Thinking Like Transformers [3]

Authors: Gail Weiss, Yoav Goldberg, Eran Yahav

Review by Ricardo Yanzon, 03671192

February 14, 2025

1 Introduction

In recent years, transformers have become foundational to various machine learning applications due to their ability to process large sequential data effectively. This is particularly important for tasks like natural language processing, which rely on understanding relationships within sequences of tokens. However, while recurrent neural networks (RNNs) have established parallels with finite state machines that help in understanding their internal mechanics, limitations, and expressive power, transformers lack an equivalent computational model. The paper "Thinking Like Transformers" by Weiss, Goldberg, and Yahav [3] addresses this gap by introducing the *Restricted Access Sequence Processing* (RASP) language, a symbolic abstraction that models the attention mechanisms of transformer encoders. The model aims to help understanding how transformers process information through sequence operations such as attention and feed-forward computations. The scope of this report is to provide a comprehensive review of the main contributions of the above mentioned paper, analyzing the proposed RASP language, the computational insights it provides into transformer-based architectures, and the broader implications for future research. The report is structured as follows: It begins with an introduction to the RASP language, describing its core components and how these map to those of a transformer encoder. Next, it analyzes RASP regarding four key aspects to understand its capabilities and limitations, namely its expressivity, convertibility to and from transformers, as well as the optimality and ability to learn of the resulting transformer. Then, a possible extension of RASP by integrating Chain-of-Thought reasoning is proposed and discussed, reflecting a current topic in transformer research. Finally, the report discusses the paper's theoretical contributions and proposes possible directions for future research.

2 RASP

The RASP language defines a transformer's computation as a composition of two primary components, attention and feed-forward operations. The paper maps these components to simple primitives, which are called *sequence operators* (s-ops). The most important s-ops are *select* and *aggregate*.[1] The *select* s-op constructs a binary matrix that captures relationships between tokens based on specified conditions, such as equality or positional comparisons. This pair-wise selection defines which tokens are *attended to* by others, much like the attention mechanism in transformers. The *aggregate* s-op then combines a selection matrix with a token sequence by gathering the values at positions where the selection condition is met and averaging them to produce a new sequence. Together, the *select* and *aggregate* operators allow a RASP program to perform tasks such as

[1] For a more complete list of s-ops as well as built-in functions from the RASP standard library, see Appendix A.

counting occurrences, sorting elements, or transferring information across positions.[2]

3 Review

The following section is an analysis of the strengths and constraints of the RASP language along four key dimensions: Its expressivity in encoding a solution for a given problem; the convertibility between RASP programs and transformer architectures; the optimality of the transformer architectures derived from RASP programs; and whether transformers derived from RASP programs ultimately converge to learn in the intended manner.

3.1 Expressivity

In this context, expressivity refers to the range of decision problems or formal languages that a RASP program can encode. To address this, a distinction between two cases needs to be done: When the input length is fixed (bounded) and when it is unbounded.

3.1.1 Bounded Input Length

When the input length is fixed at n, only a finite number of sequences exist. Assuming that a *beginning-of-sequence* (BOS) token is always included, a RASP program can be found for every possible problem. This is achieved by using RASP's pair-wise attention mechanism to iterate token-by-token through the entire sequence while always attending to the BOS token. This strategy allows to selectively transform parts of the input while leaving the rest untouched, effectively hard-coding a solution. Thus, for any fixed n, a RASP program exists to express the given problem.

3.1.2 Unbounded Input Length

The previous approach does not scale for unbounded input lengths, where the set of possible sequences is infinite. However, the authors show that RASP programs can be found for regular languages and some context-free languages (e.g. *Dyck-k*) that hold across varying input sizes. The *Dyck-k* example further illustrates that RASP program can effectively manage hierarchical structures. Assuming convertibility, this result implies that transformers derived from RASP programs are also capable of processing such hierarchies. Nevertheless, because RASP lacks loops or other mechanisms for dynamically adjusting the number of computations, each program is limited to a fixed number of operations. Consequently, RASP is not Turing-complete, which limits the scope of problems that it can express.

In [4], the authors examine how transformers derived from a RASP program generalize to input lengths beyond those seen during training. They find that transformers trained for all input lengths up to a number n tend to length-generalize well when the target task is implementable by a "short, RASP-friendly" program. To formalize "short" and "RASP-friendly", the paper introduces RASP-L, a subset of RASP that poses limits upon program size, allowed operations, and their composition. In a series of examples, the authors show how transformers trained to learn a RASP-L program can generalize beyond the limitations of the language.[3]

3.2 Convertibility

Besides some limitations, the conversion between RASP programs and transformer architectures works in both directions: *Compiling* a RASP program into a transformer and *decompiling* a

[2] Examples and visualizations to *select* and *aggregate* can be found in Appendix A.
[3] An example is provided in Appendix B.

transformer into a human-readable program.

