Mechanisms of Symbol Processing in Transformers

Paul Smolensky
Deep Learning Group
Microsoft Research
Redmond, WA 98052, USA
psmo@microsoft.com

Roland Fernandez
Deep Learning Group
Microsoft Research
Redmond, WA 98052, USA
rfernand@microsoft.com

Zhenghao Herbert Zhou
Linguistics Department
Yale University
New Haven CT 06511, USA
herbert.zhou@yale.edu

Mattia Opper
ILCC
University of Edinburgh
Edinburgh EH8 9AB, UK
M.Opper@ed.ac.uk

Adam Davies
Siebel School of Computing
UIUC
Urbana IL 61801, USA
adavies4@illinois.edu

Jianfeng Gao
Deep Learning Group
Microsoft Research
Redmond, WA 98052, USA
jfgao@microsoft.com

Abstract

We construct a 100% mechanistically-explainable transformer which perfectly performs an in-context learning task that requires inferring, and then reasoning over, latent syntactic structure. It implements a program in a symbolic, Turing-complete language in a family of leading models of the human cognitive architecture.

Note: This is a condensed version of Smolensky et al. [23]; a preprint of the unabridged paper is available at: https://arxiv.org/abs/2410.17498.

1 Introduction

1.1 Research goals

Most language tasks are heavily laden with semantics. In AI, traditional symbolic theories of semantic knowledge have been largely displaced by statistical inference on numerical vector representations induced from massive quantities of natural-language data. Neural networks, especially transformers, have shown extraordinary capacity for such statistical inference, and semantically-laden tasks have been the subject of extensive research in the AI community.

But natural language processing also demands *semantics-free* inference, most notably, in syntax, which has been best characterized in terms of purely formal, abstract relationships between words and phrases, captured by grammars using symbolic computations that are, by design, blind to the meanings of elements, sensitive only to the formal patterns of combination among these elements. As emphasized decades ago by cognitive scientists [10, 20], neural networks appear to be singularly unsuited for such computation, and should fail catastrophically on the ultimate challenge for abstract structure processing: generating complex, syntactically valid natural language. Yet neural language models with transformer architectures [26] dramatically out-perform symbolic-computation-based language processing models, generating rich, syntactically complex English, virtually flawlessly [3].

While transformer LMs generate essentially error-free syntax, they and other neural models do not seem to have an equal mastery of compositionality and systematicity [7]. Perhaps these networks have implemented symbolic computational capabilities, but if so, these abilities are clearly limited. ***Our goal*** *in this work is to understand the mechanisms within transformers that provide them the symbol processing abilities they do possess, and to see how these mechanisms can be strengthened to overcome the limitations in symbol processing that transformer models still have.*

We develop the ***Transformer Production Framework, TPF***, by studying a purely formal, semantics-free In-Context Learning (ICL) task that we introduce, a task which requires inferring and then

39th Conference on Neural Information Processing Systems (NeurIPS 2025) Workshop: CogInterp.

reasoning over latent syntactic structure. The result — a novel type of attention head making structured use of the residual stream and exploiting certain types of recurrence — enables powerful symbolic computation. Future work merges these new features with the traditional transformer architecture to produce a new type of transformer that preserves the current power for semantically-laden inference while infusing enhanced power for abstract, meaning-free symbol processing.

1.2 Contributions

Seen through Marr's Levels [15] for computational-system description,[1] our contributions introduce:

- *Computational/Functional Level*: a formally-defined, purely semantics-free task which transformer LLMs can perform to a significant degree — the ***Templatic Generation Task, TGT***.[2]
- *Algorithmic Level (higher):* a symbolic programming language in the family of Production Systems, a mainstay of symbolic architectures for modeling human cognition — the ***Production System Language, PSL***; we then provide a PSL program that solves the TGT.
- *Algorithmic Level (lower):* a symbolic abstraction of the transformer — the ***QKV Machine***.
- *Implementation Level:* a variant of the transformer using purely numerical computation in which the residual stream, and attention, are discretized — the ***Discrete-Attention-Only Transformer, DAT*** — which is 100% mechanically explicable. DAT attention heads perform precise formal inference, and in future work they can be combined with standard heads to work jointly with informal statistical inference.
- *Tools:* ***compilers*** for translating programs in PSL to those for the QKV Machine, and those in turn to weights for the DAT; ***visualization*** of DAT computation, displaying symbolically-interpreted activation and attention vectors[3]

The high-level result of this work is a style of symbolic computation that transformers are well-suited to implementing. This style is powerful — PSL is Turing complete, applicable to many tasks.

2 The Transformer Production Framework

2.1 Computational/Functional Level: the Templatic Generation Task (TGT)

The purely formal, semantics-free ICL processing we target is illustrated by (1).

