

Mechanisms of Symbol Processing for In-Context Learning in Transformer Networks

PAUL SMOLENSKY*, Microsoft Research Deep Learning, USA and Johns Hopkins Cognitive Science, USA
ROLAND FERNANDEZ, Microsoft Research Deep Learning, USA
ZHENGHAO HERBERT ZHOU, Yale University Linguistics, USA
MATTIA OPPER, University of Edinburgh ILCC, UK
ADAM DAVIES, University of Illinois School of Computing, USA
JIANFENG GAO, Microsoft Research Deep Learning, USA

Large Language Models (LLMs) have demonstrated impressive abilities in symbol processing through in-context learning (ICL). This success flies in the face of decades of critiques asserting that artificial neural networks cannot master abstract symbol manipulation. We seek to understand the mechanisms that can enable robust symbol processing in transformer networks, illuminating both the unanticipated success, and the significant limitations, of transformers in symbol processing. Borrowing insights from symbolic AI and cognitive science on the power of Production System architectures, we develop a high-level Production System Language, PSL, that allows us to write symbolic programs to do complex, abstract symbol processing, and create compilers that precisely implement PSL programs in transformer networks which are, by construction, 100% mechanistically interpretable. The work is driven by study of a purely abstract (semantics-free) symbolic task that we develop, Templatic Generation (TGT). Although developed through study of TGT, PSL is, we demonstrate, highly general: it is Turing Universal. The new type of transformer architecture that we compile from PSL programs suggests a number of paths for enhancing transformers' capabilities at symbol processing. We note, however, that the work we report addresses computability, and not learnability, by transformer networks.

JAIR Associate Editor: Sriraam Natarajan

JAIR Reference Format:

Paul Smolensky, Roland Fernandez, Zhenghao Herbert Zhou, Mattia Opper, Adam Davies, and Jianfeng Gao. 2025. Mechanisms of Symbol Processing for In-Context Learning in Transformer Networks. *Journal of Artificial Intelligence Research* 84, Article 23 (November 2025), 69 pages. DOI: [10.1613/jair.1.17469](https://doi.org/10.1613/jair.1.17469)

1 Paper Synopsis: How can Transformers Perform Complex Symbol Processing?

The unprecedented performance of generative language models (LMs) such as those of the GPT family (Radford et al. 2018, et seq.) creates a dilemma for the science of intelligence. Fundamental to virtually all classic theories of natural and artificial intelligence are structured symbolic representations and symbolic processing

*Corresponding Authors: P. Smolensky and R. Fernandez.

Authors' Contact Information: Paul Smolensky, ORCID: [0000-0003-2420-182X](https://orcid.org/0000-0003-2420-182X), psmo@microsoft.com, Microsoft Research Deep Learning, Redmond, WA 98052, USA and Johns Hopkins Cognitive Science, Baltimore, MD 21218, USA; Roland Fernandez, ORCID: [0000-0002-8032-6646](https://orcid.org/0000-0002-8032-6646), rfernand@microsoft.com, Microsoft Research Deep Learning, Redmond, WA 98052, USA; Zhenghao Herbert Zhou, ORCID: [0009-0000-6221-0132](https://orcid.org/0009-0000-6221-0132), herbert.zhou@yale.edu, Yale University Linguistics, New Haven, CT 06511, USA; Mattia Opper, ORCID: [0009-0001-9065-8812](https://orcid.org/0009-0001-9065-8812), M.Opper@ed.ac.uk, University of Edinburgh ILCC, Edinburgh EH8 9AB, UK; Adam Davies, ORCID: [0000-0002-0610-2732](https://orcid.org/0000-0002-0610-2732), adavies4@illinois.edu, University of Illinois School of Computing, Urbana, IL 61801, USA; Jianfeng Gao, ORCID: [0000-0002-5702-6143](https://orcid.org/0000-0002-5702-6143), jfgao@microsoft.com, Microsoft Research Deep Learning, Redmond, WA 98052, USA.



This work is licensed under a [Creative Commons Attribution International 4.0 License](https://creativecommons.org/licenses/by/4.0/).

© 2025 Copyright held by the owner/author(s).
DOI: [10.1613/jair.1.17469](https://doi.org/10.1613/jair.1.17469)

of such representations (Hinzen et al. 2012, see Sec. 3). These provide the basis for explaining the pervasive compositionality of intelligent cognition (see Box 1) (Prince and Pinker 1988). Neural networks appear to be singularly unsuited for such computation, and should fail catastrophically, for example, at compositional language processing, including the ultimate challenge for abstract structure processing, generating complex, syntactically valid natural language (Pinker and Prince 1988). Yet neural language models with transformer architectures (Vaswani et al. 2017) dramatically out-perform all symbolic-computation-based models of language processing, generating rich, syntactically complex English, virtually flawlessly (Chang and Bergen 2024).

Box 1. Compositionality and Systematicity in Cognition

Human cognition copes effectively with a huge range of phenomena by representing complex entities as structured assemblies of simpler entities (Russin et al. 2024). Knowledge derived from previous experience with simpler entities can give rise to knowledge about a novel complex entity by composing existing knowledge of the simpler entities of which the complex entity is composed. This is *compositional generalization*.

This is all so fundamental and natural that we take it for granted. Yet even current state-of-the-art neural networks do not display the extremely robust compositional generalization that is characteristic of human cognition.

Formalizing compositional structures as discrete symbol structures such as parse trees or knowledge graphs gave classic symbolic AI excellent compositional generalization – to the extent that natural phenomena could be successfully decomposed into discrete parts. However, adequately decomposing natural complex entities into recombinable discrete constituent parts typically proved beyond the capabilities of discrete symbolic methods. If English syntax could be fully formalized with a discrete rewrite-rule grammar, then a symbolic NLP system in possession of that grammar would exhibit perfect compositional generalization across the entire language. Despite decades of attempts, however, reducing natural language to such compositional rules has failed to adequately cover the actual richness of language use.

Yet compositional analysis still provides the deepest understanding of the portion of natural language, and cognition generally, that it is able to cover. And while contemporary neural AI systems display extraordinary coverage, their often dramatic failures of compositional generalization suggest that however these systems ‘understand’ the world, they do so in a fundamentally different way than we do.

This cursory discussion of compositionality in cognition is necessarily greatly oversimplified, but faithfully captures the core ideas. Relatedly, *systematic generalization*, often seen as an aspect of compositional generalization, consists of generalizing, to new concepts, the capabilities already learned for other concepts.

As predicted by influential critiques from the 1980s, transformer networks do often struggle with compositionality (Dziri et al. 2023). Nonetheless, they sometimes perform rather well on certain tests of systematicity and compositionality (Sinha et al. 2024) – in particular, when tested via ‘in-context learning’ (ICL), the subject of the work presented here (Brown et al. 2020). As illustrated in Box 2, in the type of ICL we study, a transformer is given an input symbol string which is a ‘prompt’ that includes an example in ‘question-answer’ format; this exemplifies a symbolic template, specific to that particular prompt, for mapping the ‘question’ into the ‘answer’. The input next provides new symbolic material in the ‘question’ format, which must be inserted into that template to generate a ‘completion’, the corresponding symbol string in the ‘answer’ format. (A formal presentation is given in Sec. 4.1.)

Box 2. Symbol Manipulation with In-Context Learning: Templatic Generation

Exemplifying ‘in-context learning’, Large Language Models can take the prompt (1) and generate a continuation (2) (Sec. 4.2.2), consistent with (i) recognizing the initial Q/\mathcal{A} in the prompt as the template (3) – the English-passive-voice-to-predicate-logical-form transformation – with $x =$ ‘the program’ $V =$ ‘translated’, $y =$ ‘a compiler’, and (ii) using this template to generate a continuation by inserting $x =$ ‘my big dog’, $V =$ ‘chased’, $y =$ ‘a small black cat’.

- (1) *Prompt:* Q the program was translated by a compiler \mathcal{A} translated (a compiler , the program)
 Q my big dog was chased by a small black cat \mathcal{A}
- (2) *Continuation:* chased (a small black cat , my big dog)
- (3) *Template:* Q x was V by y \mathcal{A} V (y , x)

Another type of template is a simple inference rule in propositional logic.

- (4) *Prompt:* Q $x \Rightarrow y$ \mathcal{A} y or not x Q (u and v) $\Rightarrow z$ \mathcal{A}
- (5) *Continuation:* z or not (u and v)
- (6) *Template:* Q $p \Rightarrow q$ \mathcal{A} q or not p

Or a basic algebraic equality:

- (7) *Prompt:* Q $\log \{ x * y \}$ \mathcal{A} $\log (x) + \log [y]$ Q $\log \{ 3a * b^2 \}$ \mathcal{A}
- (8) *Continuation:* $\log (3a) + \log [b^2]$
- (9) *Template:* Q $\log \{ u * v \}$ \mathcal{A} $\log (u) + \log [v]$

Or even a nonsensical pattern:

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

Most language tasks are heavily laden with semantics. While traditional theories of semantics (Heim and Kratzer 1998), and classical AI systems (Barr et al. 1981), attempted to capture semantic knowledge with discrete symbolic representations and rules (e.g., expert systems, Cowan, 2001), the recent revolutionary progress in AI has shown the greater power of 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 (Goodfellow et al. 2016).

1.1 Semantics-Free Symbol Processing

Semantically-laden tasks have of course been the subject of extensive research in the AI community. But language processing also demands *semantics-free* inference, most notably, in syntax. Natural language syntax 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. Chomsky famously made this point vividly with his meaningless but syntactically valid sentence, *colorless green ideas sleep furiously* (Chomsky 1957).

Influential critiques of neural network models of cognition have, since the rise to prominence of these models in the 1980s, argued that neural computation is ill-suited to just this kind of abstract, meaning-free computation, so purely captured by symbolic computation (Fodor and Pylyshyn 1988, with many thousands of citations; Marcus 2001). Yet the text generated by neural language models is essentially perfect syntactically. Perhaps classical theory has been mistaken to believe that mastering syntax requires symbolic computation. But perhaps neural LMs can actually implement a good approximation to symbolic computation where needed, and this is what enables their syntactic mastery. This is the hypothesis we pursue in this work.

While LMs' syntax is virtually error-free, they and other neural models do not seem to have an equal mastery of compositionality and systematicity (Dziri et al. 2023). If these networks have implemented symbolic computational capabilities, 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 will end up developing a type of attention head that enables symbolic computation. In the future work we propose in Sec. 9.4, these heads are to be combined with traditional attention heads in a new type of transformer which retains the current power for semantically-laden inference but fuses this with enhanced power for abstract meaning-free symbol processing.

Our new attention heads result from studying what can enable a transformer to perfectly perform a task that is purely semantics-free: the *Templatic Generation Task, TGT*. The examples of TGT given in Box 2 start off being patterns with some meaning for us, in terms of natural language syntax or logical or mathematical inference. The task we study encompasses such interesting cases, but we are not actually concerned with knowledge of that sort; the final example in the Box (10 – 12) achieves the meaning-free symbol manipulation capability we are targeting: the symbol strings through which the templates are exemplified in the prompt are randomly-generated sequences of randomly-generated symbols.

The distinction between semantics-laden and semantics-free inference can be clarified by considering (4). Generating the continuation in (5) via identifying the template in (6) requires only copying and rearranging the symbols in the prompt; no knowledge is required of the semantics of the symbol '=' or of the notion of material implication in logic. Genuinely testing for reasoning ability calls for a model to recognize from the meaning of the phrases in a passage that certain concepts stand in an implicational relation and knowing that this relation licenses drawing new conclusions about those concepts. Testing ICL with (4), as we do, is testing not for reasoning abilities, but rather, abstract-symbol-processing abilities.¹

In this context, consider previous work on ICL that has examined prompts such as 'Q France \mathcal{A} Paris Q Spain \mathcal{A} ' (Hendel et al. 2023), or prompts that provide numerical-vector pairs (x, y) where y is a hidden affine function of x , which the model must infer from the given pairs and apply to a novel value of x (Akyurek, Schuurmans, et al. 2022; Garg et al. 2022, see Secs. C.3–C.4). Now following natural-language-semantic associations, e.g., between countries and their capitals, and inferring affine numerical functions, are exactly the kinds of abilities neural networks have long been known to possess. In contrast, semantics-free ICL tasks such as the Templatic Generation Task test networks for just the type of abstract processing that classic critiques have claimed is beyond neural computation. As a minimal illustration motivating the use of our new task, TGT, note that if we present the prompt 'Q twice x \mathcal{A} x x Q twice a \mathcal{A} ' and we get the desired continuation 'a a', we want to be sure it is the result of correctly filling in the given template, and not simple application of the model's prior knowledge (e.g., from massive English pretraining data) of the semantics of 'twice'; in fact, we'd want the same continuation

¹Thus the model we design here is not to be tested on standard reasoning tasks, but rather, on the purely semantics-free symbol-processing task we design, TGT.

from the prompt ‘ Q thrice $\times \mathcal{A} \times \times Q$ thrice a \mathcal{A} ’, in violation of the English semantics of ‘thrice’, and indeed the same continuation from ‘ Q GBq3 $\times \mathcal{A} \times \times Q$ GBq3 a \mathcal{A} ’. (For a rare study of meaning-free-ness in a LM, see Lasri et al., 2022.)

The semantics-free TGT is instantiated in the paper’s primary case-study, which is inspired by cases such as (1 – 3): the Swap task, presented in Sec. 3.2. This task allows us to focus entirely on how abstract, meaning-free pattern recognition over abstract, meaning-free tokens is possible within neural computation. The TGT not only covers many interesting cases like those illustrated in Box 2, it also demands nearly all the abstract symbol-processing capabilities, provided for free by symbolic computation, which have long been claimed to be both necessary for human-level cognition and beyond the capabilities of neural computation; we present these in (24).

The main contribution of this paper is an explicit demonstration of how transformers can actually provide an implementational platform for fundamental aspects of symbol processing: a platform which, of course, supports extremely powerful learning. To the extent that future work reveals that the neural mechanisms we bring to light here are at work in trained language models, this would constitute a vindication of symbolic theory, which, when embedded in neural computation, would have been proved capable of arising through data-driven learning – a result that has yet to be achieved with purely symbolic computation.

1.2 Research Questions

Our goal is to understand the mechanisms that provide transformers the capabilities that they do possess for abstract, semantics-free symbol processing. For guidance in this challenging pursuit, we turn to Richard Feynman’s famous dictum, “what I cannot create, I do not understand” (Gleick 1993): we design and hand-program a type of transformer network that demonstrably performs ICL of the type illustrated in Box 2. In this network, the contribution that is made by every neuron and every connection to producing this capability is perfectly well understood – because we designed the network ourselves.

Our research questions are formulated in (13).

- (13) Our questions
- a. How *could* any neural network employing fairly standard, general-purpose mechanisms do ICL?
 - b. Can general-purpose operations used by *the transformer architecture* help design such a network?
 - c. Can insights from classic, *symbolic AI* help design such a network?

A preview of the answers we offer is given in (16).

1.3 The Transformer Production Framework (TPF)

The primary contribution of the present work is the *Transformer Production Framework (TPF)*; we use it to study in-context learning to fill in meaningless, randomly-generated symbolic templates, but it can be applied much more widely. In TPF, a computational system is described at multiple levels: we follow the proposal that contemporary machine-learning work take advantage of the three levels of description of a computational system famously proposed by the legendary cognitive scientist David Marr 1982, both in training (Hamrick and Mohamed 2020) and interpreting (A. Davies and Khakzar 2024) foundation models. These levels are: the *functional* level (what Marr somewhat confusingly called the ‘computational’ level), which abstractly specifies the function a computational system computes; the *algorithmic* level, specifying the system’s algorithm for computing this function; and the *implementational* level, specifying how that algorithm is implemented in a physical system.

For our algorithmic level, we develop two symbolic languages for transformer computation. The higher-level programming language, the *Production-System Language (PSL)*, borrows the notion of production system architecture from classic symbolic computation theory, briefly summarized in (14) (Jones and Ritter 2003). Production systems include Emil Post’s rewrite-rule systems (Post 1943) and the string-rewriting systems at the

foundation of the theory of formal grammars (Book and Otto 1993), all key to the classic theory of (symbolic) computation (Hopcroft et al. 2000). Early, special-purpose neural models — very different from the transformer — were developed in the late 1980s for implementing production systems (Dolan and Smolensky 1989; Touretzky and Hinton 1988).

(14) Production Systems

- a. A *production* is a *Condition-Action* pair: when the symbols in a common workspace meet the purely formal (meaning-free) requirements of the Condition, the Action may be taken, which adds or deletes symbols from the workspace.
- b. In the simplest case, the productions are linearly ordered and apply in sequence, the sequence being executed repeatedly until some termination condition is met.
- c. That human intelligence is best modeled by a production system is the founding principle of multiple leading theories of the computational architecture of human cognition (Anderson 2005; Laird 2019; Ritter et al. 2019).²
- d. Cognitive Architecture production systems often have built-in complex capabilities of symbolic-pattern-matching, enabling Conditions to be quite complex; but in our production system, conditions can only require that specified variables have specified values, and actions can only write values into variables. We want to understand how complex symbol processing can emerge through the interactions of productions that individually possess much less powerful built-in symbolic capabilities: specifically, productions simple enough to be precisely implementable in neural networks with a type of transformer architecture.

An outline of the paper is given in (15); the core consists of the 3-level presentation of TPF in (15c) – (15e) [Secs. 4 – 7].

(15) Paper outline

- a. We first discuss related research in which this work is situated [Sec. 2].
- b. We next motivate the case study forming the core of the paper: Swap [Sec. 3].
- c. Adopting a functional level of description, we present a class of symbolic template-filling functions which we seek to compute, motivated by the discussion (15b); we also show that these functions can be computed, with varying success, by pre-trained LMs and by transformers trained from scratch on the task [Sec. 4].
- d. Moving to the algorithmic level, we present an algorithm, PARGEN, for computing such functions using two abstract symbolic machines that we design:
 - i. first, the algorithm is stated in a Production System Language (PSL) for a Production System Machine (PSM) [Sec. 5];
 - ii. then the algorithm is compiled into a program for the QKV Machine, a type of symbolic transformer architecture [Sec. 6].
- e. An implementation-level description is produced by a compiler for exactly implementing QKV programs in a version of the transformer neural network architecture that deploys the residual stream/hidden state discretely, and uses a type of discrete attention: DAT, a Discrete-Attention-only Transformer. Experiments verify the correctness of the algorithms and their implementation. [Sec. 7].

²See Ryu & Lewis 2021 for explication of a transformer’s sentence-processing behavior in terms of a theory of human sentence processing (Lewis and Vasishth 2005) couched in a general production-system-based cognitive architecture (Anderson 2005).

- f. The generality of the framework is established by theorems asserting the Turing-Universality of the language PSL [Sec. 8] (see Sec. 1.4 below).
- g. A general discussion of the work is offered, including how it can promote mechanical interpretability of LMs and trained transformers. Future work is suggested, including a discussion of extending compositionality further [Sec. 9].
- h. Appendices within the article provide
 - A. a walk-through of the symbolic computation implicit in the Swap task
 - B. details of our ICL algorithm, PARGEN
 - C. further related work
 - D. a formal grammar for Templatic Generation Tasks
 - E. a formal grammar for the Production-System Language PSL
 - F. details on the operation of the DAT
 - G. the system-prompt prefix used to test trained transformers on the Templatic Generation Task
 - H. exploratory testing of the GPT-4 LLM model on the TGT dataset
- i. Online appendices provide
 - 1. a detailed discussion of Weiss et al.'s 2021 RASP language for programming transformers, and relations between RASP and our TPF
 - 2. a formal description of QKVL programs
 - 3. a formal compiler for translating a PSL program to an equivalent QKVL program
 - 4. a generalization of the analysis which exploits Tensor Product Representations
 - 5. exploratory from-scratch training and testing of various sequence-to-sequence architectures on the TGT dataset

As a brief preview, our results yield the following general answers to our questions (13).

- (16) Our questions: preview of answers
- a. How *could* any neural network employing fairly standard, general-purpose mechanisms do ICL?

By using hidden-state — or ‘residual-stream’ (Elhage, Nanda, et al. 2021) — encodings of state variables describing each input symbol, including structural variables that encode the role of the symbol in an abstract parse tree.
 - b. Can general-purpose operations used by *the transformer architecture* help design such a network?

The network can target relevant previous symbols for information access by ‘query-key matching’ of variable values (crucially including structural variables), and can set new values for variables through ‘value vectors’ returned by attention.
 - c. Can insights from classic, *symbolic AI* help design such a network?

Just as Production System architectures have enabled powerful symbolic systems for AI and for modeling human higher-level cognition, a Production System programming language can be designed which (i) can be used to write a general, fully interpretable symbolic program for templatic text generation through ICL, and (ii) can be implemented in a transformer by using key-query matching to satisfy production Conditions, and value vectors to perform production Actions.

In more detail, our work develops the correspondences spelled out in (17), some of which are familiar from previous work. (We return to these in Sec. 9.1.)

(17) How can ICL in a transformer perform symbolic templatic text generation?

Via the following [transformer element] ~ [symbolic element] correspondences:

- a. a cell's residual stream ~ a variable-value structure³
 - i. a subspace of the hidden space ~ a state variable
 - ii. a vector component within a variable's subspace ~ a value of that variable
- b. a layer's internal connections ~ a production⁴
 - i. query-key matching in attention ~ evaluating the condition of the layer's production
 - ii. value vectors ~ the production's action
 - iii. query-key matching on a subspace corresponding to a goal ~ conditional branching for goal-directed action
- c. a nested set of structural variables ~ hierarchical data structure
 - i. sharing the value of a level-*l* structural variable ~ in the same (type of) level-*l* constituent — adapted from Hinton 2023
- d. a sequence of structural-variable values (at the 'field' level) ~ the sequence of abstract roles defining a template

It remains for future work to determine whether TPF can shed light on the ICL performed by trained language models. To this end, we suggest concrete hypotheses, and propose methods for testing them, in Sec. 9.2. How language-model training gives rise to such computation is beyond the scope of the current paper: the work reported here addresses *computability*, and not *learnability*, by transformer networks — despite the potentially confusing 'L' in 'ICL'.

1.4 Turing-Universality of the Results

Although the presentation here focuses heavily on the ICL of templatic text generation, our results on what transformers can compute is actually much more general. We show here how programs in the Production System Language PSL can be naturally compiled into transformer networks whose behavior is completely explainable through the PSL programs they implement. But how general is the class of computations that can be expressed as PSL programs? In fact, *every computable function* can be computed by a PSL program: PSL is Turing complete. This is shown in Sec. 8. Note that this is Turing-completeness of a language for programming transformers, a higher-level result than the Turing-completeness of transformer networks themselves — the Turing-completeness of DAT transformers follows from that of PSL; the Turing-completeness of hard-attention transformers shown by Pérez et al. 2021 is a closely-related result. Thus the present work speaks not only to how transformers can carry out the powerful symbolic computation involved in ICL templatic text generation, but also how transformers can in fact function as universal computers.

2 Related Work

This work complements other ongoing work on understanding neural computation, and specifically, in-context learning in transformers. Unlike most mechanistic interpretability work, we are not analyzing trained models, probing for trees or other data structures. Our question is one of *mechanistic computability*: what can the transformer architecture compute in ICL, and exactly how? We focus on using ICL for the general but tightly constrained task of templatic text generation, as defined in 4.1. Our approach is to design an algorithm for

³As when defining an environment for evaluating a function or creating a function closure.

⁴Simple production systems are like NNs in lacking global control structure beyond mere sequencing of productions/layers.

performing templatic generation, in a symbolic form, which can be compiled into weights for a modified transformer architecture. The resulting network is, by construction, fully mechanistically interpretable.

The work pursued here is complementary to much existing work in another respect: we focus on NL-semantics-free, number-free ICL, as emphasized in Sec. 1.1.

2.1 Mechanistic Interpretation of ICL

The concept of in-context learning (ICL) was introduced by Brown et al. 2020, who showed that their GPT-3 transformer could solve new tasks from a few examples in the prompt, without any gradient updating or fine-tuning. Most tasks depended on text meaning, although one task required repairing syntactic errors. The tasks did not include any cases of the pure symbolic manipulation of Templatic Generation, where the generated text is required to be analogous to an example contained in the prompt in that both were generable from a common template, inserting different text strings to the template’s slots.

In their quest for mechanistic interpretability of ICL, Elhage et al. 2021 report the discovery of “induction heads” in small Transformer models with simplified architectures, with subsequent work finding similar mechanisms across a variety of Transformer architectures and parameter scales (Olsson et al. 2022). These logical heads are actually specialized pairs of attention heads (in separate layers) that use past sequence pairs in the context to predict the successor to the current column: they predict that the symbol that will follow the most recently-generated symbol *S* is the same as the symbol that followed the previous instance of *S*. The authors propose that induction heads might constitute the mechanism for the majority of all ICL in large transformer models; they present detailed (but indirect) evidence to support this hypothesis. Our algorithm for Templatic Generation uses a procedure `CONTFIELD` that is identical to induction heads. The abstract symbol-processing abilities of this algorithm, however, derive from a procedure `NEXTFIELD` that is an abstraction of the induction head: it predicts that the *abstract syntactic category* of the to-be-generated symbols is the same as the category that followed the previous instance of the category of the most recently-generated text. Yang et al. 2025 find “symbolic induction heads” that perform a similar function, predicting whether the role of the symbol that will follow *S* is the same as the role of the symbol following the last occurrence of *S*.

Other work in interpreting ICL has argued that LLMs’ capacity to perform ICL is more attributable to “function vector” heads (Hendel et al. 2023; Todd et al. 2024), which transport a compact representation of ICL task mappings from example outputs to generated outputs. Such “function vectors” (activations of corresponding attention heads in the presence of ICL exemplars) can be used to induce the same task behaviors by inserting them into the forward pass of models in a variety of contexts, including zero-shot prompts containing only task inputs. Most recently, Yin & Steinhardt 2025 empirically demonstrate that function vector heads play a much larger role in ICL performance than induction heads, also showing that many induction heads from early stages of LLM pre-training eventually learn to perform as function vector heads instead. In our algorithm, the “function” to be performed for a given prompt is conveyed by its template, which our `PARSE` algorithm identifies and represents in a vector encoding of an abstract parse structure; the heads that perform this are analogous to “function vector” heads.

Another relevant family of work in (mechanistic) interpretability concerns how pre-trained models learn to structure internal representations of discrete variables. For instance, the *linear representation hypothesis* (LRH) argues that models encode distinct variables in separate low-dimensional linear subspaces of the residual stream (Alain and Bengio 2017; Bolukbasi et al. 2016; Geiger et al. 2024; Makelev et al. 2024; Vargas and Cotterell 2020).⁵

⁵Note that there is considerable debate regarding the definition of the LRH — e.g., it is often understood specifically in terms of 1-dimensional subspaces (Elhage, Hume, et al. 2022; Engels et al. 2025; Lee et al. 2025; Park et al. 2024), or defined negatively as the belief that nonlinear methods are unnecessary for interpreting model representations (Canby et al. 2024; A. Davies, J. Jiang, et al. 2023; Pimentel, Valvoda, Hall Maudslay, et al. 2020; Pimentel, Valvoda, Stoehr, et al. 2022; Sutter et al. 2025). Here, we state the version of the hypothesis that is most relevant to TPF.