3.2.1 RASP to Transformer

The authors show how basic primitives in RASP correspond to components of a transformer. In particular, each *select-aggregate* pair in a RASP program is mapped to one attention head. Multiple such operations can be parallelized when they are independent of each other, meaning that they are assigned to different heads within the same transformer layer. On the other hand, if a *select-aggregate* pair depends on the output of an earlier *select-aggregate* pair, its corresponding attention head is placed in a subsequent layer. Consequently, the depth of a transformer in terms of layers is determined by the maximum number of dependent operations in the RASP program, while the number of attention heads per layer equals the number of independent *select-aggregate* operations. Further, element-wise operations (e.g. arithmetic or boolean functions) are handled within the feed-forward sublayer of the same transformer layer. This method has been validated on tasks such as *sorting*, *histogram computation*, and *recognizing Dyck-k languages*, confirming that the derived transformers were capable of solving the RASP-encoded problems. However, it is important to note that these results assume the presence of a BOS token and attention supervision during training, where an additional loss term forces the transformer to adapt to a desired attention pattern. Without these conditions, there is no guarantee that a transformer will learn the RASP-intended solution.[4]

3.2.2 Transformer to RASP

Building on RASP, the authors of [1] propose a method for the reversed conversion: Building transformers such that they can be translated into Python code. To achieve this, the authors define program-like discrete building blocks that they use to construct transformers. After training, each head or layer of such a transformer can be *decompiled* into `if-else` statements or lookup tables. They demonstrate high accuracy of the resulting programs on short inputs for tasks like *reverting a string*, *sorting*, and *Dyck-k language recognition*. While a full discussion of their approach is beyond the scope of this paper, their work underscores the feasibility of converting transformers into RASP-like representations despite inherent limitations. In the future, this approach could provide an alternative to existing interpretability methods, that often require extensive manual effort while providing misleading or incomplete insights.

3.3 Optimality

The optimality of transformers derived from RASP programs is another important consideration. While the conversion process guarantees an upper bound on the number of layers and attention heads required, it does not ensure that the resulting transformer architecture is optimal, meaning that it is also minimal. For example, in the case of *sorting* as mentioned in the paper, the transformer compiled from the RASP program used a pairwise comparison approach, resulting in a multi-layer architecture. However, after removing one layer from the transformer, it was still able to produce correct results and even improved accuracy. Analyzing its attention heatmaps, the authors concluded that the transformer discovered a simpler *bucket-sort* algorithm during training, which was more efficient than the original RASP program's approach. This highlights how an inefficient implementation in RASP consequently results in a suboptimal transformer in terms of size and complexity. To further emphasize this, two implementations for counting vowels in an input sequence are provided in the Appendix: An efficient implementation that *compiles* into a single head, single layer architecture; and a less efficient variant, which includes some redundant

[4] Example visualizations can be found in Appendix C.

operations, increasing the total size to two layers and three heads.[5] Moreover, larger transformer architectures, while more expressive, can introduce additional challenges during training, such as slower convergence and increased optimization difficulty. In contrast, smaller architectures, when sufficient for the task, tend to converge faster and more reliably.

3.4 Learnability

Learnability assesses whether a randomly initialized transformer obtained from a RASP program, can reliably converge to the intended solution during training. Although RASP defines specific attention patterns to solve a task, gradient-based optimization does not guarantee convergence to the exact distribution defined by the RASP program, especially in the absence of attention supervision. Still, the likelihood and intensity of divergence depends to some degree on the complexity of the task and the efficiency of the RASP implementation. For simpler tasks or when attention supervision is applied, transformers tend to align more closely with the intended attention patterns. In contrast, for more complex tasks or without supervised attention, the optimization process is more likely to adopt shortcuts, partial solutions or get stuck in local minima that deviate from the RASP attention. For instance, in the previously mentioned *sorting* example, the reduced transformer discovered a more efficient algorithm than the one encoded in the RASP program. Even with attention supervision, small deviations in inner-layer distributions can persist, suggesting that the loss function might not penalize these deviations sufficiently.[6] Finally, the setting of hyperparameters, such as training epochs, weight initialization, and the choice of an optimizer, can further influence the ability of a transformer to learn a RASP attention pattern.

4 Chain-of-Thought

Chain-of-Thought (CoT) reasoning enables large language models (LLMs) to solve complex problems by generating intermediate reasoning steps, much like human step-by-step problem-solving. This approach improves performance on tasks known to be challenging for LLMs, e.g., solving mathematical problems, and enhances transparency by making the reasoning process more explicit. [2] Therefore, integrating CoT into RASP offers the potential to model more complex, multi-step tasks while enhancing interpretability and expressivity.[7] However, as shown in [2], while very large models[8] gain significantly from CoT reasoning, smaller models often suffer decreased performance because of logical inconsistencies in the generated reasoning chains. Since RASP is designed to provide insights for simpler tasks and smaller models rather than to construct large, complex LLMs, the modest benefits of incorporating CoT do not justify its extension within the RASP language.