(1) Passive→Logical

a. *Prompt:* $\mathcal{Q}$ the program was translated by a compiler $\mathcal{A}$ translated (a compiler , the program)
$\mathcal{Q}$ my big dog was chased by a small black cat $\mathcal{A}$

b. *Continuation:* chased (a small black cat , my big dog)

c. *Template:* $\mathcal{Q}$ x was V by y $\mathcal{A}$ V (y , x)

We recognize this as a linguistically-motivated transformation from Passive Voice to Logical Form, but doing the task requires no knowledge of these concepts, or the meanings of the words: just infer the latent formal template in (1c) as the generator of the Question-Answer example in the prompt (1a); recognize that the prompt cues a continuation in which x = 'a small black cat', V = 'chased', and y = 'my big dog'; then generate by inserting these values into the corresponding slots of the Answer portion of the template. Pure meaningless symbol manipulation.

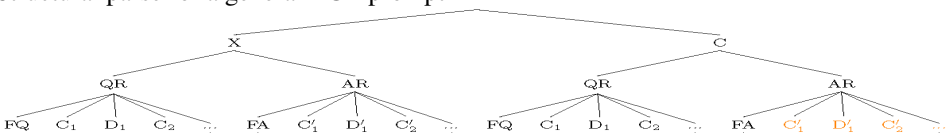
Prompts like (1) motivate the task we devise: in TGT, each prompt is generated by randomly inserting meaningless symbols into the slots in a randomly-generated parse structure (template) of the general form shown in (2); (3) shows the particular case of (1), which for expository purposes uses English rather than random symbols — crucially, our experimental dataset uses random symbols rather than words to preclude any reliance on semantics; see Appendix A.)

[1] See [12, 5] for background on Marr's levels in the context of deep learning models.

[2] Publicly available dataset at `https://huggingface.co/datasets/microsoft/templatic_generation_tasks`.

[3] Tools are a publicly-accessible python package at `https://github.com/microsoft/discrete_attn_transformer/`

(2) Structural parse for a general TGT prompt



The constituent X is the example, containing a question region QR and an answer region AR; constituent C is the cued-continuation: QR is given in the prompt, while AR needs to be generated. The function computed by our system takes as input X + the QR portion of C, and produces as output the AR portion of C. The variable slots C'_j in AR are a rearrangement of those in QR, C_i (e.g., $C'_1 = C_2$).

(3) Parse for (1)

a. $[_{\mathrm{X}}\ [_{\mathrm{QR}}\ [_{\mathrm{FQ}}\ \mathrm{Q}]\ [_{\mathrm{C}_1}\text{the program}]\ [_{\mathrm{D}_1}\text{was}]\ [_{\mathrm{C}_2}\text{translated}]\ [_{\mathrm{D}_2}\text{by}]\ [_{\mathrm{C}_3}\text{a compiler}]\ ]$
$[_{\mathrm{AR}}\ [_{\mathrm{FA}}\ \mathrm{A}]\ [_{\mathrm{C}_2}\text{translated}]\ [_{\mathrm{D}'_1}(\ ]\ [_{\mathrm{C}_3}\text{a compiler}]\ [_{\mathrm{D}'_2},]\ [_{\mathrm{C}_1}\text{the program}]\ [_{\mathrm{D}'_3})]\ ]\ ]$
$[_{\mathrm{C}}\ [_{\mathrm{QR}}\ [_{\mathrm{FQ}}\ \mathrm{Q}]\ [_{\mathrm{C}_1}\text{my big dog}]\ [_{\mathrm{D}_1}\text{was}]\ [_{\mathrm{C}_2}\text{chased}]\ [_{\mathrm{D}_2}\text{by}]\ [_{\mathrm{C}_3}\text{a small black cat}]\ ]$

b. $[_{\mathrm{AR}}\ [_{\mathrm{FA}}\ \mathrm{A}]\ [_{\mathrm{C}_2}\text{chased}]\ [_{\mathrm{D}'_1}(\ ]\ [_{\mathrm{C}_3}\text{a small black cat}]\ [_{\mathrm{D}'_2},]\ [_{\mathrm{C}_1}\text{my big dog}]\ [_{\mathrm{D}'_3})]\ ]\ ]$

c. $[_{\mathrm{QR}}\ [_{\mathrm{FQ}}\ \mathrm{Q}]\ [_{\mathrm{C}_1}x]\ [_{\mathrm{D}_1}\text{was}]\ [_{\mathrm{C}_2}V]\ [_{\mathrm{D}_2}\text{by}]\ [_{\mathrm{C}_3}y]\ [_{\mathrm{AR}}\ [_{\mathrm{FA}}\ \mathrm{A}]\ [_{\mathrm{C}_2}V]\ [_{\mathrm{D}'_1}(\ ]\ [_{\mathrm{C}_3}y]\ [_{\mathrm{D}'_2},]\ [_{\mathrm{C}_1}x]\ [_{\mathrm{D}'_3})]\ ]$

The lowest-level constituents in the parse tree (2) are *fields*: these dominate variable-length symbol strings, and we regard them as variables whose values are the strings they dominate. The fields D, D′ are *delimiters*, which retain their values throughout the prompt (including the continuation). The fields C are *constituents*, which have a fixed value within each QR, but distinct values in X and C.

