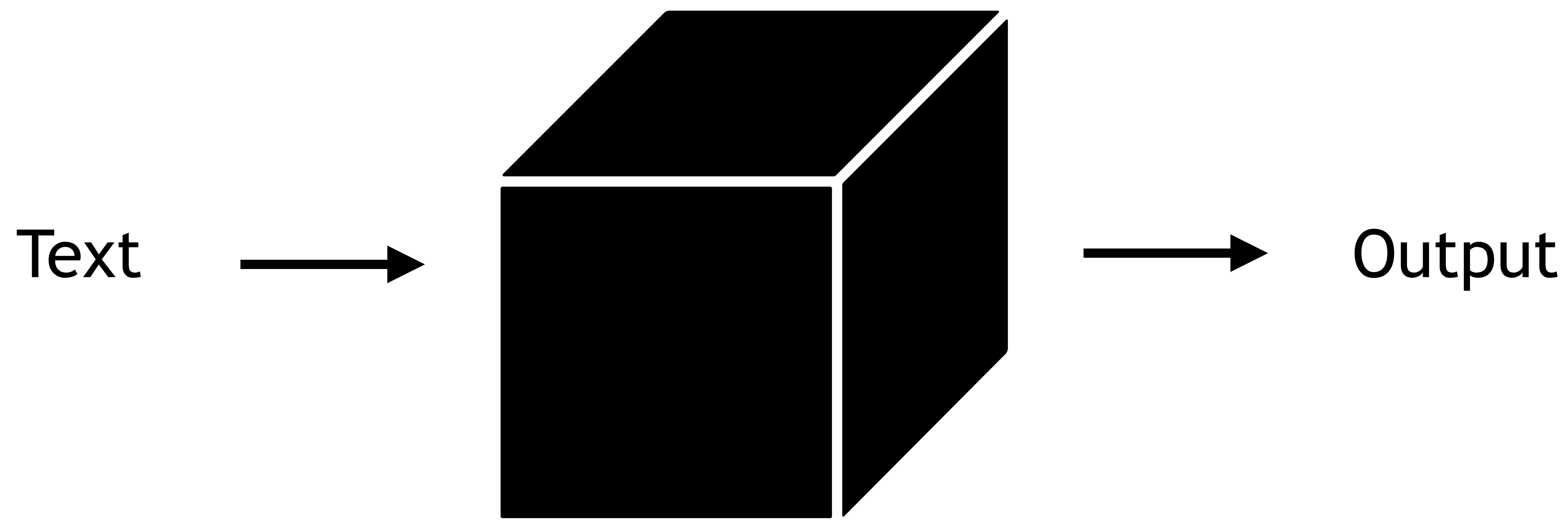
Learning Transformer Programs

Dan Friedman, Alexander Wettig, and Danqi Chen

Department of Computer Science, Princeton University

NeurIPS 2023

NLP systems are black boxes



Problem: Black box systems are difficult to **audit**, **debug**, and **trust**

- Audit for potentially unsafe behavior
- Predict and debug failure cases
- Trust that the model does what we want

If we want to **rely** on this technology, we need to have a better understanding of **how NLP models make decisions**

How can we understand NLP systems?

- ***Prior work***: Post-hoc interpretability
- Probing, feature importance, instance attribution
  - Partial insight, but not complete/faithful descriptions of how the model makes decisions
- Growing body of work on *mechanistic interpretability*
  - Manual effort; still prone to “interpretability illusions”

Our approach: Instead of trying to explain black-box models, modify Transformers to be **mechanistically interpretable by design**

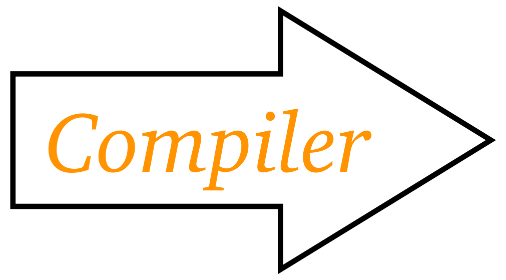
Method: **Optimize** a model to solve a task, and **automatically decompile** it into a **human-readable program**

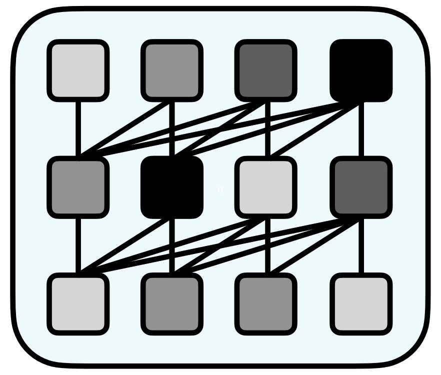
Approach: Transformers as programs

- RASP: A programming language for the Transformer

```
opp = length - indices - 1
flip = select(indices, opp, ==)
reverse = aggregate(flip, tokens)
```

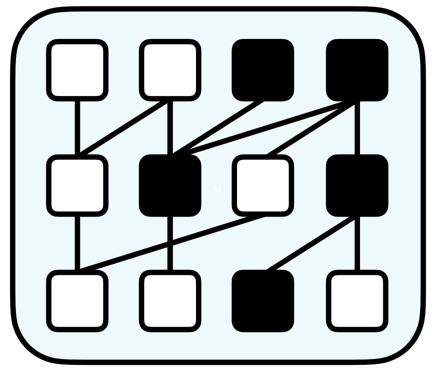
Human-written program





Transformer

This work: Can we train a (modified) Transformer and then automatically *decompile* it into a human-readable program?



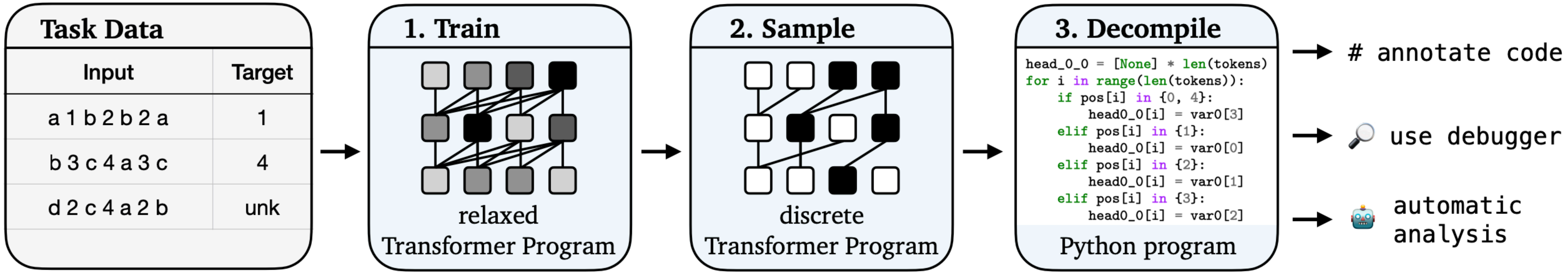
(Modified) Transformer


```
var0 = length - indices - 1
attn = select(indices, var0, ==)
output = aggregate(attn, tokens)
```

Human-*readable* program

Weiss et al., 2021. Thinking like Transformers.
Lindner et al., 2023. Tracr: Compiled Transformers as a Laboratory for Interpretability.