5 Conclusion & Discussion

The paper "Thinking Like Transformers" [3] provides a significant step forwards understanding the computational mechanisms of transformer architectures by introducing the *Restricted Access Sequence Processing* (RASP) language. This framework formalizes the attention and feed-forward processes of transformers into symbolic operations (*select* and *aggregate*), offering an interpretable abstraction to model their inner workings. RASP further enables the systematic design of transformer architectures tailored to specific computational tasks, ensuring a clear mapping of sequence operations to attention heads and layers.

[5] The RASP implementations and visualizations can be found in Appendix D.
[6] See the example of *reversing a sequence* in Appendix E.
[7] A full proposal for integrating CoT in RASP can be found in Appendix F.
[8] From the paper [2], models with over 100 billion parameters.

However, RASP's scope is limited. Its lack of Turing-completeness and incapacity to dynamically adapt to varying inputs restrict its ability to handle broader and more complex tasks. Additionally, reliance on attention supervision during training highlights challenges in achieving learnability and generalization. [4] The paper also demonstrates that transformers derived from RASP programs can discover more efficient solutions during optimization, highlighting the gap between symbolic programs and the learning of neural based transformers.

Finally, this work highlights the potential for refining RASP and its compilation process to produce more optimal transformer architectures. Future research could focus on advancing also the translation from transformers to RASP representations, enhancing current interpretability methods while providing deeper insights into transformer learning. [1] Moreover, exploring how symbolic abstractions like RASP can be scaled to larger models, and then potentially also integrated with Chain-of-Thought, could expand its applicability to real-world applications. By addressing these challenges, future advancements in RASP could serve as a foundational model for bridging interpretable symbolic abstractions and scalable neural computations, ultimately enhancing our understanding of transformers and their computational boundaries.

References

[1] D. Friedman, A. Wettig, and D. Chen. Learning transformer programs, 2023. URL https://arxiv.org/abs/2306.01128.

[2] J. Wei, X. Wang, D. Schuurmans, M. Bosma, B. Ichter, F. Xia, E. Chi, Q. Le, and D. Zhou. Chain-of-thought prompting elicits reasoning in large language models, 2023. URL https://arxiv.org/abs/2201.11903.

[3] G. Weiss, Y. Goldberg, and E. Yahav. Thinking like transformers, 2021. URL https://arxiv.org/abs/2106.06981.

[4] H. Zhou, A. Bradley, E. Littwin, N. Razin, O. Saremi, J. Susskind, S. Bengio, and P. Nakkiran. What algorithms can transformers learn? a study in length generalization, 2023. URL https://arxiv.org/abs/2310.16028.

Appendices

The Appendix section is structured as follows:

1. Appendix A provides some additional content regarding the RASP language, including examples to *select* and *aggregate*, explanations to some built-in RASP functions, code snippets from the RASP Standard library, as well as visualizations to some used examples from [3]
2. Appendix B contains an example taken from [2].
3. Appendix C describes attention supervision and visualizes how it is used in [3]
4. Appendix D shows the own implementations of the counting vowels example, described in the Optimality section of the review, and provides the resulting transformer architectures.
5. Appendix E shows how transformers trained on RASP programs can diverge in inner layers, while converging in outer.
6. Appendix F provides a proposal on how to integrate Chain-of-Thought into RASP.
7. Appendix G finally includes all the RASP code examples implemented during the writing of this review.

All visualizations used in the appendices are either from the original paper [3] or from heatmaps created by using the RASP built-in `draw` function. All code snippets are either from the official repository or from own implementations provided to help understanding of key aspects of this report.

The original repository can be found at: http://github.com/tech-srl/RASP.

The forked repository can be found at: https://github.com/ricayanzon/RASP.

A new file **additional_examples.rasp** in the root directory of the forked repository was created, where all of the added implementations can be found.

A RASP

The following contains examples and visual representations of some key built-in functions from the RASP language used for writing this report.

The first example shows how the *select* keyword can be used on the input sequences `[0,1,2]` and `[1,2,3]`, with the *smaller than* condition. Calling `select([0,1,2], [1,2,3], <)` compares each element of the first sequence `[0,1,2]` with each each element of the second sequence `[1,2,3]`. It then returns a $3x3$ selection matrix, where each entry is the result of either `0` or `1` (false or true) of the respective comparison.

$$\text{select}([0,1,2],[1,2,3],<) = \begin{bmatrix} \text{T} & \text{F} & \text{F} \\ \text{T} & \text{T} & \text{F} \\ \text{T} & \text{T} & \text{T} \end{bmatrix}$$

The second example builds upon the that and aims to explain the functionality of the *aggregate* function based on the resulting selection matrix from the above example, and the sequence `[10, 20, 30]`. For instance, when applied to the selection matrix from the previous example and the sequence `[10, 20, 30]`, it outputs `10` for the first row (only the first element is selected), `15` for

the second row (first and second elements are selected), and 20 for the third row (all values are selected).

$$\text{aggregate}\left(\begin{bmatrix} \text{T} & \text{F} & \text{F} \\ \text{T} & \text{T} & \text{F} \\ \text{T} & \text{T} & \text{T} \end{bmatrix}, [10, 20, 30]\right) = [10, 15, 20]$$

Other built-in s-ops that are used in own implementations include: *in*, which checks whether a token is in a given list; *indicator*, which converts a boolean sequence into its numeric values by mapping false/true to 0/1 respectively; *indices*, which returns the position index of each token; *length*, which provides the size of the sequence; as well as element-wise boolean and arithmetic operators. Additionally, RASP includes a standard library (rasplib.rasp) with functions such as *full_s*, which takes an input sequence and produces a selection matrix with every entry as true; *count*, which returns the number of occurrences of a specified token in a sequence; and *sort*, which arranges tokens in a defined order, among others.