Table 1 shows the TGT performance of LLMs (best over varying 'thought' levels). We also train models to perform TGT from scratch (see Appendix B), finding that Transformers outperform non-transformer LMs, suggesting that the transformer architecture provides some leverage on this type of formal symbol manipulation. Our hand-programmed transformer DAT (§2.4) performs the task perfectly.

Table 1: LLMs on TGT (no fine tuning, tested Aug. 2025)

GPT-5	GPT-4o	GPT-4	Opus 4.1	Gemini 2.5 pro	Grok 4	DeepSeek R1
0.25	0.54	0.71	0.18	0.72	0.51	0.31

2.2 Algorithmic Level (higher): the Production System Language (PSL)

The higher-level symbolic machine we introduce, the Production-System Machine (PSM), has a sequence of representations for each token in the input, like a transformer: these are layers formed by unrolling in time the successive states of the machine. These representations are symbolic *state-variable/value* structures, so that in the representations above the first symbol $\mathcal{Q}$ in (1a), the state variable symbol (s) has value Q, the variable position (p) has value 1: these are given as the input at the lowest layer of model. Higher up in this sequence of representations, the *structural state variables* are assigned values through processing; these define the parse, and include the region variable (r) which has value QR and the field variable (f) with value FQ: see (3).

Generating the text requires producing the sequence of fields that define the Answer region, inserting as values for these fields the symbols provided in the Q-region of the continuation-cue C. This requires consulting the field values in AR within X, then consulting the symbolic fillers of these fields in C. This is done with *productions* — Condition/Action pairs — like those in (4).

(4) NextField1 *Condition:* n, N satisfy $f^*[n] == f[N]$, $r^*[n] ==$ XA, $d[n] == 0$
Action: set $\hat{f}[N] := f[n]$
NextField2 *Condition:* n, N satisfy $f[n] == \hat{f}[N]$, $r[n] ==$ CQ, $d[n] == 0$
Action: set $\hat{s}[N] := s[n]$

A given layer (processing step) is controlled by a particular production: whenever a pair of positions n, N meet the requirement of Condition then the corresponding Action is effected — this writes values into the variables at position N, using information retrieved from position n. (In detail, $f^*[n]$ denotes the field of the position preceding n, so the NextField1 Condition in (4) requires finding a position n that immediately follows the field of the most-recently generated symbol at N, $f[N]$; the

field $f[n]$ is the next field that must be generated — it is assigned to $\hat{f}[N]$. NextField2 then takes this field value and finds the position (new) n that has the desired field but is in the Continuation-Cue, the region $r[n] =$ CQ; this new n provides the symbolic filler material to generate the new field, assigned as the value of $\hat{s}[N]$. The variable d has value 0 at field-initial symbols.)

Values of variables are propagated up through layers, and during generation, from the final layer for the just-generated symbol to the initial layer for the next position. A PSL program is a sequence of productions, each executed at a layer of the machine, in parallel across all prompt positions.

For a given N, a production may only select a single n (the production specifies left- or right-most selection). Generally, this is called 'conflict resolution' in Production Systems, since multiple symbolic operations cannot in general be performed simultaneously. A production may, or may not, specify causal interaction ($n \leq N$). A subsequence of consecutive productions may be specified as forming a repetition block; these are repeatedly executed in sequence until state-variables stabilize.

2.3 Algorithmic Level (lower): the QKV Machine

Bridging PSL to the neural network transformer at the implementation level is the QKV Machine, a symbolic abstraction of the transformer. Updates are not done with productions that search for n, N pairs that meet a condition, but rather, by each position N issuing a *query* which is matched to *keys* at other positions n, a perfect match resulting in propagating a *value* from n to N. A production is compiled into queries $q[N]$ and keys $k[n]$ that match when the production's Condition is satisfied, and the value $v[n]$ is then used to perform the production's Action, updating variables at N. Position N's query is a symbol structure specifying target values for variables; position n's key is a symbol structure whose values may match those of a query; a value is a symbol structure specifying the new values to be assigned to variables at N.

2.4 Implementation Level: the Discrete-Attention-only Transformer (DAT)

Finally, the QKV program compiled from our PSL program for computing the TGT function is compiled into a numerical transformer. The current implementation uses the simplest method for vector-embedding, through a series of registers, the symbolic variable/value structures of the QKV queries, keys, values and residual stream, shown in (5).