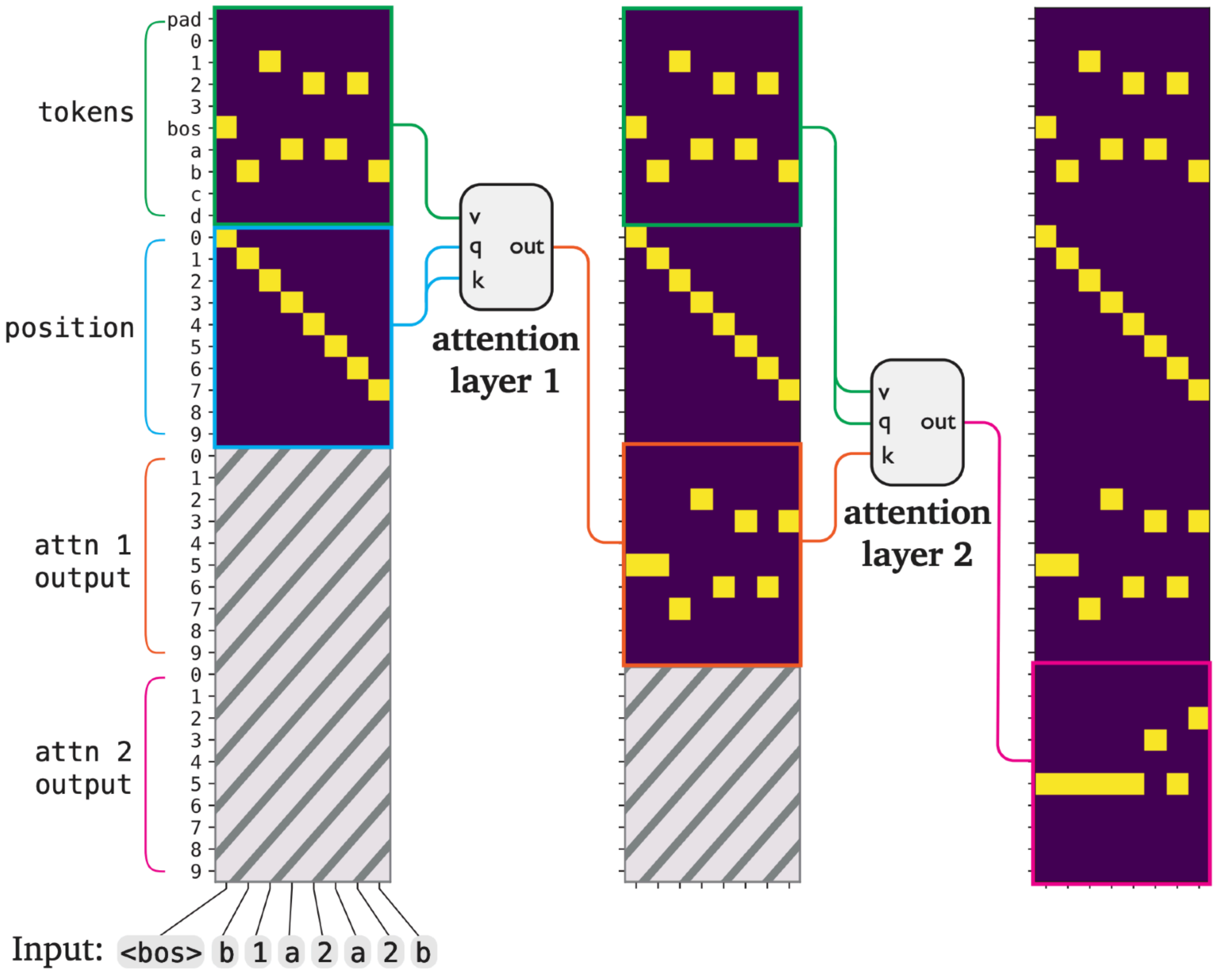
Method: Learning Transformer Programs



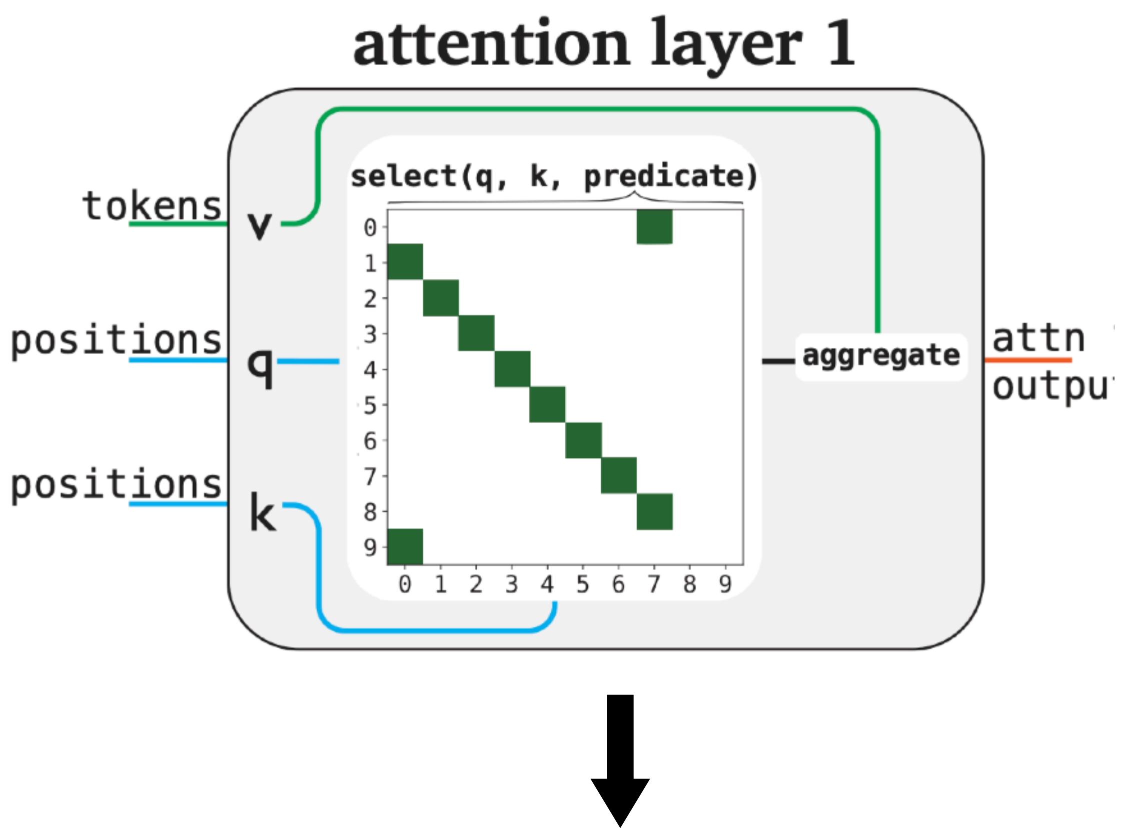
1. Define **constraints** on the network to ensure there is a mapping to a discrete, rule-based program, and **train** a continuous relaxation
2. **Discretize** the weights
3. **Decompile** the discrete model into a Python program

Overview: Constraints

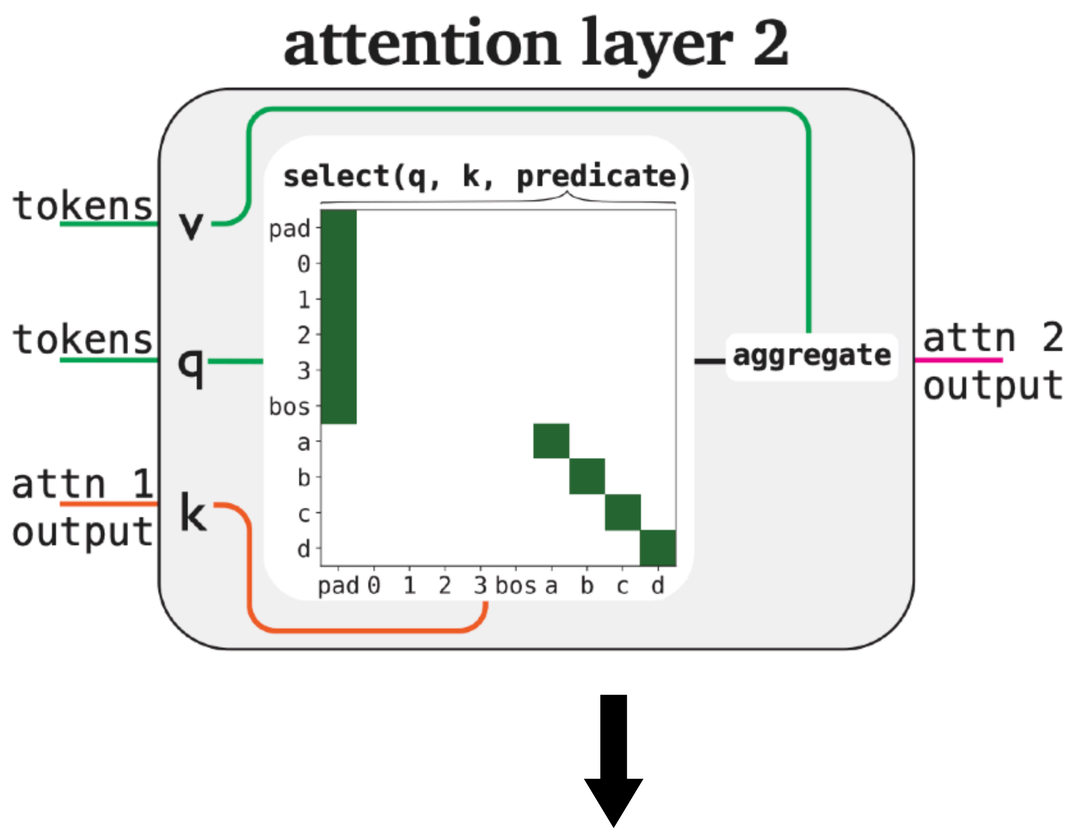
Constraint 1: Disentangled residual stream



Constraint 2: Interpretable sublayers

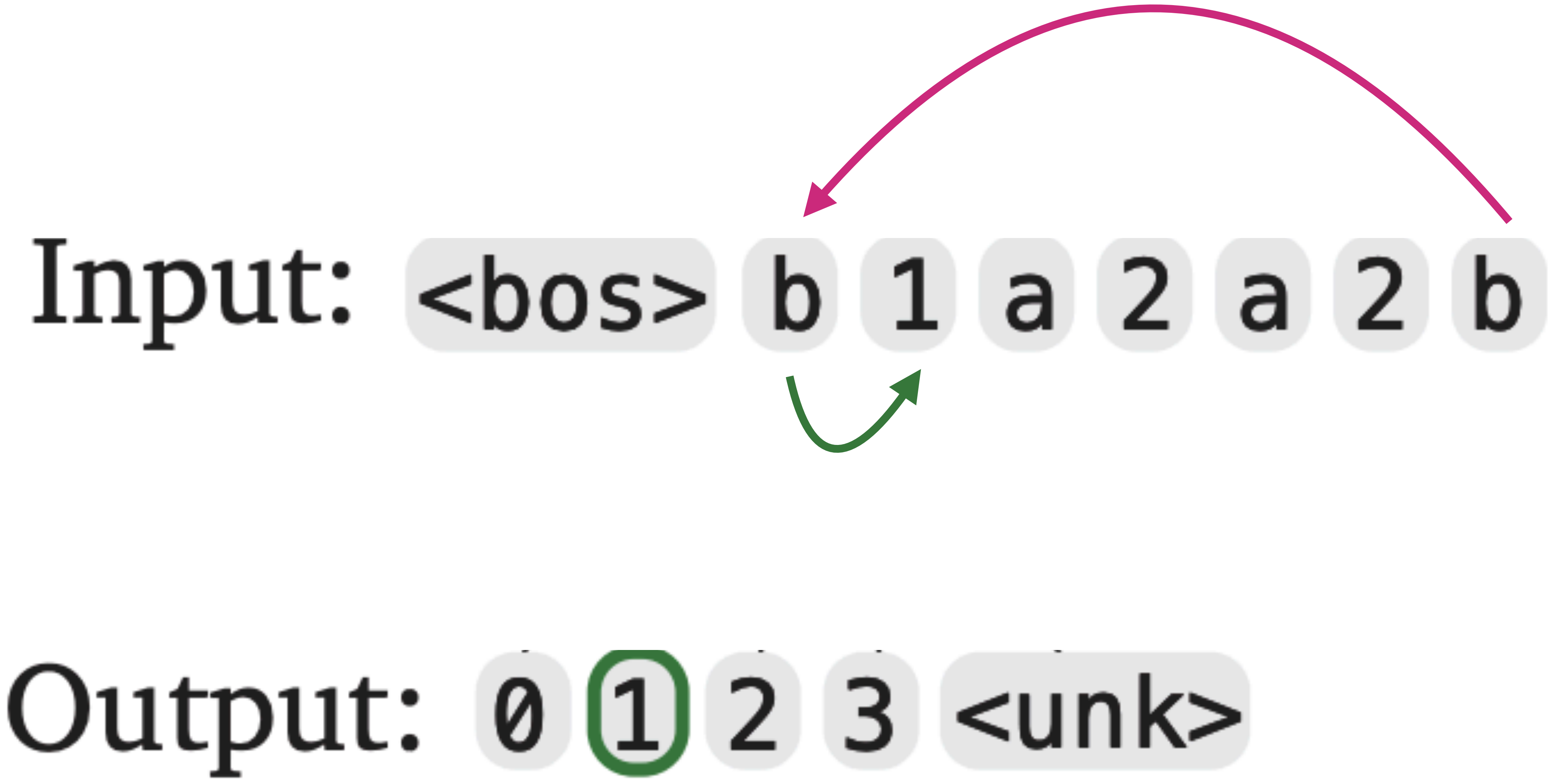



```
def predicate_1(q_position, k_position):
    if q_position in {0, 8}:
        return k_position == 7
    if q_position in {1, 9}:
        return k_position == 0
    if q_position in {2}:
        return k_position == 1
    if q_position in {3}:
        return k_position == 2
    if q_position in {4}:
        return k_position == 3
    if q_position in {5}:
        return k_position == 4
    if q_position in {6}:
        return k_position == 5
```




```
def predicate_2(token, attn_1_output):
    if token in {
        "<pad>", "0", "1", "2", "3", "<s>"
    }:
        return attn_1_output == "<pad>"
    if token in {"a"}:
        return attn_1_output == "a"
    if token in {"b"}:
        return attn_1_output == "b"
    if token in {"c"}:
        return attn_1_output == "c"
    if token in {"d"}:
        return attn_1_output == "d"
```