In DAT, we represent state structures as a concatenation of 1-hot vectors encoding the value taken by a given variable (66). As such, DAT fully satisfies the LRH, where the subspace representing each variable is simply the span of standard basis vectors from its first to final index in the concatenated vector (i.e., its register).

2.2 The RASP Language for Programming Transformers

Since our approach to understanding symbol-processing in transformers is to propose a higher-level language for programming them to do such processing, we focus the remainder of this discussion of related work on a particularly pertinent line of previous research, that based in the Restricted Access Sequence Processing language, RASP⁶, proposed by Weiss et al. 2021. For a more detailed discussion of RASP-based work, including an introduction to the language, see Online Appendix 1; for a wider discussion of related work, see App. C.

RASP-based work and TPF share the approach of deriving a high-level symbolic programming language which is compiled into a transformer network, with the goal of advancing mechanistic explanation of transformer computation. Intuitively, however, on several conceptual dimensions the approaches differ.

(18) TPF vs. RASP-based work

- a. TPF development has focused on text generation, rather than the types of sequence-analysis functions centrally studied with RASP.⁷ Rather like the sequence-processing tasks studied with RASP, the PARSE program in TPF (Sec. A.1) analyzes the prompt by assigning abstract structural properties to symbols in the prompt, but these properties are not produced as output values per se; rather they are embedded as keys and values to drive attention during text generation by the GEN program (Sec. A.2).
- b. As in the implementation of RASP programs, in TPF the residual stream is decomposed into subspaces encoding the values of variables, but whereas the variables in the RASP work are associated to the nodes of a computation graph (Lindner et al. 2023, p. 5, point 5), the variables in TPF are associated only to individual symbol positions in the prompt and have a declarative, rather than a procedural, character — static properties of the symbols which are needed to support generation of new text.
- c. The TGT requires identifying symbol sequences in the prompt as values of variables in a template, and moving them to new positions without alteration. Unlike for RASP, there is no use of MLPs in the current version of TPF because symbols are only copied, not modified — not used to generate different symbols.
- d. In RASP work, the matrix of attention values can be defined by invoking an arbitrary predicate relating the query- and key-positions. In TPF, attention is determined only by exactly matching values for variables specified in the query and key. Multiple variables are specified in an individual query or key, so attention can be conditioned on complex combinations of variable values; in RASP, attention is conditioned solely on pairwise relations between variables. A single PSL production can also write values into multiple variables, whereas each operation in RASP writes to only a single variable.
- e. TPF provides an exact neural implementation of all of PSL, whereas a complete transformer implementation of RASP has not proved possible (Lindner et al. 2023).
- f. Unlike for RASP, for TPF, publicly released software provides a comprehensive visualization tool for simultaneously viewing the program and network implementing it.

⁶Not to be confused with the random-access stored-program machine, also known as a RASP machine (Hartmanis 1971): a random-access-memory version of the Universal Turing Machine (Sec. 8.2).

⁷Among the tasks that Weiss et al. 2021 provide RASP programs to solve is computing, for each input token t , the number of distinct token types in the string that occur with the same frequency in the string as does t , and Dyck- k , detecting balanced brackets in a string with k distinct types of brackets.

3 Motivating a Case-Study of In-Context Learning

We are attempting to understand how neural networks can perform symbolic computation, but just what is ‘symbolic computation’ in this context? To address this, this section follows the outline in (19).

- (19) Section outline
- identify fundamental properties of symbolic computation [Sec. 3.1]
 - present an illustrative in-context learning task – Swap – that calls on the functionality expressed in these properties [Sec. 3.2]
 - preview the remainder of the paper, which presents: a general function class that Swap exemplifies; the symbolic languages PSL and QKVL for expressing algorithms to compute these functions at two different levels of description; and a compiler that translates these programs into a novel type of transformer neural network. [Sec. 3.3]

3.1 Fundamental Properties of Symbolic Computation

As in Box 2 (1 – 3), consider the following mapping, a simple instance of semantic parsing in which an English sentence in passive voice is mapped to a predicate-calculus-style logical form:

- (20) the program was translated by a compiler \mapsto translated(a compiler, the program)

This exemplifies a general schema or template (with variables in italics and constants in roman font):

- (21) x was V by $y \mapsto V(y, x)$

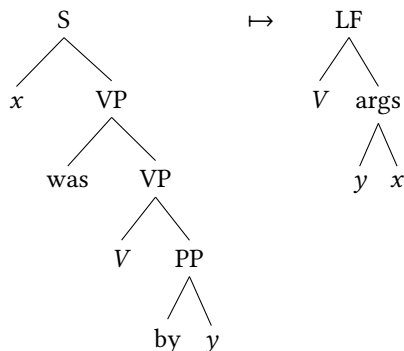
The variables take values that are *strings* of symbols: this binding of values to variables is of particular interest because it has been argued to be a capability crucial for intelligence but beyond the capabilities of the simple neural architectures typical of cognitive and AI models prior to the advent of the transformer (e.g., Marcus, 2001; cf. Smolensky 1987, 1990).

Henceforth we drop the punctuation marks used above to aid human readability, so the template becomes

- (22) x was V by $y \mapsto V y x$

This can be cast in binary tree-to-tree form as an instance of a function we call $\text{Passive} \rightarrow \text{Logical}$:

- (23) $\text{Passive} \rightarrow \text{Logical}$



The symbols constituting the values of x , y are themselves trees, moved as wholes from their positions in the input (syntactic) tree to their positions in the output (LF) tree.

In addition to variable binding, *Passive*→*Logical* displays a constellation (24) of crucial capabilities of symbol processing that are beyond the abilities of simple neural models (Fodor and Pylyshyn 1988, Marcus 2001; cf. Smolensky 2012). We will see explicitly the mechanisms by which these abilities can in fact be naturally achieved in a class of transformer networks. These properties collectively characterize much of the systematicity and compositionality that gives symbolic computation such great power for explaining — and generating — intelligent behavior (Box 1).

(24) Fundamental properties of symbolic computation (Newell 1980)

- a. Representations have *part/whole structure*: they are composed of *constituents* which
 - i. function as wholes in themselves for processing (e.g., they can, as wholes, be moved, copied, deleted, compared for equality);
 - ii. preserve their identities across different positions;
 - iii. have *types*: each constituent is a member — a *token* — of a particular category of constituents.
- b. Representations typically have hierarchical structure: as a whole in its own right, a constituent may be composed of subconstituents.
- c. Representations have abstract roles that constituents fill:
 - i. the type of a constituent is characterized by a sequence of distinct *roles* that its subconstituents *fill* (e.g., the binary tree type can be specified with two roles: left-child, right-child);
 - ii. a role is a *variable*, and in a particular structural instance, it is *bound* to a *value*, its *filler*: the constituent that fills it;
 - iii. copying a constituent as a whole (24a-i) entails copying its sequence of roles, each bound to the particular structure that fills it in the constituent token being copied.
- d. Representations contain symbols: a constituent that has no subconstituent is an atomic element — a *symbol*, a token of its atomic-constituent type (e.g., in many grammar formalisms, a word's part-of-speech)
- e. Representations may be *recursive*: a constituent may have the same type as one of its subconstituents, or subsubconstituents, etc. (e.g., VP in (23)).
- f. Processing of representations:
 - i. can include conditional process *branching*, conditioned on a representation's structure, as well as its content;
 - ii. can include building a structure by *binding* its roles (variables) to particular fillers (values);
 - iii. can include extraction from a structure by *unbinding* one of its roles, yielding that role's particular filler as output;
 - iv. is *compositional*: a constituent is processed by processing its subconstituents and combining the results into a new structure following a composition procedure determined by the type of the constituent.

Experimental evidence that transformers can implement these properties (24) is provided in Sec. 4.2, which shows that mappings like (22) can be performed by language models such as GPT-4 with in-context learning: based only on its pretraining, given a prompt in question (Q)/answer (A) form like (25),

- (25) *Passive*→*Logical*-like GPT prompt
Q J was V by K \mathcal{A} V K J *Q* B was V by C \mathcal{A}

GPT-4 (gpt-4-0613, 2023) can correctly continue (25) with (26):

(26) Passive→Logical-like GPT continuation
 $V C B$

(Throughout the paper we follow the convention of setting the prompt string in blue text and the continuation in orange text. This color contrast, and the special font used for Q and \mathcal{A} here, are simply to aid the human reader; they are not part of the formalism of TPF that we are developing.)

Although examples of symbolic mappings are easier for *us* to process when the symbols are words, and the strings are phrases, and punctuation characters are used (as in Box 2), we will focus on prompts in which symbols are typically simply individual characters, as in (27a) below. This is because in this work, as emphasized in Sec. 1.1, we seek to understand the capabilities of transformer networks to perform pure symbol-manipulation tasks in which symbol meanings are irrelevant or non-existent. Pretraining gives language models great facility with English, which can be exploited to complete prompts in English without necessarily behaving strictly on the basis of meaning-free patterns of symbols. As noted above, completing Q twice $x \mathcal{A} x x Q$ twice $a b \mathcal{A}$ with $a b a b$ could reflect either knowledge of the semantics of English ‘twice’ acquired in LM pretraining, or an ability to perform abstract, semantics-free templatic generation — both are of considerable interest, but the work here is focused on the latter, as most other work on ICL assesses semantically-laden knowledge, and the present research is intended to be complementary to that large body of other work.

Recognizing the pattern in the prompt (25) as the instantiation of a template (22) with variables x and y assigned values J and K , and then generating a continuation by reassigning the variables the new values B and C , is already a non-trivial instance of symbol processing. A considerably more challenging class of such tasks allows the variables to take on values that are not just single symbols, but symbol strings: these must be parsed out of the input to identify the template structure; this was already the case in (20), where each of the variables x and y in (22) had two-symbol values: “the program” and “a compiler”, respectively. We will study prompts in which constituents take on values with variable numbers of symbols; this will be referred to as the ‘length’ of constituents. In addition, prompts will vary in the number of constituents they employ (number of variable ‘slots’ in the templates).

3.2 A Case Study: Swap

An instance of this more challenging type of task is Swap, which will provide a primary case study for the remainder of the paper. An illustrative prompt for Swap is given in (27a). The location of V in the answer substring has now been shifted relative to Passive→Logical so that the pattern now more closely resembles the task Passive→Active, where the passive form “the program was translated by a compiler” is mapped to the active form “a compiler translated the program”: the linear positions of the subject and object have been swapped.

(27) Instance of Swap
 a. Prompt: $Q B C V D E \mathcal{A} D E V B C Q F G V J K L \mathcal{A}$
 b. Continuation: $J K L V F G$

The template for Swap is simply

(28) Swap template: $Q x V y \mathcal{A} y V x$

The symbols Q , V , and \mathcal{A} function as fixed *delimiters*, delimiting the strings providing the values of the variable *constituents* (or ‘slots’, or ‘arguments’) x and y . We will be studying templates with varying numbers of constituents: Swap has 2 (28).

To make examples more transparent to readers, we will typically follow the convention used in (27a) according to which the value of a template slot (x or y here) is a string of individual characters in alphabetic sequence. The Swap task we study does not require this: aside from the reserved delimiter symbols \mathcal{Q} and \mathcal{A} which respectively initiate Question- and Answer-regions of the prompt, the identities of the individual symbols in the examples are arbitrary, of no relevance to the task; in particular, these symbols need not be single characters and could be words or non-alphanumeric symbols, as in (1), or nonsensical strings as in (10).

The prompts we study are a concatenation of two strings. First, an initial question-answer *example* string, which we label X – starting at a prompt-initial token of the reserved symbol \mathcal{Q} and terminating before a second token of \mathcal{Q} . Next is a *continuation-cue* string, which we label C: this consists of a Question-region string followed by the reserved symbol \mathcal{A} , a prompt for completing an Answer-region; the continuation-cue string starts at the prompt’s second \mathcal{Q} and continues through the prompt-final \mathcal{A} . Note that we will reserve the terms *example* and (*continuation-*)*cue* for this usage. When training or testing models on ICL, we will refer to the entire input as a ‘prompt’, reserving ‘example’ for the portion of the initial portion of the prompt that instantiates the template driving the prompt’s continuation. (Thus training set size is measured by the number of ‘prompt/completion’ pairs it contains, rather than the number of ‘examples’.)

For (27a), this structure is shown in (29) (and in more complete tree-form below in (30)).

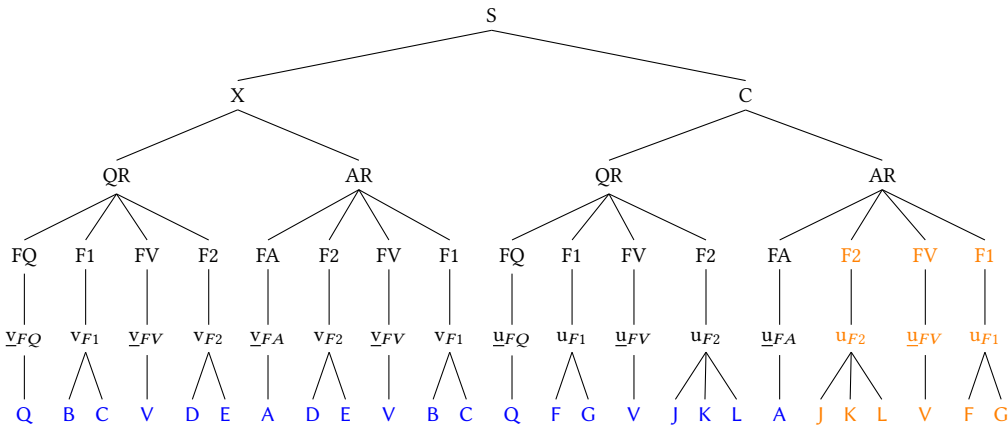
- (29) Example-Cue structure of (27)
 - a. example X = $\mathcal{Q} B C V D E \mathcal{A} D E V B C$
 - b. cue C = $\mathcal{Q} F G V J K L \mathcal{A}$

The (prompt, continuation) pairs we study are the (input, output) pairs of some function in a class \mathcal{F} we define below in (32). In the following informal discussion, when we mention some structural property of the prompt or continuation, that property is imposed by the definition of \mathcal{F} .

3.3 Preview of the Symbolic Computation Implicit in the Swap Task

How to rationalize the continuation in (27)? Intuitively, in the example that initiates the prompt, we recognize the Q-region (QR) substring $\mathcal{Q} B C V D E$ as instantiating the template $\mathcal{Q} x y$ of (28), and from this, recognize the A-region (AR) substring $\mathcal{A} D E V B C$ as instantiating $\mathcal{A} y x$. Then in the continuation cue, using the template given in the Q-region of the example, we recognize a new instance of the template in which x now has value $F G$ while y now has value $J K L$. Inserting these values into the template for the example’s A-region, $\mathcal{A} y x$, determines that the continuation from the final \mathcal{A} should be $J K L V F G$.

- (30) Swap instance (27) full (prompt + continuation) structure: Target



Supporting this intuitive analysis is the structure in (30). A walk-through of our algorithm for Swap, showing how this parse structure is produced and then utilized for generation, is provided in Appendix A; the full algorithm is written out in Appendix B. The walk-through shows how this task is a strong test of symbolic computational capabilities as it calls on nearly all the properties of such computation listed in (24).

Our algorithm for the generation process calls for executing two procedures. NEXTFIELD determines which abstract position type in the template like (28) needs to be generated next: perhaps a constant delimiter symbol such as \mathbf{V} , or a variable slot such as x . The continuation cue determines what string provides the value for a variable slot, and NEXTFIELD looks this up and generates the first symbol in that string. Then, if the string contains more than one symbol, CONTFIELD fills in the remaining symbols in the cued string. Note that CONTFIELD is the functionality provided by an *induction head* (Olsson et al. 2022), and NEXTFIELD is a generalization of this functionality to abstract categories.

The algorithms referred to here and given as informal walk-throughs in Appendix A will be presented formally below when we discuss the algorithmic level of TPF in Secs. 5 and 6. But first we must formally present TPF at the highest level, the functional level.

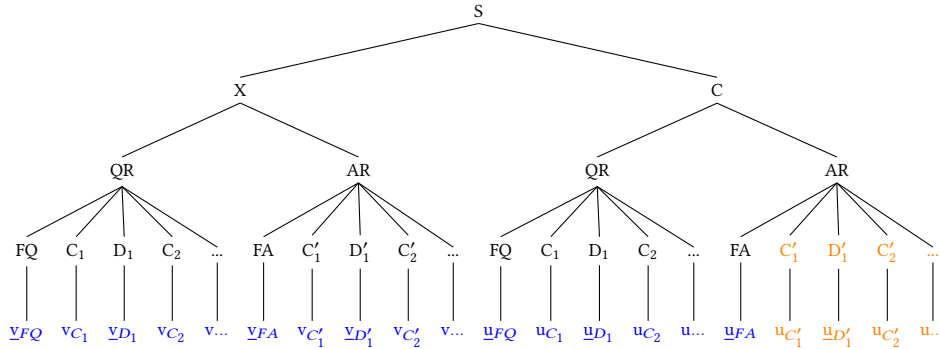
4 TPF, Functional Level. A Class of In-Context Learning Tasks: Templatic Generation

Appendix A shows that the Swap task implicitly incorporates 11 of the 13 properties of symbolic computation given in (24).⁸ This motivates the study of a class \mathcal{F} of ICL functions including Swap which we formalize in this section. \mathcal{F} provides the functional-level description of the Transformer Production Framework TPF we now develop. We call \mathcal{F} *templatic generation* tasks: TGT. (Recall the examples given in Box 2.)

4.1 Templatic Generation Defined

Each input-output mapping in \mathcal{F} defining an instance of our ICL templatic generation task is generated from an instance of the structural template in (31).

(31) The **prompt-continuation** (input-output) structure of templatic generation \mathcal{F}



The yield (sequence of terminal symbols) of the non-orange portion of (31) is the *prompt* P : the concatenation of the symbol sequences $\underline{v}_{FQ} \underline{v}_{C_1} \cdots \underline{u}_{FA}$. This is the input sequence to the function $f \in \mathcal{F}$ being computed by our TPF system. The output $f(P)$ is the *continuation* — the yield of the orange portion: $\underline{u}_{C_1} \underline{u}_{D_1} \cdots$.

(32) specifies the tree template depicted in (31). [In square brackets are comments concerning the motivation for some of the specifications.] A formal grammar for the TGT is provided in App. D.

⁸Higher-level compositionality (24f-iv) across multiple templates given through multiple examples in the prompt, and recursion (24e), are important properties to be incorporated in future work: see the preliminary discussion in Sec. 9.3.1.

- (32) TPF at the functional level: the **prompt-continuation (input-output)** structure S
- a. Region structure
 - i. S is a sequence of two subconstituents: the *example* (X) followed by the *continuation-cue* (C). [A characteristic of typical ‘1-shot’ ICL tasks.]
 - ii. Each of X and C is a sequence of two subconstituents called *regions*: the *Q(uestion)-region* (QR) followed by the *A(nswer)-region* (AR).
 - b. Field structure
 - i. Each region is a sequence of subconstituents called *fields*. A field may not appear more than once in a region. The sequence of fields comprising a QR constituent is the same for the QR within X and for the QR within C; the same is true for the AR. [This is the sense in which the continuation follows the template established by the example; the to-be-generated completion of the AR within C consists of the same sequence of fields as that of the AR within the example given in the prompt.]
 - ii. Fields fall into two classes: *delimiters* (D) and *constituents* (C).
 - iii. The sequence of fields constituting a region alternate between D and C fields.
 - iv. The first field of a QR (resp. AR) is the delimiter field FQ (resp. FA).
 - c. Field values
 - i. A field can be viewed as a variable which takes a symbol string as its *value*. Within any region, no symbol may appear more than once. [A provisional assumption simplifying the matching and copying of symbol-string values between regions.]
 - ii. Within X, we denote the value of any field F by v_F ; within C, by u_F . If F is a delimiter field, we underline the value name (\underline{v} or \underline{u}).
 - iii. Within S, a given delimiter field has a fixed value. [Together with (32b-iii), this assures that the delimiters can be used to parse out the strings that are the varying constituent-field values.] The fixed value of FQ is always \underline{Q} ; of FA, \underline{A} .
 - iv. Within X, a given field type has a unique value (a symbol string) [i.e., the same in QR and AR].
 - d. Field constraints
 - i. Within X, the constituent (non-fixed-value) fields comprising AR — call them C'_k — are a subset of those comprising QR — C_j . Thus each C'_k field in AR is identical to a corresponding field C_j in QR: $C'_k = C_j$ for some function $\varphi : k \mapsto j$.
 - ii. It follows that $v_{C'_k} = v_{C_j}$ where $j = \varphi(k)$: the symbol string which is the value of C'_k in AR is a repetition of the value in QR of the corresponding C_j [it has been ‘copied’ from position j in QR to position k in AR].
 - iii. Note that φ need not be onto: a C_j in QR may be absent in AR. [The symbol string v_{C_j} in the question QR has then been ‘deleted’ from the answer AR.]
 - iv. As within X, within C, a given field type has a unique value. The values of a given *constituent*-field type in X and in C must be different (but from (32c-iii), the value of a given *delimiter* field type is the same in X and C — i.e., it is fixed throughout S). [It follows that the to-be-generated value of field C'_k in AR of the completion C — $u_{C'_k}$ — is the given value of the field C_j within QR of C — u_{C_j} — for $j = \varphi(k)$.]

Here, and throughout the theory development in the text, we are considering ‘1-shot’ ICL in which the prompt provides a single example X of the target input-output template. The ‘ k -shot’ case is the simple generalization

in which the structure S has not just one, but a sequence of k examples X preceding the cue constituent C . Unlike many tasks explored with ICL, in our task (which is semantics-free, purely symbolic: Sec. 1.1), a single example uniquely determines the correct continuation: so in developing the theory, we put aside the additional complexities of multiple examples. While logically unnecessary for a model that has mastery of the task, additional ‘shots’ may be helpful for trained models: we explore this in App. H.

4.2 Relevance of the Templatic Generation Task for Symbolic Computation in Transformers

We are studying how mechanisms within the transformer architecture enable advanced symbol processing, and we have seen through the case study of Swap how the Templatic Generation Task calls on most of the general capabilities involved in symbol processing. But is this task, as formalized above, a task that transformer models can actually perform? In this section we examine this question with respect to both pre-trained language-model transformers and transformers trained from scratch to perform the task.

4.2.1 The TGT Dataset. In order to assess how baseline models train and perform on templatic generation tasks, we created a synthetic dataset, generating prompts according to the TGT grammar and constraints outlined in App. D. Each line in a dataset split file contains a prompt and the associated correct completion. Note that, except where indicated otherwise, in this dataset, within a constituent, each symbol is a “random-letter ‘word’” (‘rlw’) – a random 2-letter sequence of lower-case letters; within delimiters, each symbol is an individual special character – see (10) and illustrations below. (Recall our semantics-free focus, Sec. 1.1.) This dataset is publicly available.⁹

The dataset consists of the tasks and the splits shown in Table 1. Each line of a file containing a split contains a prompt/continuation pair generated by an associated prompt-template. The prompt-template defines the number of constituents and delimiters, and their order of appearance, in the Q and A of the examples given in the prompt. Within a split file, each prompt/continuation line has 2-3 parts, separated by a <tab> character:

```
x   a prompt
y   the correct completion
info optional text identifying the example type (for possible filtering during training)
‘Echo’ prompts are used to introduce out-of-distribution vocabulary symbols to the model (in the train split).
```

Echo prompt.

```
Q ZW A ZW . Q VI A <tab> VI . <tab> {"type": "echo"}
```

breakdown:

```
1 example Q-region:  Q ZW
1 example A-region:  A ZW .
continuation-cue:   Q VI A
target continuation: VI .
prompt info:        {"type": "echo"}
```

One-shot example. Here is a prompt/continuation line from the train split of the 1-shot_rlw task:

```
Q oy xf kq be ` ? jp A jp = . Q jf ty zu np ` ? cx A <tab> cx = .
<tab> {"cons_count": "Q2A1", "cons_len": "Q41.Q41"}
```

breakdown:

```
1 example Q-region:  Q oy xf kq be ` ? jp
1 example A-region:  A jp = .
```

⁹https://huggingface.co/datasets/microsoft/templatic_generation_tasks

```

continuation-cue:    Q jf ty zu np ` ? cx A
target continuation: cx = .
prompt info:        {"cons_count": "Q2A1", "cons_len": "Q41.Q41"}

```

Table 1. TGT tasks and dataset splits

Task	Description
1_shot_rlw	each prompt is 1 example (Q/\mathcal{A} , input/output, pair) + continuation-cue
2_shot_rlw	each prompt is 2 examples + cue
3_shot_rlw	each prompt is 3 examples + cue
5_shot_rlw	each prompt is 5 examples + cue
10_shot_rlw	each prompt is 10 examples + cue
1_shot_eng	symbols in constituents are English words (vs. 2-letter random sequences)
1_shot_rlw_10x	same as 1_shot_rlw but with 10x as many training prompts

Split	Description
train	contains 1, 2, or 4 constituents; each with 1, 2, or 4 symbols
dev	contains 1, 2, or 4 constituents; each with 1, 2, or 4 symbols
test	contains 1, 2, or 4 constituents; each with 1, 2, or 4 symbols
ood_lexical	the constituent symbol vocabulary is absent from training examples (UPPER CASE 2-letter random sequences)
ood_cons_len_3	all template constituent values contain 3 symbols
ood_cons_len_5	all template constituent values contain 5 symbols
ood_cons_len_7	all template constituent values contain 7 symbols
ood_cons_len_10	all template constituent values contain 10 symbols
ood_cons_count_3	all templates have 3 constituents
ood_cons_count_5	all templates have 5 constituents
ood_cons_count_7	all templates have 7 constituents
ood_cons_count_10	all templates have 10 constituents

In the prompt string \mathbf{x} , each example (‘shot’) begins with a “ Q ”, includes an “ \mathcal{A} ”, and ends with a period (“.”). At the end of \mathbf{x} is a continuation-cue (beginning with a “ Q ” and ending with an “ \mathcal{A} ”), to be completed by the model. The ground-truth completion is contained in the \mathbf{y} string.

The above example was randomly generated using the following prompt template generated from the TGT grammar in App. D:

```

Q <constituent 1> ‘ ? <constituent 2>
A <constituent 2> = .

```

The default size of a training split in tasks is approximately 280,000 prompts. The 1_shot_rlw_10x training set has about 2.8 million examples.

4.2.2 Performance of Pre-Trained Language Models on Templatic Generation. For our LLM Transformer testing, we choose several popular models that we had access to through various API services. We started by testing all of these models on the 1_shot_rlw task of the TGT dataset, using the test split.

The LLMs were fed a prompt from our TGT dataset, prepended with a system prompt containing a basic instruction to complete the abstract pattern (see App. G). Each model was tested using the specified number of prompts shown in Table 2.