(5) One-hot version of vector representation of state-variable structures in the DAT

○ ··· ○	○ ··· ○	○ ··· ○	○ ··· ○	○ ··· ○	○ ··· ○	○ ··· ○	○ ··· ○	○ ··· ○	○ ··· ○	○ ··· ○	○ ··· ○	○ ··· ○	○ ··· ○	○ ··· ○
position	symbol	region	field	type	index	prev_pos	prev_sym	prev_reg	prev_fld	prev_type	prev_ind	position`	symbol`	···
p	s	r	f	t	d	p^*	s^*	r^*	f^*	t^*	d^*	$p^{\backprime}$	$s^{\backprime}$	···

This is a fully local Tensor Product Representation [22], with a dedicated unit for each variable/value binding, but it is possible to choose more general, non-one-hot, TPR embeddings; these are what we can look for in trained transformers doing TGT to see if any of the techniques developed here help with mechanical analysis of trained models (as in [16, 17, 24]; see [23, Sec. 9.2]). Distributed encodings within each register enable scaling up to larger vocabularies of values; we need to detect perfect matches of values specified in queries and keys, but this does not demand one-hot, or even orthogonal, distributed encodings of symbols.[4]

Compiling PSL conditions like $f^*[n] == f[N]$ in (4) yields weight matrices with embedded identity matrices that project a value in one register (f) to the same value in another register (f^*). These identity matrices mean that the compiled productions treat all symbols equally, providing perfect generalization to novel lexical items, as demanded by the pure symbol processing of TGT.

Whereas LLMs and trained-from-scratch models struggle with TGT (Table 1 and Appendix B Table 2), our DAT performs it perfectly. More crucially, we understand exactly how every neuron and every connection contributes to this correct performance.

[4]For example, even within a 128-d space, normalized 10-hot vectors provide 227 trillion distinct representations with a maximum dot-product of 0.9 between non-identical vectors. Exact matching simply requires rejecting dot-products ≤ 0.9.

3 Discussion

Related work As mentioned in Section 1, a primary novelty of the present work is the study of semantics-free ICL, providing a complement to the large literature on semantics-laden ICL tasks.[5] Where prior work has aimed to interpret the internal mechanisms that trained transformer LLMs leverage to perform ICL, such as induction heads [8, 18, 29] and function vectors [13, 25, 30], we instead demonstrate how transformers can solve a purely symbolic ICL task in a way that is fully interpretable and generalizes perfectly. The substantial performance gap between our DAT models and current LLMs/trained models indicates two key directions for future work: (1) mechanistically interpreting the limitations of these models relative to DAT, and (2) designing improved architectures that better capture the symbol-processing capabilities of DAT. Pursing (2) yielded a learnable DAT-inspired attention head that greatly strengthens precisely-targeted, long-distance attention [19].

The repetition-block and non-causal-interaction constructs of PSL compile into a DAT with activation-looping over a block of layers [1, 6, 9, 11], and non-causal attention for prompt parsing [27]. QKV's perfect query-key match requirements compile into a novel form of hard attention [4].

Our symbol-processing-task focus leads PSL to differ in a number of ways from the RASP [28] language for programming transformers to do quite different sequence-processing tasks, although we observe the close parallel between the register structure of DAT's residual stream and the 'disentangled residual stream' of the Tracr [14] implementation of RASP.

Generality Although PSL was developed through study of the TGT, it is actually a Turing-complete language — a 5-layer DAT can simulate a Universal Turing Machine (see Appendix C) — so even the current first-generation version of PSL is potentially applicable to a wide variety of symbol-processing tasks.

As the immediate focus of the present work is abstract symbol processing, TGT is an appropriate target task. General cognitive tasks call for unifying the power of standard transformer heads for semantic/statistical inference with the power of DAT-like heads for syntactic/formal inference. An architecture in which both types of heads collaborate within every transformer cell is proposed in [23, Sec. 9.4].

References

[1] Bae, S., Kim, Y., Bayat, R., Kim, S., Ha, J., Schuster, T., Fisch, A., Harutyunyan, H., Ji, Z., Courville, A., et al. (2025). Mixture-of-recursions: Learning dynamic recursive depths for adaptive token-level computation. *arXiv preprint arXiv:2507.10524*.

[2] Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., et al. (2020). Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*.

[3] Chang, T. A. and Bergen, B. K. (2024). Language model behavior: A comprehensive survey. *Computational Linguistics*, 50(1):293–350.

[4] Csordás, R., Irie, K., and Schmidhuber, J. (2021). The neural data router: Adaptive control flow in transformers improves systematic generalization. *arXiv preprint arXiv:2110.07732*.