Illustration: Simple in-context learning



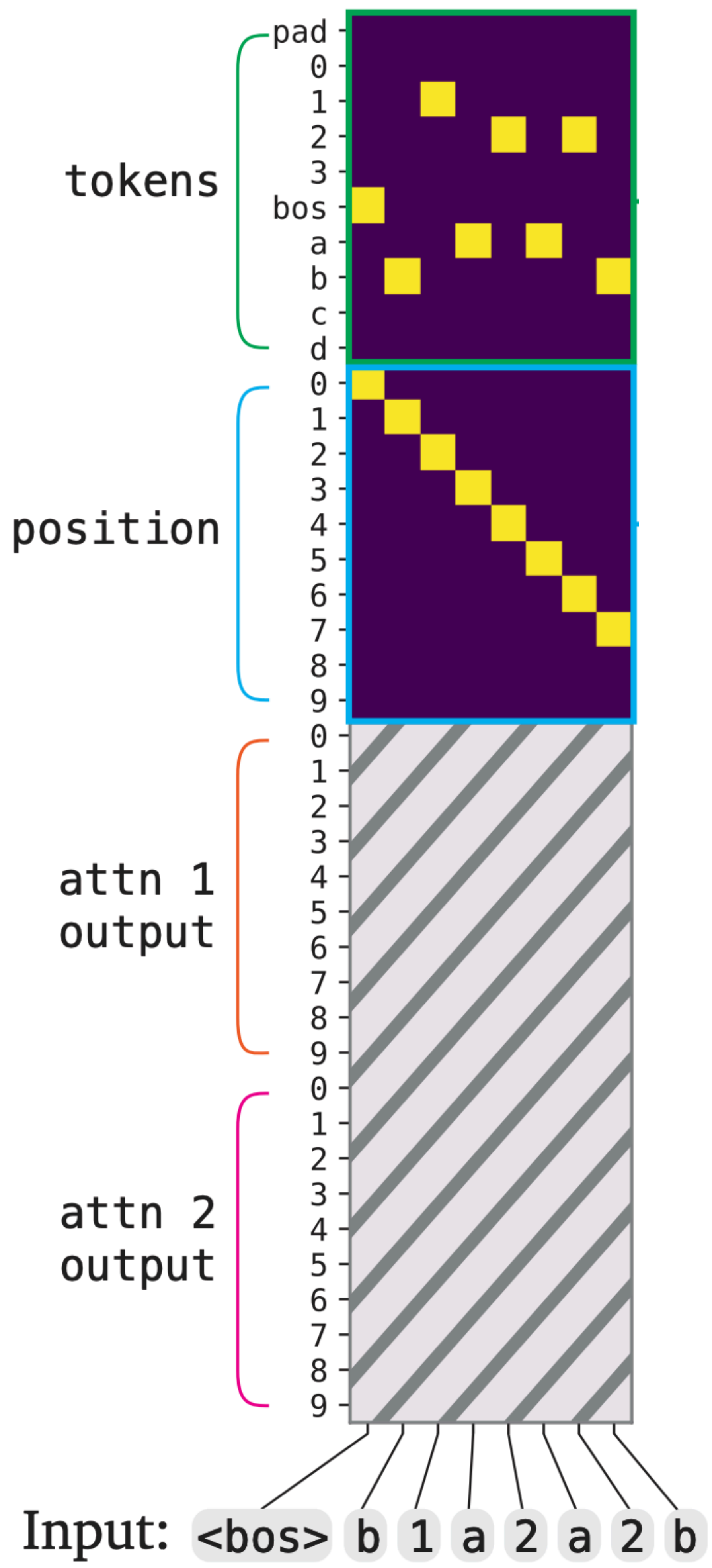
Transformer Program

- Two layers
- One attention head per-layer
- Vocab size = 10
- Sequence length = 10

Brown et al., 2020. Language Models Are Few-shot Learners.
Elhage et al., 2021. A Mathematical Framework for Tranformer Circuits.

Constraint 1: Disentangled residual stream

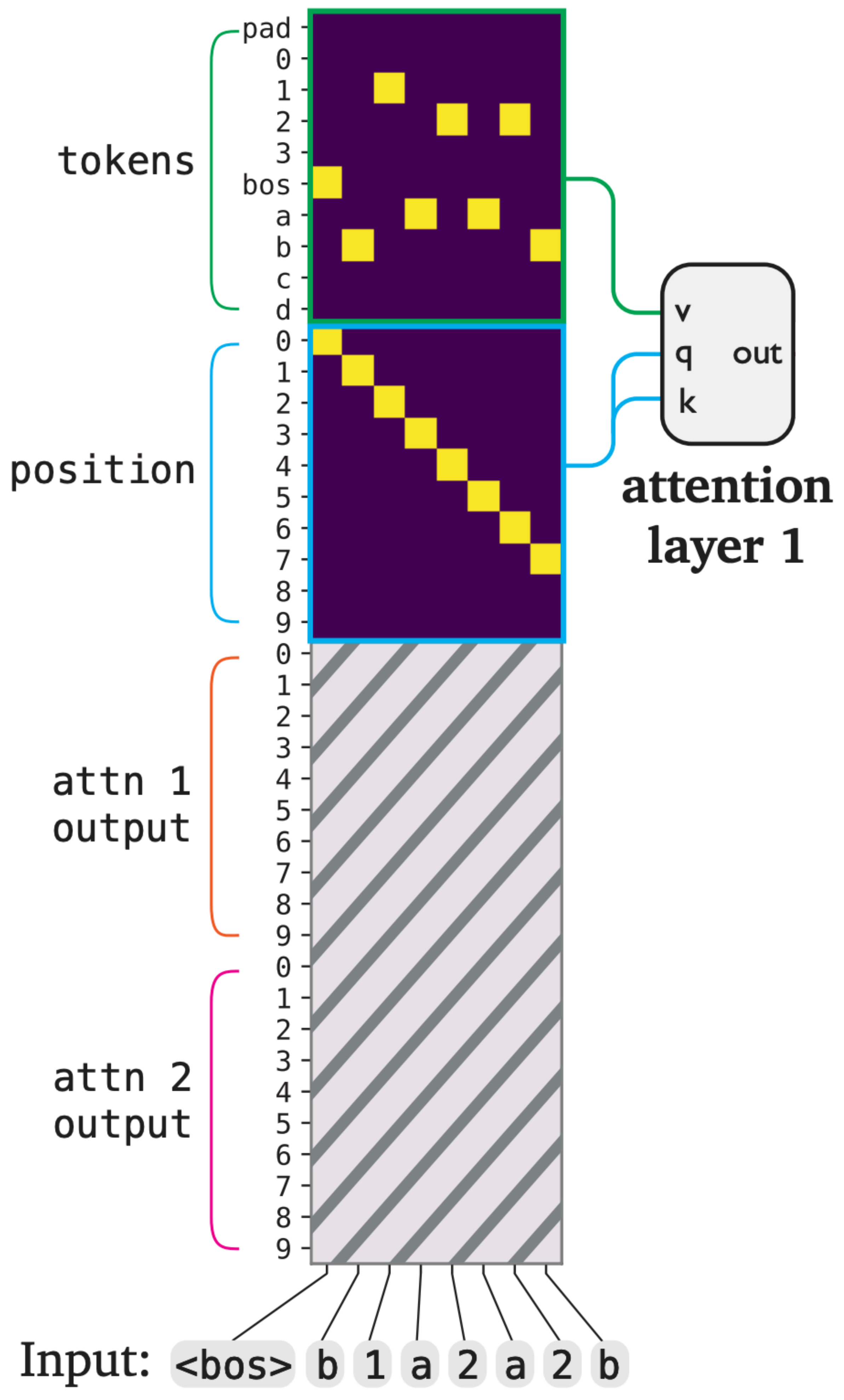
Input embeddings encode two categorical variables (*tokens* and *position*)



Constraint 1: Disentangled residual stream

Input embeddings encode two categorical variables (*tokens* and *position*)

Each attention layer *reads* a fixed set of variables



Reading from the residual stream

Given two input variables, learn:

$$\boldsymbol{\pi} \in \{0,1\}^2 : \pi_1 + \pi_2 = 1$$

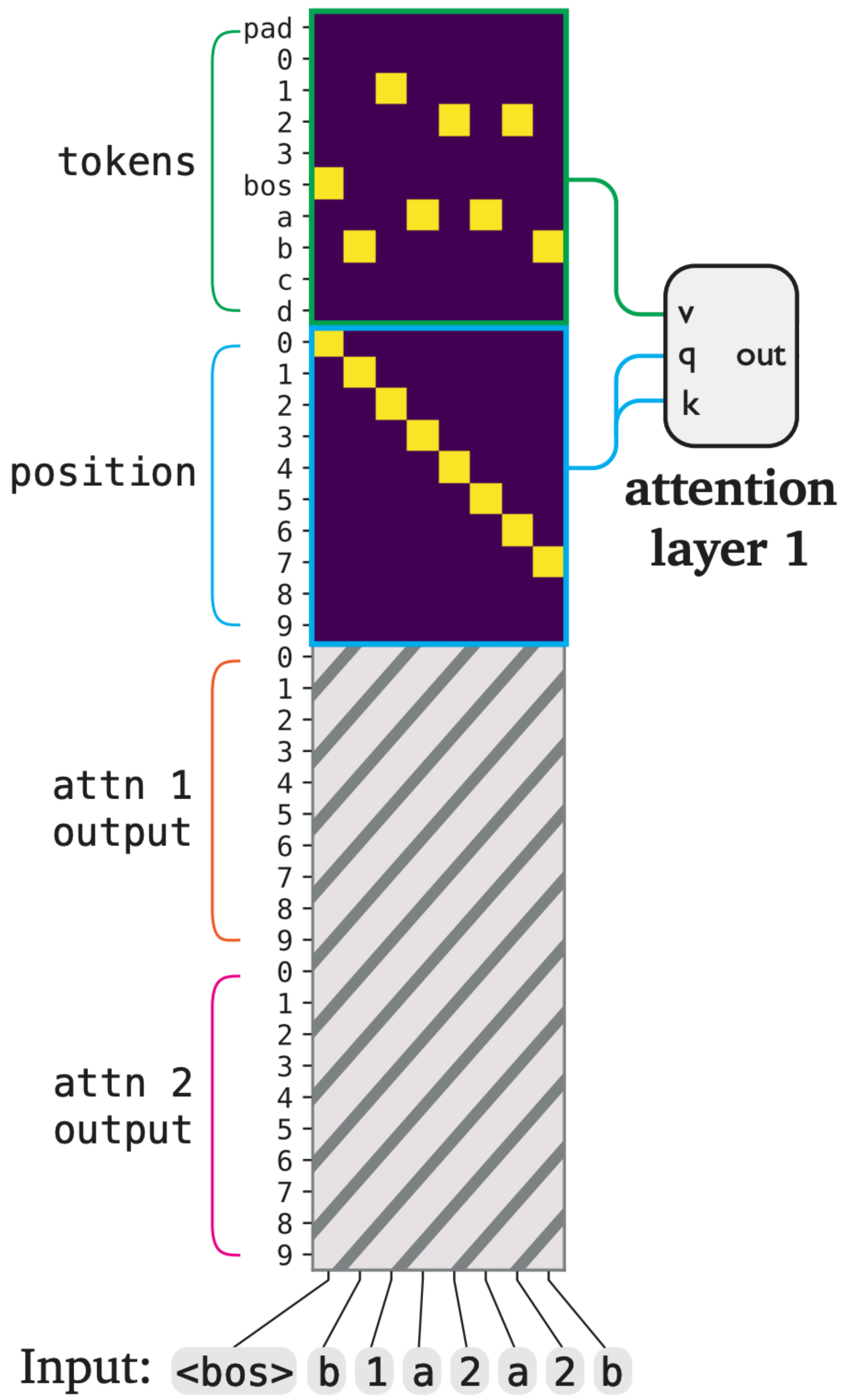
$$\mathbf{W} = [\pi_1 \mathbf{I}; \pi_2 \mathbf{I}]^\top$$