The RASP implementations of some of these functions used in own implementations, together with some other useful functions, are presented below.

A.1 Example Code from RASP Standard Library

Listing 1: rasplib.rasp

```
full_s = select(1,1,==);
length = round(1/aggregate(full_s,indicator(indices==0)));

def selector_width(sel) {
        at0 = select(indices,0,==);
        sAND0 = sel and at0;
        sOR0 = sel or at0;
        inverted = aggregate(sOR0,indicator(indices==0));
        except0 = (1/inverted)-1;
        valat0 = aggregate(sAND0,1,0);
        return round(except0 + valat0);
}

def has_focus(sel) {
        return aggregate(sel,1,0)>0;
}

def count(seq,atom) {
        return round(
                length * aggregate(
                        full_s, indicator(seq==atom)));
}

def sort(seq,key) {
        select_earlier_in_sorted =
                select(key,key,<) or (select(key,key,==) and select(indices
                    ,indices,<));
        target_position =
                selector_width(select_earlier_in_sorted);
        select_new_val =
                select(target_position,indices,==);
        return aggregate(select_new_val,seq);
}
```

A.2 Visualizations of the *reversing order* example

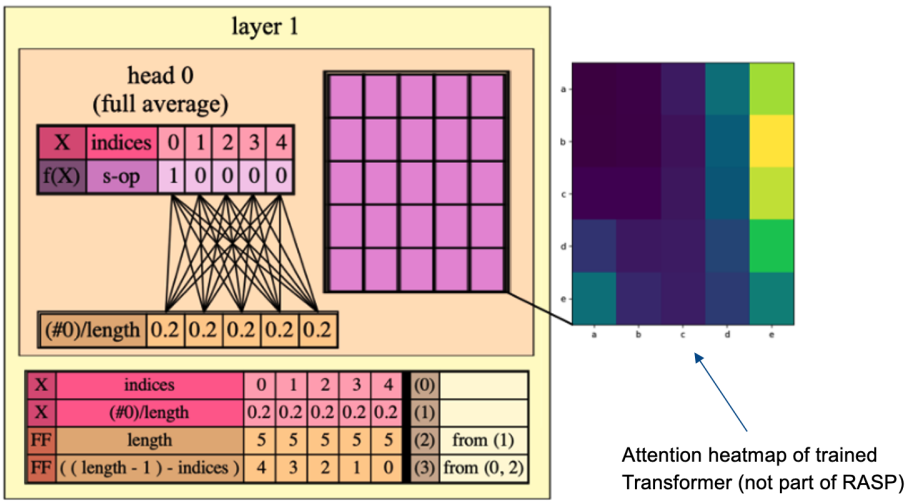

Figure 1: Reversing order: Layer 1

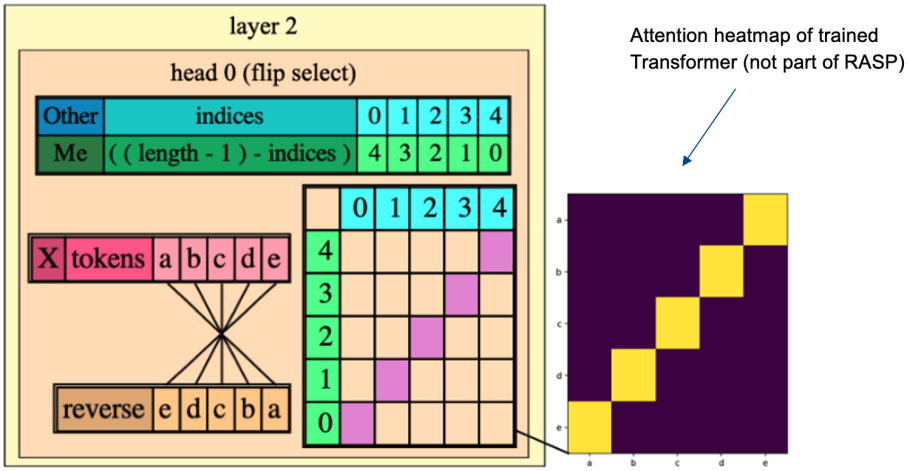

Figure 2: Reversing order: Layer 2

A.3 Example Code from Thinking Like Transformers

The following are some of the implementations from the examples provided in [3], which are referenced in this review.