[5] Davies, A. and Khakzar, A. (2024). The cognitive revolution in interpretability: From explaining behavior to interpreting representations and algorithms. *arXiv preprint arXiv:2408.05859*.

[6] Dehghani, M., Gouws, S., Vinyals, O., Uszkoreit, J., and Kaiser, Ł. (2018). Universal transformers. *arXiv preprint arXiv:1807.03819*.

[7] Dziri, N., Lu, X., Sclar, M., Li, X. L., Jiang, L., Lin, B. Y., Welleck, S., West, P., Bhagavatula, C., Le Bras, R., et al. (2023). Faith and fate: Limits of transformers on compositionality. *Advances in Neural Information Processing Systems*, 36:70293–70332.

[8] Elhage, N., Nanda, N., Olsson, C., Henighan, T., Joseph, N., Mann, B., Askell, A., Bai, Y., Chen, A., Conerly, T., et al. (2021). A mathematical framework for transformer circuits. *Transformer Circuits Thread*, 1(1):12.

[5] Several successful examples of syntactic error-repair do however appear in the seminal ICL paper [2, p. 30].

[9] Fan, Y., Du, Y., Ramchandran, K., and Lee, K. (2024). Looped transformers for length generalization. *arXiv preprint arXiv:2409.15647*.

[10] Fodor, J. A. and Pylyshyn, Z. W. (1988). Connectionism and cognitive architecture: A critical analysis. *Cognition*, 28(1-2):3–71.

[11] Giannou, A., Rajput, S., Sohn, J.-y., Lee, K., Lee, J. D., and Papailiopoulos, D. (2023). Looped transformers as programmable computers. In *International Conference on Machine Learning*, pages 11398–11442. PMLR.

[12] Hamrick, J. and Mohamed, S. (2020). Levels of analysis for machine learning. *arXiv preprint arXiv:2004.05107*.

[13] Hendel, R., Geva, M., and Globerson, A. (2023). In-context learning creates task vectors. In Bouamor, H., Pino, J., and Bali, K., editors, *Findings of the Association for Computational Linguistics: EMNLP 2023*, pages 9318–9333, Singapore. Association for Computational Linguistics.

[14] Lindner, D., Kramár, J., Farquhar, S., Rahtz, M., McGrath, T., and Mikulik, V. (2023). Tracr: Compiled transformers as a laboratory for interpretability. In Oh, A., Naumann, T., Globerson, A., Saenko, K., Hardt, M., and Levine, S., editors, *Advances in Neural Information Processing Systems*, volume 36, pages 37876–37899. Curran Associates, Inc.

[15] Marr, D. (1982). *Vision: A Computational Investigation into the Human Representation and Processing of Visual Information*. W. H. Freeman, San Francisco.

[16] McCoy, R. T. (2022). *Implicit compositional structure in the vector representations of artificial neural networks*. PhD thesis, Johns Hopkins University. `https://jscholarship.library.jhu.edu/items/7b4b8761-db3c-42a5-9e7f-3cb094eaa8ad`.

[17] McCoy, R. T., Linzen, T., Dunbar, E., and Smolensky, P. (2019). RNNs implicitly implement tensor product representations. In *International Conference on Learning Representations*. *arXiv preprint arXiv:1812.08718*.

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

[19] Opper, M., Fernandez, R., Smolensky, P., and Gao, J. (2025). TRA: Better length generalisation with threshold relative attention. *Transactions on Machine Learning Research*. arXiv:2104.10350.

[20] Pinker, S. and Prince, A. (1988). On language and connectionism: Analysis of a parallel distributed processing model of language acquisition. *Cognition*, 28(1-2):73–193.

[21] Schuurmans, D., Dai, H., and Zanini, F. (2024). Autoregressive large language models are computationally universal. *arXiv preprint arXiv:2410.03170*.

[22] Smolensky, P. (1987). Analysis of distributed representation of constituent structure in connectionist systems. In *Proceedings of the 1987 International Conference on Neural Information Processing Systems*, pages 730–739.

[23] Smolensky, P., Fernandez, R., Zhou, Z. H., Opper, M., and Gao, J. (in press). Mechanisms of symbol processing for in-context learning in transformer networks. *Journal of Artificial Intelligence Research*. arXiv:2410.17498.

[24] Soulos, P., McCoy, R. T., Linzen, T., and Smolensky, P. (2020). Discovering the compositional structure of vector representations with role learning networks. In *Third BlackboxNLP Workshop on Analyzing and Interpreting Neural Networks for NLP*, pages 238–254.

[25] Todd, E., Li, M., Sharma, A. S., Mueller, A., Wallace, B. C., and Bau, D. (2024). Function vectors in large language models. In *The Twelfth International Conference on Learning Representations*.