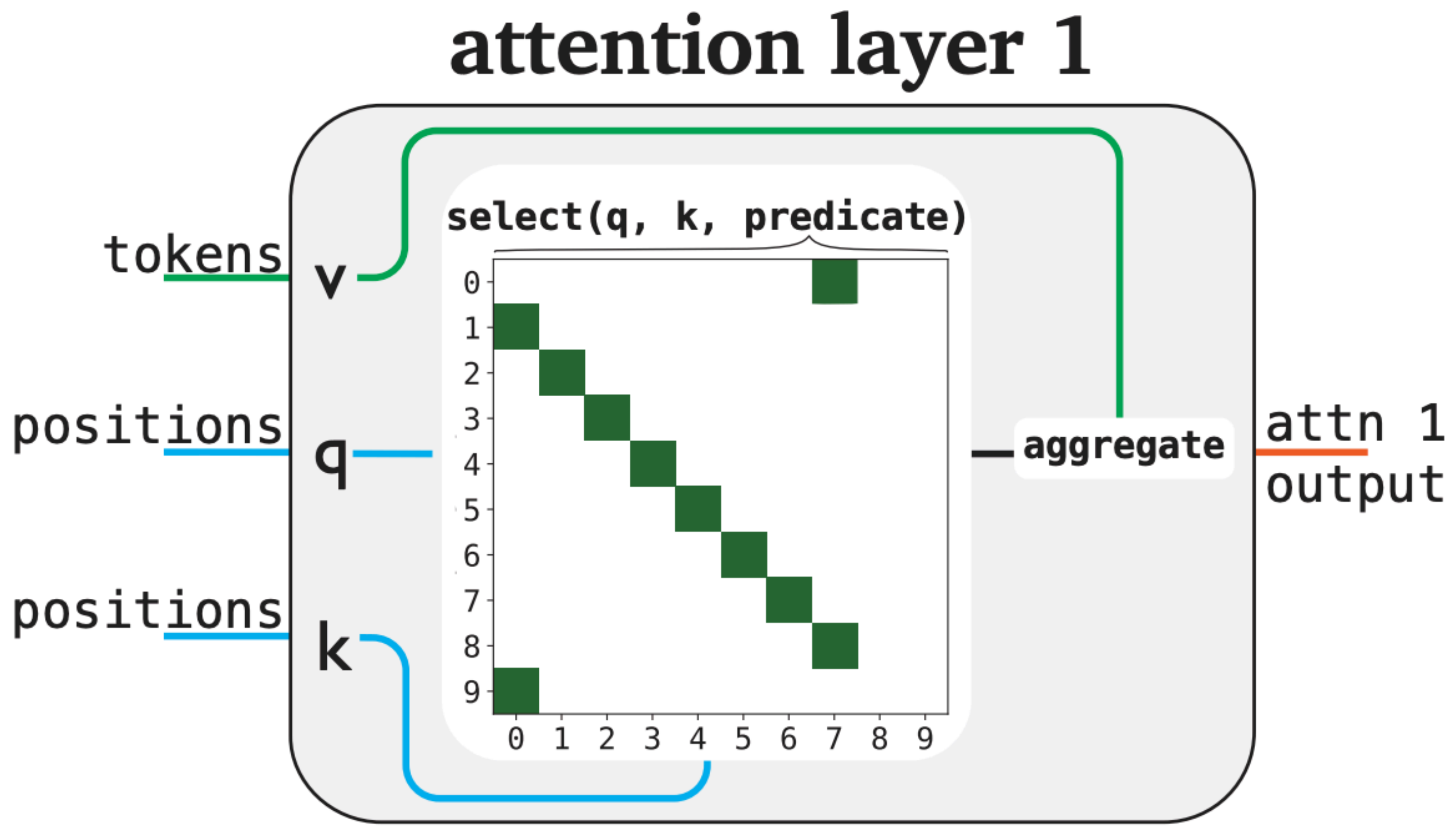
Constraint 2: Interpretable sublayers

Input embeddings encode two categorical variables (*tokens* and *position*)

Each attention layer *reads* a fixed set of variables

… applies a *learned*, *rule-based* transformation



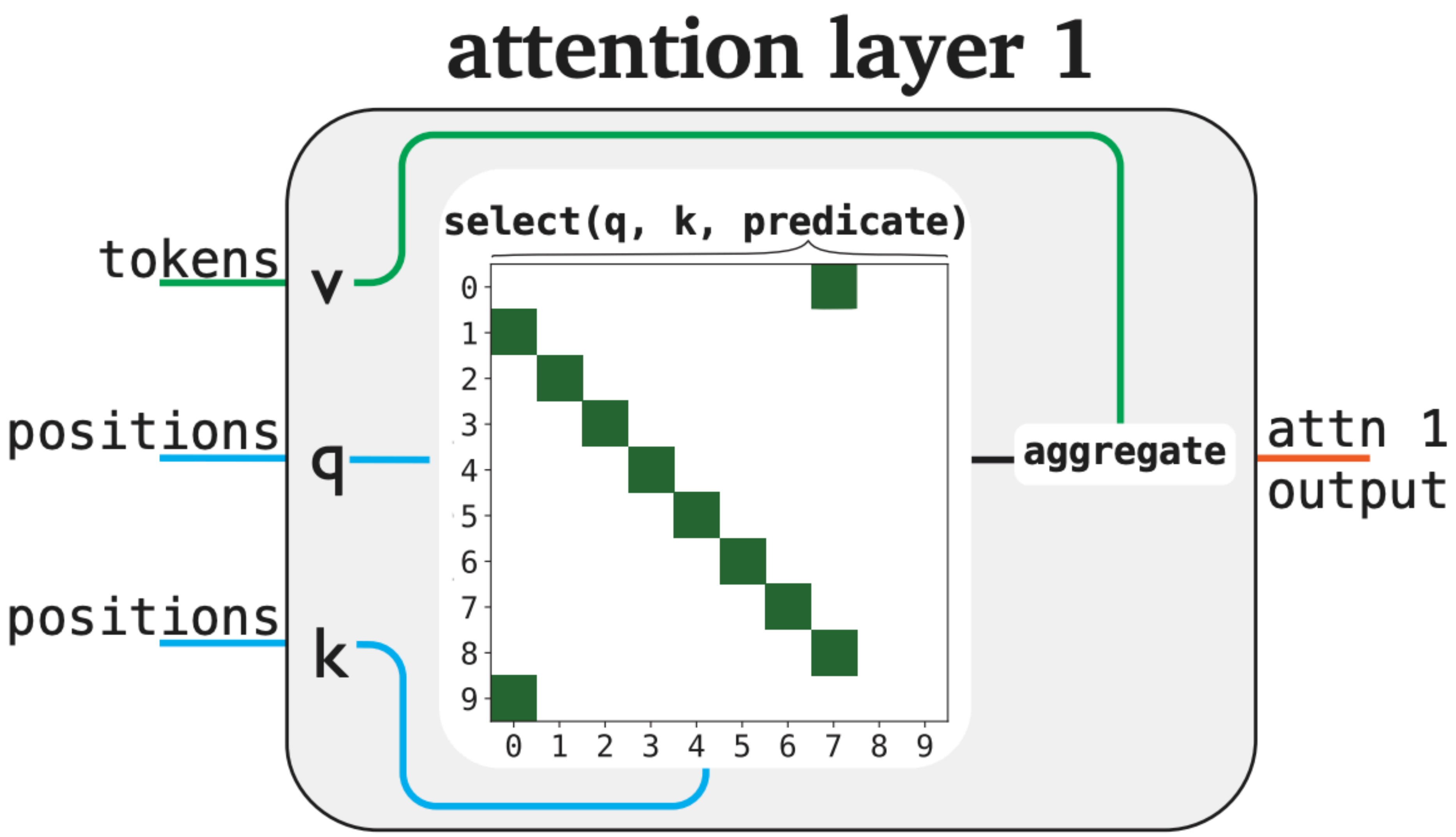


Given input variables with cardinality *10*, learn:

$$\mathbf{W}_{\text{predicate}} \in \{0,1\}^{10\times 10} \quad \text{(rows sum to one)}$$

Note: One-hot attention
- Attend to closest matching token
- Attend to BOS if there's no match

Constraint 2: Interpretable sublayers



Summary: At each position, copy the identity of the token at the previous position

```
attn_1_pattern = select_closest(
    positions, positions, predicate_1)
```

Read *positions* as the key and query variable

```
def predicate_1(q_position, k_position):
    if q_position in {0, 8}:
        return k_position == 7
    if q_position in {1, 9}:
        return k_position == 0
    if q_position in {2}:
        return k_position == 1
    if q_position in {3}:
        return k_position == 2
    if q_position in {4}:
        return k_position == 3
    if q_position in {5}:
        return k_position == 4
    if q_position in {6}:
        return k_position == 5
    if q_position in {7}:
        return k_position == 6
```

Predicate: Each position attends to the previous position

Read *tokens* as the value

```
attn_1_outputs = aggregate(attn_1_pattern, tokens)
```