Table 2. LLM Testing Summary

Model	API Service	Test Date	Prompts	Accuracy
gpt-4	OpenAI	Sep-23-2024	100	0.75
llama-3.1-405b	Together	Sep-26-2024	100	0.60
gpt-4o	OpenAI	Sep-23-2024	100	0.57
claude-3-opus	Anthropic	Sep-26-2024	100	0.48
gemini-1.5-pro	Google	Sep-26-2024	100	0.48
o1-mini	OpenAI	Sep-23-2024	100	0.45
claude 3.5 sonnet	Anthropic	Sep-26-2024	100	0.38
gemini-1.5-flash	Google	Oct-06-2024	100	0.23
o1-preview	OpenAI	Oct-06-2024	100	0.18
gpt-4o-mini	OpenAI	Sep-23-2024	100	0.12
llama-3.1-70b	Together	Sep-26-2024	100	0.12
llama-3.1-8b	Together	Sep-26-2024	100	0.05

These results show that pretrained transformer LMs can perform the Templatic Generation Task to varying degrees, but not even the best model exceeded 75% accuracy. This speaks to the relevance of TGT for understanding the symbol processing capabilities of pretrained transformer LLMs. The capability to perform TGT is present in LMs in some form, but particularly as the templates become larger and the number of symbols in the constituents grow, they struggle: there is certainly room for strengthening this capability in future work drawing on the insights from TPF: see Sec. 9.3.3. From the initial tests shown in Table 2, we chose the best performing model, GPT-4, and then did a series of exploratory tests to see how the performance varied under different task variants. See App. H for details.

4.2.3 Training Models on Templatic Generation. Given our LLM testing, 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 from scratch on the TGT 1-shot task 6 basic sequence-to-sequence models listed in Online Appendix 5. The results are presented in Table 3. The case of ‘OOD Lexical’ was already illustrated: 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.

These results show that transformers can learn to perform the in-distribution TGT tasks directly, without LM training; other neural architectures, CNNs and GRUs, struggle, although Mamba succeeds.

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 modest success. The transformers’ OOD generalization is stronger in the length of constituents (‘ConLen 7’), which is intuitively

Table 3. Results for 1_shot_rlw task

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
cnl	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

easier than increasing the number of constituents in the template that must be extracted from the prompt and suitably arranged.

The superiority of transformers at OOD generalization is consistent with the hypothesis that within the transformer architecture there are internal mechanisms that facilitate templatic generation; the work here provides detailed and comprehensive hypotheses about what exactly those mechanisms may be (Sec. 9.2) as well as suggestions for how the standard transformer architecture can be enhanced to improve OOD generalization (Sec. 9.3.3). The modified-transformer model we design below achieves 100% on all OOD tests (Sec. 7.2.4).

To further understand how altering the TGT task would affect the performance of these models, we also trained them on the other tasks in the TGT dataset. See Online Appendix 5 for details.

5 TPF, Higher Algorithmic Level: The Production System Machine

Having documented the partial success achieved by transformers on the task defined by the functional level description of TPF systems presented in Sec. 4 – the class \mathcal{F} of templatic generation functions instantiated in the TGT dataset – we now descend to the algorithmic level, which actually contains two sub-levels that are formalized as two symbolic abstract machines that compute the functions in \mathcal{F} . Closest to the functional level is the Production System Machine PSM, which we present in this section. Beneath that, closer to the implementation level, is the QKV Machine, presented in Sec. 6: like PSM, this is a symbolic abstract machine, but it can be directly implemented in a type of discrete-attention transformer network – DAT – defined in Sec. 7. In Sec. 6 we describe a compiler that takes a program for the PSM, written in the Production System Language PSL introduced next, and translates it to an equivalent program in QKVL, a language for expressing programs for the QKVM. In Sec. 7 we then describe a second compiler that translates QKVL programs into an equivalent neural network with isomorphic states and state dynamics. It is only at this lowest level that numeric neural computation appears.

5.1 The PSM Architecture

The symbolic processing in both PSM and QKVM consists of two phases: prompt processing – i.e., parsing – followed by continuation generation. These phases are described in general terms for PSM in (33) and (34).

- (33) Production-System Machine state dynamics: Parallel processing of the prompt
- a. Machine states
 - i. The state of the Production-System Machine is a sequence of cell states.
 - ii. A cell is identified by its value of the variable position (or p), a natural number.

- iii. A prompt containing P symbols is encoded in the cells in positions 1 through P , the *prompt cells*; the completion will be encoded in the subsequent *completion cells*.
- b. Cell-state structure
- i. Each cell has a *state* which is characterized by the values of a set of *state variables*, including position (p) and symbol (s). The possible values of each state variable form a discrete set.
 - ii. $s[m]$ is the type of the symbol encoded in the cell with $p = m$.
 - iii. Other state variables are introduced below. Some of them are used to encode the parse tree structure (31), as visualized in (36) – these are the *structural variables*: region (r), field (f) and index (d).
 - iv. The state of a cell is a *state structure* encoding the values of the state variables for that cell; some variables may have the null value *nil*. The space of all possible state structures is the *state-structure space* SSS.
 - v. *Notation*. For any state structure $S \in \text{SSS}$, let the value of state variable x in S be denoted $S.x$. When S is understood, we abbreviate $S.x = a$ to $x : a$. If S has, say, two state variables with non-null values, $S.x = a$ and $S.y = b$, we abbreviate S itself to $x : a, y : b$.
- c. Layer structure
- i. The dynamics of the cells is unrolled in time, so that for each step of computation there is a *layer* of cells.
 - ii. At layer 1, each cell's state includes its position value and a value for symbol that is supplied by the prompt.
 - iii. (36) depicts the state of the machine at a single time step, i.e., a single layer. Each state variable is represented as a row containing the values of the variable across the cells in that layer. The state structure for the first cell is shown in (37).
- d. Production System Language, PSL
- i. A particular PS Machine is specified by a program in the language PSL: this is a sequence of L productions, with production ℓ specifying the updating process for layer ℓ . Each layer is updated according to a single production.
 - ii. The prompt cells in a given layer are all updated in parallel (i.e., not autoregressively) by the production corresponding to that layer.
 - iii. Production ℓ is specified in PSL by: (i) a *Condition*, which defines requirements on the values of state variables in layer ℓ cells in order to allow the production to execute; and (ii) an *Action*, which assigns values to state variables of cells to be set in layer $\ell + 1$.
 - iv. Production Conditions and Actions deploy two meta-variables n and N .
 - ① When production ℓ executes, for each cell position N it sets values for state variables in cell N in layer $\ell + 1$, using the values of variables in cell n of layer ℓ (where possibly $n = N$).
 - ② A production can only execute when n and N satisfy its Condition. There is no requirement of causal interaction, $n < N$ [but see (61d)].
 - ③ When a production executes to update a cell N , if multiple positions n satisfy the production's Condition, the least value of n is used [but see (61e)]. If no position n satisfies the Condition, no Action is taken.
 - ④ If Production ℓ does not assign a value to a given state variable in cell N , that variable has the same value in layer $\ell + 1$ as it did in layer ℓ .

(37) The state structure for the position-1 cell in (36). ($\dot{}$ stands in for other state variables not shown in (36).)

r :	XQ
f :	FQ
s :	Q
p :	1
d :	0
...	$\dot{}$

The hierarchical structure encoded *explicitly* in the tree (35) is now encoded *implicitly* in the spans of the values of the *structural variables*: that the substring **B C** is the value of a field constituent (of type F1) is encoded by the *field variable* field (or f) having the same value (F1) for both **B** and **C** (and a different value for the preceding and following symbols). Similarly, that the prompt prefix **Q A B V C D** is the value of a region constituent is encoded by the *region variable* region (or r) having the same value, XQ, for all the symbols in that prefix. When implemented below in a transformer, this is the method for encoding hierarchy proposed in Hinton’s 2023 GLOM model. (Extending this scheme to enable recursive structure faces a number of challenges, but see Sec. 9.3.1.)

The final structural variable, index (d), denotes the position of each symbol within its string (38).

- (38) a. The values of index – the string-internal positions of field values – are shown as 0, 1, 2, ... in (36) (see J K L).
- b. However, our algorithms will only need the binary distinction between field-initial ($d = 0$) and non-field-initial, $d \neq 0$; therefore, henceforth we will simply index all non-field-initial symbols with $d = 1$ – e.g., see J K L in (39).

Our algorithm PARGEN for solving TGT consists of (i) a parsing algorithm PARSE to generate the encoded hierarchical structure of the input prompt in (36), and (ii) a generation algorithm GEN to use this parse to produce the output continuation string: these are respectively presented in Secs. 5.4 and 5.3. Although logically prior, because of its complexity, we postpone discussion of the parsing algorithm until after we present the generation algorithm.

5.3 Generation Algorithm GEN in the Production-System Language PSL

In Sec. A.2 we illustrate through informal walk-throughs the two operations used during generation of the continuation string: CONTFIELD and NEXTFIELD. Which of these is executed at a particular step depends on whether the most-recently-generated symbol is final in its field (calling for NEXTFIELD) or not (calling for CONTFIELD).

5.3.1 CONTFIELD. We discuss CONTFIELD first because it is simpler than NEXTFIELD, although we need to jump ahead a step into the generation process to reach a point where CONTFIELD is the appropriate operation. So suppose the first continuation symbol **J** has just been generated (along with its structural variable values $f = F2$, $r = CA$, $d = 0$). **J** is not the final symbol in its field F2, so the next symbol must be generated by CONTFIELD, which performs the actions in (40); see the visualization in (39). This operation is justified by (32d-iv), which requires that – in the notation of (30) – u_{F2} , the value of field F2 within CA, must equal u_{F2} , the value of field F2 within CQ.

(39) Use of CONTFIELD to continue generating the current field's value string

r	XQ	XQ	XQ	XQ	XQ	XQ	XA	XA	XA	XA	XA	XA	CQ	CQ	CQ	CQ	CQ	CQ	CA	CA	CA				
f	FQ	F1	F1	FV	F2	F2	FA	F2	F2	FV	F1	F1	FQ	F1	F1	FV	F2	F2	F2	FA	F2	F2			
s	Q	B	C	V	D	E	A	D	E	V	B	C	Q	F	G	V	J	K	L	A	J	K			
p	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
d	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	1	0	0	1			
																	n_0	n			N				

(40) CONTFIELD in action

- Current most-recently-generated symbol: **J** in position $N = 21$; $s[N] = \mathbf{J}$
- Match to symbol type **J** in CQ: position $n_0 = 17$; $s[n_0] = s[N]$ (where n_0 is not field-final)
- Next position: $n = n_0 + 1 = 18$
- Symbol at position n : $s[n] = \mathbf{K}$
- Update symbol[N] to **K** for subsequent propagation (34a) to the next position $N + 1 = 22$: set $s[N] = s[n] = \mathbf{K}$
- Update structural-variable values at position N for subsequent propagation to position $N + 1$: set $f[N] = f[n] = \mathbf{F2}$; $r[N] = \mathbf{CA}$, $d = 1$

Since $n_0 = n - 1$, (40) can be compactly expressed as the condition-action production rule (41).

(41) CONTFIELD as a production operating on state variables

- Condition: n, N satisfy $s[n - 1] == s[N]$ and $r[n - 1] == \mathbf{CQ}$
(where $n - 1$ is not field-final)
- Action: set $s[N] := s[n]$; $f[N] := f[n]$; $r[N] := \mathbf{CA}$, $d[N] := 1$

As pointed out above, this describes the effect of a transformer ‘induction head’ (Olsson et al. 2022).

Note that the condition that $n - 1$ not be a field-final position is equivalent to the condition that n not be field-initial; this can be simply expressed as $d(n) = 1$: recall that within the symbol strings that are values of fields, the index 0 labels the initial symbol, with index 1 labelling the non-initial symbols (39). The production (41) copies a symbol from within the same field-value string as the symbol in CQ that matches the current symbol $s[N]$ in CA: this copied symbol cannot be field-initial, so it necessarily has $d = 1$.

The production (41) makes reference to $s[n - 1]$ in its condition, and $s[n]$ in its action; these symbols are used to update symbol[N] to then be used to generate $s[N + 1]$. Throughout the paper, N will always denote the position being updated by a production, with n denoting a position containing information needed to perform the update (with possibly $n = N$). Note that a given production applies in parallel to update all positions N , and for each N , the relevant corresponding position n is determined independently of the other positions being updated.

In the transformer implementation below, position N will attend to position n to retrieve the information needed to generate the next symbol. Rather than having to attend additionally to position $n - 1$ to implement the condition (41a), it is convenient to localize all the necessary information in one position, n . To do this we introduce the auxiliary state variable s^* , setting $s^*[n] = s[n - 1]$; the condition ‘ $s[n - 1] == s[N]$ ’ now becomes ‘ $s^*[n] == s[N]$ ’ and now all needed information can be gathered at n .

More generally, for each state variable x we will define the related variable prev_x , x^* for short, defined by $x^*[n] = x[n - 1]$. Further auxiliary variables will also be needed; corresponding to state variable x will be x^\wedge , assigned values by the action of productions.

When a production executes at step ℓ of the computation, the production reads values of state variables at step ℓ and the Action writes the values of variables at step $\ell + 1$. Together with the use of the auxiliary state variables

introduced above, this means that the production (41) can be written in what we'll call its *Transformed* notation. The Transformed version of the production (41) can be written solely in terms of N (which, in the transformer implementation, will issue the appropriate query) and n (which will carry the appropriate matching key) (42).

(42) CONTFIELD as a Transformed production

- Condition: n, N satisfy $s^{*(\ell)}[n] == s^{(\ell)}[N]$, $r^{*(\ell)}[n] == \text{CQ}$, $d^{(\ell)}[n] == 1$
- Action: set $s^{(\ell+1)}[N] := s^{(\ell)}[n]$, $f^{(\ell+1)}[N] := f^{(\ell)}[n]$, $r^{(\ell+1)}[N] := \text{CA}$, $d^{(\ell+1)}[N] := 1$

Since, for step ℓ , variable values are always read from the step- ℓ state structure and written to the step- $(\ell + 1)$ state structure, these step values can be left implicit; (42) can be written in the abbreviated form (43).

(43) CONTFIELD as a Transformed production, abbreviated: ContField

- Condition: n, N satisfy $s^*[n] == s[N]$, $r^*[n] == \text{CQ}$, $d[n] = 1$
- Action: set $s[N] := s[n]$, $f[N] := f[n]$, $r[N] := \text{CA}$, $d[N] := 1$

5.3.2 NEXTFIELD. Because the newly-generated symbol **K** does not complete its field F2, the next step of generating the continuation also needs CONTFIELD; this generates the next symbol **L** (along with its structural-variables' values).

L completes the F2 field, so generating the following symbol, **V**, requires NEXTFIELD. Its action is spelled out in (45), visually supported by (44).

(44) Use of NEXTFIELD to start the generation of the next field's value-string

r	XQ	XQ	XQ	XQ	XQ	XQ	XA	XA	XA	XA	XA	XA	CQ	CQ	CQ	CQ	CQ	CQ	CA	CA	CA	CA	CA	CA	
f	FQ	F1	F1	FV	F2	F2	FA	F2	F2	FV	F1	F1	FQ	F1	F1	FV	F2	F2	F2	FA	F2	F2	F2	FV	
s	Q	B	C	V	D	E	A	D	E	V	B	C	Q	F	G	V	J	K	L	A	J	K	L	V	
p	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
d	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	1	0	0	1	1	0	
									n_1	n_2						n							N		

(45) NEXTFIELD in action

- Current most-recently-generated symbol: **L** in position $N = 23$; $s[N] = \text{L}$, with field $f[N] = \text{F2}$
- Match to field **F2** in XA, field-final position: position $n_1 = 9$; $f[n_1] = f[N]$ (where n_1 is field-final)
- Next position: $n_2 = n_1 + 1 = 10$
- Field at position n_2 : $f[n_2] = \text{FV}$
- Match to field **FV** in CQ, field-initial position, n : $f[n] = \text{FV}$; $n = 16$ (where n is field-initial)
- Symbol at position n : $s[n] = \text{V}$
- To subsequently generate symbol **V** for the continuation-string's next position $N + 1 = 24$, update $s[N] = s[n] = \text{V}$
- To subsequently generate structural-variable values for position $N + 1$, update $f[N] = f[n] = \text{FV}$; $r[N] = \text{CA}$; $d[N] = 0$

(45) can be compactly expressed with two productions (46)–(47), communicating through a new variable f' that stores the next-field name (FV here) in the state structure at position N . Note that the condition “ n_1 is field-final” is equivalent to “ $n_1 + 1 = n_2$ is field-initial”, i.e., $d[n_2] = 0$; “ n is field-initial” is simply $d[n] = 0$.

- (46) NEXTFIELD as a sequence of productions: first production
- Condition: n_2, N satisfy $f[n_2 - 1] == f[N], r[n_2 - 1] == XA, d[n_2] == 0$
 - Action: set $f^*[N] := f[n_2]$
- (47) NEXTFIELD as a sequence of productions: second production
- Condition: n, N satisfy $f[n] == f^*[N], r[n] == CQ, d[n] == 0$
 - Action: set $s[N] := s[n], f[N] := f[n], r[N] := CA$

Note that the condition “ $n_1 = n_2 - 1$ is in region XA” is equivalent to “ $r^*[n_2] = XA$ ”. Because condition-matching of each production at each position N operates independently within its own layer, the dummy variable n_2 in (46) can be replaced with n ; at any position, when the first production executes, this n will be bound to a position independently of the position n appearing in (47) that will be bound when the second production executes. Transforming the productions (46)–(47) then yields (48)–(49).

- (48) NEXTFIELD as a sequence of Transformed productions: NextField1
- Condition: n, N satisfy $f^*[n] == f[N], r^*[n] == XA, d[n] == 0$
 - Action: set $f^*[N] := f[n]$
- (49) NEXTFIELD as a sequence of Transformed productions: NextField2
- Condition: n, N satisfy $f[n] == f^*[N], r[n] == CQ, d[n] == 0$
 - Action: set $s[N] := s[n], f[N] := f[n], r[N] := CA, x[N] := 1$

The NEXTFIELD productions so far cover the case shown in our working example (44). There is another case however, which will explain the appearance of x in (49b): when XA includes a delimiter that is not present in XQ and therefore not in CQ. This occurs when the Answer inserts fixed material absent in the Question, as in the Active \rightarrow Passive template: $x V y \mapsto y$ was V by x . In this case, the last-generated continuation symbol’s field must be found in XA rather than in CQ. Thus another conditional branch is required: if the NextField2 production fails to execute (no matching field in CQ) then we need to execute NextField3, which is identical to nextField2 except that CQ is replaced by XA:

- (50) NEXTFIELD as a sequence of Transformed productions: NextField3
- Condition: n, N satisfy $f[n] == f^*[N], r[n] == XA, d[n] == 0, x[N] == 0$
 - Action: set $s[N] := s[n], f[N] := f[n], r[N] := CA$

x_temp (x for short) is a ‘branch variable’, explained shortly; it is initialized to 0 but set to 1 if NextField2 executes; this blocks NextField3 because of its Condition $x[N] == 0$.

5.3.3 *The generation algorithm GEN.* The core of the generation algorithm GEN is provided by the four productions ContField (41), NextField1 (48), NextField2 (49), and NextField3 (50). However there is a conditional branch here: IF the most-recently-generated symbol is not field-final, THEN ContField should execute (its Condition will be satisfied in that case), but NextField should *not* execute (even if its Condition as currently stated is satisfied); ELSE the NextField productions *should* execute.

We implement this branching through a *branch variable* – an additional state variable end (e for short), which is initialized in each cell to $end := 0$, and set to value 1 in cell N by the execution of ContField in that cell; the Conditions of the NextField productions are supplemented to include $end[N] == 0$. This ensures that the NextField productions will not execute if ContField has.

This gives the core of GEN in PSL as the sequence of productions (51) – (54).

- (51) ContField: final form [production G1 of (100)]
- a. Condition: n, N satisfy $s^*[n] == s[N], r^*[n] == CQ, d[n] = 1$
 - b. Action: set $s[N] := s[n], f[N] := f[n], r[N] := CA, d[N] := 1, e[N] := 1$
- (52) NextField1: final form [production G2 of (100)]
- a. Condition: n, N satisfy $f^*[n] == f[N], r^*[n] == XA, d[n] == 0, e[N] == 0$
 - b. Action: set $f^*[N] := f[n]$
- (53) NextField2: final form [production G3 of (100)]
- a. Condition: n, N satisfy $f[n] == f^*[N], r[n] == CQ, d[n] == 0, e[N] == 0, x[N] == 0$
 - b. Action: set $s[N] := s[n], f[N] := f[n], r[N] := CA, x[N] := 1$
- (54) NextField3: final form [production G3' of (100)]
- a. Condition: n, N satisfy $f[n] == f^*[N], r[n] == XA, d[n] == 0, e[N] == 0, x[N] == 0$
 - b. Action: set $s[N] := s[n], f[N] := f[n], r[N] := CA$

In addition to these four productions, additional productions are required for book-keeping purposes. One (production G0) initializes end to 0, and the others (production Gpre-1, Gpre-2) update the prev_v (v^*) state variables to correctly provide the value of $v[N - 1]$ to each cell's $v^*[N]$ state variable, after a production has altered the values of v . All 7 productions constituting GEN are given a complete PSL specification in (100) in App. B.

We now take up the parsing algorithm, which is presented below in (55); this also omits the book-keeping productions such as those for updating the prev_v variables after v values have changed. These are included in the full 24-production PSL program for PARSE presented in (99) in App. B.

5.4 The Parsing Algorithm PARSE

The generation algorithm relies heavily on the structural variables region, field, index (r, f, d): these state variables encode the parse of the input produced by an algorithm PARSE that we now discuss. PARSE is presented descriptively below in (55).

Note that the field names F1, FV, F2 which we have used previously are just more readable versions of the names assigned by this algorithm. For instance, since all field values are initialized to equal the position value (Production P0), the symbol B in position 2 of the prompt in (36) is initialized to $f = 2$, and never changed by the algorithm. Thus F1, FV, F2 above correspond to the values $f = 2, 4, 5$ resulting from the algorithm (on this particular prompt). The names given to region-delimiter fields above (FQ, FA) are, however, those used by the algorithm.

The 'P#' labels 0, 1a, 1b, ..., 11 in (55) identify particular productions in the parsing algorithm: they are given in App. B. Here we illustrate with three of these 16 productions: one typical, and two with distinguishing features.

- (55) Parsing algorithm: operations on structural variables
 Showing for each production: its goal (with a pointer to its role in the walk-through in Sec. A.1) (94) – (96); the specification(s) in (32) justifying it; its number P#; and a verbal description

Goal	(32)	P#	Description
initialize variables	mark region, type as 'unset'; give each cell a unique field value	0	everywhere set region = R (r:R), type = T (t:T), index = 1 (d:1) everywhere set field = position (fp)
identify regions (94)	mark start of Q-regions	a	1a mark position 0 with r:XQ, f:FQ, t:D
	fill in Q-regions	b-i,iv	1b mark start of CQ at repeat of symbol starting XQ with r:CQ, f:FQ, t:D
		a	2a spread r:XQ rightward to 1st delimiter (t:D), where field CQ starts [now all Ds are region-Ds; region-internal Ds are inserted later; 'XQ' includes XA for now]
		"	2b spread CQ rightward until end of prompt ['CQ' includes CA for now]
	mark start of A-regions	a-ii,b-iv	3a mark start of XA region at A in current 'XQ' region with r:XA, f:FA, t:D
		"	3b mark start of CA at A in current 'CQ' region with r:CA, f:FA
	fill in A-regions	a-ii	4 spread r:XA right until first t:D, where field CQ starts
		c-i,iii	5a mark a symbol in XQ that is repeated in CQ as a (region-internal) delimiter: t:D [non-causal]
identify region-internal delimiters, constituents (96) (95)	identify region-internal delimiters	"	5b mark a symbol in CQ that repeats a symbol in XQ as a delimiter, same field: t:D
		"	5c mark a symbol in XA that repeats a symbol in CQ as a delimiter, same field: t:D
		c-i,iv	6 identical untyped symbols in X have the same C field
assign fields 1 (96)	identify remaining delimiters	b-ii	7 mark all unset types in XA as delimiters: t:D
set remaining types	identify constituents	7'	mark all remaining unset types as constituents: t:C
assign fields 2 (96)		b-i	8 XQ (QR in X) and CQ (QR in C) have identical field-sequences: a constituent field following a particular delimiter field in CQ is the same as the one following the same delimiter field in XQ
assign indices (38)		b-iii	9 constituent fields change only at delimiters
		10	at a change in field: d:0
		11	at end of prompt, set parse=0: a:0
mark end of parsing			

Production P6 is a typical, simple production, virtually a literal translation from the English description to the formal expressions.

- (56) *Production P6.* Identical symbols are contained in the values of identical fields.

- a. Condition: n, N satisfy $s[n] == s[N]$
- b. Action: set $f[N] := f[n]$

The intended effect is that when two symbols at positions j and k match, where $j < k$, the value of f at the earlier position j is copied rightward to become the new value of f at the later position k .

Note that when the same symbol appears in cells j and k , with $j < k$, P6's condition is satisfied under multiple bindings of n and N . Suppose the fields for these positions are, for concreteness, $f[j] = F1$ and $f[k] = F2$. Then the four n, N pairs meeting P6's Condition are shown in (57).

- (57) Effect of $f[N] := f[n]$ when $j < k$, $f[j] = F1$, $f[k] = F2$

	n	N	effect if production executes	comment
a.	j	k	set $f[k] := F1 = f[n] = f[j]$	desired effect
b.	k	k	set $f[k] := F2 = f[n] = f[k]$	no effect
c.	k	j	set $f[j] := F2 = f[n] = f[k]$	undesired effect
d.	j	j	set $f[j] := F1 = f[n] = f[j]$	no effect

According to ③ of (33d-iv), when a cell N is updated, it can only read information from at most one cell n (possibly itself); if there are multiple n that meet the condition for updating N , the lowest-valued n will be used. When updating $N = k$, the lowest-valued n meeting the condition will be $n = j < k$, so case a of (57) will be the operative one, and the desired effect will occur. When updating $N = j$, the lowest-valued n meeting the condition will again be $n = j < k$, case d: this re-assigns $F1$ to $f[j]$, yielding no effect. Crucially, case d blocks the undesired effect of leftward copying from k to j that would result if c rather than d were the operative bindings. Thus the

lowest-match condition ③ of (33d-iv) forces information to only travel from earlier to later positions when P6 executes.

The third column of the table in (55), headed ‘(32)’, identifies for each production the clauses in the definition of our function class \mathcal{F} (32) that license that production. For this production P6, this is (32c-i,iv), which state that ‘no symbol may appear in the value of more than one field type’ and ‘within X, a given field type has a unique value’.¹⁰

An exceptional production is Production P5a.

- (58) *Production P5a.* Mark a symbol in XQ that is repeated in CQ as a (region-internal) delimiter: t:D.
- a. Condition: n, N satisfy $r[n] == CQ, r[N] == XQ, s[n] == s[N]$
 - b. Action: set $t[N] := D$

(58) is the only production requiring $n > N$: the to-be-updated position N , in region XQ, precedes the position n in CQ which matches the symbol $s[N]$ and hence determines that N is a delimiter position. In the implementation, this will require non-causal (forward-looking) attention. All other productions involve only causal (backward-looking) attention. Thus the DAT implementation of our PSL program for TGT shares with non-causal (prefix) language models (T. Wang et al. 2022) the property of using strictly causal attention for generation, but (extremely limited) non-causal attention for parsing the prompt prefix.