[26] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. (2017). Attention is all you need. In *Advances in Neural Information Processing Systems*, pages 5998–6008.

[27] Wang, T., Roberts, A., Hesslow, D., Scao, T. L., Chung, H. W., Beltagy, I., Launay, J., and Raffel, C. (2022). What language model architecture and pretraining objective work best for zero-shot generalization? *arXiv preprint arXiv:2204.05832*.

[28] Weiss, G., Goldberg, Y., and Yahav, E. (2021). Thinking like transformers. *arXiv preprint arXiv:2106.06981*.

[29] Yang, Y., Campbell, D. I., Huang, K., Wang, M., Cohen, J. D., and Webb, T. W. (2025). Emergent symbolic mechanisms support abstract reasoning in large language models. In *Forty-second International Conference on Machine Learning*.

[30] Yin, K. and Steinhardt, J. (2025). Which attention heads matter for in-context learning? In *Forty-second International Conference on Machine Learning*.

A TGT example

We use the TGT dataset `1_shot_rlw` to test pre-trained LLMs (and train models from scratch; see Appendix B). In `1_shot_rlw`, symbols are random 1- or 2-character sequences ('rlw'), each symbol appearing in only one field type; the number of fields, and the number of symbols within a field, varies across prompts. A typical instance is:

- *Prompt:* $\mathcal{Q} \sim$ es zd ey db ak) fx $ { tr dz , + vj kj zo % jq hu rd ag $\mathcal{A}$ _ vj kj zo $ es zd ey db ak / jq hu rd ag * fx . $\mathcal{Q} \sim$ dv he) vv bo td $ { xh dp qc my mz , + qk % hw oc cw uh $\mathcal{A}$
- *Continuation:* _ qk $ dv he / hw oc cw uh * vv bo td .
- *Template:*[6] $\mathcal{Q} \sim x$) y $ { z , + u % v $\mathcal{A}$ _ u $ x / v * y .

B Training models on TGT

Given the results in Table 1, we know that pre-trained LLMs have the ability to solve the templatic generation tasks to varying degrees, depending on the model. Can the ability to solve these tasks be learned from scratch, even by smaller models? To find out, we trained 6 basic sequence-to-sequence models from scratch on the TGT 1-shot task. The results are presented in Table 2. In 'OOD Lexical', 2-letter random uppercase symbols (RLW) were used as the values of constituents, having been seen in the training set only in single-symbol 'echo' prompts. The 'OOD ConLen 7' test used prompts containing 1, 2 or 4 constituents, each comprising a length-7 (rlw) symbol string: the models saw only constituent lengths of 1, 2, or 4 in training. The 'OOD ConCnt7' test used prompts containing 7 constituents (each comprised of 1, 2 or 4 symbols), while the training set only contained prompts with 1, 2 or 4 constituents.

Model	**Train Acc**	**Dev Acc**	**OOD Lexical**	**OOD ConLen 7**	**OOD ConCnt 7**
transformer	0.9838	0.8568	0.0052	0.6344	0.1828
nano_gpt	0.9997	0.9997	0.0074	0.6908	0.2247
nano_gpt_attn_only	0.9992	0.9989	0.0070	0.7118	0.3346
cnn	0.6284	0.4766	0.0000	0.0062	0.0025
lstm_attn	0.6995	0.6432	0.0000	0.0000	0.0069
mamba	0.9980	0.9136	0.0137	0.0000	0.0739

Table 2: Performance of models trained from scratch on the `1_shot_rlw` task

These results show that transformers can learn to perform the in-distribution TGT tasks directly, without LM pre-training, as can Mamba; the other architectures (CNNs and GRUs) struggle to learn the task even in-distribution.

All models exhibit poor OOD lexical generalization, indicating that the knowledge acquired during training is not of an abstract-pattern-based nature but instead tied rather strongly to particular symbols seen. On OOD generalization in the number of constituents ('ConCnt 7'), only transformers achieve

[6]Note that templates are not seen by models, but rather used to generate TGT instances by appropriately filling them with symbols.

modest success. The transformers' OOD generalization is stronger in the length of constituents ('ConLen 7'); intuitively, such generalization appears easier than increasing the number of constituents in the template that must be extracted from the prompt and suitably rearranged.

C Universality of the framework: TPF is Turing-complete

How general is the class of functions that can be computed by PSL programs and thereby with DAT transformer networks?[7] By the classic theory-of-computation definition of 'computable', any computable function f can be computed by some Turing Machine TM$_f$, specified by a machine-instruction table governing the evolution of the machine's internal finite-state controller, which, conditioned by the current control state and the symbol on the tape currently being read by the machine's read/write head, writes a symbol and moves the head one position left or right. We adopt the particular TM formalism in which each cell in this table (corresponding to a specific state/row, symbol/column pair $[\sigma_0, S_0]$) is an instruction of the form (1).