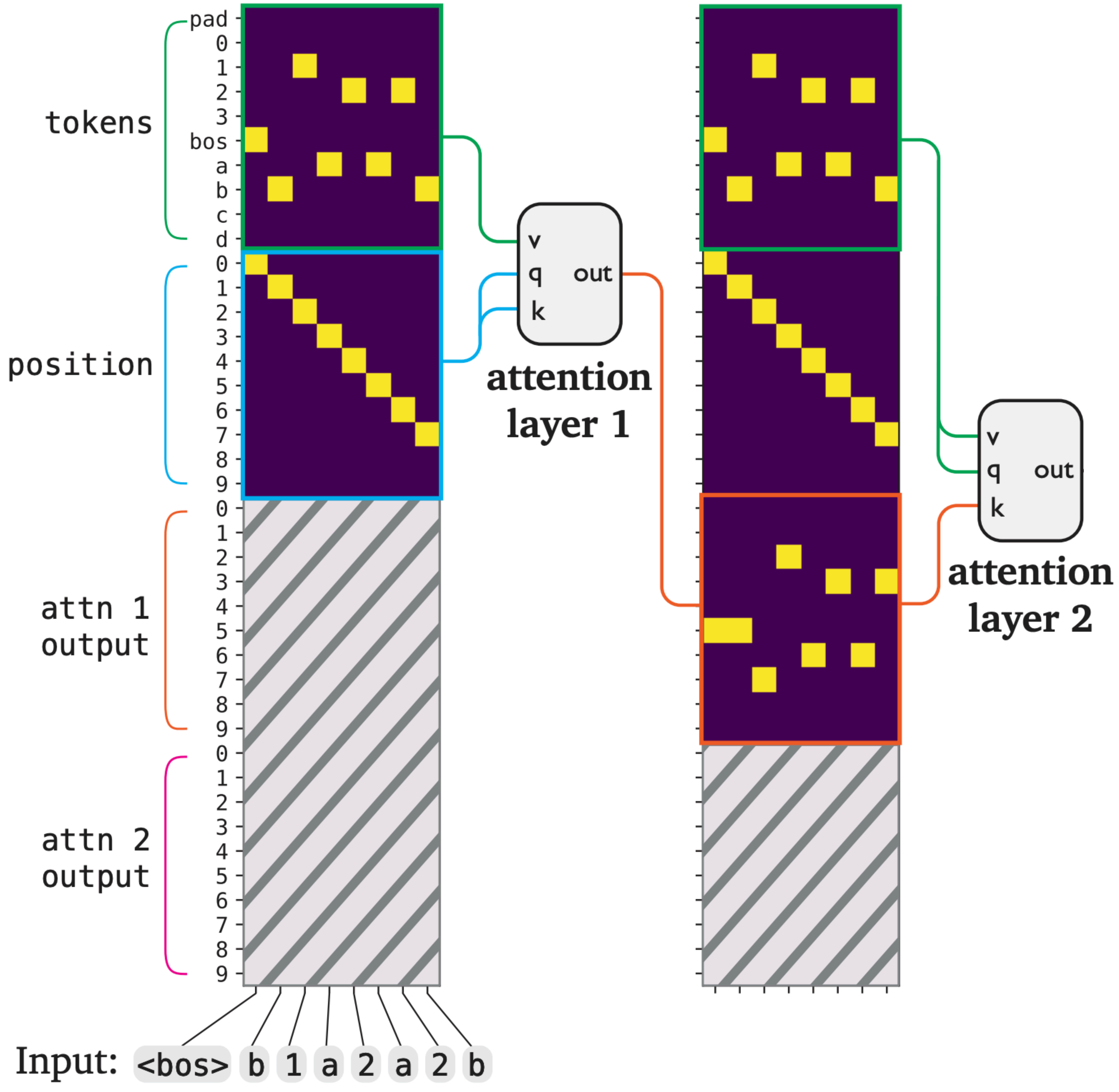
Constraint 2: Interpretable sublayers

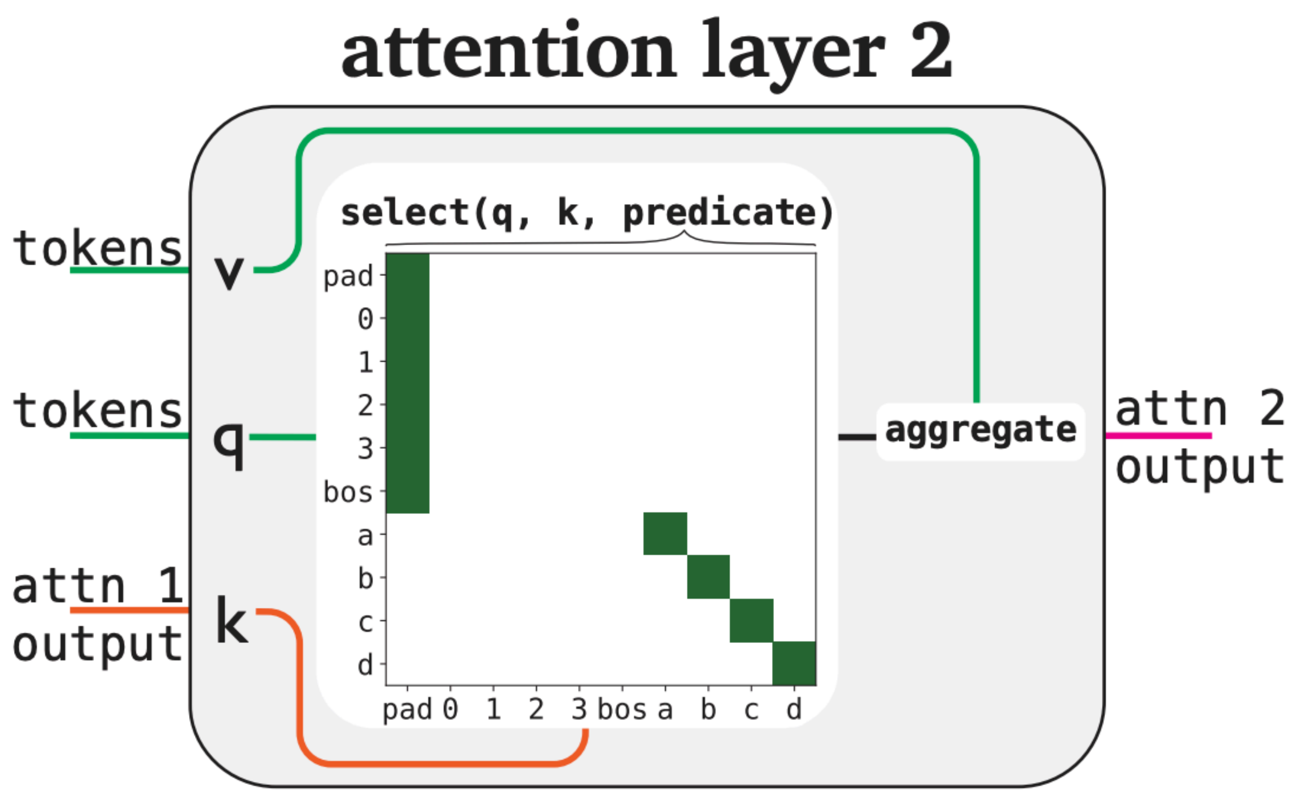
Input embeddings encode two categorical variables (*tokens* and *position*)

Each attention layer *reads* a fixed set of variables

… applies a *learned*, *rule-based* transformation

… and *writes* a new variable to a dedicated address






```
def predicate_2(token, attn_1_output):
    if token in {
        "<pad>", "0", "1", "2", "3", "<s>"
    }:
        return attn_1_output == "<pad>"
    if token in {"a"}:
        return attn_1_output == "a"
    if token in {"b"}:
        return attn_1_output == "b"
    if token in {"c"}:
        return attn_1_output == "c"
    if token in {"d"}:
        return attn_1_output == "d"
```

Summary: “induction head” mechanism

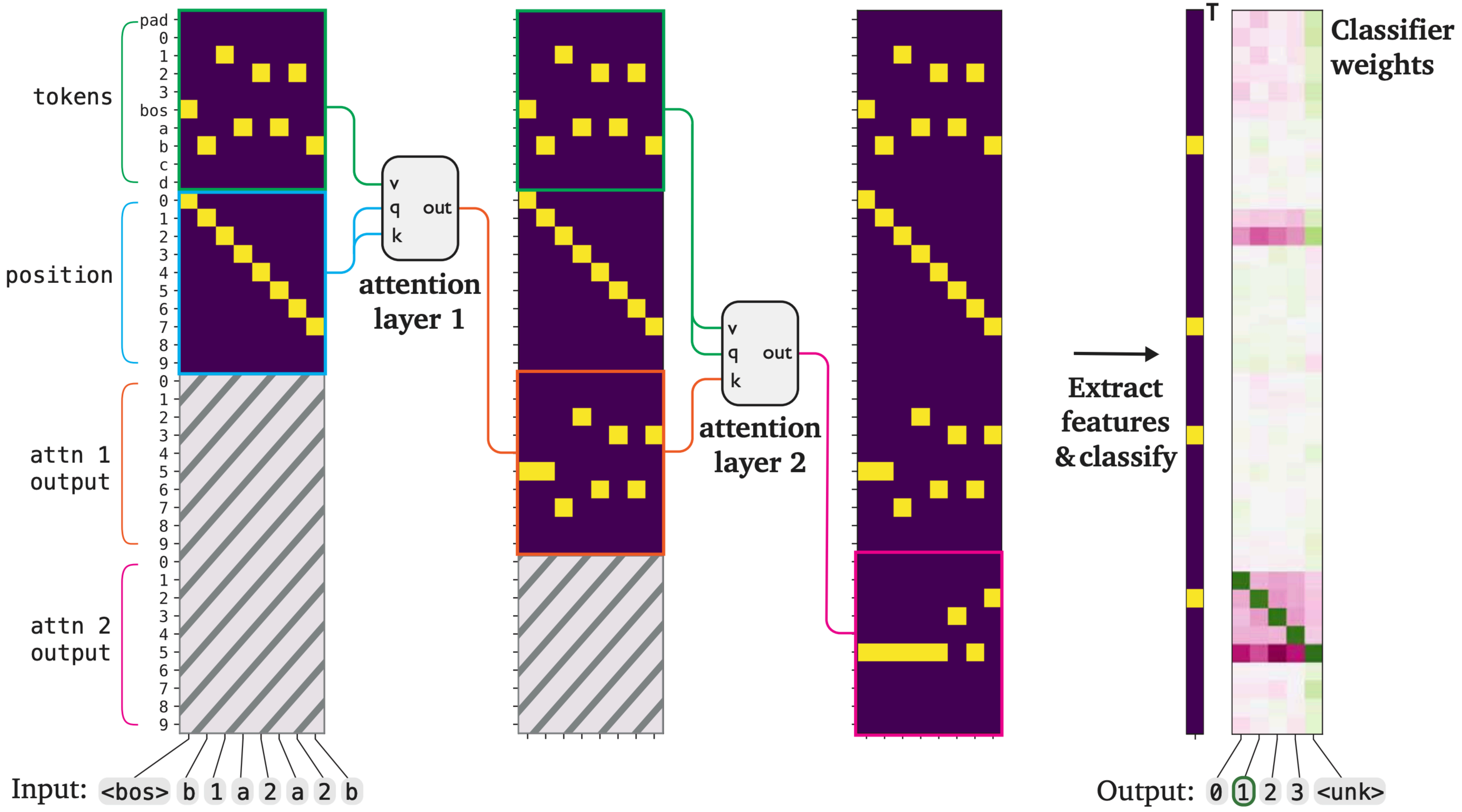
Linear classifier

Input embeddings encode two categorical variables (*tokens* and *position*)