An atypically complex production is Production P8.

- (59) *Production P8.* XQ and CQ have identical field-sequences: a constituent field following a particular delimiter field in CQ is the same as the one following the same delimiter field in XQ.
- a. Condition: n, N satisfy $r^*[n] == XQ, r[n] == XQ, t^*[n] == D, t[n] == C,$
 $r[N] == CQ, t^*[N] == D, t[N] == C, f^*[n] == f^*[N]$
 - b. Action: set $f[N] := f[n]$

(59) in fact has the most complex Condition of all the productions for parsing and generation; the complexity of the formal expressions follows that of the English description. This is the core production enabling the identification of template structure in which abstract (syntactic) categories (fields) appearing within the ‘question’ are identified with those appearing in the ‘answer’, e.g., x in the template (3).

5.5 Combining PARSE and GEN

The first goal of the PARGEN algorithm is to parse the prompt; then the second goal is to generate a continuation. As is standard in production systems (Jones and Ritter 2003), there is a state variable *parse* (a for short) encoding the current goal, which the Condition of productions evaluates. $parse = 1$ signals that the current goal is to execute the productions comprising PARSE; $parse = 0$ signals the goal to execute GEN. So every production in the PARSE program includes in its Condition $a == 1$. The final production of PARSE (P11) sets $a := 0$, which is required by the Conditions of all the productions of GEN. This production P11 only applies to the final prompt cell (housing the last symbol of the prompt); $a == 0$ is then propagated to the first generation cell and from there to all subsequent generation cells via the autoregressive updating of the generation cells (34). The end of the prompt is not marked by an end-of-prompt symbol; rather, within the final prompt cell (holding the last symbol of the prompt), a dedicated state variable z is set to the symbol EOP in the input.

¹⁰The qualifier ‘within X’ here raises a subtlety in the action of this production. Two symbols can match, meeting P6’s Condition, in two cases. If one of the symbols is contained in the value of a constituent field, then the two matching symbols must both occur in X, because the values of constituent fields in C must be different from those in X (32d-iv). If a symbol is contained in the value of a delimiter field, it need not lie within X, but the matching symbol must also be within the value of the same delimiter field, as these fields have a constant value throughout the prompt (32c-iii). P6 is actually used to handle both these cases.

5.6 The PSL Programming Language

Not just those exhibited above, but in fact nearly all the productions we use for parsing and generation share the form shown in (60); for the other productions, see (61).

- (60) For some state variables x, y, z, u, v, w and constant field values $\text{CONST}_i, 1 \leq i \leq 3$:
- Condition: n, N satisfy $x[n] == \text{CONST}_1, x[n] == y[N], z[N] == \text{CONST}_2 \dots$
 - Action: $u[N] := \text{CONST}_3, v[N] := w[n] \dots$

The ellipses ‘ \dots ’ indicate that any number of equality tests ($\dots == \dots$) of the forms shown for the Condition, and any number of assignment statements ($\dots := \dots$) of the forms shown for the Action, are permitted.

Note that the condition $x[N] == y[N]$ is not included among the possible Conditions in (60), but that effect can be achieved by combining the two allowed Conditions $x[n] == y[N], p[n] == p[N]$, as in production P10 in (99).

Thus the core of a PSL program is a sequence of productions of the form (60). Implementing such a program in a transformer network requires only implementing each production as a transformer layer, the layers sequenced to match the sequence of productions.

Additional expressiveness needed for our ICL programs is also provided by PSL (61).

- (61) Additional expressiveness in the PSL programming language
- Inequality tests.* Conditions can include inequality tests: $x[N] != \text{CONST}$, as in production P1b in (99), or $x[N] != y[N]$, as in production P10 in (99), or $x[n] != \text{CONST}$, as in production G1 in (100).
 - Operators in Conditions.* Conditions can include operators such as F in “ $x[n] == y[N]@F$ ”, i.e., $x[n]$ equals the image of $y[N]$ under the mapping F . These operators map variable values into other values, such as the function $F_{\text{pos_increment}}$ that increments by 1 any value of p , the position index, as in the production Ppre-1 in (99).¹¹
 - Repeat blocks.* A sequence of consecutive productions can be executed repeatedly until some condition (e.g., no state change, or a requirement on state variables with the form of a production’s Condition) is achieved. This is illustrated in (99) by the block containing the two-production sequence Ppre-2a, P2a.
 - Causal propagation.* Optionally, a production may be specified as requiring ‘causal’ information flow: $n < N$.
 - Rightmost selection.* Optionally, a production ℓ may specify $\text{right_match}(\ell) = \text{T}$: then, among positions n whose keys tie for a perfect match with a query, the rightmost (rather than the default case, leftmost: $\text{right_match}(\ell) = \text{F}$) position is selected for value propagation.
 - ‘In’ construction.* Rather than specifying a single value for a variable in a production’s condition, a construction ‘ x in $[x_1, x_2, \dots]$ ’ can be used to identify multiple matching values. A ‘not in’ construction is also available.

In the code base, a default production is specified as where $\langle \text{condition} \rangle: \langle \text{action} \rangle$. For the exact syntax, and specification of the optional features in (61), see the formal grammar for PSL provided in App. E.

¹¹For implementation, we will require that F be implementable as a linear transformation; this is always possible when the vectors embedding the values of y are linearly independent, as they are when they are one-hot.

6 TPF, Lower Algorithmic Level. The QKV Machine: Symbolic Attention

6.1 The QKVM Architecture

As shown in the preceding section, based in symbolic-AI-style production system computation, the PSL language can be straightforwardly deployed to write fully interpretable programs for the PSM to perform both the parsing and generation phases of PARGEN. To bridge to the level of implementation in a transformer neural network, however, we need another abstract machine closer in architecture to the transformer itself: this machine – the QKV Machine – uses query-key attention to update machine states. Like the PSM, however, it is purely symbolic. In this section, we will present a compilation process for translating PSL programs into QKVM programs. In the next section, another compiler will be presented that translates a particular QKVM into a numerical machine that uses neural computation: an attention-only transformer network using a form of discrete attention and discrete state normalization, the Discrete-Attention-only Transformer, DAT.

(62) QKV Machine state dynamics: Parallel processing of the prompt

a. Machine states

- i. Like the PSM, the QKV Machine state is a sequence of cell states. But now, each cell state is specified by the values of five *cell attributes*: query, key, value (q, k, v), as well as input (i) and output (o). The value of an attribute is a structure in SSS: the same set of state variables as for the PSM, each assigned a particular value (possibly nil).
- ii. *Notation*. Structures in SSS are specified as they are for the PSL. For a given cell, the SSS-valued attribute query, for example, might have the value notated “ $r:CA, f:FV$ ” – the state structure $S \in$ SSS with two non-null state-variable values, $S.r = CA$ and $S.f = FV$.

b. Layer structure: as for the PSM

- i. The cell dynamics is unrolled in time, so that for each step of computation there is a layer of cells.
- ii. In layer 1, the input for cells containing the prompt symbols – the *prompt cells* – are assigned appropriate values for position and symbol, the latter taken directly from the prompt string.
- iii. The prompt cells are all updated in parallel; the structure in SSS assigned to the input attribute to the cell in position p of layer 2 is the structure assigned to the output attribute for the cell in position p of layer 1. This is repeated for all layers.
- iv. A sub-sequence of layers can constitute a repeat block, in which case the layers in that block are evaluated in sequence repeatedly until a termination condition specified for that block is met.

c. Cell-state updating

Rather than by productions, in the QKVM, cell updates are performed by a uniform *discrete attention* process:

- i. The structure assigned to the output attribute of a cell N is determined by (i) $\text{input}[N]$ and (ii) the structure assigned to the value attribute of a cell n with a key that matches the query of cell N :
 $\text{query}[N] \text{ matches } \text{key}[n] \Rightarrow \text{output}[N] := \text{value}[n] \blacktriangleright \text{input}[N]$.
- ii. (definition of \blacktriangleright) $\text{output}[N]$ is a state structure in SSS which is the same as $\text{input}[N]$ except that the values of state variables that have been assigned non-null values in the structure $\text{value}[n]$ overwrite whatever values these variables may have had in $\text{input}[N]$.
- iii. A query structure (exactly) *matches* a key structure if, for every state variable x with a non-null value v in query, the value of x in key is v . (Any additional variables with non-null values in key are ignored.)

- iv. When a cell N is updated in layer ℓ , if no position n has a key that matches cell N 's query exactly, no Action is taken; if multiple positions n have a matching key, the least such value of n is used (unless $\text{right_match}(\ell) = \text{T}$, in which case the largest such value is used).
- v. Our TGT program does not exploit this additional option, but it is available: if a layer ℓ has $\text{causal_atn}(\ell) = \text{T}$, then attention to a query at position N can only match positions $n \leq N$.

d. QKV Language, QKVL

- i. A particular QKV Machine is specified by a program in the language QKVL: this specifies, for each layer $\ell = 1, \dots, L$, a map $W_q^{(\ell)}$ from SSS to SSS which takes the input structure for a cell and maps it to the query structure for that cell. All cells in layer ℓ use the same map $W_q^{(\ell)}$. There are corresponding maps $W_k^{(\ell)}$ and $W_v^{(\ell)}$ mapping from the input to key and value.
- ii. $W_q^{(\ell)}$ (and likewise $W_k^{(\ell)}$ and $W_v^{(\ell)}$) is specified by identifying, for some set of state variables, what value those variables take in the SSS state structure $q^{(\ell)}[N]$. These values are specified in terms of the values of state variables in $i^{(\ell)}[N]$. For example, if the space of possible values V_x, V_y for the state variables x and y are the same, V , the value of x in $q^{(\ell)}[N]$ might be specified as the value of y in $i^{(\ell)}[N]$: $q^{(\ell)}[N].x = i^{(\ell)}[N].y$. When it is understood that it is the level- ℓ state structure of q that is being specified, we abbreviate this to $x : y$.
- iii. QKVL allows specifications of the form $q^{(\ell)}[N].x = F(i^{(\ell)}[N].y)$ for some specified function F on the value-space V . (Also written $x : y@F$.) (E.g., for $y = p$, $q^{(\ell)}[N].x = F_{\text{pos_increment}} i^{(\ell)}[N].p$ where $F_{\text{pos_increment}}$ increases by 1 any position value for p .)
- iv. QKVL also allows inequality specifications of the form $q^{(\ell)}[N].x \neq i^{(\ell)}[N].y$; this entails that q does not match any key k for which $k.x = i^{(\ell)}[N].y$, but otherwise the value of $k.x$ is ignored in evaluating a match. Similarly, $q^{(\ell)}[N].x \neq x_k$ is permitted, with x_k a fixed possible value of x .
- v. See (64) for further discussion of QKVL.

(63) QKV Machine state dynamics: Autoregressive generation of the continuation [as for PSM (34)].

- a. Given a prompt of length P , the level-1 states of cells with $p = 1, \dots, P$ are determined by the prompt (62b-ii). These are processed in parallel according to (62). The continuation cells are updated autoregressively. The level-1 state of cell $P + 1$ is the level- L state of cell P (except that $p := P + 1$). The values of all state variables (except p) are copied from P to $P + 1$, not just the symbol variable s .
- b. Cell $P + 1$ is processed through all L layers, and then the level-1 state of cell $P + 2$ is set equal to the level- L state of cell $P + 1$ (except that $p := P + 2$). This process iterates until a termination condition is met (e.g., the generation of a special termination symbol).
- c. The generated continuation string is read from the level-1 continuation cells as the sequence of values of input.symbol.

6.1.1 The QKV programming language, QKVL.

(64) QKVL Programming Language

- a. For a given layer ℓ , specify each of the $q/k/v$ structures in V_{SSS} via instructions of the form *target-variable : source-variable*, such as
 - $x : y$
which means that for any cell, in the $q/k/v$ structure being specified, the variable x is given the value that variable y has in the input structure for that cell, or
 - $x : x_i$
which means that x is assigned the fixed value x_i . This specifies the mappings $W_{q/k/v}^{(\ell)}$ of (62d).
- b. More general specifications are also possible:
 - $x : y@F$
which means that x is assigned the value $F(\text{input}.y)$ where F is a function over the shared space of possible values for x and y that is defined in a library accessible to QKVL (62d.iii). And
 - $x != x_i$
means that the specified state structure will match any value of x except x_i (62d.iv).
- c. *Note:* In PSL, expressions such as “ $x[n] : y[N]$ ” make reference to the values of the variables x, y at two locations n, N ; but in QKVL code, the instruction “ $x : y$ ” references the values of both variables at the same location. The y value is taken from the input to a cell, and this is assigned to the value of x in that cell’s query, key, or value structure, whichever is being specified by the instruction.

In the code base, a QKVL program is a description of QKVM layers, in the JSON data format. (See Online Appendix 2.) At the highest level, it is an array of layers. Each layer is a Python dictionary with the fields “layer_comment” (a text description of the layer’s purpose), “causal_attn” (a boolean specifying whether causal attention is enabled for the layer), “right_match” (a boolean specifying whether right-most attention selection should be applied to the layer), and most crucially “weights” – a dictionary describing the three mappings $W_{q,k,v}$ of (62d-i) which will next be compiled to determine the numerical weights in a layer of a DAT transformer implementing the QKV program. These mappings are specified by instructions of the form given in (64).

An example layer dictionary is: {“layer_comment”: “// parse step pre 1. set prev_position and prev_symbol”, “causal_attn”: false, “right_match”: false, “weights”: {“q”: {“p”: “p”, “a”: “a”}, “k”: {“p”: “p@pos_decrement”, “a”: “1”}, “v”: {“p*”: “p”, “s*”: “s”}}}

6.2 Compiling PSL Code to QKVL Code

(65) shows how we compile PSL productions of the form (60) into lower-level QKVL instructions. (See also Online Appendix 3.)

(65) From PSL to QKVL

PSL Condition	QKV instructions		Comment
	q[N]	k[n]	
$z[N] == C_z$	$z : z[N]$	$z : C_z$	C_z is a constant value of z
$x[n] == y[N]$	$x^{\setminus} : y[N]$	$x^{\setminus} : x[n]$	$y[N]$ may be replaced by a constant value of x or a transformed value $y[N]@F$ (64b)
Action	$v[n]$		
$u[N] := w[n]$	$u : w[n]$		$w[n]$ may be replaced by a constant value of u

In the instructions, “(n)” and “(N)” are redundant, since for the query, the state variables are always evaluated at the cell being updated (N), and the information used to do the updating (k, v) always comes from the cell n meeting the Condition relating it to N. We therefore typically omit “(n)” and “(N)” when giving QKVL instructions, as in (99) – (100) in App. B. Note that the specifications of q, k, v here apply to every cell in the layer realizing the given production (62d-i): the use of the labels N, n is only to indicate which of all the cells with the specified q, k, v values correspond to the cells labelled N, n in the PSL production being compiled.

It is not hard to see why the compilation of a PSL production to QKVL instructions follows (65), which should be read as follows. The cell attributes q, k, v are all state structures. The structure given in (65) for cell N’s query has two non-null state variable values: z – which has the value assigned to z in N’s input attribute, input[N].z, here called z[N] – and x’ – which has the value assigned to y in input[N], written y[N]. Similarly, (65) specifies for the attribute key at cell n the two non-null state variable values z : C_z, x’ : x[n]. For cell N’s query to match cell n’s key, both non-null-valued state variables must match: for z values to match, we must have z[N] == C_z, and for x’ values to match, we must have y[N] == x[n]. Thus these two values in q[N] and k[n] implement the desired Condition.

Any number of equality constraints may be present in the Condition of a PSL production, and for each such constraint, we simply insert into the state structure for query and key the state-variable values given in (65). This is amply exemplified in (99) – (100).

The value attribute is straightforward: for a pair of cells n, N for which the Condition is met, (65) specifies that value[n] is the state structure in which state variable u is assigned the value that w has in input[n] – this is notated “u : w[n]”. When the cells n, N meet the Condition, the Action uses this state structure to determine output[N], which is the same as input[N] except that the value of state variable u in this structure is set to w[n], over-writing any value u may have in input[N]. As with the Condition, the Action of a production may have any number of assignments of the form u[N] := w[n], and for each one, u : w[n] is inserted into value[n].

As mentioned in the Comment column, the translations in (65) also cover the cases in which y[N] is replaced with a constant or a transformed value y[N]@F in a Condition, or w[n] is replaced by a constant in an Action. In fact the specifications in (65) produce a translation for any PSL production having the form (60). The extensions in (61) are handled as follows.

Inequality conditions. When an inequality condition in a PSL production $z[N] \neq C_z$ is translated into QKVL, q is specified as $z : z$, and k as $z \neq C_z$. The semantics of ‘!’ here is that this key does not match a q[N] for any position N where $z[N] = C_z$. Similarly, the PSL inequality condition $z[n] \neq C_z$ is translated into QKVL by specifying k as $z : z$ and q as $z \neq C_z$; this q fails to match k[n] at any position n where $z[n] = C_z$.

Operators in Conditions. Already treated in (65).

Repeat blocks. Just like PSL, QKVL allows a sequence of layers to be tagged as a repeat block which functions just as in PSL.

This covers all the PARSE and GEN productions given in (99) – (100).

7 TPF, Implementational Level: DAT, a Discrete-Attention-Only Transformer Network

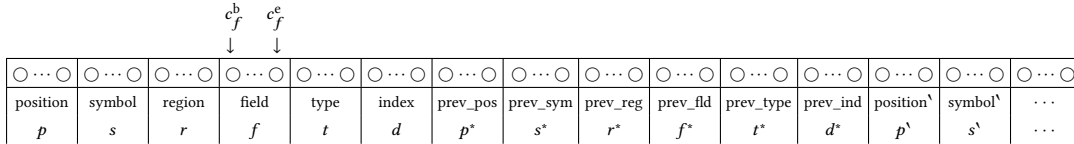
We are finally ready to produce our Discrete-Attention-only Transformer network, DAT, which computes ICL functions in \mathcal{F} . We need to convert the discrete symbolic instructions of QKVL into matrices of numerical weights that generate the vectors **q**, **k**, **v**. For this we need an embedding of the discrete state structures of QKVM in SSS into a vector space of transformer hidden states, V_{SSS} .

7.1 Embedding Abstract Machine Cell-State Structures as Transformer-Cell Vector States

- (66) Embedding state structures $s \in \text{SSS}$ as vectors $\mathbf{s} \in V_{\text{SSS}}$: Fully local (1-hot) case
- \mathbf{s} is a concatenation of vectors \mathbf{v}_x over all state variables x . Within \mathbf{s} , \mathbf{v}_x begins with neuron c_x^b and ends with neuron c_x^e : this is the x -register V_x within \mathbf{s} .
 - The possible values $\{x_i\}_{i=1}^{d_x} \equiv V_x$ of variable x are encoded as 1-hot (i.e., localist) vectors $\{\vec{x}_i\}_{i=1}^{d_x} \subset \mathbb{R}^{d_x}$. We can order the neurons so that \vec{x}_i is the i^{th} coordinate vector: $[\vec{x}_i]_j = \delta_{ij}$.
 - The vector encoding of the binding of variable x to the value $x_i \in V_x$ is $\vec{x} : \vec{x}_i$; this is the 1-hot vector \vec{x}_i located in the register V_x for variable x (from neuron c_x^b to neuron c_x^e).
 - If there are M state variables encoded in the hidden state, $\{v_m\}_{m=1}^M$, and variable v_m has d_{v_m} possible values, then $V_{\text{SSS}} \equiv \bigoplus_{m=1}^M V_{v_m} \cong \bigoplus_{m=1}^M \mathbb{R}^{d_{v_m}} \cong \mathbb{R}^D$ where $D \equiv \sum_{m=1}^M d_{v_m}$.
 - This can be analyzed as a tensor product representation (TPR) with 1-hot embeddings of the state variables serving as TPR *role vectors* and 1-hot embeddings of state variable values serving as TPR *filler vectors*.

For explanation of the TPR analysis of this embedding, see Online Appendix 4. The TPR analysis is helpful for showing how our analysis of DAT generalizes from the fully local embedding of (66) to more distributed embeddings (69). Although we do not explicitly use these distributed embeddings for the hand-programming reported here, the analysis applies equally well to these embeddings, and is likely to prove necessary for probing for state vectors in embeddings that are learned by transformer models: these are expected to be distributed, not local (see discussion in Sec. 9.2). In the body of the paper we will restrict attention to the simplest case, the fully local embedding, which is visualized in (67). This is essentially the ‘disentangled residual stream’ used in the RASP implementations of Lindner et al. (2023) and Friedman et al. (2023).

- (67) Fully local embedding of SSS visualized



- (68) Properties of the fully local embedding of SSS in V_{SSS}
- In this embedding, a state variable x (a TPR role) corresponds to a subspace V_x of V_{SSS} : the subspace spanned by the unit vectors for dimensions c_x^b through c_x^e .
 - The subsequence of neurons encoding a variable x – spanning from neuron c_x^b through neuron c_x^e – constitutes the x -register within the overall vector \mathbf{s} .
 - When two state variables x and y have the same set of possible values $V = \{v_1, \dots, v_d\}$, there is an isomorphism between $V_x \cong \mathbb{R}^d$ and $V_y \cong \mathbb{R}^d$: for any value $v_i \in V$, $\vec{x} : \vec{v}_i \in V_x$ corresponds to $\vec{y} : \vec{v}_i \in V_y$, these being the same vector \vec{v}_i (the embedding of v_i – the i^{th} coordinate vector – in \mathbb{R}^d) located within the x and y registers, respectively. This is what it means for x and y to “have the same value”. (In the TPR analysis, x and y are roles bound to the same filler v_i ; see Online Appendix 4).
- (69) Distributed embeddings summary – from Online Appendix 4
- There are two types of distributed embeddings which are isomorphic, under orthogonal linear transformations, to the fully local embedding of (66), and under the TPR analysis take the identical form, with only these differences:

- a. Semi-local orthonormal embedding: identical, except the filler vectors \vec{v}_i are orthonormal (but not necessarily 1-hot). There are localized registers, but the vector embeddings of values for the state variables $\{\vec{v}_i\}_{i=1}^{d_x}$ are, in general, dense distributed – not 1-hot – vectors.
- b. Fully distributed orthonormal embedding: identical, but the role vectors as well as the filler vectors are orthonormal (and not necessarily 1-hot). When the role vectors are not 1-hot, there are no localized registers: all state variables' embeddings are superimposed over the complete set of neurons spanning V_{SS} . In this case, the hypothesis that a learned hidden representation has the form specified here requires special methods for uncovering full distributed TPRs: see the discussion of the DISCOVER technique in Sec. 9.2.

7.2 The DAT Architecture

To perform ICL, the DAT is given as input an appropriate prompt: a symbol string of variable length P , with $s[p] \in S$ being the type of the symbol in position $p \in \{1, \dots, P\} \equiv P$, and S the vocabulary of symbol types. The input layer of DAT, like that of a decoder-only transformer, extends beyond the P *prompt cells* hosting the prompt; the layer extends up to a fixed total of T cells, with the cells following the initial P prompt cells – the *continuation cells* – ready to host the continuation string that constitutes the model's output.

- (70) Discrete-Attention-only Transformer (DAT) state dynamics: Parallel processing of the prompt
 - a. Machine states
 - i. Layers $1, \dots, L$, each a sequence of subnetworks called *cells* with positions $1, \dots, T$
 - ii. Each cell has a state specified by 5 *attribute vectors*, each in a vector space V_{SS} :
 - input vector \mathbf{i}
 - query, key, value vectors $\mathbf{q}, \mathbf{k}, \mathbf{v}$
 - output vector \mathbf{o}
 - iii. The query, key and value vectors for any cell are affine transformations of its input vector. The weights in these transformations are the parameters specifying the layer (they are constant throughout a layer).
 - iv. Unlike standard transformers, there are no MLP or LayerNorm sublayers; DAT is an attention-only transformer architecture (with a single attention head).
 - b. Inputs in layer 1
 - i. The layer-1 prompt cell with position p has input vector $\mathbf{i}^{(1)}[p] = \overrightarrow{s : s[p]} + \overrightarrow{p : p} + \overrightarrow{a : 1}$ where $s[p]$ is the type of the symbol at position p , and a is the parse flag which is initialized to 1 for each prompt cell. (These are the cells subject to the productions of PARSE given in (99).)
 - ii. The input vector of the layer-1 cell carrying the final symbol of the prompt has an additional variable set, the end-of-prompt flag variable z which is set to EOP: $z : \overrightarrow{EOP}$.
 - iii. Note that in the local embedding, $\overrightarrow{s : s[p]}$ is the 1-hot vector embedding of $s[p]$ in the s -register V_s and $\overrightarrow{p : p}$ is the 1-hot vector embedding of p in the p -register V_p ; summation of these vectors in two orthogonal subspaces is equivalent to concatenation.
 - c. Cell-state update computation: Discrete Attention (single-headed)
 - i. *Attention vectors; DATmax.* In any layer, cell N 's output vector $\mathbf{o}[N]$ is determined by (i) the cell's input vector $\mathbf{i}[N]$ and (ii) the value vector $\mathbf{v}[n]$ of a cell n with a key vector $\mathbf{k}[n]$ that matches the non-null variable values encoded in the query vector $\mathbf{q}[N]$ of cell N ; this n is given by the function

α defined as follows. For all $N = 1, \dots, T$, let the raw and normalized query-key dot-product vectors $\delta[N]$, $\hat{\delta}[N] \in \mathbb{R}^T$ have components, for $n = 1, \dots, T$:

$$\delta[N]_n \equiv \mathbf{q}[N] \cdot \mathbf{k}[n] \quad \hat{\delta}[N]_n = \delta[N]_n / \|\mathbf{q}[N]\|^2$$

Then, for the following reason, we define

$$\alpha(N) \equiv \min \{n \in \{1, \dots, T\} \mid \hat{\delta}[N]_n = 1\}$$

Because of the 1-hot encodings we are using in V_{SSS} , $\mathbf{q}[N] \cdot \mathbf{k}[n]$ is the number of variable registers set in $\mathbf{q}[N]$ to non-null values for which the corresponding variable registers in $\mathbf{k}[n]$ have the same value; the total number of such non-null values in $\mathbf{q}[N]$ is just $\|\mathbf{q}[N]\|^2$, which means that $\hat{\delta}[N]_n \leq 1$, with equality holding only when the desired perfect matching occurs between the query and key vectors. Hence the requirement that $\hat{\delta}[N]_n = 1$.¹² If no perfectly matching n exists, $\alpha[N]$ is undefined, and attention at N will return $\mathbf{0}$; if multiple exact matches occur, $\alpha[N]$ is (by default) the first/leftmost match: hence the min over n . (In the non-default case of a layer ℓ with $\text{right_match}(\ell) = \top$, in the definition of α , min is replaced with max.)¹³ So α implements exactly the attentional selection process of the QKV machine (62c). Thus the attention-weight vector computed in standard transformers by the softmax over normalized query-key dot-products is replaced in DAT by

$$\text{DATmax}(\hat{\delta}[N]) \equiv \mathbf{1}\text{-hot}(\alpha(N))$$

where $\mathbf{1}\text{-hot}(n)$ is the one-hot vector in \mathbb{R}^T with value 1 at location n .