(1) Turing-table instruction
   a. If the machine state is σ_0 and the symbol at the current head position is S_0,
   b. then write the symbol S_1, change the state to σ_1, and move the head on the tape one position in direction δ (= 'L' or 'R'; left or right).

   Abbreviated: $\sigma_0,\ S_0 \Rightarrow \sigma_1,\ S_1,\ \delta$

C.1 DAT(TM) implementation of a given TM

Once we see how to express a TM-table instruction in PSL, TPF lets us compile a PSL program implementing all TM$_f$'s instructions into DAT(TM$_f$): a DAT that exactly emulates TM$_f$. We first show how the general instruction (1) can be expressed as a sequence of five productions in PSL. Treating the entire instruction table of TM$_f$ as a list of such instructions (arbitrarily ordered), and translating each instruction into the corresponding five productions, gives a PSL program for computing f. The entire set of productions is one large repeat block: productions keep applying until none are able to apply. The TM tape is realized as the sequence of cells in the Production System Machine PSM (Sec. 2.2), one cell of the PSM for each cell of the tape. The three state variables we use in the PSM are given in (2).

(2) State variables for PSM realizing a Turing Machine
   a. $s[N]$ = current symbol-type in tape position N
   b. $c[N] = 0$ if N is *not* the current head position (otherwise 1, L, or R).
   c. $\sigma[N]$ = current TM state (for all N)

As illustrated in (8) below, the Turing-table instruction (1) $[\sigma_0,\ S_0 \Rightarrow \sigma_1,\ S_1,\ \delta]$ is translated into the following five productions; if the direction of head movement δ is L, *Production* P_2(L) will be activated; otherwise, P_2(R). Here, writing values to state variables is not autoregressive — the PSM cell being updated, N, is wherever the TM head is located (wherever $c[N] \neq 0$ at the time of update) — and attention is not causal — crucially, $n > N$ holds often; e.g., see (4).

(3) *Production* P_1. In the currently active tape position, update the state and symbol
   a. Condition: $c[N] == 1, \sigma[N] == \sigma_0, s[N] == S_0$
   b. Action: set $c[N] := \delta, \sigma[N] := \sigma_1, s[N] := S_1$

(4) *Production* P_2(L). Move head left, update state there[8]
   a. Condition: $c[n] ==$ L, $p[n] == p[N]@pos\_increment$
   b. Action: set $c[N] := 1, \sigma[N] := \sigma[n]$

(5) *Production* P_2(R). Move head right, update state there

[7] We thank Rick Lewis for pointing the way towards the results presented here.

[8] $p[N]@pos\_increment$ is the position $N+1$; $p[N]@pos\_decrement$ is the position $N-1$. The condition requires N to be the position left of the current head position n, flagged for leftward-movement with $c[n] ==$ L.

a. Condition: $c[n] ==$ R, $p[n] == p[N]@pos\_decrement$
b. Action: set $c[N] := 1, \sigma[N] := \sigma[n]$

(6) *Production P_3*. Remove moved-head mark[9]
a. Condition: $c[N]$ in [L, R]
b. Action: set $c[N] := 0$

(7) *Production P_4*. Broadcast new state globally
a. Condition: $c[n] == 1$
b. Action: set $\sigma[N] := \sigma[n]$

The input to the PSM is determined as follows. The sequence of symbols $s[N]$ in the input to the PSM is the sequence of symbols on the input tape of TM$_f$. The TM-state variable $\sigma[N]$ is set to the TM's start state, for all positions N. If the TM head starts at position N_0 on the TM tape, in the initial state of the PSM, $c[N]$ is initialized to 1 for $N = N_0$ and to 0 for all other N.

A minimal illustration of the operation of a single TM instruction — on a tape containing only 3 positions, with the head originating in the middle position — is provided in (8).

(8) Example of a Turing-Machine update

TM-table-cell instruction: σ_0, B $\Rightarrow \sigma_1$, X, R

a. start

s	A	B	C
σ	σ_0	σ_0	σ_0
c	0	1	0

b. P_1

		N	
s	A	X	C
σ	σ_0	σ_1	σ_0
c	0	R	0

c. P_2(R)

		n	N
s	A	X	C
σ	σ_0	σ_1	σ_1
c	0	R	1

d. P_3

		N	
s	A	X	C
σ	σ_0	σ_1	σ_1
c	0	0	1

e. P_4

	N	(N)	$n(N)$
s	A	X	C
σ	σ_1	σ_1	σ_1
c	0	0	1

Thus we have:

(9) $DAT(TM_f)$ Theorem: TM$_f$ in DAT weights
DAT(TM$_f$) implements TM$_f$ using a repeat block of five layers for each cell of the TM$_f$ instruction table.