Listing 2: rasplib.rasp

```
# The reversing a string program

_flip_s = select(indices,length-1-indices,==);
reverse = aggregate(_flip_s,tokens_str);

```

```
# Dyck-1, -2 and -3

def _dyck1_ptf() {
        up_to_self = select(indices,indices,<=);
        n_opens = round((indices+1)*aggregate(up_to_self,indicator(
            tokens_str=="(")));
        n_closes = round((indices+1)*aggregate(up_to_self,indicator(
            tokens_str==")")));
        balance = n_opens - n_closes;
        prev_imbalances = aggregate(up_to_self,indicator(balance<0));
        return "F" if prev_imbalances>0 else
                        ("T" if balance==0 else "P");
}

dyck1_ptf = _dyck1_ptf();

def dyckk_ptf(paren_pairs) {
        # paren pairs should come as list of strings of length 2, e.g.:
            ["()","{}"]
        openers = [p[0] for p in paren_pairs];
        closers = [p[1] for p in paren_pairs];
        opens = indicator(tokens_str in openers);
        closes = indicator(tokens_str in closers);
        up_to_self = select(indices,indices,<=);
        n_opens = round((indices+1)*aggregate(up_to_self,opens));
        n_closes = round((indices+1)*aggregate(up_to_self,closes));
        depth = n_opens - n_closes;
        delay_closer = depth + closes;
        depth_index = selector_width(select(delay_closer,delay_closer,==)
            and up_to_self);
        open_for_close = select(opens,True,==) and
                                        select(delay_closer,delay_closer
                                            ,==) and
                                        select(depth_index,depth_index
                                            -1,==);
        matched_opener = aggregate(open_for_close,tokens_str,"-");
        opener_matches = matched_opener+tokens_str in paren_pairs;
        mismatch = closes and not opener_matches;
        had_problem = aggregate(up_to_self,indicator(mismatch or (depth<0))
            )>0;
        return "F" if had_problem else ("T" if depth==0 else "P");
}

dyck2_ptf = dyckk_ptf(["()","{}"]);
dyck3_ptf = dyckk_ptf(["()","{}","[]"]);
```

B Expressivity

An example described in the paper [4] is that of *counting*: Given an input sequence consisting of a BOS and two tokens a and b holding numeric values, count from a to b, returning $b-a$ successive tokens (one per number) and marking the end with an end-of-sequence token. After training the derived transformer on sequences of all lengths up to $n = 50$, the transformer generalized almost

perfectly to sequences up to length 100. Although the insights gathered by this paper into the expressive power of RASP are questionable, the example demonstrates how transformers trained to learn a RASP program can generalize beyond the limitations of the language.

C Appendix: Convertibility

The following visualization illustrates how attention supervision is used during training to force the transformer to learn the RASP attention pattern.

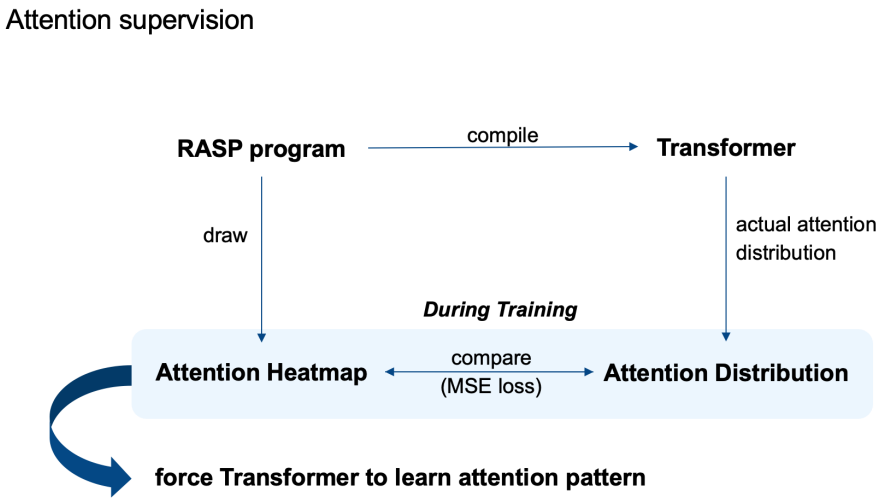

Figure 3: Attention Supervision

The next figure shows how the different attention maps of three RASP programs look like when visualized. This follows the approach of attention supervision as mentioned in [3].