Each attention layer *reads* a fixed set of variables…

… applies a *learned*, *rule-based* transformation

… and *writes* a new variable to a dedicated address

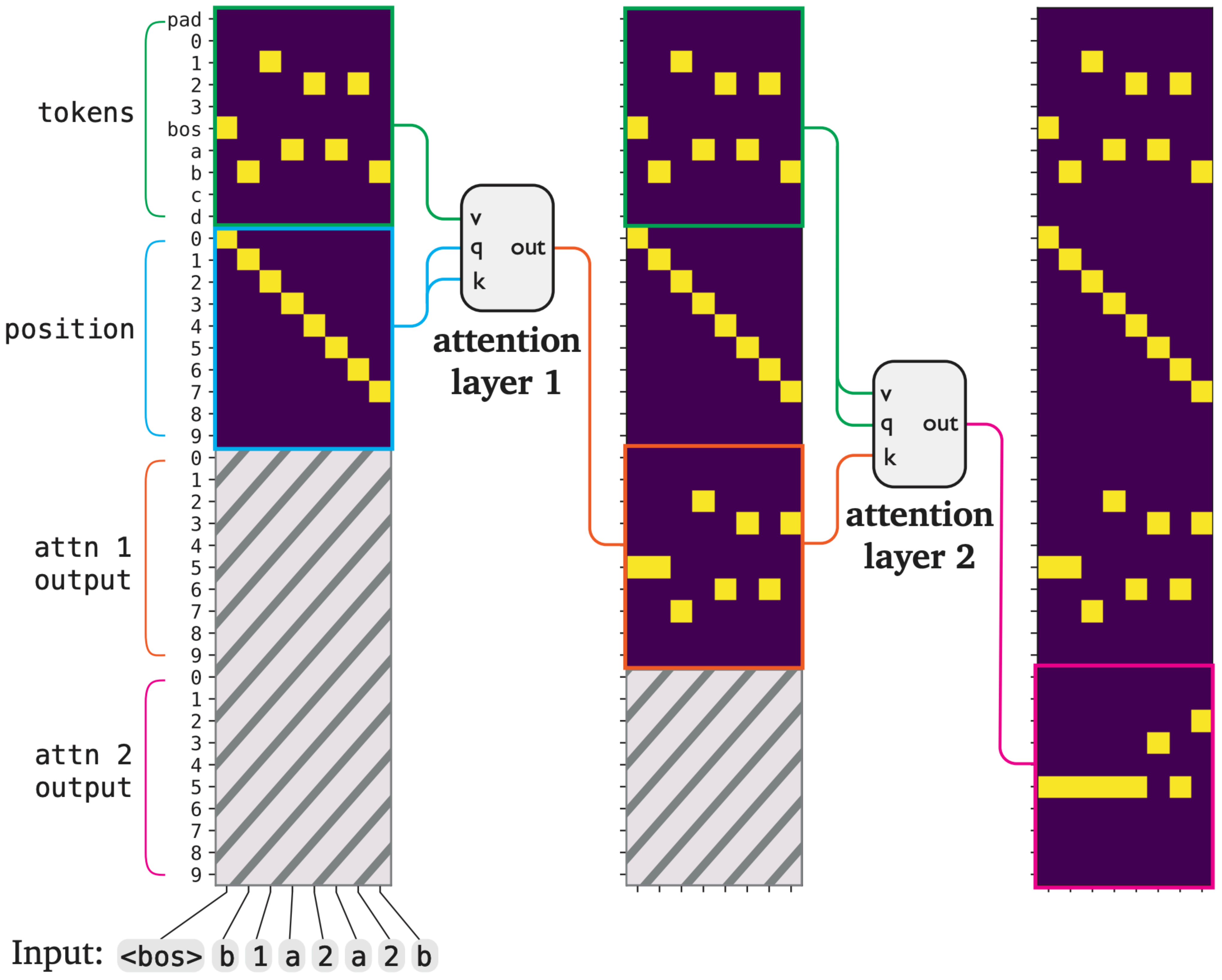


Discrete feature extractor

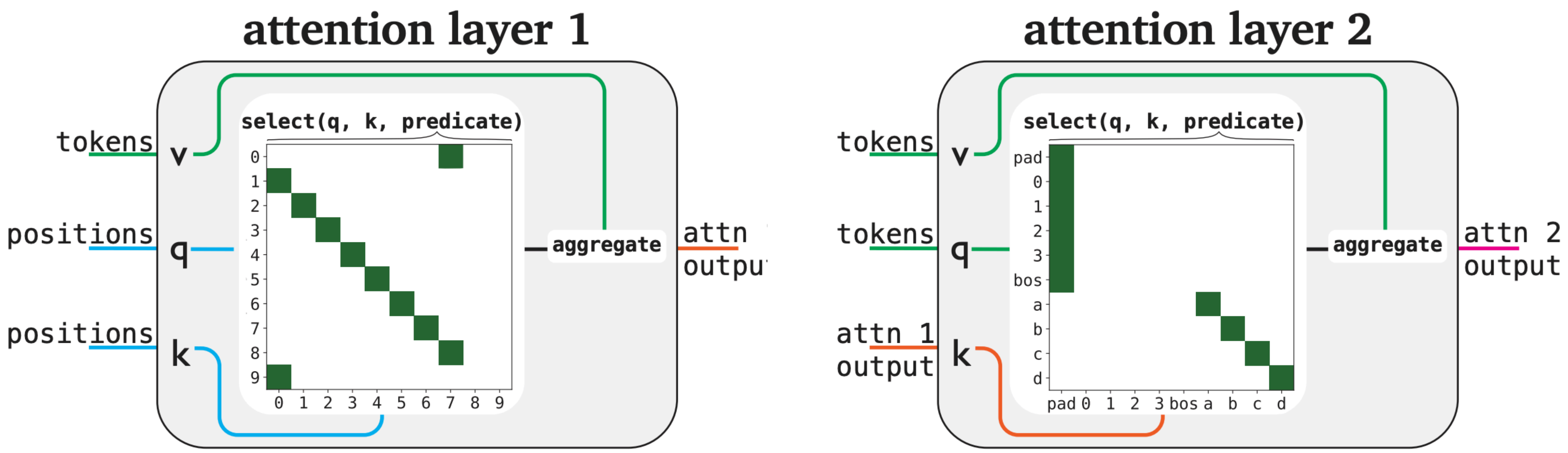
Continuous feature classifier

Method: Learning Transformer Programs

Constraint 1: Disentangled residual stream



Constraint 2: Interpretable sublayers



Extensions: Other program modules

- Attention with numerical values
- Feed-forward layers
- Categorical word embeddings

(See paper for details)

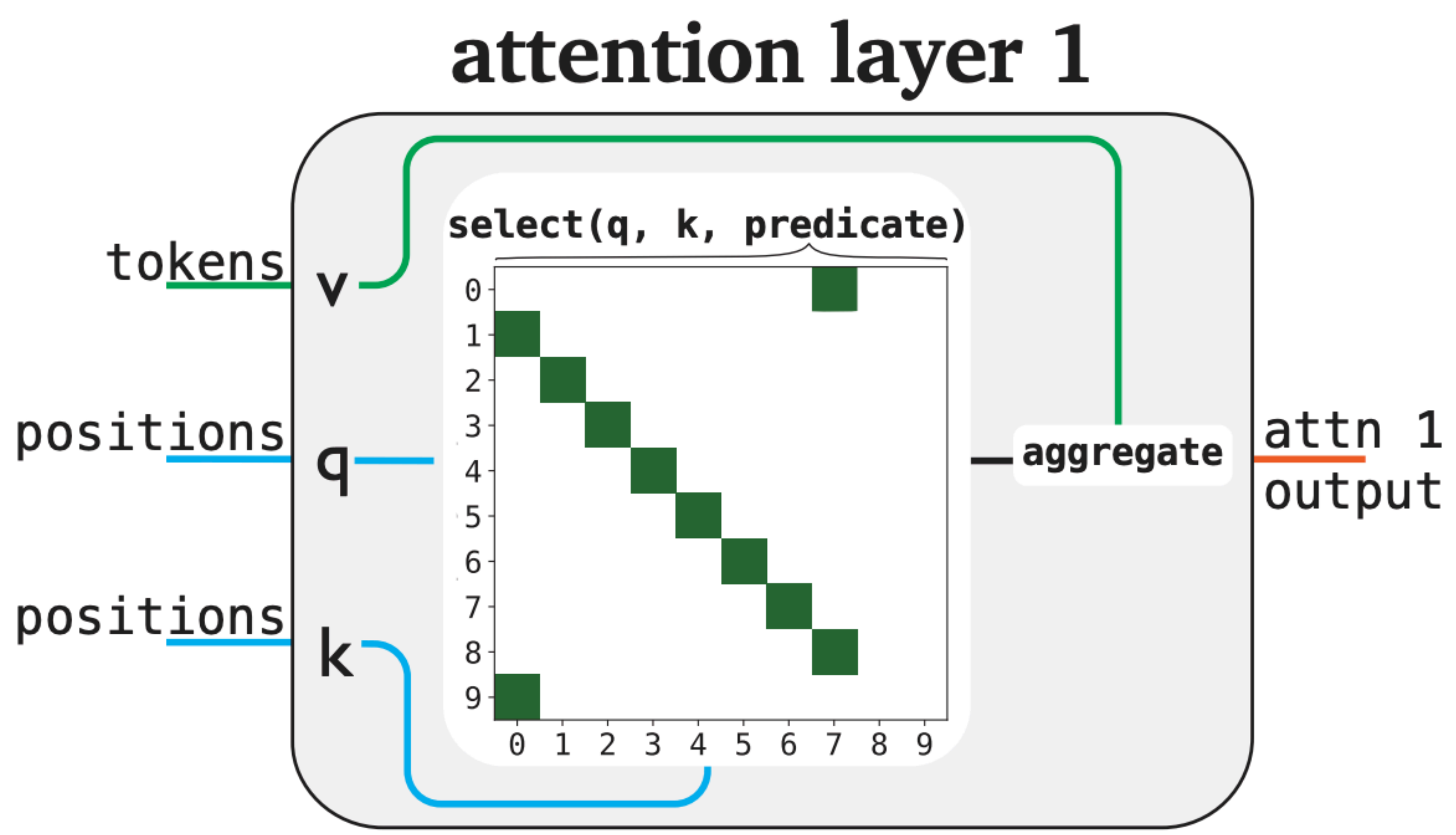
Optimization

Discrete weights

π_V

π_Q

π_K



$$\mathbf{W}_{\text{predicate}} \in \{0,1\}^{10\times 10}$$

Continuous parameters

$$\phi_V, \phi_Q, \phi_K \in \mathbb{R}^2$$

$$\pi_K \sim \text{One-hot}(\text{Categorical}(\phi_K))$$

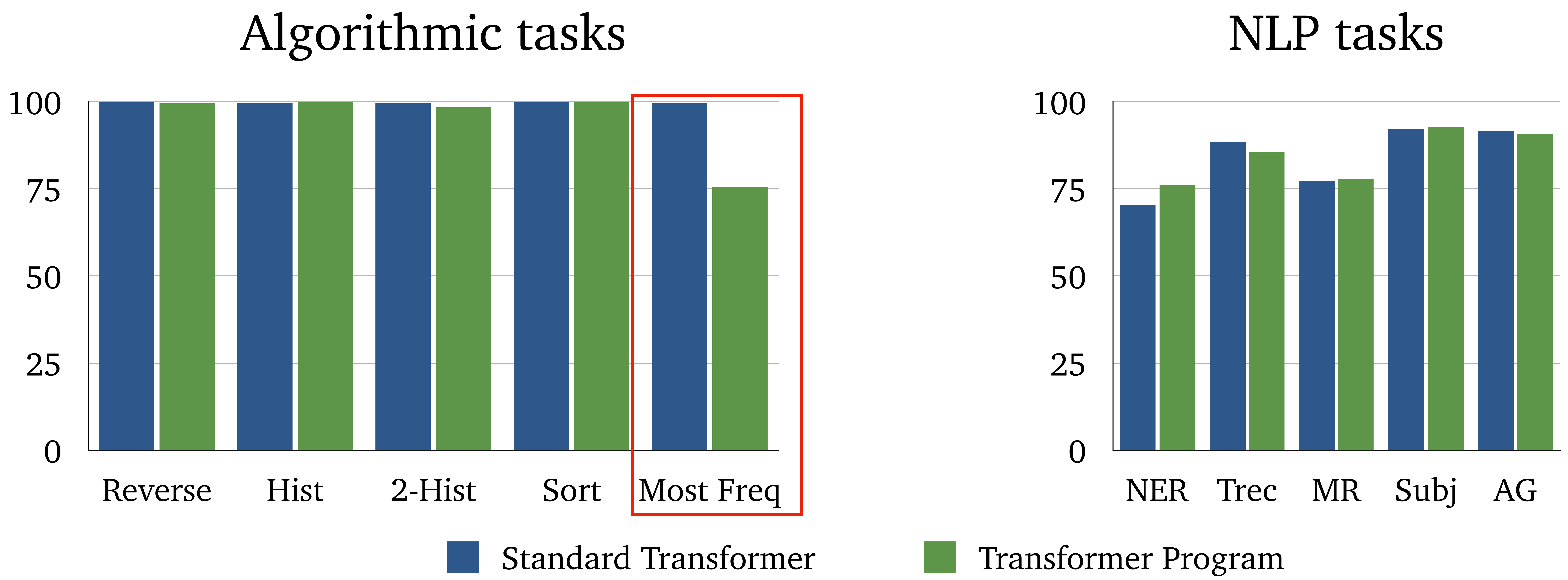
$$\psi_1, \ldots, \psi_{10} \in \mathbb{R}^{10}$$

$$\mathbf{W}_{\text{predicate},i} \sim \text{One-hot}(\text{Categorical}(\psi_i))$$

- Define a distribution over discrete program weights
- Optimize using Gumbel reparameterization

Jang et al., 2017. Categorical Reparameterization with Gumbel-Softmax.

Experiments: Can we learn effective programs?



- In the paper: More analysis of where Transformer Programs struggle

Are the programs interpretable?

- We can interpret the solutions by reading the code
- *Example*: Recognizing balanced parenthesis languages

```python
# First attention head: copy previous token.
def predicate_0_0(q_position, k_position):
    if q_position in {0, 13}:
        return k_position == 12
    elif q_position in {1}:
        return k_position == 0
    elif q_position in {2}:
        return k_position == 1
    elif q_position in {3}:
        return k_position == 2
    elif q_position in {4}:
        return k_position == 3
    elif q_position in {5}:
        return k_position == 4
    elif q_position in {6}:
        return k_position == 5
```

1. Copy the previous token

```python
# MLP: reads current token and previous token
# Outputs 13 if it sees "(}" or "{)".
def mlp_0_0(token, attn_0_0_output):
    key = (token, attn_0_0_output)
    if key in {(")", ")"),
               (")", "}"),
               ("{", ")"),
               ("}", ")"),
               ("}", "}")}:
        return 4
    elif key in {(")", "{"),
                 ("}", "(")}:
        return 13
```