[9]The Condition utilizes an option in the PSL syntax where "$x[n]$ in [L, R]" does the work of two productions, identical except that in one production, the Condition includes "$x[n] ==$ L" while in the other, it includes "$x[n] ==$ R". In the QKV Machine, this can be achieved in a single layer in which the key is 2-hot, with a 1 at the loci of both L and R, allowing a 'perfect match' with a query containing either $s : L$ or $s : R$. (The normalization of **k** however needs to count only 1 possible match, not 2, since a perfect match is *either* L *or* R, not both, in **q**.)

In DAT(TM$_f$), the TM program for f is implemented in the *weights*; next we see how such a program can be implemented instead in *activations*: a kind of ICL.

The universality result (9) establishes that, despite the focus here on the particular Templatic Generation Task defined in Sec. 2.1, the TPF has potentially broad relevance for mechanistic interpretation of transformer computation.

C.2 DAT(UTM): DAT implementation of a Universal TM

The approach presented in Sec. C.1 could be used to implement a TM that is universal: one that can take as input a tape that includes the TM table for computing a function f as well as an argument string $\boldsymbol{s}$, and can write on the tape the value $f(\boldsymbol{s})$. However, the random-access memory capability of the TPF allows a natural way to directly construct a more efficient Turing-Universal version of the DAT: DAT(UTM). For DAT(UTM), the cells of the PSM are used, like the tape of any universal TM, to store the TM table for computing f as the initial segment of the prompt, followed by the argument string $\boldsymbol{s}$ as the remainder of the prompt. The completion generated by the machine is then $f(\boldsymbol{s})$. This is the TM analog of the use of ICL for TGT, with the prefix of the prompt encoding a TM program rather than a text-generation template.

In detail, for computing $f(\boldsymbol{s})$, the prompt is specified as in (10). Each single entry in the TM$_f$ table is stored in the initial state-variable values of a single cell of the PSM (these are never overwritten); these state variables are $\Sigma 0, \Sigma 1$, *S0*, *S1*, and Δ. The only deviation from the standard DAT architecture defined above is that the prefix of the initialization (the input layer) — where the instructions of TM$_f$ are written — assigns values to state-variables other than s and p; the remainder of the input layer — the suffix of the prompt — is set as usual for PSM, assigning values only to state-variables s and p to provide the argument symbol string $\boldsymbol{s}$ for computing $f(\boldsymbol{s})$.

(10) Universal Turing-Machine input prompt for PSM(UTM) for computing $f(\boldsymbol{s})$
   a. Prefix (program)
      i. For each TM$_f$ instruction: $\sigma_0,\ S_0 \Rightarrow \sigma_1,\ S_1,\ \delta$ (1),
      ii. assign initial state-variable values of a single cell of the PSM prompt:
      $$\Sigma 0 : \sigma_0,\ \textit{S0} : \mathrm{S}_0,\ \Sigma 1 : \sigma_1, \textit{S1} : \mathrm{S}_1,\ \Delta : \delta$$
   b. Suffix (data)
      Encode the argument-string $\boldsymbol{s}$ as usual, assigning the state-variable $s[n]$ the value of the symbol type for the n^{th} position $p[n]$ in s.

We then use the same productions as for the fixed-function TM$_f$ in the previous section (3 – 7), except that the one production that encodes the content of a TM$_f$ table instruction, P_1, is replaced by the production U_1 (11) which 'looks up' the relevant instruction parameters within the activations in the current prompt rather than having those parameters built into the weights of all the layers implementing P_1 for each of the instructions of the TM$_f$ table.

(11) *Production U_1.* Same as *Production P_1* (3) but with states, symbols, and head-movement-direction looked-up from the prompt-prefix cell n encoding the TM instruction in which the current-state and current-symbol parameters $\Sigma 0, S0$ match those of the currently-updating prompt-suffix cell N (emulating the current TM-head position)
   a. Condition: $c[N] == 1,\ \Sigma 0[n] == \sigma[N],\ \textit{S0}[n] == s[N]$
   b. Action: $\sigma[N] := \Sigma 1[n],\ s[N] := \textit{S1}[n],\ c[N] := \Delta[n]$

This establishes a second universality result for TPF:[10]

(12) *$DAT(UTM)$ Theorem: TM in DAT activations*
DAT(UTM) is a version of DAT that implements a UTM in a single repeat block of five layers.

[10] In concurrent work, Schuurmans et al. [21] presents the construction of a Universal Turing Machine using a pre-trained LLM rather than a hand-programmed transformer (and 2027 rather than 5 productions). It's possible that their method for inducing an LLM to execute formal productions given in the prompt might be adapted to the type of UTM-emulation-by-ICL presented here.