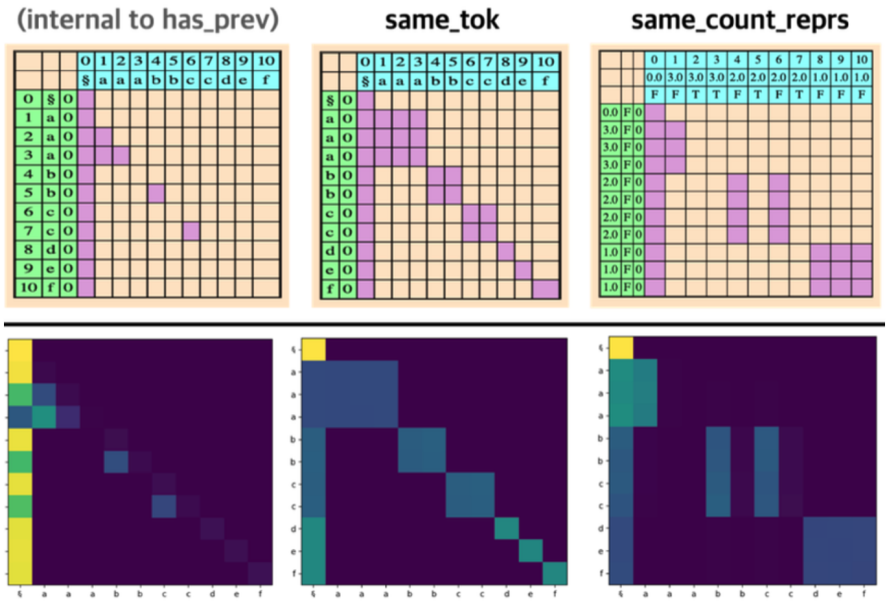

Figure 4: Attention Heatmaps

D Appendix: Optimality

The implementations mentioned in the Optimality section of this review are in the following, together with the resulting visualizations when created with the RASP built-in `draw` function. The problem they solve is that of counting vowels in an input sequence. To be precisely, each token in the input sequence is compared to the following list of tokens: `[\a", \e", \i", \o", "u", \A", \E", \I", \O", \U"]`

Listing 3: Efficient Implementation of the Counting Vowels Example

```
# The (efficient) implementation without select & aggregate:
# Compiles to 1 layer, 1 attention head
def _count_vowels_efficient(tks) {
  is_vowel = tks in ["a", "e", "i", "o", "u", "A", "E", "I", "O", "U"];
  return count(is_vowel, True);
}
count_vowels_efficiently = _count_vowels_efficient(tokens);
```

Listing 4: Inefficient Implementation of the Counting Vowels Example

```
# The (less efficient) implementation with select & aggregate:
# Compiles to 2 layer, 3 attention heads
def _count_vowels_inefficient(tks) {
  is_vowel = tks in ["a", "e", "i", "o", "u", "A", "E", "I", "O", "U"];
  is_vowel_binary = indicator(is_vowel);
  select_self = select(indices, indices, ==);
  is_vowel_binary = aggregate(select_self, is_vowel_binary, 0);
  return count(is_vowel_binary, 1);
}
count_vowels_inefficiently = _count_vowels_inefficient(tokens);
```