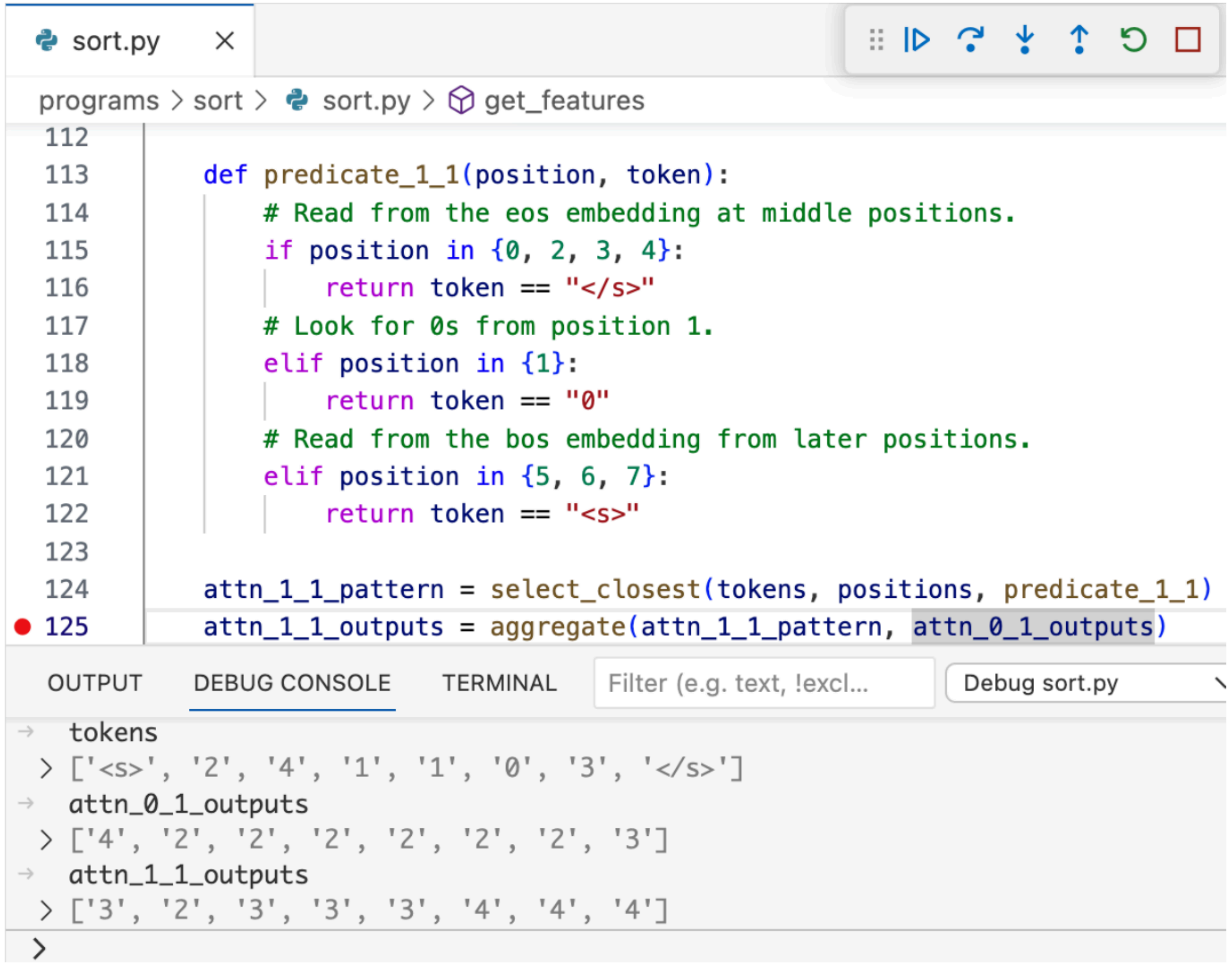
2. Check for invalid bigrams

```python
# 2nd layer attention: check for "(}" or "{)"
def predicate_1_2(position, mlp_0_0_output):
    if position in {0, 1, 2, 4, 5, 6, 7, 8, 9,
                    10, 11, 12, 13, 14, 15}:
        return mlp_0_0_output == 13
    elif position in {3}:
        return mlp_0_0_output == 4
```

3. Propagate the result to later positions

Are the programs interpretable?

- Computer code can still be difficult to understand…
- But we can use off-the-shelf tools for code analysis

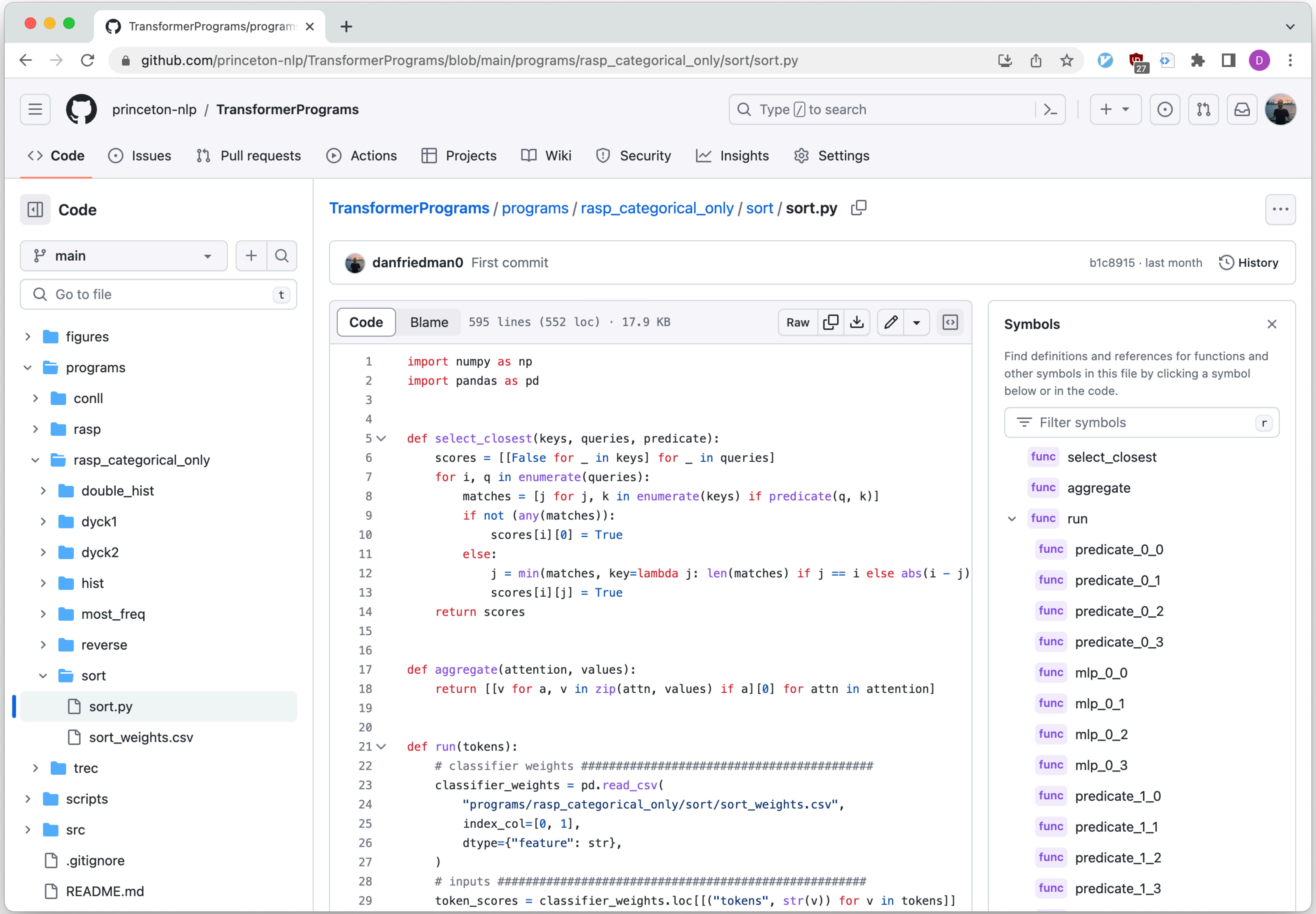


Leave comments

Set breakpoints

Inspect intermediate variables

Full programs are on GitHub:



github.com/princeton-nlp/TransformerPrograms

Summary

- Learn Transformers that are **mechanistically interpretable by design**
- This method **can learn non-trivial programs** (for small-scale tasks)
- The programs are **easy to interpret,** e.g. using standard code analysis tools
- Directions for future work
  - Addressing **discrete optimization challenges**
  - Introducing **more expressive** modules
  - Tools for automatic **program analysis**
- See our paper for more **examples**, **analysis**, and **discussion**

Paper: https://arxiv.org/abs/2306.01128
Code: https://github.com/princeton-nlp/TransformerPrograms
Contact: dfriedman@princeton.edu

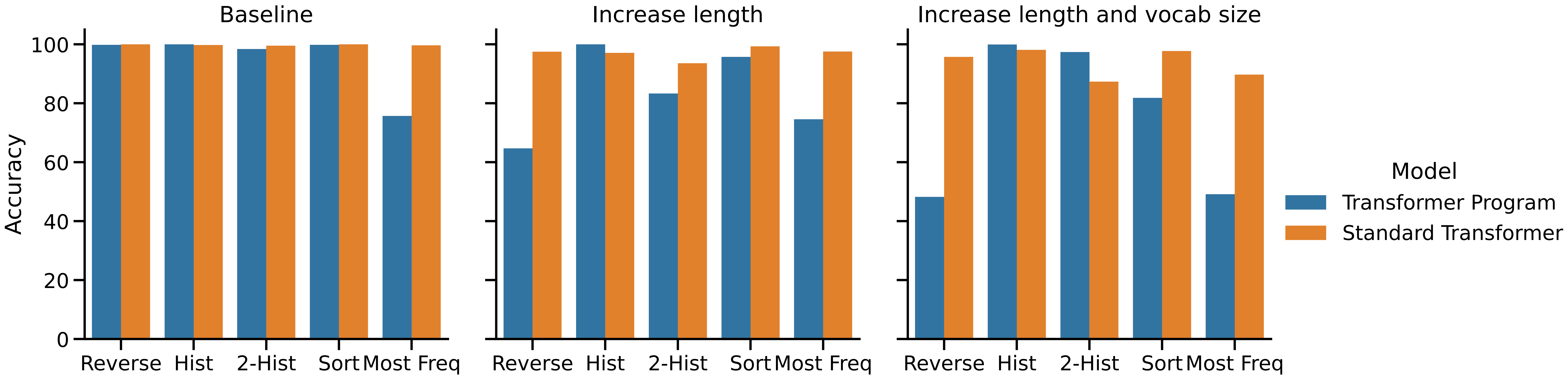
Link to the **paper**

Programs and **code**

Extra slides

Scaling Transformer programs

- What are the obstacles to scaling this approach?