- ii. *Inequality conditions.* Recall from Sec. 6.2 that a PSL production with an inequality condition $z[N] \neq z_i$ is translated into QKVL by specifying \mathbf{q} as $z : z_i$, and \mathbf{k} as $z \neq z_i$; this key does not match a query for which the value of z is z_i . In DAT, for all cells n in the layer implementing this production, the activity of each neuron μ in the z register of $\mathbf{k}[n]$ is set to $[\mathbf{k}[n]]_\mu = 1 - [\vec{\mathbf{v}}_i]_\mu$, the ones'-complement of $\vec{\mathbf{v}}_i$. Within the z -register, this will yield a raw-attention dot-product $\delta[N]_n \equiv \mathbf{q}[N] \cdot \mathbf{k}[n]$ of 0 when $\mathbf{q}[N] = \vec{\mathbf{v}}_i$, but a dot product of 1 whenever $\mathbf{q}[N] = \vec{\mathbf{v}}_j$ with $j \neq i$.
- iii. *DATnorm.* Given a vector $\mathbf{u} \in V_{SSS}$, define $\text{DATnorm}(\mathbf{u})$ so that for each state variable x , the register subspace V_x — hosted by neurons c_x^b through c_x^e — is independently normalized within $\text{DATnorm}(\mathbf{u})$ to the 1-hot vector with activity 1 at the neuron in position $\text{argmax}_{i \in V_x} \{\mathbf{u}_i\}$, where ' $i \in V_x$ ' abbreviates ' $c_x^b \leq i \leq c_x^e$ '; if $\mathbf{u}_x = \mathbf{0}$, we set $\text{DATnorm}(\mathbf{u}_x) = \mathbf{0}$. As used in the DAT architecture in (70c-iv), when $\mathbf{u} \neq \mathbf{0}$, there will never be a tie for the maximum element of a vector $\{\mathbf{u}_i\}_{i \in V_x}$ used as an argument to argmax for computing DATnorm . Note that DATnorm operates within each register independently, quite unlike LayerNorm . Thus the register structure within DAT's hidden (attribute) vectors impose real structure on DAT computation.

¹²The DAT programs produced by our compiler will always yield $\|\mathbf{q}[N]\| = \|\mathbf{k}[n]\|$ for all N, n in the same layer. Thus we can equivalently define $\hat{\delta}[N]_n = \hat{\mathbf{q}}[N] \cdot \hat{\mathbf{k}}[n]$ while normalizing both \mathbf{q} and \mathbf{k} to unit length. For extension of these mathematical properties of attention in DAT to non-one-hot encodings, see Online Appendix 4.

¹³In future work exploring in-weights learning in the DAT, it may improve gradient propagation to implement left- (or right-)mostness differentially as follows. Suppose that before computing final attention weights for N we preprocess the normalized key-query dot-product vector $\hat{\delta}[N]$ by setting to 0 all values at n with $\hat{\delta}[N]_n < 1$, e.g. via $\hat{\delta}[N] - \text{ReLU}(\vec{\mathbf{1}} - \hat{\delta}[N])$; then it remains to pick the left- (or right-)most of the remaining $\hat{\delta}[N]$ values (if any). Consider the leftmost case. We add to $\hat{\delta}[N]_n$ a quantity $\lambda(n)$ which smoothly decreases as n increases and which has an overall magnitude small enough to ensure that the addition of λ will break exact ties in $\hat{\delta}[N]$ but never alter the location of $\text{argmax}_n \{\hat{\delta}[N]_n + \lambda(n)\}$. Using a new state variable λ , this could be arranged by inserting, for small enough ϵ , $\lambda(n) : \epsilon/n$ into $\text{key}[n]$ and $\lambda : 1$ into $\text{query}[N]$ for all N (n is the numerical value of the position variable $\text{input}[n].p$). An analogous maneuver could yield right-mostness via a new state variable ρ assigned small values that smoothly *increase* with n . Then each production could select for left- or right-most attention by specifying either $\lambda : 1$ (and ρ set to null) or $\rho : 1$ (λ null), respectively, in its particular query. With this modification in place, DATnorm could be differentially approximated by softmax with a low temperature parameter, returning at N a nearly-1-hot vector of attention weights, 'hot' at $\text{argmax}_n \{\delta[N]_n + \lambda(n)\}$ (or $\text{argmax}_n \{\delta[N]_n + \rho(n)\}$), the left- (or right-)most location n of a 1 in $\hat{\delta}[N]_n$.

iv. *The output vector* $\mathbf{o}[N]$ of a cell N is given by

$$\mathbf{o}[N] = \text{DATnorm}(\mathbf{i}[N] + \kappa \mathbf{v}[n]) \quad n \equiv \alpha(N)$$

where $\mathbf{v}[n]$ is the value attribute vector of cell n ; α is defined in (70c.i), and if it is undefined, we have simply $\mathbf{o}[N] = \text{DATnorm}(\mathbf{i}[N]) = \mathbf{i}[N]$. We can choose any $\kappa > 1$; we let $\kappa = 2$. (As in standard transformers, the term $\mathbf{i}[N]$ is the contribution of a residual connection which adds the input vector to the output.) Three cases exist for each state variable x :

if $\mathbf{v}[n]$ does not have x set to a non-null value, then – within register V_x – $\mathbf{v}[n] = \mathbf{0}$ and therefore $\mathbf{o}[N] = \mathbf{i}[N]$;

if $\mathbf{v}[n]$ has x set to a non-null value v_k , and in $\mathbf{i}[N]$, x has a different non-null value v_j , then within register V_x , $\mathbf{o}[N] = \vec{v}_k$, the 1-hot embedding of v_k . This is because, in the x register, $\mathbf{i}[N] + \kappa \mathbf{v}[n]$ has value 1 on the j^{th} neuron (from $\mathbf{i}[N]$) and value $\kappa = 2 > 1$ on the k^{th} neuron (from $\mathbf{v}[n]$), so the DATnorm operation resets the former to 0 and the latter to 1.

if $\mathbf{v}[n]$ has x set to a non-null value v_k that is the same value as $\mathbf{i}[n].x$, then $\mathbf{i}[N] + \kappa \mathbf{v}[n] = (1 + 2)\vec{v}_k$ which is similarly reset by DATnorm to simply \vec{v}_k .

Thus this implements the update rule for the QKV machine (62c).

d. Cell updating sequence

- i. The prompt cells are all updated in parallel; the input attribute vector $\mathbf{i}^{(2)}[p]$ of the cell in position p of layer 2 is the output attribute vector $\mathbf{o}^{(1)}[p]$ for the cell in position p of layer 1. This is repeated for all layers.
- ii. A sub-sequence of layers can constitute a repeat block, in which case the layers in that block are evaluated in sequence repeatedly until a termination condition specified for that block is met.

(71) DAT Machine state dynamics: Autoregressive generation of the continuation [as for QKVM (63)].

- a. Given a prompt of length P , the level-1 states of cells with $p = 1, \dots, P$ are determined by the prompt (70b.i). These are processed in parallel according to (70). The continuation cells are updated autoregressively. The level-1 state of cell $P + 1$ is the level- L state of cell P (except that $p := P + 1$). The values of all state variables (except p) are copied from P to $P + 1$, not just the symbol variable s .
- b. Cell $P + 1$ is processed through all L layers, and then the level-1 state of cell $P + 2$ is set equal to the level- L state of cell $P + 1$ (except that $p := P + 2$). This process iterates until a termination condition is met (e.g., the generation of a special termination symbol).
- c. The generated continuation string is read from the level-1 continuation cells as the sequence of symbols that are embedded as the vectors $\mathbf{i}^{(1)}[p].s$, $p = P, P + 1, \dots$

A DATL program specifies a sequence of layers, some of which may be marked as forming a repeat block with a specified termination condition. Each layer ℓ is specified by a set of query/key/value weights, which determine each of a cell n 's attribute vectors $\mathbf{q}[n]$, $\mathbf{k}[n]$, $\mathbf{v}[n]$ through an affine transformation of the cell's input vector $\mathbf{i}[n]$ via a matrix $\mathbb{W}_{\mathbf{q},\mathbf{k},\mathbf{v}}^{(\ell)}$ and a bias vector $\mathbf{b}_{\mathbf{q},\mathbf{k},\mathbf{v}}^{(\ell)}$ that implement the mappings $W_{q,k,v}$ of (62d-i).¹⁴

7.2.1 Compiling from QKVL to DATL. As spelled out in (72), compiling a QKVL program produces a set of affine-transformation weights (in a matrix and bias vector): for each layer, these transform a cell's input vector \mathbf{i} into its query/key/value attribute vectors \mathbf{q} , \mathbf{k} , \mathbf{v} . Compiling proceeds layer by layer as follows. A QKVL instruction for a layer, a *target-variable* : *source-variable* pair such as $x : y$, is compiled into a set of weights

¹⁴The current default size of each bias vector in the `dat_explorer` simulator, Sec. 7.2.3, is 3936 (41 registers each of size 96); the size of each matrix is [3936, 3936].

that can be loaded into the DAT Transformer. Weights are built up starting from zero, adding for each QKVL instruction a contribution to the weight matrices and biases. $[\mathbb{W}_k]_x^y$ denotes the block submatrix (within the key-generating matrix \mathbb{W}_k) which maps the subspace V_y of V_{SSS} embedding values of a source variable y into the subspace V_x embedding values of the target variable x . \mathbf{b}_k is the constant bias term in the affine transformation mapping \mathbf{i} to \mathbf{k} : $\mathbf{k} = \mathbf{i}\mathbb{W}_k + \mathbf{b}_k$.

(72) Compiling from QKVL to DATL, for attribute \mathbf{k} (attributes \mathbf{q} , \mathbf{v} operate identically)

QKVL	DAT result	attribute vector	contribution to weights $\mathbb{W}_k, \mathbf{b}_k$
a. $x : y$	$\mathbf{k}[n].x := \mathbf{i}[n].y$	$\mathbf{k}[n] := \mathbf{i}[n] \mathbb{W}_k$	$[\mathbb{W}_k]_x^y = \mathbb{I}_{d_x}$ (block identity matrix $V_y \rightarrow V_x$) $d_x = \#$ possible values for $x = d_y$
b. $x : y@F$	$\mathbf{k}[n].x := (\mathbf{i}[n].y) \mathbb{F}$	$\mathbf{k}[n] := \mathbf{i}[n] \mathbb{W}_k$	$[\mathbb{W}_k]_x^y = \mathbb{F}$ $\mathbb{F} =$ matrix implementing value-transform F
c. $x : x_i$	$\mathbf{k}[n].x := \vec{x}_i$	$\mathbf{k}[n] := \mathbf{b}_k$	$[\mathbf{b}_k]_x = \vec{x}_i$ the x register V_x of \mathbf{b}_k is \vec{x}_i
d. $x != x_i$	$\mathbf{k}[n].x := \vec{1} - \vec{x}_i$	$\mathbf{k}[n] := \mathbf{b}_k$	$[\mathbf{b}_k]_x = \vec{1} - \vec{x}_i$ the ones'-complement of \vec{x}_i
e. $x != y$	$\mathbf{k}[n].x := \vec{1} - \mathbf{i}[n].y$	$\mathbf{k}[n] := \mathbf{i}[n] \mathbb{W}_k$	$[\mathbb{W}_k]_x^y = \mathbf{1} - \mathbb{I}_{d_x}$ the ones'-complement of \mathbb{I}_{d_x}

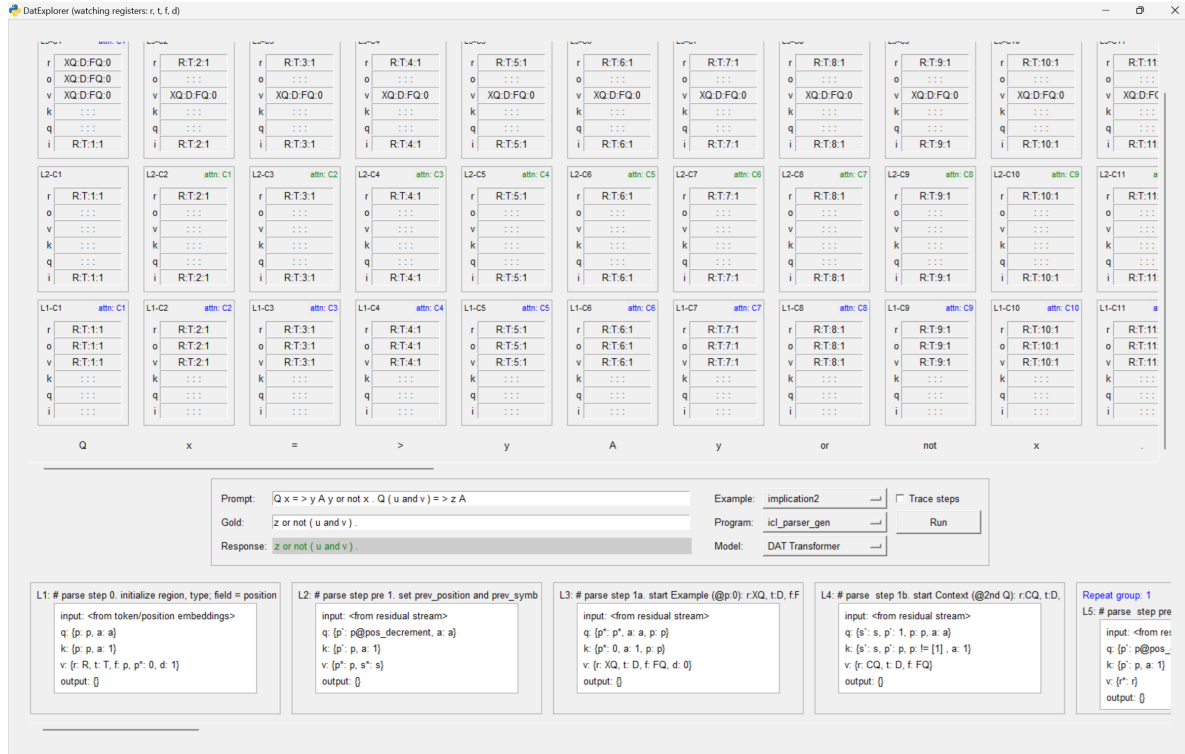
$\vec{1}$ is the vector with 1 in every position; thus $\vec{1} - \vec{x}_i$ is the ones'-complement of \vec{x}_i , the vector with 0 in position i and 1 in every other position: this matches the vector embedding of an x value \vec{x}_j if and only if $j \neq i$. $\mathbf{1}$ is the matrix of all 1s. (Note 11, page 30, points out that the \mathbb{F} needed for (72b) always exists, given a linearly-independent, e.g., one-hot, encoding of variable values.)

7.2.2 *DAT Operation.* Implementation details of the operation of the DAT, viewed as a modification of a standard transformer, are presented in App. F.

7.2.3 *The dat_explorer Simulator.* The TPF is implemented in a publicly-accessible python package on github;¹⁵ it consists of several programs: `psl_compiler`, `weights_compiler`, `dat_interpreter`, `dat_transformer`, and `dat_explorer`. The `dat_explorer` enables users to load a PSL program, compile it into a QKVL program, compile that into the numerical weights for a DAT, enter a prompt and run the DAT, visualizing each of its resulting cell states (for each column and layer) using the symbolic interpretation of vectors and matrices.

Here is a screenshot of the program on the propositional-logic inference prompt (4):

¹⁵https://github.com/microsoft/discrete_attn_transformer/



Visualization features include:

- pull-down menu for selecting a PSL program for producing a prompt continuation
- scrollable view of columns and layers of the DAT running the compiled program, displaying each cell as a box containing a row for each cell attribute
- a selectable set of watch-registers that are displayed symbolically in each cell of a layer
- dynamically built tooltips to see all registers of a cell row
- locus of attention $\alpha(N)$ for each cell N (70c.i)
- currently-active prompt, with generated and gold continuation
- scrollable view of the running QKVL code (by production)

7.2.4 *Experiments.* We tested our DAT implementation of the PARGEN algorithm using the following prompts:

Name	Prompt	Gold Continuation
tgt 1	Q the big bear ? a red car A the big bear \$. Q some baby cub ? one small house A	some baby cub \$.
tgt 2	Q - his green bird A his green bird . Q - some light monkey A	some light monkey .
j loves m	Q john loves mary A mary hugs john . Q sue loves bill A	bill hugs sue .
act2pass	Q x V y A y Was V By x . Q u V w A	w Was V By u .
act2pass 2	Q x V y A y Was V -en By x . Q u1 u2 V w1 w2 w3 A	w1 w2 w3 Was V -en By u1 u2 .
swap	Q B V D E A D E V B . Q F G H V K L A	K L V F G H .
cross mult	Q (x / y) /// ((u // v)) A (x * v) / ((y ** u)) . Q (a / b) /// ((c // d)) A	(a * d) / ((b ** c)) .
implica	Q x => y A y or not x . Q (u and v) => z A	z or not (u and v) .
implica 2	Q x = > y A y or not x . Q (u and v) = > z A	z or not (u and v) .

The correct completion sequence for all of these prompts was predicted by the DAT.¹⁶ In addition to the above examples, *DAT achieved 100% accuracy* when tested on the first 1000 examples from each of the following splits of the 1_shot_rlw task: test, ood_lexical, ood_cons_len_7, and ood_cons_count_7. Recall that learned models struggled with these OOD tests (Sec. 4.2.3).

8 Universality of the Framework: TPF is Turing-Complete

How general is the framework we have developed? Specifically, we return to the question raised in Sec. 1.4: how general is the class of functions that can be computed by PSL programs and thereby with DAT transformer networks?¹⁷ 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 (73).

(73) 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: $\sigma_0, S_0 \Rightarrow \sigma_1, S_1, \delta$

¹⁶Note regarding cross mult: It is currently not possible to give the template in the natural form $Q(x/y)/(u/v)A(x*v)/(y*u)$. Multiple distinct symbols for parentheses and division — $($, $)$, and $/$, etc. — must be used because, in the current initial state of work in the TPF, the space \mathcal{F} of ICL functions studied constrains prompts so that every symbol type occurs in only one field type within a single Q- or A-region (32c.i), and each field type occurs only once within a given Q- or A-region (32b.i). This is also manifest in (7). Note that ‘**’ is a variant of ‘*’ intended to denote multiplication, not exponentiation.

¹⁷We thank Rick Lewis for pointing the way towards the results presented here.

8.1 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 (73) 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 production 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. 5), one cell of the PSM for each cell of the tape. The three state variables we use in the PSM are given in (74).

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

As illustrated in (80) below, the Turing-table instruction (73) $[\sigma_0, S_0 \Rightarrow \sigma_1, S_1, \delta]$ 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 (76).

- (75) *Production* P_1 . In the currently active tape position, update the state and symbol
- Condition: $c[N] == 1, \sigma[N] == \sigma_0, s[N] == S_0$
 - Action: set $c[N] := \delta, \sigma[N] := \sigma_1, s[N] := S_1$
- (76) *Production* $P_2(L)$. Move head left, update state there
- Condition: $c[n] == L, p[n] == p[N]@pos_increment$
 - Action: set $c[N] := 1, \sigma[N] := \sigma[n]$
- (77) *Production* $P_2(R)$. Move head right, update state there
- Condition: $c[n] == R, p[n] == p[N]@pos_decrement$
 - Action: set $c[N] := 1, \sigma[N] := \sigma[n]$
- (78) *Production* P_3 . Remove moved-head mark¹⁸
- Condition: $c[N]$ in [L, R]
 - Action: set $c[N] := 0$
- (79) *Production* P_4 . Broadcast new state globally
- Condition: $c[n] == 1$
 - Action: set $\sigma[N] := \sigma[n]$

¹⁸The Condition utilizes an option in the PSL syntax described in (61f), 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**.)

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 in the middle position — is provided in (80).

(80) Example of a Turing-Machine update

TM-table-cell instruction: $\sigma_0, B \Rightarrow \sigma_1, X, R$

a.	<table style="border-collapse: collapse; display: inline-table;"> <tr> <td style="border: none; padding-right: 5px;">s</td> <td style="border: 1px solid black; padding: 2px; text-align: center;">A</td> <td style="border: 1px solid black; padding: 2px; text-align: center;">B</td> <td style="border: 1px solid black; padding: 2px; text-align: center;">C</td> <td style="border: none; padding-left: 5px;">start</td> </tr> <tr> <td style="border: none; padding-right: 5px;">σ</td> <td style="border: 1px solid black; padding: 2px; text-align: center;">σ_0</td> <td style="border: 1px solid black; padding: 2px; text-align: center;">σ_0</td> <td style="border: 1px solid black; padding: 2px; text-align: center;">σ_0</td> <td style="border: none;"></td> </tr> <tr> <td style="border: none; padding-right: 5px;">c</td> <td style="border: 1px solid black; padding: 2px; text-align: center;">0</td> <td style="border: 1px solid black; padding: 2px; text-align: center;">1</td> <td style="border: 1px solid black; padding: 2px; text-align: center;">0</td> <td style="border: none;"></td> </tr> </table>	s	A	B	C	start	σ	σ_0	σ_0	σ_0		c	0	1	0	
s	A	B	C	start												
σ	σ_0	σ_0	σ_0													
c	0	1	0													

b.	<table style="border-collapse: collapse; display: inline-table;"> <tr> <td style="border: none; padding-right: 5px;">s</td> <td style="border: 1px solid black; padding: 2px; text-align: center;">A</td> <td style="border: 1px solid black; padding: 2px; text-align: center;">X</td> <td style="border: 1px solid black; padding: 2px; text-align: center;">C</td> <td style="border: none; padding-left: 5px;">P_1</td> </tr> <tr> <td style="border: none; padding-right: 5px;">σ</td> <td style="border: 1px solid black; padding: 2px; text-align: center;">σ_0</td> <td style="border: 1px solid black; padding: 2px; text-align: center;">σ_1</td> <td style="border: 1px solid black; padding: 2px; text-align: center;">σ_0</td> <td style="border: none;"></td> </tr> <tr> <td style="border: none; padding-right: 5px;">c</td> <td style="border: 1px solid black; padding: 2px; text-align: center;">0</td> <td style="border: 1px solid black; padding: 2px; text-align: center;">R</td> <td style="border: 1px solid black; padding: 2px; text-align: center;">0</td> <td style="border: none;"></td> </tr> </table>	s	A	X	C	P_1	σ	σ_0	σ_1	σ_0		c	0	R	0	
s	A	X	C	P_1												
σ	σ_0	σ_1	σ_0													
c	0	R	0													

c.	<table style="border-collapse: collapse; display: inline-table;"> <tr> <td style="border: none; padding-right: 5px;">s</td> <td style="border: 1px solid black; padding: 2px; text-align: center;">A</td> <td style="border: 1px solid black; padding: 2px; text-align: center;">X</td> <td style="border: 1px solid black; padding: 2px; text-align: center;">C</td> <td style="border: none; padding-left: 5px;">$P_2(R)$</td> </tr> <tr> <td style="border: none; padding-right: 5px;">σ</td> <td style="border: 1px solid black; padding: 2px; text-align: center;">σ_0</td> <td style="border: 1px solid black; padding: 2px; text-align: center;">σ_1</td> <td style="border: 1px solid black; padding: 2px; text-align: center;">σ_1</td> <td style="border: none;"></td> </tr> <tr> <td style="border: none; padding-right: 5px;">c</td> <td style="border: 1px solid black; padding: 2px; text-align: center;">0</td> <td style="border: 1px solid black; padding: 2px; text-align: center;">R</td> <td style="border: 1px solid black; padding: 2px; text-align: center;">1</td> <td style="border: none;"></td> </tr> </table>	s	A	X	C	$P_2(R)$	σ	σ_0	σ_1	σ_1		c	0	R	1	
s	A	X	C	$P_2(R)$												
σ	σ_0	σ_1	σ_1													
c	0	R	1													

d.	<table style="border-collapse: collapse; display: inline-table;"> <tr> <td style="border: none; padding-right: 5px;">s</td> <td style="border: 1px solid black; padding: 2px; text-align: center;">A</td> <td style="border: 1px solid black; padding: 2px; text-align: center;">X</td> <td style="border: 1px solid black; padding: 2px; text-align: center;">C</td> <td style="border: none; padding-left: 5px;">P_3</td> </tr> <tr> <td style="border: none; padding-right: 5px;">σ</td> <td style="border: 1px solid black; padding: 2px; text-align: center;">σ_0</td> <td style="border: 1px solid black; padding: 2px; text-align: center;">σ_1</td> <td style="border: 1px solid black; padding: 2px; text-align: center;">σ_1</td> <td style="border: none;"></td> </tr> <tr> <td style="border: none; padding-right: 5px;">c</td> <td style="border: 1px solid black; padding: 2px; text-align: center;">0</td> <td style="border: 1px solid black; padding: 2px; text-align: center;">0</td> <td style="border: 1px solid black; padding: 2px; text-align: center;">1</td> <td style="border: none;"></td> </tr> </table>	s	A	X	C	P_3	σ	σ_0	σ_1	σ_1		c	0	0	1	
s	A	X	C	P_3												
σ	σ_0	σ_1	σ_1													
c	0	0	1													

e.	<table style="border-collapse: collapse; display: inline-table;"> <tr> <td style="border: none; padding-right: 5px;">s</td> <td style="border: 1px solid black; padding: 2px; text-align: center;">A</td> <td style="border: 1px solid black; padding: 2px; text-align: center;">X</td> <td style="border: 1px solid black; padding: 2px; text-align: center;">C</td> <td style="border: none; padding-left: 5px;">P_4</td> </tr> <tr> <td style="border: none; padding-right: 5px;">σ</td> <td style="border: 1px solid black; padding: 2px; text-align: center;">σ_1</td> <td style="border: 1px solid black; padding: 2px; text-align: center;">σ_1</td> <td style="border: 1px solid black; padding: 2px; text-align: center;">σ_1</td> <td style="border: none;"></td> </tr> <tr> <td style="border: none; padding-right: 5px;">c</td> <td style="border: 1px solid black; padding: 2px; text-align: center;">0</td> <td style="border: 1px solid black; padding: 2px; text-align: center;">0</td> <td style="border: 1px solid black; padding: 2px; text-align: center;">1</td> <td style="border: none;"></td> </tr> </table>	s	A	X	C	P_4	σ	σ_1	σ_1	σ_1		c	0	0	1	
s	A	X	C	P_4												
σ	σ_1	σ_1	σ_1													
c	0	0	1													

Thus we have:

(81) $DAT(TM_f)$ 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.

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 (81) establishes that, despite the focus here on the particular Templatic Generation Task defined in Sec. 4, the TPF has potentially broad relevance for mechanistic interpretation of transformer computation (see Sec. 9.3.3).

