance degradation as a function of the number of enqueues earlier in the conversation. This could suggest that the task is easier for the models to learn when they can determine the cycle number as a function of previous null outputs, rather than having to count the number of null inputs.

### D.2 Copying mechanisms

In Fig. 13, we plot the training curves corresponding to the experiments described in §4.3. Models generalize the worst to data with the highest degree of internal repetition ($\alpha_{\text{test}} = 0.01$); this data also takes models longer to learn. This agrees with the findings of Zhou et al. (2023) and could suggest that induction-head style mechanisms are easier for Transformers to learn compared to mechanisms that rely on position arithmetic.

In Fig. 14, we recreate the results from Fig. 5c, but plotting the results separately for each final-layer attention head. As discussed in § 4.3, in this plot, positive values indicate that the attention head has a preference for attending on the basis of position rather than content, and negative values indicate a preference for attending based on content (i.e., to tokens that have the same $n$-gram prefix as the current token), rather than position. Interestingly, within each model, the majority of attention heads show broadly similar patterns, perhaps indicating that the models encode the same mechanism redundantly across multiple heads. This result echoes the findings of Singh et al. (2024), who find that models learn multiple parallel induction heads. Fig. 14 also illustrates that none of the attention cleanly corresponds to one of our hypothesized mechanisms, underscoring the challenges of aligning real-world Transformers with interpretable symbolic mechanisms.

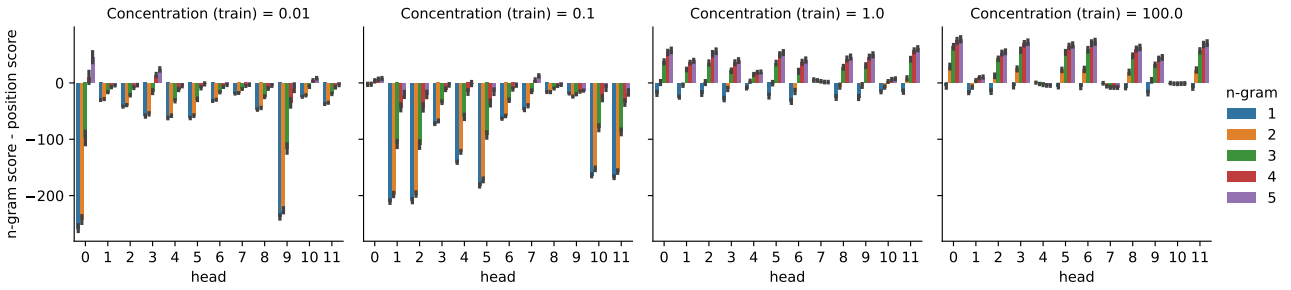



Figure 14: Which mechanism do Transformers use to copy segments of the user's input? At each copying step, we can identify the position in the input we should read from next by counting the number of tokens in each decomposition group. To investigate whether models use this mechanism, we compare the difference in the average attention score between queries and keys under two conditions: either the key has same $n$-gram prefix as the current output, but appears at the wrong position; or the key appears at the target position but has a different $n$-gram prefix. In Fig. 5c, we averaged this metric over all 12 attention heads in the final layer; here, we show the results for each final-layer attention head individually. Each column corresponds to a model trained on data generated with a different concentration parameter $\alpha$, with lower values corresponding to sentences that are more likely to repeat the same $n$-grams multiple times. For each model, the majority of attention heads show broadly similar patterns, suggesting that similar mechanisms are implemented redundantly by multiple heads.