- Future work needed for:
  - Better optimization methods
  - Making the programs more expressive

Optimization challenges: case study

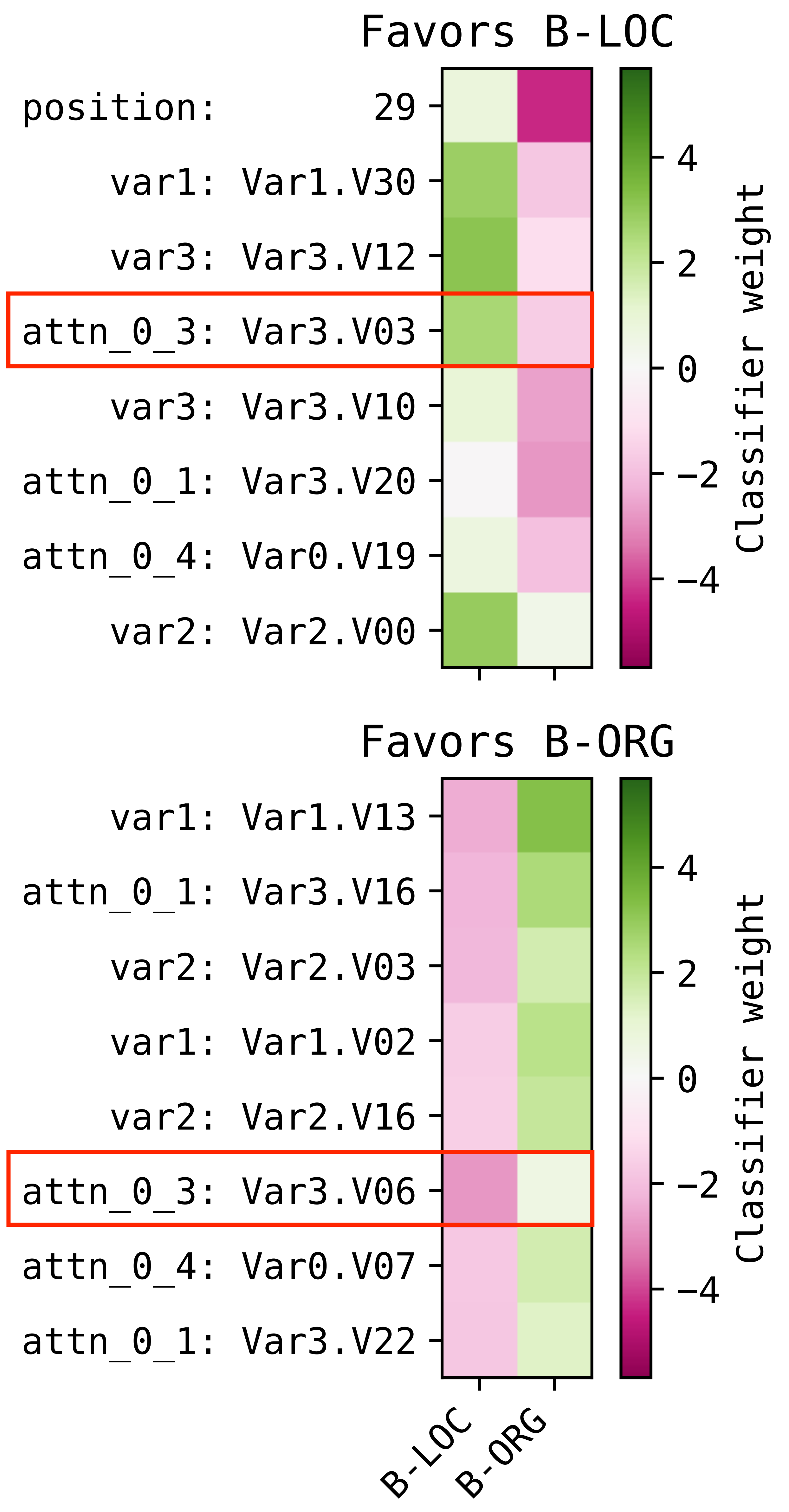
Attention 1		MLP 1	Attention 2		
Read	*Predicate*	*Read*	*Read*	*Predicate*	Accuracy
-	-	-	-	-	23.2/23.6/24.1
✔	✔	✔	✔	✔	99.9/99.9/80.8
✔	✔	✔	✔	-	37.9/40.3/18.5
✔	✔	✔	-	✔	17.1/13.7/20.2
✔	✔	-	✔	✔	95.1/94.1/95.3
✔	-	✔	✔	✔	99.1/83.9/78.2
-	✔	✔	✔	✔	35.5/44.1/41.8

Results on the *Reverse* task (vocab size = length = 16) after initializing the model to encode a generalizing solution (below). Each component is initialized either manually (✔) or randomly (-).

```
# First-layer attention
length = aggregate(select(tokens, tokens, lambda q, k: k == "</s>"), positions)

# First-layer MLP
targets = one_hot(length - positions)

# Second-layer attention
output = aggregate(select(targets, positions, ==), tokens)
```



(a) Feature weights.

```
# attn_0_3: Copy var3 from previous token
def predicate_0_3(q_position, k_position):
    if q_position in {2}:
        return k_position == 1
     if q_position in {3}:
        return k_position == 2
    if q_position in {4}:
        return k_position == 3
      if q_position in {5}:
        return k_position == 4
    if q_position in {6}:
        return k_position == 5
    # ...
attn_0_3_pattern = select_closest(positions, positions, predicate_0_3)
attn_0_3_outputs = aggregate(attn_0_3_pattern, var3_embeddings)
```

(b) Code for the attention features.

```
class Var3(Enum):
    V00 = ['German', 'television', 'Foreign', 'newspaper', ...]
    V01 = ['<unk>', 'Johnson', 'Morris', 'Service', ...]
    V02 = ['<s>', '</s>', 'Bank', 'York', 'Commission', ...]
    V03 = ['at', 'AT', 'In', 'Saturday', 'match', 'At', ...]
    V04 = ['/', 'up', 'no', 'newsroom', 'Attendance', ...]
    V05 = ['during', 'leader', 'quoted', 'manager', 'came', ...]
    V06 = ['Akram', 'TORONTO', 'BALTIMORE', 'BOSTON', ...]
    V07 = ['said', "'s", 'has', '@th', 'other', 'shares', ...]
    V08 = ['second', 'told', 'b', 'did', 'spokesman', ...]
    V09 = ['Australia', 'France', 'Spain', 'England', ...]
    V10 = ['Netherlands', 'Finland', 'countries', 'Kurdish', ...]
    # ...
```

(c) The most common words assigned to different values of the `Var3` embedding variable.

Double histogram

Description: For each token, the number of unique tokens with the same histogram value.

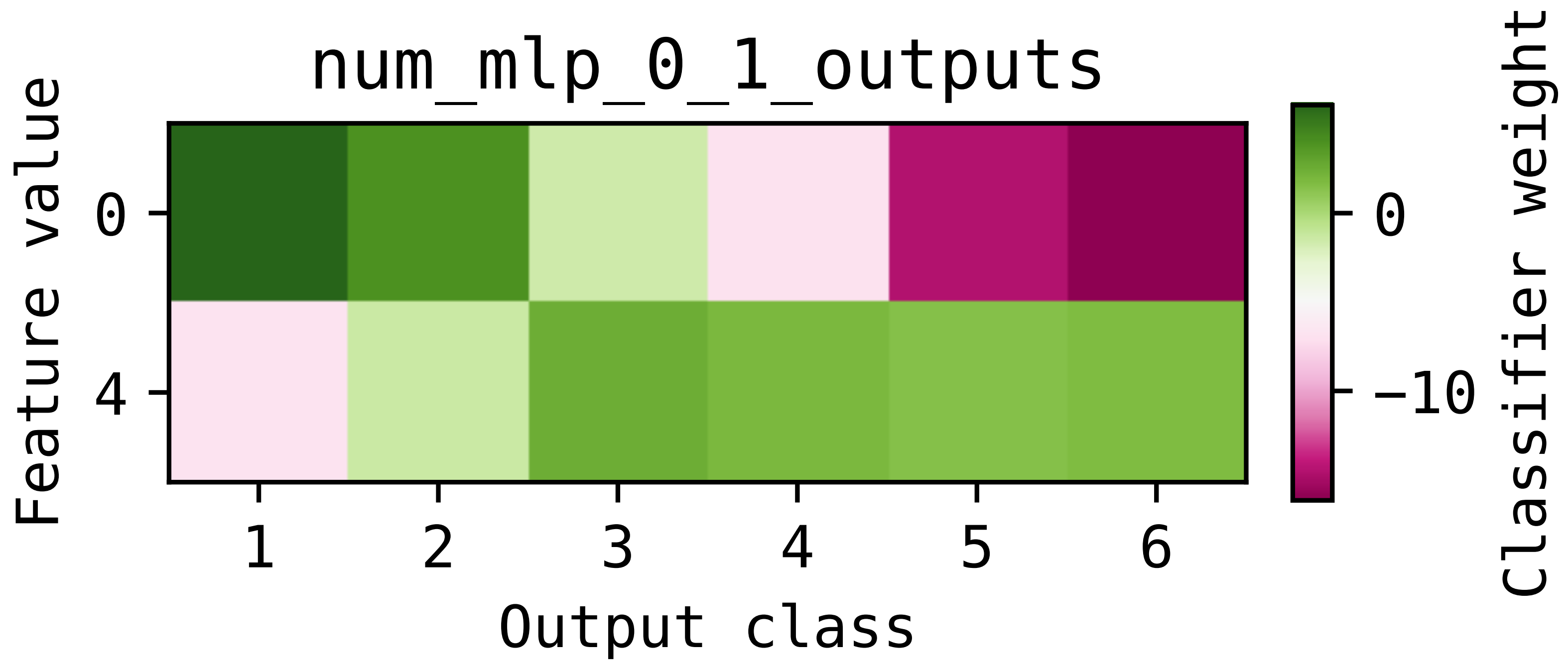
Example: hist2("abbc") = 2112

```
def num_predicate_0_1(q_token, k_token):
    if q_token in {"0"}:
        return k_token == "0"
    elif q_token in {"1"}:
        return k_token == "1"
    elif q_token in {"2"}:
        return k_token == "2"
    elif q_token in {"3"}:
        return k_token == "3"
    elif q_token in {"4"}:
        return k_token == "4"
    elif q_token in {"5"}:
        return k_token == "5"
    elif q_token in {"<s>"}:
        return k_token == "<pad>"

num_attn_0_1_pattern = select(
    tokens, tokens, num_predicate_0_1)
num_attn_0_1_outputs = aggregate_sum(
    num_attn_0_1_pattern, ones)
```

```
def num_mlp_0_1(num_attn_0_1_output):
    key = num_attn_0_1_output
    if key in {0, 1}:
        return 4
    return 0

num_mlp_0_1_outputs = [
    num_mlp_0_1(k0)
    for k0 in num_attn_0_1_outputs]
```



Dyck-2

Description: For each position i, is the input up until i a valid string in Dyck-2 (T); a valid prefix (P); or invalid (F).

Example: dyck2("()(}") = PTPF

```python
# First attention head: copy previous token.
def predicate_0_0(q_position, k_position):
    if q_position in {0, 13}:
        return k_position == 12
    elif q_position in {1}:
        return k_position == 0
    elif q_position in {2}:
        return k_position == 1
    elif q_position in {3}:
        return k_position == 2
    elif q_position in {4}:
        return k_position == 3
    elif q_position in {5}:
        return k_position == 4
    elif q_position in {6}:
        return k_position == 5
    elif q_position in {7}:
        return k_position == 6
    elif q_position in {8}:
        return k_position == 7
    elif q_position in {9}:
        return k_position == 8
    elif q_position in {10}:
        return k_position == 9
    elif q_position in {11}:
        return k_position == 10
    elif q_position in {12}:
        return k_position == 11
    elif q_position in {14}:
        return k_position == 13
    elif q_position in {15}:
        return k_position == 14
attn_0_0_pattern = select_closest(positions, positions,
                                  predicate_0_0)
attn_0_0_outputs = aggregate(attn_0_0_pattern, tokens)
```

```python
# MLP: reads current token and previous token
# Outputs 13 if it sees "(}" or "{)".
def mlp_0_0(token, attn_0_0_output):
    key = (token, attn_0_0_output)
    if key in {(")", ")"),
               (")", "}"),
               ("{", ")"),
               ("}", ")"),
               ("}", "}")}:
        return 4
    elif key in {(")", "{"),
                 ("}", "(")}:
        return 13
    elif key in {("(", ")"),
                 ("(", "}"),
                 (")", "("),
                 ("{", "}"),
                 ("}", "{")}:
        return 0
    return 7
mlp_0_0_outputs = [
    mlp_0_0(k0, k1) for k0, k1 in
    zip(tokens, attn_0_0_outputs)
]

# 2nd layer attention: check for "(}" or "{)"
def predicate_1_2(position, mlp_0_0_output):
    if position in {0, 1, 2, 4, 5, 6, 7, 8, 9,
                    10, 11, 12, 13, 14, 15}:
        return mlp_0_0_output == 13
    elif position in {3}:
        return mlp_0_0_output == 4
attn_1_2_pattern = select_closest(
    mlp_0_0_outputs, positions, predicate_1_2)
attn_1_2_outputs = aggregate(
    attn_1_2_pattern, mlp_0_0_outputs)
```