8.2 DAT(UTM): DAT Implementation of a Universal TM

The approach presented in Sec. 8.1 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 s , and can write on the tape the value $f(s)$. 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 s as the remainder of the prompt. The completion generated by the machine is then $f(s)$. 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. The analogy is even closer for the task of Nested Function Evaluation presented below (89).

In detail, for computing $f(s)$, the prompt is specified as in (82). 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 $\Sigma_0, \Sigma_1, S_0, S_1$, 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 s for computing $f(s)$.

(82) Universal Turing-Machine input prompt for PSM(UTM) for computing $f(s)$

a. Prefix (program)

i. For each TM_f instruction: $\sigma_0, S_0 \Rightarrow \sigma_1, S_1, \delta$ (73),

ii. assign initial state-variable values of a single cell of the PSM prompt:

$$\Sigma_0 : \sigma_0, S_0 : S_0, \Sigma_1 : \sigma_1, S_1 : S_1, \Delta : \delta$$

b. Suffix (data)

Encode the argument-string 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 (75 – 79), except that the one production that encodes the content of a TM_f table instruction, P_1 , is replaced by the production U_1 (83) 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.

(83) *Production U_1* . Same as *Production P_1* (75) 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 Σ_0, S_0 match those of the currently-updating prompt-suffix cell N (emulating the current TM-head position)

a. Condition: $c[N] == 1, \Sigma_0[n] == \sigma[N], S_0[n] == s[N]$

b. Action: $\sigma[N] := \Sigma_1[n], s[N] := S_1[n], c[N] := \Delta[n]$

This establishes a second universality result for TPF:¹⁹

(84) *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.

9 Discussion and Future Work: Mechanistic Interpretation and Enhancement of Transformers

9.1 What have we Learned?

A number of findings were previewed in the theoretical summary (17), which we reprise here, adding pointers to the relevant discussion in the paper.

¹⁹During the finalization of this paper, Schuurmans et al. 2024 presented 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.

(85) How can ICL in a transformer perform symbolic templatic text generation?

Via the following [transformer element] ~ [symbolic element] correspondences:

- a. a cell's residual stream ~ a variable-value structure (67)
 - i. a subspace of the residual stream ~ a state variable
 - ii. a vector component within a variable's subspace ~ a value of that variable
- b. a layer's internal connections ~ a production (65 – 72)
 - i. query-key matching in attention ~ evaluating the condition of the layer's production
 - ii. value vectors ~ the production's action
 - iii. query-key matching on a subspace corresponding to a goal ~ conditional branching for goal-directed action (Sec. 5.5)
- c. a nested set of structural variables ~ hierarchical data structure (36)
 - i. sharing the value of a level- l structural variable ~ in the same (type of) level- l constituent (adopted from Hinton, 2023)
- d. a sequence of structural-variable values (at the 'field' level) ~ the sequence of abstract roles defining a template (93)

A high-level summary of our results, with pointers to relevant sections of the paper, is given in (86).

(86) A type of transformer — a Discrete-Attention-only Transformer (DAT) can:

- a. encode a cell state as a vector-embedded structure that encodes the values for a set of symbolic state-variables (Sec. 7.1);
- b. employ this same structure for queries, keys and values (62d);
- c. encode, in a cell's state, abstractions like the role of a symbol in a parse of the prompt, via a set of structural variables; encode a hierarchical parse structure in the state of an entire layer (Sec. 5.2);
- d. use simple weight matrices to generate \mathbf{q} , \mathbf{k} , \mathbf{v} vectors that implement symbolic productions: condition/action rules that read and write values of state variables (Sec. 7.2.1);
- e. implement a **higher-level symbolic programming language for transformer computation**, PSL (Sec. 5);
- f. implement programs in this language which parse the prompt, e.g., as a template (Sec. 5.4);
- g. implement programs in this language for generating text that is sensitive to this parse structure, including a type of ICL (Sec. 5.3).

9.2 Mechanistic Analysis of Symbol Processing in Trained Transformer Models

The TPF results on how to construct a transformer to perform templatic text generation through ICL — answering our questions (13) — suggests many hypotheses about how *trained* transformers perform this task; these hypotheses are all precisely specified and so formally testable. The hypotheses fall into two groups: those based on the specific algorithm for TGT, PARGEN, that we have programmed into our system, Sec. 9.2.1, and those based on the general ICL framework formalized by TPF, Sec. 9.2.2. The universality results of Sec. 8 encourage us to apply these latter hypotheses to networks trained for tasks other than TGT — e.g, LMs.

9.2.1 Specific Hypotheses Concerning TGT.

- (87) Specific PARGEN-algorithm-derived hypotheses about how trained transformers perform TGT
- a. Performing the operation of a single production — two classes of testable hypotheses for each of 29 productions P :
 - i. attention pattern: the loci of some head’s maximal attention for symbols in some layer are fit by the attention pattern produced by P ;
 - ii. attribute vectors: the \mathbf{i} , \mathbf{q} , \mathbf{k} , \mathbf{v} vectors of some head in some layer, when projected to an appropriate subspace, are fit, under an appropriate linear transformation, by the corresponding attribute vectors implementing P in the DAT.
 - b. Encoding the parse:
 - i. For each constituent, on some subspace of the residual stream, the projection on that subspace is the same for all symbols within that constituent.
 - ii. For each constituent type, on some subspace of the residual stream, the projection on that subspace is the same for all symbols within constituents of that type.

(87a-i) can be used as a kind of screening method for picking out which attention heads at which layers are candidates for interpretation as a given production. For (87a-ii), there is already publicly-available software to perform the analysis. Exploiting the framework of Tensor Product Representations (TPRs, Online Appendix 4), the DISCOVER package (McCoy et al. 2019) can take as input a set of hidden vectors from a ‘target model’ under analysis, as well as a hypothesis about the symbol structures that these vectors embed: here, a set of roles (variables) that have been assigned fillers (values). In the DISCOVER technique (the ‘decoder’ variant, McCoy, 2022, Sec. 7.3.1), a trainable auxiliary neural network takes the hidden vector to be interpreted and decomposes it according to the symbol structure hypothesized as its interpretation: this auxiliary network learns a vector embedding for the hypothesized roles and their possible fillers, and a linear transformation to map the target hidden vector to a space of the dimensionality of a TPR constructed using those vector embeddings. Optimization adjusts these embeddings, and the linear transformation, so that the structure resulting from decomposition best approximates the hypothesized symbol structure. If the result is a good approximation, then the hypothesis is confirmed and, given the hypothesized role:filler structure for any target hidden vector, we have a closed-form equation for a TPR which is the component of the hidden state lying in the subspace targeted by the DISCOVER interpretation.

The closed-form equation for the hidden vector as a TPR makes it possible to manipulate that vector by subtracting a role:filler pair’s contribution to that hidden vector and replacing it with the appropriate contribution for a different filler for that role, with the objective of making a controlled alteration of the output of the target model, thereby testing the causal role of the symbolic structure hypothesized as the interpretation of the hidden state. For example, in Soulos et al. 2020, studying the SCAN compositionality task, the output, initially correct for an original input jump twice after run left, changes to a new output that would be correct for jump thrice after run left, when the contribution of twice to the hidden vector is replaced with the contribution that would have been made by thrice; multiple such alterations can be performed in sequence to ultimately end up with the output appropriate for, say, walk thrice after look right. That is, the TPR approximation gave a purely formal analysis of the target model’s hidden encoding vector that provided a causally efficacious decomposition: altering the constituents in the internal encoding caused the model to decode it into just the novel output that is appropriate for the given altered constituents.

This same technique could be used to test the 29×4 fully-precisely-specified hypotheses stated in (87a-ii). Note that the ‘decoder’ variant of DISCOVER finds, within the representational space of the target vector being

interpreted, a subspace that can be linearly mapped to a TPR — the TPR need not explain the entire representational space, which is appropriate here because we expect our algorithm’s variable:value pairs to explain only part of the variance in the hidden vectors; the hidden vector may well encode many other properties as well: a LM’s internal representation, for example, will likely contain considerable semantic information along with the semantics-free information processed in our purely symbolic algorithm which provides the hypotheses (87a-ii). This is particularly expected because even a trained network that does encode something like our productions will not dedicate an entire layer to a single production; successfully trained transformers of Sec. 4.2.3 perform TGT with many fewer layers than the 29 deployed in our one-production- and one-head-per-layer compiled DAT.

The type of encoding structure hypothesized in (87a) has in fact been identified in pre-trained LLMs performing variable binding (X. Davies et al. 2023; Wu et al. 2025), entity tracking (Q. Dai et al. 2024; Feng and Steinhardt 2023), and belief-tracking (Prakash et al. 2025) tasks. For instance, Feng & Steinhardt 2023 consider tasks like ‘P lives in WA. G lives in MD. Where does P live?’ The example region of the prompt creates a binding problem, where P’s residence must be bound to WA, G’s residence must be bound to MD, and accurately answering ‘where does x live?’ requires retrieving the correct binding for x . Each of the two facts in the prompt involves a separate binding of person to place; Feng & Steinhardt found that the two facts were in essence assigned two identifiers (analogous to ‘fact_1’, ‘fact_2’) and the entities in each fact were bound to their corresponding identifier in the residual stream — by addition, with a particular orthogonal subspace dedicated to encoding the identifiers. These ‘identifier labels’ are embedded in the residual stream just as are our ‘region labels’, to which they directly correspond.

9.2.2 General Hypotheses Concerning ICL. Rather than testing the specific PARGEN-derived hypotheses realized in our DAT for TGT (87), we can reformulate the hypotheses in more general terms and use existing methods to discover structures hidden within a trained model performing another ICL task.

- (88) General TPF-framework-derived hypotheses about how trained transformers perform ICL
- a. For some head in some layer, when projected to an appropriate subspace, the trained model’s \mathbf{i} , \mathbf{q} , \mathbf{k} , \mathbf{v} vectors are fit, under an appropriate linear transformation, by an unknown (to-be-discovered) set of state-variable:value structures
 - i. In this variant of (87a), the variable:values are unknown structures to be discovered by analysis
 - ii. Here we can use an extension of the DISCOVER technique in which there is no predetermined human-generated hypothesis for the role:filler structure of the target hidden states under analysis. A second auxiliary network called ROLE is used to learn to assign roles and fillers for the interpretation of a target hidden state: here, the variables and values of a state structure. In its original application (Soulos, McCoy, et al. 2020), this was successfully applied to an encoder-decoder RNN solving the SCAN compositional-generalization task: ROLE learned to assign one of a set of initially meaningless roles to each symbol in an input sequence, such that the hidden vector produced by the target RNN’s encoder network was accurately modeled as a linearly-transformed TPR built from: a learned set of vector embeddings for the roles; a learned set of vector embeddings for the input symbols; and a learned linear transformation of the resulting TPR.
 - b. For each constituent, on some subspace of the residual stream, the projection on that subspace is the same for all symbols within the same constituent of an unknown (to-be-discovered) parse tree;
 - i. In this variant of (87b), the parse is not hypothesized a priori, but discovered empirically by the analyst, as in the work of Murty et al. 2022, 2023.
 - ii. As mentioned in Sec. 9.2.1, Feng & Steinhardt 2023 discovered a case of this in which the constituents are distinct facts given in the prompt.

- c. At some layer and for some head, there is systematic matching between subspaces of queries and keys at positions systematically linked by the trained transformer’s attention mechanism; across different prompts, as the query varies, the key varies congruently:
 - i. under an orthogonal transformation \mathbb{T} ,²⁰ implementing a to-be-discovered production’s Condition (“ $x[n] == y[N]$ ”): $\mathbf{k}[n] = \mathbf{q}[N] \mathbb{T}$ on appropriately chosen subspaces of the \mathbf{k}, \mathbf{q} vector spaces –
 - ii. possibly with an orthogonality-imposing transformation \mathbb{N} intervening: $\mathbf{k}[n] = \mathbf{q}[N] \mathbb{T} \mathbb{N}$, (analogous to the one’s-complement operation implementing “ $x[n] != y[N]$ ”; \mathbb{N} can be any anti-symmetric matrix, $\mathbb{N}^\top = -\mathbb{N}$).
- d. When decomposed using the subspaces discovered in (88c), some weight matrices $\mathbb{W}_{\mathbf{q},\mathbf{k},\mathbf{v}}$ contain embedded orthogonal block sub-matrices, analogous to the embedded identity matrices \mathbb{I} of DAT for copying values between variables (or an embedded \mathbb{N} matrix for negating a variable).

9.3 Enhancing TPF and Transformer Architectures

Of the many directions for possible future research building on the work presented here, we mention one that significantly extends the framework TPF itself (Sec. 9.3.1), two that pursue how features of the standard transformer architecture might be incorporated into the DAT (Sec. 9.3.2), and three that apply what we have learned from TPF to suggest potential improvements to the standard transformer architecture (Sec. 9.3.3).

9.3.1 Extending the TPF: Composition and Recursion. The task, TGT, defined by the functional-level description of the system studied here (32), is to identify a single template, illustrated by a single example at the beginning of the input prompt, and to generate text by instantiating that template with new symbol strings filling its slots. The powerful symbolic-computation operation of *embedding* would, in this context, consist of multiple templates in a prompt, some embedded within the slots of others. To pursue this we are examining a variation of the task in which templates are replaced by symbolic-function definitions: in place of $Q \text{ swap } x \ y \ \mathcal{A} \ y \ x$ we have $\text{swap}(x, y) \rightarrow y \ x$. This extended task, *Nested Function Evaluation (NFE)*, naturally allows embedding of functions/templates within others, as in the recursive example in (89).

(89) Nested Function Evaluation: NFE

Prompt: $\text{swap}(x, y) \rightarrow y \ x; \text{twice}(x) \rightarrow x \ x; \text{swap}(\text{twice}(a \ b), \text{swap}(c, d)) \rightarrow$
Continuation: $\text{swap}(\text{twice}(a \ b) \rightarrow a \ b \ a \ b, \text{swap}(c, d) \rightarrow d \ c) \rightarrow \text{swap}(a \ b \ a \ b, d \ c)$
 $\rightarrow d \ c \ a \ b \ a \ b$

Preliminary results of current work show that the methods developed here for TGT extend naturally to NFE. TPF can employ productions that cycle between two behaviors which we term *copy* and *eval*. The *copy* behavior operates similarly to CONTFIELD: it simply repeats symbols from the prompt that are too complex to be resolved immediately. This process continues until a segment of the prompt is found which can be resolved using the templates already specified (e.g. $\text{twice}(a \ b)$ in the example above). At this point the model switches to the *eval* behavior, resolving the nested function according to the examples in the prompt. By iterating between these two behaviors, DAT can resolve complex functions through their simple constituents and thereby accommodate the two remaining capabilities of symbolic computation not captured by TGT itself: composition (24f-iv) and recursion (24e). Presuming that these preliminary results hold up, the insights from the modestly-extended TPF suffice to understand how *all* the fundamental compositional elements of symbolic computation (24) can be naturally embedded in the continuous numerical computation of which transformer-style networks are capable.

²⁰ \mathbb{T} maps the subspace for register y to the subspace for register x (or x').

9.3.2 *Enhancing the DAT Architecture.* Certain features of the standard transformer architecture might profitably be incorporated into the DAT architecture in future work.

(90) Enhancing DAT with architectural features of standard transformers

- a. *Multiple attention heads.* A production might be implemented as one head within an extension of DAT deploying multi-headed attention. When multiple productions' Conditions are met and their Actions conflict, symbolic production systems invoke a conflict-resolution procedure preventing multiple conflicting productions from acting simultaneously. In the multi-headed DAT, pre-programming of the production system could ensure that such conflict cannot arise, ensuring that such conflicting productions are separated into different layers, with the appropriate ordering. Non-conflicting productions could be implemented in different heads within a single layer. As long as there remains no MLP sublayer in the DAT, there is little computational difference between productions being implemented in multiple heads within a single layer or as single heads across multiple layers.
- b. *MLPs.* In our TGT formalization, there is no need to modify symbols or identify relations between them other than equality, so we have no need of MLPs in our DAT. Of course it is generally presumed that MLPs play a large role in LMs and the work here does not speak to their role. Many analogy-inspired ICL tasks do incorporate symbol modification, and assume prior knowledge of symbol mappings, as in $Qxyz \mathcal{A}XYZ$ or $Qpqrs \mathcal{A}tuvw$. Addition of symbol-mapping sublayers into the DAT (e.g., an MLP) would open the door to studying this larger class of interesting ICL tasks and developing a general framework that correspondingly extends the version of TPF presented here.

9.3.3 *Enhancing the Transformer Architecture.* The following considerations relating our DAT to trained transformers encourage several possible directions for future work on enhancing standard transformer architectures.

(91) Enhancing standard transformers with architectural features of DAT

- a. *Repeat blocks.* While DAT lacks the MLP and multi-headedness of standard transformers, it has an architectural feature lacking in standard transformers, repeat blocks. The power of such structure has been explored in the Universal Transformer (Dehghani et al. 2018), the Looped Transformer (Y. Fan et al. 2024; Giannou et al. 2023) in which all layers together form a single repeat block, and adaptively at scale in Mixture-of-Recursions (Bae et al. 2025).
- b. *Discretization.* DATmax and DATnorm could provide important inductive biases for transformers learning to perform ICL in something like the way our discretely-functioning DAT does.
 - i. *DATmax* discretizes the processing (attention) to the retrieval of values from a single (by default, leftmost) locus (70c.i). Csordás et al. 2021 have shown substantial improvements in learning embedding-depth generalization on algorithmic tasks by similarly constraining attention.
 - ii. *DATnorm* discretizes the data to a single value per variable (70c.iv). DATnorm imposes a disciplined use of the residual stream to encode discretely-valued state variables — a TPR, not necessarily deploying one-hot embeddings (see Online Appendix 4; this generalizes the 'disentangled residual stream' notion of Friedman et al., 2023).

These features of discreteness (91b) could be inserted into standard transformers softly, as biases implemented as regularization terms added to the loss function.

- c. *Residual-stream recurrence.* During generation, the input to a column of the DAT is not just the single 'next symbol' predicted by the previous column: it is the full residual stream at the top of the previous column. This allows direct propagation of computed features of tokens such as role in a parse (e.g., field); such recurrence gives much power to RNNs and can potentially do the same for enhanced

transformer architectures — at the cost of complicating (or preventing) training by teacher forcing (see also Fan et al., 2020).²¹

9.4 Unifying Semantics-Free and Semantics-Permeated Knowledge

As mentioned in Sec. 1.1, the ultimate goal of this research program is to develop architectures that intimately unify both (i) TPF’s type of formal, semantics-free processing and (ii) standard transformer processing. As merely an illustration of the large space of possibilities for unification, here we present just one simple approach. We propose a transformer architecture part of which is biased (through regularization terms in the loss function) towards DAT-like processing; the rest of the network is a typical transformer with no such bias.

- (92) Integrating DAT and standard-transformer processing: The Semi-formal Transformer architecture
- a. The residual stream V is the direct sum of two subspaces, the *biased* subspace V_b and the *unbiased* subspace V_u .
 - b. There are two types of attention heads: biased heads H_b and unbiased heads H_u .
 - c. Only biased heads can write into V_b (i.e., only they can have value vectors with non-zero components in V_b): V_b , like DAT, is constrained to host a TPR with subspaces encoding symbolic values for variables; writing into the residual stream is performed only by biased heads that use DATnorm to enforce the constraint that registers have a unique symbolic value.
 - d. All heads can read from the entire residual stream, i.e., queries and keys can contain vector components in both V_b and V_u .
 - e. Attention for biased heads is subject to a regularization term in the loss function that biases the model to weights that approximate DATmax, focusing attention towards a single source.
 - f. Layers contain an MLP sublayer that operates exclusively on V_u . Any LayerNorm operations that may be present also apply only on V_u .

In this Semi-formal Transformer architecture, V_b is processed by compositionality-respecting operations, but control decisions selecting what operations to perform are carried out by unconstrained neural processes: this division of labor has proved to be very effective as embodied in the Differentiable Tree Machine (Soulos, Conklin, et al. 2024; Soulos, Hu, et al. 2023).

For a particular task domain, TPF shows how we could use PSL to pre-program the H_b with domain-useful, abstract formal operations; the unbiased portion of the model then uses powerful neural computation to determine how to deploy these formal operations. A promising application domain for this new architecture is the processing of semi-formal documents, which feature natural-language descriptions of properties joined with logical combinators, such as rules for what constitutes an acceptable comment for an online forum, which must not have objectionable property A or B and must contain certain required elements C and D. Standard LM heads determine whether the NL-described properties A–D are true or false of a target comment, writing these values into V_b , and formal heads are programmed to evaluate AND, OR and NOT combinations, given the LM-estimated truth values of these properties that they read from V_b .

9.5 Wrapping Up

The neurocompositionality hypothesis (Smolensky, McCoy, et al. 2022a,b) proposes that human-level intelligence arises from a deep synergy between two principles previously viewed as incompatible: representations have

²¹Note that, unlike the task for standard transformer LMs, the generation task in TGT is deterministic, and our production system results in a unique symbol being assigned non-zero generation probability; thus symbol-generation in a given column does not need to know which of multiple potential symbols was randomly sampled for generation in the previous column.

compositional structure that drives their processing, and representations lie in a continuous space supporting similarity-based generalization and gradient-based learning. We suspect that the astounding success of transformer-based AI results from these systems satisfying the neurocompositionality hypothesis.

The Transformer Production Framework (TPF) presented here (Secs. 4 – 7) shows with complete precision and completeness how a characteristically powerful ability of transformers, templatic text generation through in-context learning (Box 2; Sec. 4.1), can result from neural networks deploying representations that are continuous and compositional, and processing that, while numerical and continuous, exploits the compositional structure of the representations: these networks implement a kind of abstract program that has been a central model of human higher-level cognition – a production system (14). This production-system level of description endows the networks that implement it with great power and generality – it is Turing Universal (Sec. 8).

In addition to enabling hand-programming to create a fully mechanistically-interpretable concrete model capable of powerful ICL, TPF provides a rich set of precise, formally-testable hypotheses about the mechanisms that power ICL in transformers that are trained from data (Sec. 9.2). If even a fraction of these hypotheses are confirmed in future work, significant light will have been shed on the inner workings of the black boxes powering contemporary generative AI.

Finally, while a number of the mechanisms deployed in this paper have been discussed to some extent in the literature – e.g., sparse/discrete attention (Meister et al. 2021; Peters et al. 2019), repeat blocks of layers (Csordás, Irie, and Schmidhuber 2021; Csordás, Irie, Schmidhuber, et al. 2024) and structured residual streams (Mak and Flanigan 2025), the TPF framework presented may prove beneficial for architecture design by allowing us to bridge the gap between theory and practice. Researchers can design tasks that require a certain form of abstract symbolic reasoning, and obtain an existence proof for their solvability by writing the generalizing program and obtaining the corresponding compiled DAT model.

This opens up a number of interesting questions. If we are unable to write a generalizing program without a specific architectural component, is the same true of learned models? If we are able to write the generalizing program, but cannot correspondingly learn it – why not? The TPF framework can be uniquely beneficial here, as it allows us to study potential solutions at an extremely granular level (down to the individual attention head if needed), and thereby bring theory and empirical validation far closer than might otherwise be possible, giving us a principled basis on which to study what components, and what combinations of them, contribute towards generalization.

As an example, both geometric attention (realized in TPF through DATmax) and sharp attention (realized in TPF through DATnorm) have separately been the subject of considerable study. However, the way they combine in the TPF framework (strict condition match followed by positional tie breaker) unlocks a new form of reasoning which had not, to the best of our knowledge, been discussed in the literature at time of writing. An attention mechanism based on – and inspired by – this combination has since been released (Oppen et al. 2025), and enables novel forms of generalization, which are trivial for the DAT, but had previously been unattainable by contemporary transformer variants.

Acknowledgments

For insightful comments and suggestions we wish to thank members of the Johns Hopkins Neurocompositional Computing research group, the Yale Computational Cognitive Science group, and at Microsoft Research (MSR) Redmond, the Deep Learning, RiSE and SIGLR groups, especially Matthai Philipose and Nikolaj Bjrner; we also thank the participants and organizers of the 2024 Weinberg Institute Symposium on Cognitive Science at the University of Michigan (Smolensky, Fernandez, et al. 2024), especially Rick Lewis for discussion of the universality of PSL. Thanks too to Geoff Hinton for discussion of encoding hierarchical structure in transformers, to Tal Linzen for discussion of syntactic performance of LLMs, and to Tom McCoy for discussion of the decoder version

of DISCOVER. For valuable contributions to online resource release we are grateful to Swadheen Shukla at MSR. Finally we acknowledge crucial support of internships at Microsoft for authors M. Opper and A. Davies.