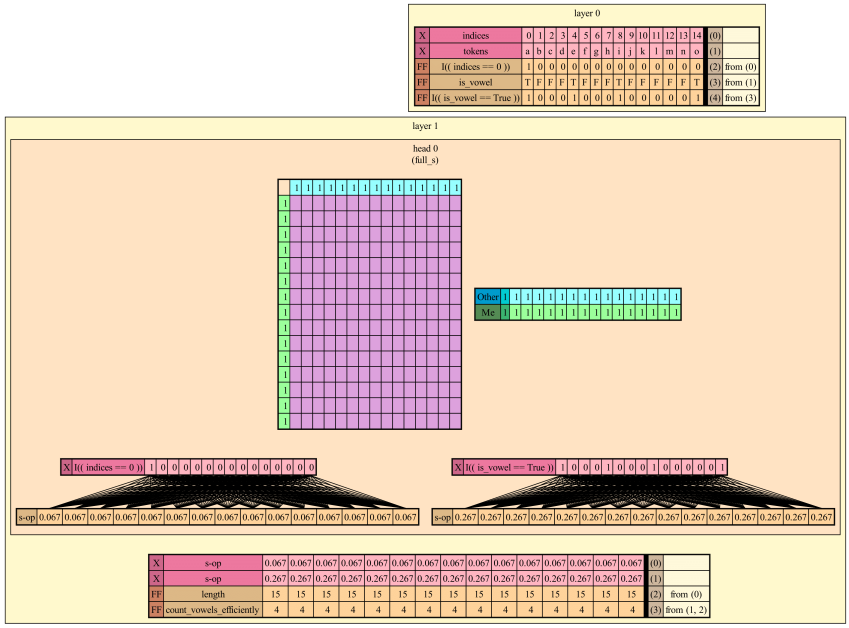

Figure 5: Count Vowels Efficiently

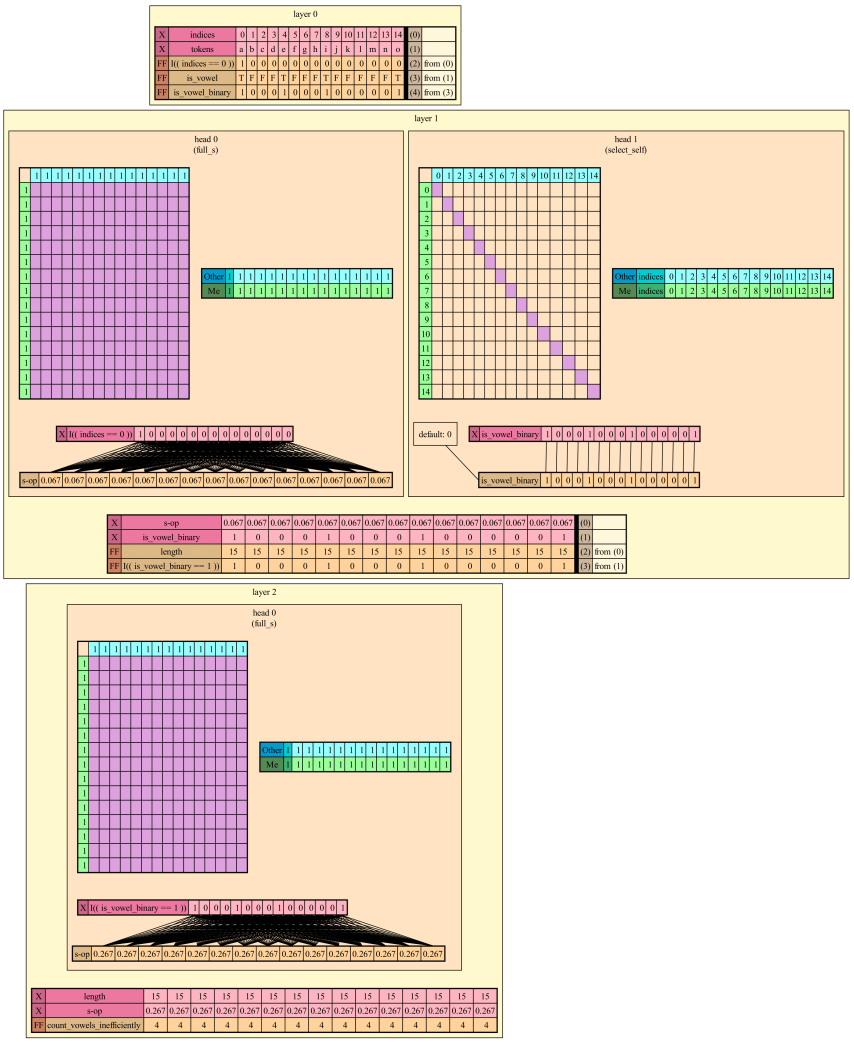

Figure 6: Count Vowels Inefficiently

E Appendix: Learnability

For example, for the transformer trained on *reversing a sequence*, where small deviations in the attention distribution of inner layers persisted, while the outer layers exhibited near-perfect alignment.