References

- E. Akyurek, D. Schuurmans, J. Andreas, T. Ma, and D. Zhou. 2022. “What learning algorithm is in-context learning? Investigations with linear models.” *arXiv:2211.15661*.
- E. Akyurek, B. Wang, Y. Kim, and J. Andreas. 2024. “In-context language learning: Architectures and algorithms.” *arXiv:2401.12973*.
- G. Alain and Y. Bengio. 2017. *Understanding intermediate layers using linear classifier probes*. (2017). <https://openreview.net/forum?id=ryF7rTqgl>.
- J. R. Anderson. 2005. “Human symbol manipulation within an integrated cognitive architecture.” *Cognitive Science*, 29, 313–341.
- S. Bae et al. 2025. “Mixture-of-Recursions: Learning Dynamic Recursive Depths for Adaptive Token-Level Computation.” *arXiv preprint arXiv:2507.10524*.
- A. Barr, E. A. Feigenbaum, and P. R. Cohen. 1981. *The handbook of artificial intelligence*. Vol. 1. HeurisTech Press.
- K. Bhatia, A. Narayan, C. D. Sa, and C. Ré. 2023. “Tart: A plug-and-play Transformer module for task-agnostic reasoning.” *arXiv:2306.07536*.
- T. Bolukbasi, K.-W. Chang, J. Y. Zou, V. Saligrama, and A. T. Kalai. 2016. “Man is to Computer Programmer as Woman is to Homemaker? Debiasing Word Embeddings.” In: *Advances in Neural Information Processing Systems*. Ed. by D. Lee, M. Sugiyama, U. Luxburg, I. Guyon, and R. Garnett. Vol. 29. Curran Associates, Inc. https://proceedings.neurips.cc/paper_files/paper/2016/file/a486cd07e4ac3d270571622f4f316ec5-Paper.pdf.
- R. V. Book and F. Otto. 1993. *String-rewriting systems*. Springer.
- T. B. Brown et al. 2020. “Language models are few-shot learners.” *arXiv preprint arXiv:2005.14165*.
- M. Canby, A. Davies, C. Rastogi, and J. Hockenmaier. 2024. “How Reliable are Causal Probing Interventions?” *arXiv preprint arXiv:2408.15510*.
- S. C. Chan, A. Santoro, A. K. Lampinen, J. X. Wang, A. K. Singh, P. H. Richemond, J. L. McClelland, and F. Hill. 2022. “Data Distributional Properties Drive Emergent In-Context Learning in Transformers.” *arXiv preprint arXiv:2205.05055*.
- T. A. Chang and B. K. Bergen. 2024. “Language model behavior: A comprehensive survey.” *Computational Linguistics*, 50, 1, 293–350.
- N. Chomsky. 1957. *Syntactic Structures*. Mouton.
- J. Coda-Forno, M. Blinz, Z. Akata, M. Botvinick, J. X. Wang, and E. Schulz. 2023. “Meta-in-context learning in large language models.” *arXiv:2305.12907*.
- R. Cowan. 2001. “Expert systems: aspects of and limitations to the codifiability of knowledge.” *Research Policy*, 30, 9, 1355–1372.
- R. Csordás, K. Irie, and J. Schmidhuber. 2021. “The neural data router: Adaptive control flow in transformers improves systematic generalization.” *arXiv preprint arXiv:2110.07732*.
- R. Csordás, K. Irie, J. Schmidhuber, C. Potts, and C. D. Manning. 2024. “Moeit: Mixture-of-experts universal transformers.” *Advances in Neural Information Processing Systems*, 37, 28589–28614.
- D. Dai, Y. Sun, L. Dong, Y. Hao, S. Ma, Z. Sui, and F. Wei. 2022. “Why Can GPT Learn In-Context? Language Models Implicitly Perform Gradient Descent as Meta-Optimizers.” *arXiv:2212.10559*.
- Q. Dai, B. Heinzerling, and K. Inui. Nov. 2024. “Representational Analysis of Binding in Language Models.” In: *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*. Ed. by Y. Al-Onaizan, M. Bansal, and Y.-N. Chen. Association for Computational Linguistics, Miami, Florida, USA, (Nov. 2024), 17468–17493. doi:10.18653/v1/2024.emnlp-main.967.
- A. Davies, J. Jiang, and C. Zhai. 2023. “Competence-Based Analysis of Language Models.” *arXiv preprint arXiv:2303.00333*.
- A. Davies and A. Khakzar. 2024. “The cognitive revolution in interpretability: From explaining behavior to interpreting representations and algorithms.” *arXiv preprint arXiv:2408.05859*.
- X. Davies, M. Nadeau, N. Prakash, T. R. Shaham, and D. Bau. 2023. “Discovering variable binding circuitry with desiderata.” *arXiv preprint arXiv:2307.03637*.
- M. Dehghani, S. Gouws, O. Vinyals, J. Uszkoreit, and Ł. Kaiser. 2018. “Universal transformers.” *arXiv preprint arXiv:1807.03819*.
- C. P. Dolan and P. Smolensky. 1989. “Tensor Product Production System: A modular architecture and representation.” *Connection Science*, 1, 1, 53–68.
- N. Dziri et al. 2023. “Faith and fate: Limits of transformers on compositionality.” *Advances in Neural Information Processing Systems*, 36, 70293–70332.
- N. Elhage, T. Hume, et al. 2022. “Toy Models of Superposition.” *Transformer Circuits Thread*. https://transformer-circuits.pub/2022/toy_model/index.html.
- N. Elhage, N. Nanda, et al. 2021. “A mathematical framework for transformer circuits.” *Transformer Circuits Thread*, 1, 1, 12.
- J. Engels, E. J. Michaud, I. Liao, W. Gurnee, and M. Tegmark. 2025. “Not All Language Model Features Are One-Dimensionally Linear.” In: *The Thirteenth International Conference on Learning Representations*. <https://openreview.net/forum?id=d63a4AM4hb>.
- A. Fan, T. Lavril, E. Grave, A. Joulin, and S. Sukhbaatar. 2020. “Addressing some limitations of transformers with feedback memory.” *arXiv preprint arXiv:2002.09402*.

- Y. Fan, Y. Du, K. Ramchandran, and K. Lee. 2024. "Looped Transformers for Length Generalization." *arXiv preprint arXiv:2409.15647*.
- J. Feng and J. Steinhardt. 2023. "How do Language Models Bind Entities in Context?" *arXiv preprint arXiv:2310.17191*.
- J. A. Fodor and Z. W. Pylyshyn. 1988. "Connectionism and cognitive architecture: A critical analysis." *Cognition*, 28, 1-2, 3-71.
- D. Friedman, A. Wettig, and D. Chen. 2023. "Learning transformer programs." *Advances in Neural Information Processing Systems*, 36.
- S. Garg, D. Tsipras, P. S. Liang, and G. Valiant. 2022. "What can transformers learn in-context? A case study of simple function classes." *Advances in Neural Information Processing Systems*, 35, 30583-30598.
- A. Geiger, Z. Wu, C. Potts, T. Icard, and N. Goodman. 2024. "Finding alignments between interpretable causal variables and distributed neural representations." In: *Causal Learning and Reasoning*. PMLR, 160-187.
- A. Giannou, S. Rajput, J.-y. Sohn, K. Lee, J. D. Lee, and D. Papailiopoulos. 2023. "Looped transformers as programmable computers." In: *International Conference on Machine Learning*. PMLR, 11398-11442.
- J. Gleick. 1993. *Genius: The life and science of Richard Feynman*. Vintage.
- I. Goodfellow, Y. Bengio, and A. Courville. 2016. *Deep learning*. MIT press.
- J. Hamrick and S. Mohamed. 2020. "Levels of analysis for machine learning." *arXiv preprint arXiv:2004.05107*.
- J. Hartmanis. 1971. "Computational complexity of random access stored program machines." *Mathematical Systems Theory*, 5, 3, 232-245.
- I. Heim and A. Kratzer. 1998. *Semantics in generative grammar*. Blackwell.
- R. Hendel, M. Geva, and A. Globerson. Dec. 2023. "In-Context Learning Creates Task Vectors." In: *Findings of the Association for Computational Linguistics: EMNLP 2023*. Ed. by H. Bouamor, J. Pino, and K. Bali. Association for Computational Linguistics, Singapore, (Dec. 2023), 9318-9333. doi:10.18653/v1/2023.findings-emnlp.624.
- G. Hinton. 2023. "How to represent part-whole hierarchies in a neural network." *Neural Computation*, 35, 3, 413-452.
- W. Hinzen, E. Machery, and M. Werning, (Eds.) . 2012. *The Oxford Handbook of Compositionality*. Oxford University Press.
- J. E. Hopcroft, R. Motwani, Rotwani, and J. D. Ullman. 2000. *Introduction to Automata Theory, Languages and Computability*. (2nd ed.). Addison-Wesley Longman Publishing Co., Inc., USA. ISBN: 0201441241.
- G. Jones and F. E. Ritter. 2003. "Production systems and rule-based inference." *Encyclopedia of cognitive science*, 3, 741-747. Ed. by L. Nadel.
- J. E. Laird. 2019. *The Soar cognitive architecture*. MIT press.
- K. Lasri, O. Seminc, A. Lenci, and T. Poibeau. 2022. "Subject verb agreement error patterns in meaningless sentences: Humans vs. BERT." *arXiv preprint arXiv:2209.10538*.
- S. Lee, A. Davies, M. E. Canby, and J. Hockenmaier. 2025. "Evaluating and Designing Sparse Autoencoders by Approximating Quasi-Orthogonality." *arXiv preprint arXiv:2503.24277*.
- R. L. Lewis and S. Vasishth. 2005. "An Activation-Based Model of Sentence Processing as Skilled Memory Retrieval." *Cognitive Science*, 29, 375-419.
- Y. Li, M. E. Ildiz, D. Papailiopoulos, and S. Oymak. 2023. "Transformers as Algorithms: Generalization and Stability in In-context Learning." *arXiv:2301.07067*.
- D. Lindner, J. Kramár, S. Farquhar, M. Rahtz, T. McGrath, and V. Mikulik. 2023. "Tract: Compiled transformers as a laboratory for interpretability." In: *Advances in Neural Information Processing Systems*. Ed. by A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine. Vol. 36. Curran Associates, Inc., 37876-37899. https://proceedings.neurips.cc/paper_files/paper/2023/file/771155abaae744e08576f1f3b4b7ac0d-Paper-Conference.pdf.
- B. Mak and J. Flanagan. 2025. "Residual Matrix Transformers: Scaling the Size of the Residual Stream." *arXiv preprint arXiv:2506.22696*. <https://arxiv.org/abs/2506.22696> arXiv: 2506.22696 (cs.LG).
- A. Makelov, G. Lange, A. Geiger, and N. Nanda. 2024. "Is This the Subspace You Are Looking for? An Interpretability Illusion for Subspace Activation Patching." In: *The Twelfth International Conference on Learning Representations*. <https://openreview.net/forum?id=Ebt7JgMHv1>.
- G. Marcus. 2001. *The algebraic mind: Integrating connectionism and cognitive science*. MIT press.
- D. Marr. 1982. *Vision: A Computational Investigation into the Human Representation and Processing of Visual Information*. W. H. Freeman, San Francisco.
- R. T. McCoy, T. Linzen, E. Dunbar, and P. Smolensky. 2019. "RNNs implicitly implement Tensor Product Representations." In: *International Conference on Learning Representations*. *arXiv preprint arXiv:1812.08718*.
- R. T. McCoy. 2022. "Implicit compositional structure in the vector representations of artificial neural networks." Ph.D. Dissertation. Johns Hopkins University. <https://jscholarship.library.jhu.edu/items/7b4b8761-db3c-42a5-9e7f-3cb094eaa8ad>.
- C. Meister, S. Lazov, I. Augenstein, and R. Cotterell. 2021. "Is Sparse Attention more Interpretable?" *arXiv preprint arXiv:2106.01087*. <https://arxiv.org/abs/2106.01087> arXiv: 2106.01087 (cs.CL).
- S. Min, M. Lewis, L. Zettlemoyer, and H. Hajishirzi. 2022. "MetalCL: Learning to Learn In Context." *arXiv preprint arXiv:2110.15943*.
- S. Murty, P. Sharma, J. Andreas, and C. D. Manning. 2022. "Characterizing intrinsic compositionality in transformers with tree projections." *arXiv preprint arXiv:2211.01288*.
- S. Murty, P. Sharma, J. Andreas, and C. D. Manning. 2023. "Grokking of hierarchical structure in vanilla transformers." *arXiv preprint arXiv:2305.18741*.
- A. Newell. 1980. "Physical symbol systems." *Cognitive Science*, 4, 2, 135-183.

- C. Olsson et al. 2022. “In-context learning and induction heads.” *arXiv preprint arXiv:2209.11895*.
- M. Oppen, R. Fernandez, P. Smolensky, and J. Gao. 2025. “TRA: Better Length Generalisation with Threshold Relative Attention.” *arXiv preprint arXiv:2104.10350*. <https://arXiv.org/abs/2503.23174> arXiv: 2503.23174 (cs.LG).
- K. Park, Y. J. Choe, and V. Veitch. July 21–27, 2024. “The Linear Representation Hypothesis and the Geometry of Large Language Models.” In: *Proceedings of the 41st International Conference on Machine Learning (Proceedings of Machine Learning Research)*. Ed. by R. Salakhutdinov, Z. Kolter, K. Heller, A. Weller, N. Oliver, J. Scarlett, and F. Berkenkamp. Vol. 235. PMLR, (July 21–27, 2024), 39643–39666. <https://proceedings.mlr.press/v235/park24c.html>.
- J. Pérez, P. Barceló, and J. Marinkovic. 2021. “Attention is Turing-Complete.” *Journal of Machine Learning Research*, 22, 75, 1–35. <http://jmlr.org/papers/v22/20-302.html>.
- B. Peters, V. Niculae, and A. F. T. Martins. 2019. “Sparse Sequence-to-Sequence Models.” *CoRR*, abs/1905.05702. <http://arXiv.org/abs/1905.05702> arXiv: 1905.05702.
- T. Pimentel, J. Valvoda, R. Hall Maudslay, R. Zmigrod, A. Williams, and R. Cotterell. July 2020. “Information-Theoretic Probing for Linguistic Structure.” In: *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*. Association for Computational Linguistics, Online, (July 2020), 4609–4622. doi:10.18653/v1/2020.acl-main.420.
- T. Pimentel, J. Valvoda, N. Stoehr, and R. Cotterell. Dec. 2022. “The Architectural Bottleneck Principle.” In: *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*. Ed. by Y. Goldberg, Z. Kozareva, and Y. Zhang. Association for Computational Linguistics, Abu Dhabi, United Arab Emirates, (Dec. 2022), 11459–11472. doi:10.18653/v1/2022.emnlp-main.788.
- S. Pinker and A. Prince. 1988. “On language and connectionism: Analysis of a parallel distributed processing model of language acquisition.” *Cognition*, 28, 1-2, 73–193.
- E. L. Post. 1943. “Formal reductions of the general combinatorial decision problem.” *American Journal of Mathematics*, 65, 2, 197–215.
- N. Prakash, N. Shapira, A. S. Sharma, C. Riedl, Y. Belinkov, T. R. Shaham, D. Bau, and A. Geiger. 2025. “Language models use lookbacks to track beliefs.” *arXiv preprint arXiv:2505.14685*.
- A. Prince and S. Pinker. 1988. “Subsymbols aren’t much good outside of a symbol-processing architecture.” *Behavioral and Brain Sciences*, 11, 1, 46–47.
- A. Radford, K. Narasimhan, T. Salimans, I. Sutskever, et al. 2018. “Improving language understanding by generative pre-training.”
- F. E. Ritter, F. Tehranchi, and J. D. Oury. 2019. “ACT-R: A cognitive architecture for modeling cognition.” *Wiley Interdisciplinary Reviews: Cognitive Science*, 10, 3, e1488.
- J. Russin, S. W. McGrath, D. J. Williams, and L. Elber-Dorozko. 2024. “From Frege to chatGPT: Compositionality in language, cognition, and deep neural networks.” *arXiv preprint arXiv:2405.15164*.
- S. H. Ryu and R. L. Lewis. 2021. “Accounting for Agreement Phenomena in Sentence Comprehension with Transformer Language Models: Effects of Similarity-based Interference on Surprisal and Attention.” In: *Proceedings of the Workshop on Cognitive Modeling and Computational Linguistics*, 61–71.
- D. Schuurmans, H. Dai, and F. Zanini. 2024. “Autoregressive Large Language Models are Computationally Universal.” *arXiv preprint arXiv:2410.03170*.
- S. Sinha, T. Premsri, and P. Kordjamshidi. 2024. “A Survey on Compositional Learning of AI Models: Theoretical and Experimental Practices.” *arXiv preprint arXiv:2406.08787*.
- P. Smolensky. 1987. “Analysis of distributed representation of constituent structure in connectionist systems.” In: *Proceedings of the 1987 International Conference on Neural Information Processing Systems*, 730–739.
- P. Smolensky. 2012. “Symbolic functions from neural computation.” *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 370, 1971, 3543–3569.
- P. Smolensky. 1990. “Tensor product variable binding and the representation of symbolic structures in connectionist systems.” *Artificial Intelligence*, 46, 1-2, 159–216.
- P. Smolensky, R. Fernandez, and J. Gao. 2024. “The Current Compositionality Crisis in Cognitive Science.” In: video: <https://lsa.umich.edu/weinberginstitute/symposium/symposium/2024-symposium>. Weinberg Institute Symposium, University of Michigan.
- P. Smolensky, R. T. McCoy, R. Fernandez, M. Goldrick, and J. Gao. 2022a. “Neurocompositional computing in human and machine intelligence: A tutorial.” *Microsoft Technical Report MSR-TR-2022*.
- P. Smolensky, R. T. McCoy, R. Fernandez, M. Goldrick, and J. Gao. 2022b. “Neurocompositional computing: From the Central Paradox of Cognition to a new generation of AI systems.” *AI Magazine*, 43, 3, 308–322. *arXiv preprint arXiv:2205.01128*.
- P. Soulos, H. Conklin, M. Oppen, P. Smolensky, J. Gao, and R. Fernandez. 2024. “Compositional Generalization Across Distributional Shifts with Sparse Tree Operations.” In: *The Thirty-eighth Annual Conference on Neural Information Processing Systems*. <https://openreview.net/forum?id=fOQunr2E0T>.
- P. Soulos, E. J. Hu, K. Mccurdy, Y. Chen, R. Fernandez, P. Smolensky, and J. Gao. 2023. “Differentiable Tree Operations Promote Compositional Generalization.” In: *Proceedings of the 40th International Conference on Machine Learning (Proceedings of Machine Learning Research)*. Ed. by A. Krause, E. Brunskill, K. Cho, B. Engelhardt, S. Sabato, and J. Scarlett. Vol. 202. PMLR, 32499–32520. <https://proceedings.mlr.press/v202/soulos23a.html>.

P. Soulos, R. T. McCoy, T. Linzen, and P. Smolensky. 2020. “Discovering the compositional structure of vector representations with Role Learning Networks.” In: *Third BlackboxNLP Workshop on Analyzing and Interpreting Neural Networks for NLP*, 238–254.

D. Sutter, J. Minder, T. Hofmann, and T. Pimentel. 2025. “The Non-Linear Representation Dilemma: Is Causal Abstraction Enough for Mechanistic Interpretability?” *arXiv preprint arXiv:2507.08802*.

S. Swaminathan, A. Dedieu, R. V. Raju, M. Shanahan1, M. Lázaro-Gredilla, and D. George. 2022. “Schema-learning and rebinding as mechanisms of in-context learning and emergence.” *arXiv preprint arXiv:2307.01201*.

E. Todd, M. Li, A. S. Sharma, A. Mueller, B. C. Wallace, and D. Bau. 2024. “Function Vectors in Large Language Models.” In: *The Twelfth International Conference on Learning Representations*. <https://openreview.net/forum?id=AwyxyMwaG>.

D. S. Touretzky and G. Hinton. 1988. “A distributed connectionist production system.” *Cognitive Science*, 12, 3, 423–466. doi:[https://doi.org/10.1016/0364-0213\(88\)90029-8](https://doi.org/10.1016/0364-0213(88)90029-8).

F. Vargas and R. Cotterell. Nov. 2020. “Exploring the Linear Subspace Hypothesis in Gender Bias Mitigation.” In: *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Association for Computational Linguistics, Online, (Nov. 2020), 2902–2913. doi:[10.18653/v1/2020.emnlp-main.232](https://doi.org/10.18653/v1/2020.emnlp-main.232).

A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin. 2017. “Attention is all you need.” In: *Advances in Neural Information Processing Systems*, 5998–6008.

T. Wang, A. Roberts, D. Hesslow, T. L. Scao, H. W. Chung, I. Beltagy, J. Launay, and C. Raffel. 2022. “What Language Model Architecture and Pretraining Objective Work Best for Zero-Shot Generalization?” *arXiv preprint arXiv:2204.05832*. <https://arXiv.org/abs/2204.05832> arXiv:2204.05832 (cs. CL).

G. Weiss, Y. Goldberg, and E. Yahav. 2021. “Thinking Like Transformers.” *arXiv preprint arXiv:2106.06981*.

N. Wies, Y. Levine, and A. Shashua. 2023. “The Learnability of In-Context Learning.” *arXiv preprint arXiv:2303.07895*.

Y. Wu, A. Geiger, and R. Millière. 2025. “How Do Transformers Learn Variable Binding in Symbolic Programs?” In: *Forty-second International Conference on Machine Learning*. <https://openreview.net/forum?id=kVtyv7bpnw>.

S. M. Xie, A. Raghunathan, P. Liang, and T. Ma. 2022. “An Explanation of In-context Learning as Implicit Bayesian Inference.” *arXiv preprint arXiv:2111.02080*.

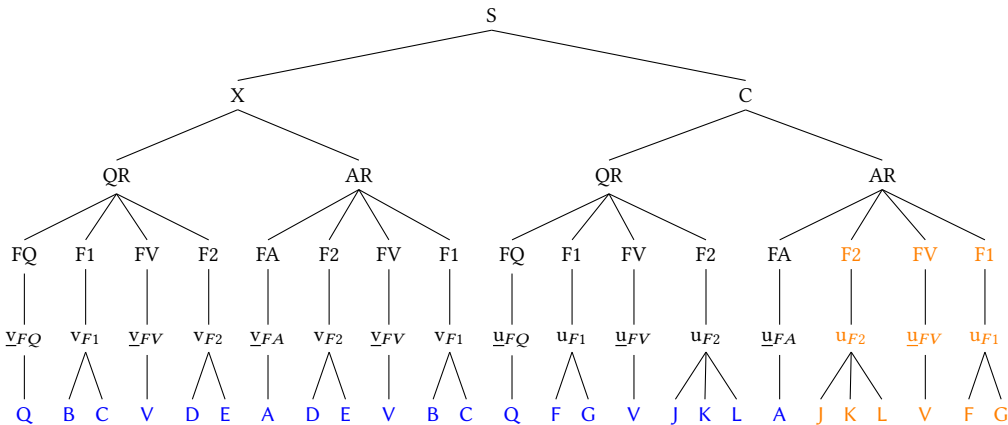
Y. Yang, D. I. Campbell, K. Huang, M. Wang, J. D. Cohen, and T. W. Webb. 2025. “Emergent Symbolic Mechanisms Support Abstract Reasoning in Large Language Models.” In: *Forty-second International Conference on Machine Learning*. <https://openreview.net/forum?id=y1SnRPDWx4>.

K. Yin and J. Steinhardt. 2025. “Which Attention Heads Matter for In-Context Learning?” In: *Forty-second International Conference on Machine Learning*. <https://openreview.net/forum?id=C7XmEByCFv>.

A A Walk-Through of the Symbolic Computation Implicit in the Swap Task

We now describe the role of this structure in the definition and execution of the task Swap, identifying the essential roles in the algorithm of the key properties of symbolic computation spelled out in (24). In this informal section, we will make free use of the properties of the function class \mathcal{F} providing the functional-level description of our TPF system (32): these properties are formally defined below in (32), and (55) indicates how particular steps in the algorithm are justified by that formal definition.

(93) Swap instance (27) full (prompt + continuation) structure: Target



Within the Q-region QR of the example X (subconstituent QR within constituent X), the substrings of the template (28) – Q, x, V, y – are treated as four subconstituents we will call *fields*: they are labelled FQ, F1, FV, F2. The *delimiter fields* FQ and FV alternate in linear position with the *constituent fields* F1 and F2. The A-region AR contains the same fields as QR, but with F1 and F2 swapped, and with the initial field now FA rather than FQ.

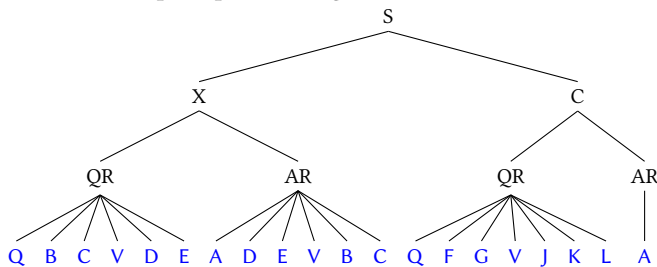
The field structure found in the Q-region QR of the example string is identical to that of the QR in the continuation-cue string C (subconstituent QR within constituent C), but the values of F1 and F2 are different strings. These new field values are then inserted into a copy of the field structure of the example A-region (AR within X) to generate the output, completing the A-region of the continuation (AR within C).

The previous paragraph is a high-level overview of the algorithm in our TPF framework that will generate the continuation using the complete parsed template structure (93): PARGEN. This will be decomposed into an initial *parsing algorithm* PARSE which generates the portion of the tree in (93) that dominates the input provided by the prompt (blue symbols). This will be followed by a *generation algorithm* GEN that uses this parse to build the remainder of (93) (the orange symbols). We begin by walking through the parsing algorithm.

A.1 Parsing Algorithm PARSE: Informal Walk-Through

A first step towards building the Swap structure (93) is parsing the prompt into four regions, shown in (94): in order, they are the Q-region of the example, the A-region of the example, the Q-region of the cue, and the A-region of the cue, which contains only the symbol \mathcal{A} initially, but will be extended by the generation algorithm. The start of each region is marked by a reserved delimiter symbol, either \mathcal{Q} or \mathcal{A} . This illustrates property (24b); S is composed of two subconstituents of type X and C; the X and C subconstituents are in turn each composed of two subsubconstituents of type QR and AR. Thus we have two distinct tokens of both type QR and type AR: property (24a-iii). The beginning of each of the two type-QR constituents is signalled by a token of the atomic Q type (24d); it is important that these two symbols are recognizably tokens of the same type: property (24a-ii). Throughout this section, we identify [in square brackets] the properties of symbolic computation listed in (24) which are crucial to performing this task. We see that nearly all the properties in (24) are demanded by the task, showing that it provides a strong test of symbolic computational abilities.

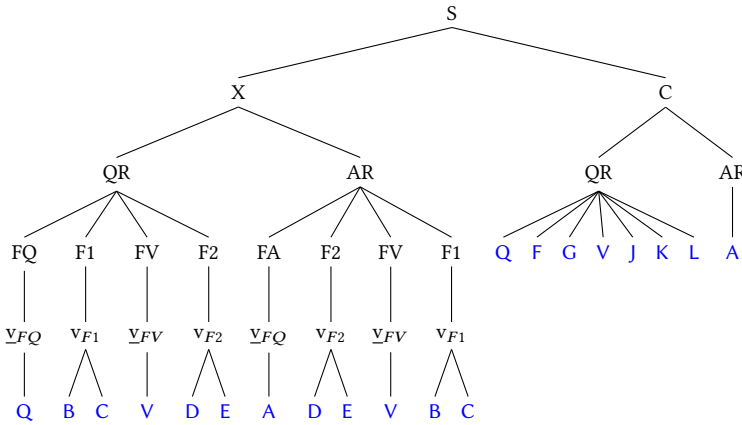
(94) Swap instance prompt (27a): region structure



A next step is parsing the Q-region into fields, shown in (95); as already mentioned, these are the substrings into which the region must be divided to capture the templatic pattern (28) which the generation process will later need to fill. Each substring is taken to be the value assigned to a variable, the field that it fills. The delimiter \mathcal{Q} is the value of the delimiter field FQ. \mathcal{V} is also a delimiter, the value of the field variable FV: *all delimiter fields have a constant value throughout the input structure S*. That \mathcal{V} repeats in the Q-region of the cue signals that it is a delimiter: in the task we study, non-delimiter fields must change value between the example and the continuation-cue. FV separates the two argument slots of the template: x and y in (28). The first of these is the field F1, with value $\mathcal{B}\mathcal{C}$; the second is F2, with value $\mathcal{D}\mathcal{E}$. These non-delimiter fields will be called *constituent*

fields: their values are not fixed; they are the open slots in the template, which take different values in the cue and in the example.