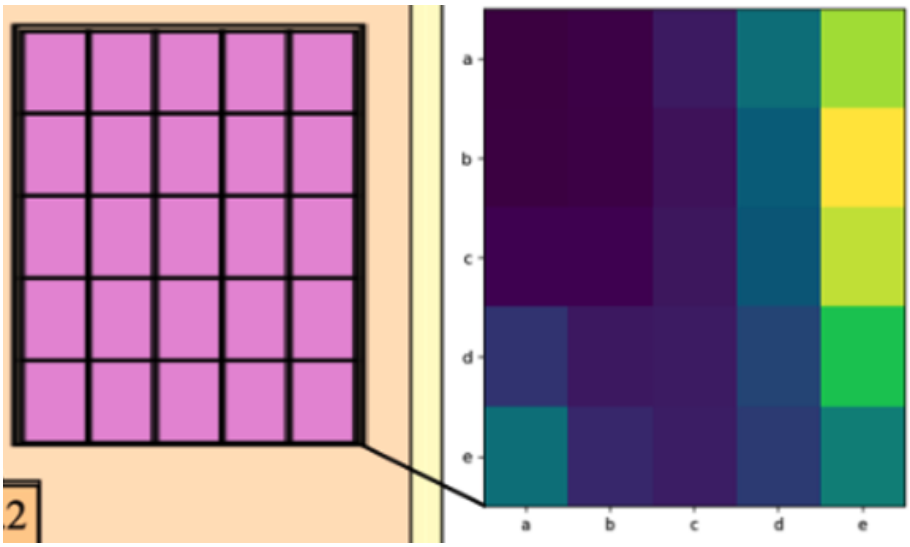

Figure 7: Reversing Order Attention Heatmaps: Layer 1

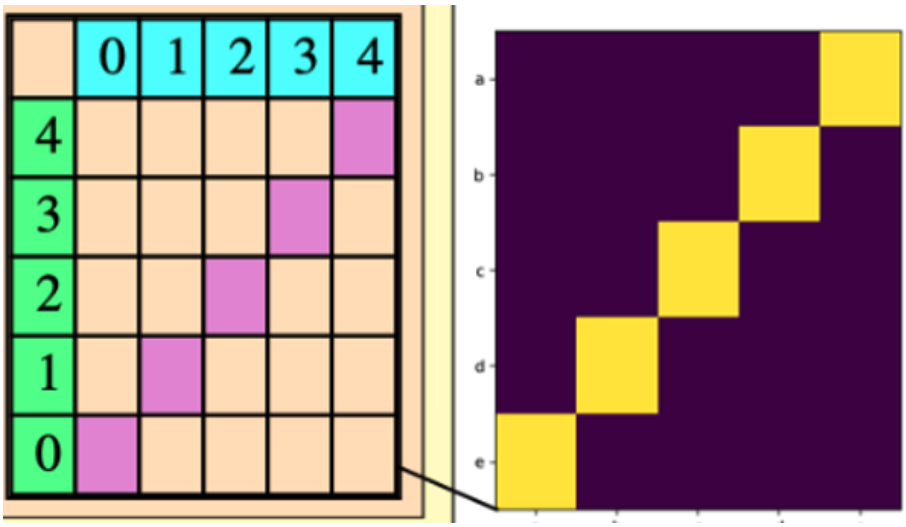

Figure 8: Reversing Order Attention Heatmaps: Layer 2

F Chain-of-Thought

The integration of CoT into RASP would involve introducing new keywords to facilitate step-by-step reasoning. Specifically, the following extensions to RASP are proposed:

1. Reasoning Steps: A new *reason_step* s-op could be introduced, allowing the explicit definition of intermediate reasoning steps. Each step would return a sequence of tokens representing the intermediate result.
2. Ordered Reasoning: To enhance readability and logical flow, a *reason_sequence* keyword could be added. This would allow reasoning steps to be ordered explicitly, ensuring that the

operations are performed in the exact sequential order.

3. Conditional Branching: To handle tasks requiring decision-making based on intermediate results, *reason_sequence* would support conditional branching. This would enable the RASP program to adapt dynamically based on the outcomes of previous steps.

The mapping of the new keywords in RASP to transformer architectures would follow a similar approach to the existing RASP-to-transformer compilation process. Each *reason_step* would correspond to a layer in the transformer, following the same order as they appear in the *reason_sequence* . Attention heads within each *reason_step* would implement the $select - aggregate$ operations and branching logic, while feed-forward sublayers would handle element-wise computations. This mapping ensures that the transformer architecture mirrors the logical structure of the CoT reasoning process, while adhering to the existing compilation process.

G Appendix: Additional RASP implementations

This section shows the complete RASP code added to the original RASP code.

Listing 5: additional_examples.rasp

```
def _has_matching_end_token(seq) {
  sel_equal = select(seq, seq, ==);
  sel_all_backwards = select(indices, length - indices - 1, ==);
  sel_matching_tokens = sel_equal and sel_all_backwards;
  return aggregate(sel_matching_tokens, 1);
}
is_palindrome = count(_has_matching_end_token(tokens_str), 1) == length;


# Selects tokens until the first occurrence of the end_char
def substring_until_last(seq, end_char) {
        later = select(indices, indices, >);
        substring_sel = select(seq, end_char, ==) and later;
  substring_until = aggregate(substring_sel, 1);
  return substring_until * seq;
}

# Selects tokens starting from the first occurrence of the start_char
def substring_from_first(seq, start_char) {
        earlier = select(indices, indices, <);
        substring_sel = select(seq, start_char, ==) and earlier;
  substring_from = aggregate(substring_sel, 1);
  return substring_from * seq;
}

# Sequences need to be separated by separator token
def _check_are_anagrams(seq, sep) {
  seq_1 = substring_until_last(seq, sep);
        seq_1_sorted = sort(seq_1, seq_1);
  seq_2 = substring_from_first(seq, sep);
        seq_2_sorted = sort(seq_2, seq_2);
        matches = seq_1_sorted == seq_2_sorted;
  matches_count = count(matches, True);
  return matches_count == length;
}
```

```
are_anagrams = _check_are_anagrams(tokens, "#"); # only callable with the
    set example

def _count_unique_tokens(str) {
  earlier = select(str, str, ==) and select(indices, indices, <);
  first_occurrences = not aggregate(earlier, 1);
  return count(first_occurrences, True);
}
count_unique_tokens = _count_unique_tokens(tokens_str);

# Example for not being a lower bound for transformer architecture:
# A program without any select and aggregate, that compiles to a
# certain architecture, that can also be implemented using select
# and aggregate and therefore compiles into another architecture,
# consisting of different numbers of attention heads and layers.

# The (efficient) implementation without select & aggregate:
# Compiles to 1 layer, 1 attention head
def _count_vowels_efficient(tks) {
  is_vowel = tks in ["a", "e", "i", "o", "u", "A", "E", "I", "O", "U"];
  return count(is_vowel, True);
}
count_vowels_efficiently = _count_vowels_efficient(tokens);

# The (less efficient) implementation with select & aggregate:
# Compiles to 2 layer, 3 attention heads
def _count_vowels_inefficient(tks) {
  is_vowel = tks in ["a", "e", "i", "o", "u", "A", "E", "I", "O", "U"];
  is_vowel_binary = indicator(is_vowel);
  select_self = select(indices, indices, ==);
  is_vowel_binary = aggregate(select_self, is_vowel_binary, 0);
  return count(is_vowel_binary, 1);
}
count_vowels_inefficiently = _count_vowels_inefficient(tokens);
```