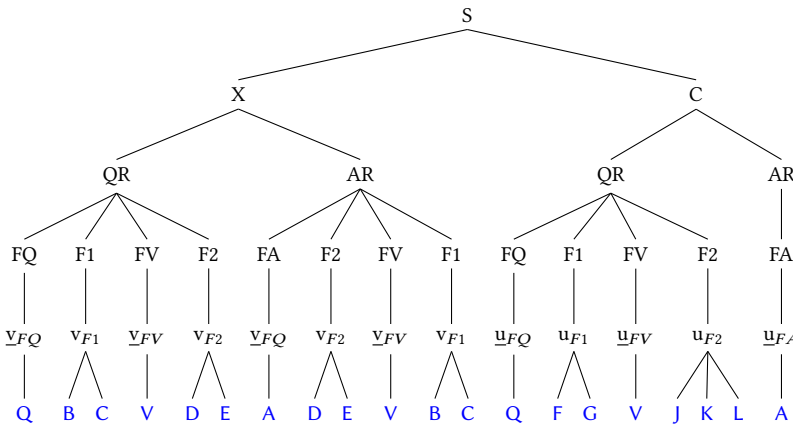
(95) Swap instance prompt (27a): example region field structure (delimiters underlined)



In order that their boundaries be well-defined, constituent fields in the example Q-region must be separated by delimiter fields (which we mark by underlining). Thus B C and D E are the values of constituent fields, which we are calling F1 and F2; they are separated by the delimiter field FV. B C is labeled as the value of F1 by being grouped under the node v_{F1} ; likewise for D E and v_{F2} . That B C and D E are the values of constituent fields is signalled by their appearance as substrings of both the Q- and the A-regions of the example: these symbol-sequences retain their identities – are detectable as identical – across the different regions in which they appear (24a-ii). The values of constituent fields vary across inputs, but within a given Q/A pair within a single input – i.e., separately within each of X and C – these fields have the same values. Like the Q-region of the example, the A-region of the example has $v_{F1} = \text{B C}$ as the value of F1, and $v_{F2} = \text{D E}$ as the value of F2: but the ordering of fields is different in the Q- and A-regions of the example. These field sequences are respectively the subconstituent role sequences characteristic of constituents of type QR and AR [property (24c-i)].

Since the parsing of the Q-region of the example identifies the sequence of roles (here, fields) characteristic of type-QR constituents, the Q-region of the cue can be parsed using the same field sequence: this is shown in (96).

(96) Swap instance prompt (27a): complete field structure

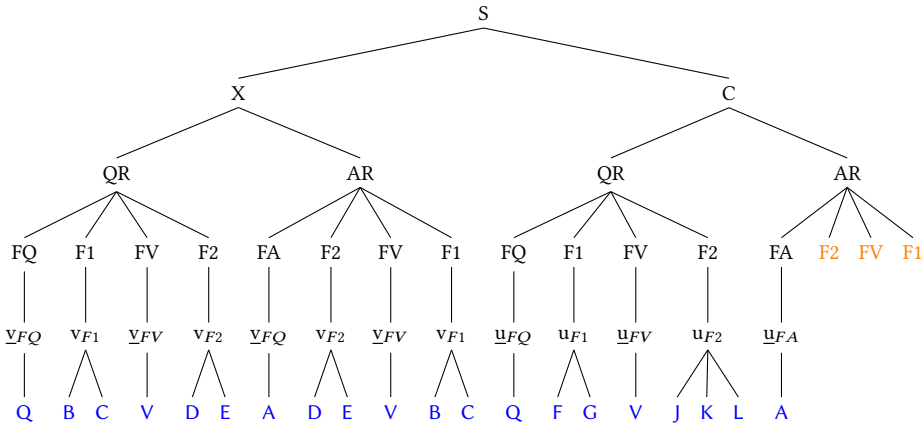


Given the sequence of fields of the Q-region of the cue, we can identify the values filling those fields. The delimiter fields FQ, FV are matched to their fixed string values **Q**, **V** (the same in the cue as in the example); and *the substrings between those delimiters are parsed as the new values of the constituent fields F1, F2 in the cue*: these are $u_{F1} = \mathbf{F\ G}$, $u_{F2} = \mathbf{J\ K\ L}$. [Illustrating property (24c-i), constituents of type QR have a characteristic sequence of roles (here, ‘fields’): FQ F1 FV F2, whether embedded within an X or a C constituent (24a-ii). Constituents of type AR have the role-sequence: FA F2 FV F1, not only in the given AR embedded within X, but also in the to-be-generated AR embedded within C.]

A.2 Generation Algorithm GEN: Informal Walk-Through

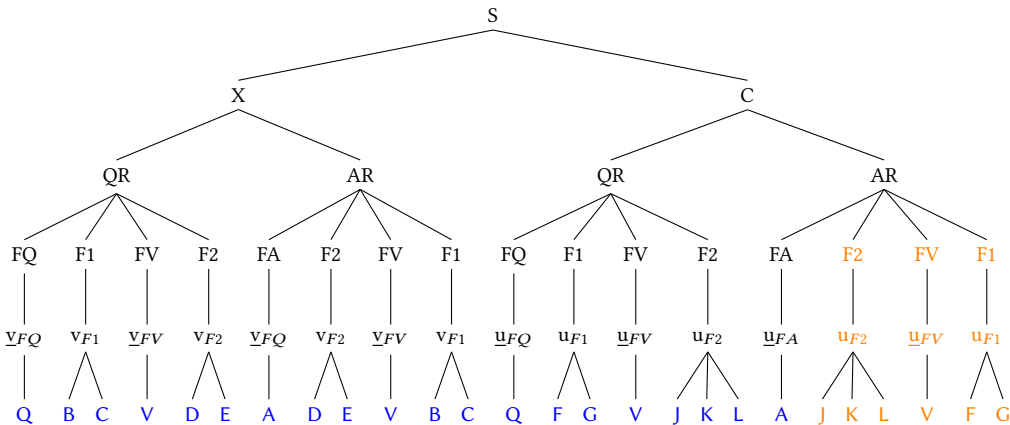
Having identified the field sequence of an AR constituent from the AR within the example, the same sequence populates the AR in the cue: (97) [properties (24a-ii) and (24c-i)].

(97) Continued Swap instance (27): field structure



Within a single QR/AR pair, each field has the same value in both the QR and the AR subconstituents, so the new values of the fields identified in the QR of the cue – u_{F1} and u_{F2} (along with the fixed value of the delimiter field FV) – must serve as the values of those same fields in the completion’s AR (98) [illustrating (24c-iii)].

(98) Swap instance (27) full structure: Generated



This determines the continuation string **J K L V F G**.

If the continuation string is to be produced one symbol at a time — as in the transformer implementation in our TPF system — generating the first symbol **J** involves generating the next field, **F2**, and unbinding it [property (24f-iii)] to get the first symbol of its value string in the *Q-region of the cue*, u_{F2} ; this is then bound to the new instance of **F2** in the *A-region of the cue* [property (24f-ii)]. This is the first of our two generation operations: starting the next field, NEXTFIELD. The second operation, continuing a field that has already been initiated — CONTFIELD — generates the next symbol in the current field: **K**. Choosing the appropriate operation is a case of conditional branching conditioned on representational structure [property (24f-i)].

CONTFIELD is in fact the operation performed by an *induction head* (Olsson et al. 2022): predict that the symbol following some symbol Σ in the continuation will be of the same type as the symbol following the most recent previous token matching the type of Σ . While it is known how to implement CONTFIELD in a transformer using induction attention heads, NEXTFIELD is a considerably more abstract operation: predict that the *field* following Σ in the continuation will be of the same type as the *field* following the previous occurrence of Σ 's field type — within the *example A-region*, not the *cue Q-region*. Given the next field type, generating the next symbol requires determining the value string for that field type — within the *cue Q-region*, not the *example Q-region*.

To generate the next continuation symbol, the generation algorithm we give below (see also Sec. 5.3.3) decides whether to apply NEXTFIELD or CONTFIELD; for this, it must determine whether the value string for the field currently being generated has been completed: if so, NEXTFIELD is called for; otherwise, CONTFIELD is needed. This is a structure-sensitive choice [property 24f-i]. The results of processing the subconstituents (fields) of AR with NEXTFIELD and CONTFIELD are composed together in the field-sequence prescribed by the parse structure of AR [property 24f-iv].

For NEXTFIELD to generate the next field in the AR of C under construction, the last-generated field must be matched with a field in the AR of the example X; the field following that matching field within X gives the field type that needs to be generated next in the continuation.

B PSL and QKVL Programs for Templatic Parsing and Generation with the PARGEN Algorithm

(99) PARSE

P#	gloss	PSL Production		QKVL instructions		
		Condition	Action	q	k	v
		all have parse[N] == 1		all have a:a	all have a:1	
0	initialize region, type; field = position	position[N] == position[N]	region[N] = R_INIT type[N] = T_INIT field[N] = position[N] prev_position[N] = 0 index[N] = 1	p:p	p:p	r:R, t:T, f:p, p*:0, d:1
pre-1	set prev_position and prev_symbol	position[n] == position[N]@pos_decrement	prev_position[N] = position[n] prev_symbol[N] = symbol[n]	p^:p@pos_ decrement	p^:p	p*:p, s*:s
1a	start Example (at p:0): r:XQ, t:D, f:FQ, d:0	prev_position[N] == 0	region[N] = XQ type[N] = D field[N] = FQ index[N] = 0	p*:p*, p:p	p*:0, p:p	r:XQ, t:D, f:FQ, d:0
1b	start Cue (at 2nd Q): r:CQ, t:D, f:FQ	symbol[n] == symbol[N] position[n] == 1 position[N] != 1	region[N] = CQ type[N] = D field[N] = FQ	s':s, p':1, p:p	s':s, p':p, p!=1	r:CQ, t:D, f:FQ
pre-2a	<i>start repeat block</i> set prev_region	position[n] == position[N]@pos_decrement	prev_region[N] = region[n]	p^:p@pos_ decrement	p^:p	r*:r
2a	propagate r:XQ to 1st delimiter (t:D; starts r:CQ) <i>end when NO_CHANGE</i>	prev_region[N] == XQ region[N] == R_INIT	region[N] = XQ	r*:r*, r:r, p:p	r*:XQ, r:R, p:p	r:XQ
pre-2b	<i>start repeat block</i> set prev_region	position[n] == position[N]@pos_decrement	prev_region[N] = region[n]	p^:p@pos_ decrement	p^:p	r*:r
2b	propagate r:CQ to input end <i>end when NO_CHANGE</i>	prev_region[N] == CQ region[N] == R_INIT	region[N] = CQ	r*:r*, r:r, p:p	r*:CQ, r:R, p:p	r:CQ
3a	start r:XA (f:FA) at A in current r:XQ	symbol[N] == A region[N] == XQ	region[N] = XA type[N] = D field[N] = FA	s:s, r:r, p:p	s:A, r:CQ, p:p	r:XA, t:D, f:FA
3b	start r:CA (f:FA) at A in current r:CQ	symbol[N] == A region[N] == CQ	region[N] = CA type[N] = D field[N] = FA	s:s, r:r, p:p	s:A, r:XQ, p:p	r:CA, t:D, f:FA
pre-4	<i>start repeat block</i> set prev_region	position[n] == position[N]@pos_decrement	prev_region[N] = region[n]	p^:p@pos_ decrement	p^:p	r*:r
4	propagate XA to 1st t:D (starts r:CQ) <i>end when NO_CHANGE</i>	prev_region[N] == XA region[N] == XQ type[N] == T_INIT	region[N] = XA	r*:r*, r:r, t:t, p:p	r*:XA,t:T, r:XQ, p:p	r:XA
5a	symbol in XQ later repeated in CQ; set t:D	region[n] == CQ region[N] == XQ symbol[n] == symbol[N]	type[N] = D	r^:CQ, r:r, s^:s	r^:r, r:XQ, s^:s	t:D
5b	symbol in CQ that repeats from XQ; set t:D, same field	region[n] == XQ region[N] == CQ symbol[n] == symbol[N]	type[N] = D field[N] = field[n]	r^:XQ, r:r, s^:s	r^:r, r:CQ, s^:s	t:D, f:f
5c	symbol in XA that repeats from CQ; set t:D, same field	region[n] == CQ region[N] == XA symbol[n] == symbol[N]	type[N] = D field[N] = field[n]	r^:CQ, r:r, s^:s	r^:r, r:XA, s^:s	t:D, f:f
6	identical untyped symbols in X have the same C field	symbol[n] == symbol[N] region[n] == XQ region[N] == XA type[N] == T_INIT	type[N] = C field[N] = field[n]	s^:s, r^:XQ, r:r, t:t	s^:s, r:XA, r^:r, t:T	t:C, f:f
7	unset types in XA are delimiters	region[N] == XA type[N] == T_INIT	type[N] = D	r:r, t:t, p:p	r:XA, t:T, p:p	t:D
7'	remaining unset types are con- stituents	type[N] == T_INIT	type[N] = C	t:t, p:p	t:T, p:p	t:C

P#	gloss	PSL Production		QKVL instructions		
		Condition	Action	q	k	v
pre-8	set prev_region, prev_type, prev_field	all have parse[N] == 1	prev_region[N] = region[n]	all have aa	all have a:1	
		position[n] == position[N]@pos_decrement	prev_type[N] = type[n] prev_field[N] = field[n]	p`p@pos_decrement	p`p	r`r, t`t, f`f
8	field sequence is the same in XQ and CQ	prev_region[n] == XQ region[n] == XQ prev_type[n] == D type[n] == C region[N] == CQ prev_type[N] == D type[N] == C prev_field[n] == prev_field[N]	field[N] = field[n]	r`^:XQ, r`:XQ t`^:D, t`:C, r:r, t`^:t, t:t, f`^:f*	r`^:r*, r`:r, f:f t`^:t*, t`:t, r:CQ, t`^:D, t:C, f`^:f*	
pre-9	<i>start repeat block</i> set prev_field	position[n] == position[N]@pos_decrement	prev_field[N] = field[n]	p`p@pos_decrement	p`p	f`f
9	constituent fields change only at t:D	prev_type[N] == C type[N] == C	field[N] = prev_field[N]	t`^:t*, t:t, p:p	t`^:C, t:C, p:p	ff*
10	<i>end when NO_CHANGE</i> change in f => d:0	prev_field[N] != field[N]	index[N] = 0	f`^:f*, p:p	f`^:f, p:p	d:0
11	set parse = 0 at prompt's last token	z_temp[N] == EOP	parse[N] = 0	z:z, p:p	z:EOP, p:p	a:0

(100) GENERATE

G#	gloss	Production		QKV instructions		
		Condition	Action	q	k	v
0	set end = 0, x_temp = 0 globally	all have parse[N] == 0 position[N] == position[N]	end[N] = 0 x_temp[N] = 0	p:p	p:p	e:0
pre-1	update prev_symbol and prev_field	position[n] == position[N]@pos_decrement	prev_symbol[N] = symbol[n] prev_field[N] = field[n]	p`p@pos_decrement	p`p	s`s, f`f
1	IF symbol in CQ matching current symbol is not field-final THEN [CONTFIELD] copy next symbol in CQ, set end[N]=1, x_temp[N]=0 to prevent application of remaining productions	region[n] == CQ prev_symbol[n] == symbol[N] prev_field[n] == field[N] index[n] != 0	end[N] = 1 x_temp = 0 region[N] = CA symbol[N] = symbol[n] field[N] = field[n] type[N] = type[n] index[N] = index[n]	r`:CQ, s`^:s, f`^:f, d`^:0	r`:r, s`^:s*, f`^:f*, d`:d	e:1, x:0, r:CA, s:s, ff, t:t, d:d
pre-2	update prev_field and prev_region	position[n] == position[N]@pos_decrement	prev_field[N] = field[n] prev_region[N] = region[n]	p`p@pos_decrement	p`p	f`f, r`r
2	ELSE (end[N] = 0) [NEXTFIELD] find final position of field in XA matching current field, assign following field label to y_temp[N]	end[N] == 0 region[n] == XA prev_field[n] == field[N] index[n] == 0 prev_region[n] == XA	y_temp[N] = field[n]	e:e, r`^:XA, f`^:f, d`^:0, r`^:XA	e:0, r`:r, y:y f`^:f*, d`:d, r`^:r*	
3	ELSE con't: IF find initial position of field y_temp[N] in CQ, copy that symbol and its structural variables (except r) and set flag x_temp:=1 to block P3'	end[N] == 0 region[n] == CQ field[n] == y_temp[N] index[n] == 0	x_temp[N] = 1 region[N] = CA symbol[N] = symbol[n] field[N] = field[n] type[N] = type[n] index[N] = index[n]	e:e, r`:CQ, f`f:y, d`:0	e:0, r`:r, f`f: d`:d	x:1, r:CA, s:s, ff, t:t, d:d
3'	ELSE (x_temp=0): find initial position of the field y_temp[N] in XA, copy that symbol and its structural variables (except r)	end[N] == 0 x_temp[N] == 0 region[n] == XA field[n] == y_temp[N] index[n] == 0	region[N] = CA symbol[N] = symbol[n] field[N] = field[n] type[N] = type[n] index[N] = index[n]	e:e, x:x, r`^:XA, f`f:y, d`^:0	e:0, x:0, r`:r, f`f, d`^:d	r:CA, s:s, ff, t:t, d:d

C Further Related Work

Here we widen the discussion of related work from that initiated in Sec. 2.

C.1 ICL Mechanisms

Bayesian learning: Xie et al. 2022 posit that ICL can emerge in models when pre-training documents possess long-range coherence derived from a document-level ‘concept’. They study knowledge-based ICL examples with 0-64 shots that must predict a single correct output token. They explain that models must infer a latent concept from the prompt’s n -shot input/output pairs to predict the correct output token, describing this as an implicit form of Bayesian inference. In our work, the ‘concept’ providing coherence to a particular prompt is not semantically defined, but an abstract symbolic template, which governs the generation of an entire output string; this template must be inferred from each prompt’s single example. We analyze the inference process involved in completing such prompts, but not the conditions enabling such a process to be learned.

Gradient descent: Dai et al. 2022 report finding a duality between attention in transformer models and gradient descent, and posit that in-context learning can be understood as implicit fine-tuning by adjusting the model weights using attention in the feedforward inference process. The contribution to attention from the in-context material can be viewed as an adjustment to the trained attention-governing weights, with the value vectors playing the role of back-propagated error signals in a fine-tuning update restricted to query- and key-generating weights. They study ICL classification tasks, with up to 32-shot examples. Our work is focused on how a pre-determined algorithm for sophisticated symbol-processing can be applied using 1-shot examples to predict an entire output sequence, not just a classification label.

CSCG: Swaminathan et al. 2022 view in-context learning through a different sequence-learning model called the clone-structured causal graph (CSCG), using the mechanisms of schema learning, recall, and rebinding. They posit and confirm that similar mechanisms could exist in transformers. They study both knowledge-based classification tasks as well as abstract sequence-prediction tasks.

C.2 Influence of Pre-Training Data

Training data requirements for ICL: Chan et al. 2022 establish 3 requirements for LM-training data to give rise to a model exhibiting ICL: data burstiness (not uniformly distributed), data with a large number of rarely occurring classes, and data with dynamic meaning. Using the Omniglot dataset, they show that ICL usually competes with “in-weights learning”, with one of the two winning.

C.3 ICL Function Learning

Simple numerical functions: Garg et al. 2022 train decoder-only transformers from scratch to perform ICL in simple numerical function classes: linear functions, sparse linear functions, decision trees, and two-layer ReLU neural networks (using 20–40 shot examples of 20-dimensional inputs). They show that ICL achieves performance comparable to an optimal least squares estimator, and is robust to certain types of train-test distribution shift.

Language learning: Akyürek et al. 2024 introduce a new task for studying ICL — stochastic languages — and compare the performance of transformers to state space models and their variants. They find that transformers perform best on this task, and posit that induction heads are important. They show improvements on the other models when the equivalent of induction heads are added. This lends credibility to the hypothesis that, for

trained models, counterparts of our operations of `CONTFIELD`, identical to induction heads, and `NEXTFIELD`, the abstraction of induction heads from symbol matching to syntactic category matching, may have an important role.

C.4 ICL Learning Process

Generalization and Stability: Li et al. 2023 study the generalization and stability of transformers pre-trained using multi-task learning (MTL) with n -shot-style examples. Through proofs with mild assumptions, they obtain generalization bounds for ICL that depend on the stability of the transformer algorithm, showing that as the ICL prompt length increases, the ICL predictions become more stable. They also find that the transfer risks on unseen examples depend on the number of examples and complexity of the MTL tasks, but, surprisingly, not on the complexity of the transformer architecture.

ICL algorithms for linear regression: Akyurek et al. 2022 study linear regression, exploring whether ICL uses one of three known algorithms to learn the linear function latent in the examples in a given ICL prompt. They identify 4 operations over the hidden states of a transformer layer that can be implemented in a single transformer layer (move-subvector, matrix-multiply, scalar-divide, affine-transform) and prove by construction (programming the transformer with these 4 instructions) that a transformer can emulate 3 classical solutions to linear regression: gradient descent, closed-form ridge regression, and exact least-squares regression. They then pre-train transformers on linear regression tasks with an ICL-style objective and examples, and compare the resulting behavior to the 3 previously programmed models. They find that their trained transformers transition between the classical algorithms as depth and dataset noise vary, and converge to optimal Bayesian decisions as the width and number of layers grow.

PAC in-context learnability: Wies et al. 2023 extend the PAC framework to prove, under mild assumptions, that ICL efficiently ‘learns’ tasks through examples. Their pre-training data is a mixture of multiple latent downstream tasks, presented in n -shot-style prompts, with consistent delimiters in each prompt. They conclude that “in-context learning is more about identifying the task than about learning it” [p. 1]. They find polynomial sample complexity in the number of shots per prompt.

C.5 Improving ICL

Meta training: Min et al. 2022 fine-tuned a GPT-2 large transformer tasks from 142 NLP datasets reformatted as ICL style tasks. This resulted in increased performance over baselines on the test set (containing 52 unique target tasks), sometimes beating models with 8x larger parameter count. Performance increased with the number and diversity of the fine-tuning tasks. Best performance resulted from fine-tuning with both instructions and the meta-training tasks.

Meta ICL: Coda-Forno et al. 2023 introduced a technique to improve ICL by preceding the normal n -shot examples of a task in the prompt with K n -shot examples from other, related tasks. They study this technique using the pre-trained GPT-3 model and 2 tasks: 1-dimensional regression and a 2-armed bandit. Through analysis of the ICL activations, they show how these K additional-task examples reshape the model’s prior over expected tasks.

Reasoning module: Bhatia et al. 2023 analyze ICL failure cases, relative to task specific fine-tuning, and posit that ICL has all the information it needs – the right representations for the task – but fails due to its inability to perform simple probabilistic reasoning over the representations to predict the next token. They

create a separate, task agnostic reasoning module (a decoder-only transformer), trained only on synthetic logistic regression tasks. After training, the models are composed by feeding the output of the base model through a PCA/averaging layer and then to the reasoning module. An additional variant called leave-one-out (LOO) embedding improves the model further. They also demonstrate that the reasoning module is not just model- and task-agnostic, but also modality-agnostic, by using it for binary classification tasks with audio and image inputs.

D Templatic Generation Task (TGT) Grammar

The grammar describing Templatic Generation tasks is shown here in Backus-Naur form:

```

<tgt>                ::= <example> <cue>
<example>           ::= Q <question> A <answer>
<cue>               ::= Q <question> A
<question>          ::= <dc sequence>
<answer>            ::= <dc sequence>

<dc sequence>       ::= <constituent list> [ <delimiter> ]
<constituent list> ::= <constituent> [ <delimiter> <constituent list> ]

<delimiter>        ::= <symbol> [ <delimiter> ]
<constituent>      ::= <symbol> [ <constituent> ]

```

Additional constraints:

- no symbol can be repeated within the <question>, within the <answer>, or within the <cue>.
- symbols in the <constituents> of the <cue> cannot overlap with symbols in the <constituents> of the <question>.

E PSL Grammar

The grammar for the PSL Language is shown here in Backus-Naur form:

```

<program>           ::= <declarations> <statements>
<declarations>     ::= <declaration> [ <declarations> ]
<declaration>      ::= <register map> | <constant map> | <system map> | <watch list>

<register map>      ::= registers "{" <register entry list> "}"
<register entry list> ::= <register entry> [ "," <register entry list> ]
<register entry>    ::= <register name> ":" <register short name>
<register name>     ::= <unquoted string>
<register short name> ::= <quoted string>

<constant map>     ::= constants "{" <constant entry list> "}"
<constant entry list> ::= <constant entry> [ "," <constant entry list> ]
<constant entry>   ::= <constant name> [ ":" <quoted string> ]
<constant name>    ::= <unquoted string>

```

```

<system map> ::= system "{" <system entry list> "}"
<system entry list> ::= <system entry> [ "," <system entry list> ]
<system entry> ::= <system register name> ":" <register name>
<system register name> ::= symbol | position | output | parse | eop

<watch list> ::= watch "[" <register name> [ "," <register name> ] "]"

<statements> ::= <statement> [ <statements> ]
<statement> ::= <causal attn statement> | <where statement> | <repeat statement>

<causal attn statement> ::= causal_attn ":" <boolean value>
<boolean value> ::= true | false

<where statement> ::= <where variant> <conditions> ":" <assignments>
<where variants> ::= where | where_lm | where_rm

<conditions> ::= <condition> [ and <conditions> ]
<condition> ::= <simple condition> | "(" <conditions> ")"

<simple condition> ::= <bool compare> | <bool in>
<bool compare> ::= <left cond> <compare op> <right cond>
<left cond> ::= <register name> "[" <register index> "]"
<register index> ::= N | n
<compare op> ::= "=" | "!="
<right cond> ::= <constant name> | <right reg> [ <weight func> ]
<right reg> ::= <register name> "[" <register index> "]"
<weight func> ::= "@" <weight function>
<weight function> ::= pos_increment | pos_decrement

<bool in> ::= <left cond> <in op> "[" <constant list> "]"
<in op> ::= in | not in
<constant list> ::= <constant name> [ "," <constant list> ]

<assignment list> ::= <assignment> [ <assignment list> ]
<assignment> ::= <assign left> "=" <assign right>
<assign left> ::= <register name> "[" N "]"
<assign right> ::= <constant name> | <right reg>

<repeat statement> ::= repeat <statements> until <stop condition>
<stop condition> ::= NO_CHANGE | <conditions>

```

F DAT Operation

This appendix summarizes how the DAT Transformer operates (during inference, i.e., in-context learning; in-weights-learning has been left for future work). The explanation takes as a starting point a conventional

transformer, and describes the alterations that lead to the DAT.

F.1 High-Level Architecture

- We start with a normal decoder-only transformer.
- We remove the Feedforward module in each layer.
- In each column of the transformer, the residual stream, and vectors for inputs, queries, keys, and values, are maintained as `n_registers` registers, each of which is a 1-hot (or 0-hot) vector over the vocabulary symbols.
- We load and freeze all Multi-Head Attention (MHA) weights (DAT only operates in inference mode).

F.2 Transformer-Level Operation

- As part of the DAT initialization, the tensor weights (compiled from QKVL code that was compiled from PSL code) are loaded into the Q, K, V weights (matrices and bias vectors) for each layer. These weights are then frozen.
- The `forward()` method of the transformer contains an additional parameter `input_embeddings` that is used as follows:
 - When the given prompt is being processed (the initial prompt, before any new columns have been generated), `input_embeddings` is set to `None`.
 - When the remainder of the prompt is being processed, `input_embeddings` for the current column is set to the output of the DAT for the previous column.
- When the `forward()` method of the transformer is called, we build the embeddings for all columns as follows:
 - Compute the symbol embeddings using a frozen matrix that translates each vocab index into its 1-hot vector representation.
 - Compute a “one” embedding (using the vocab index for the symbol “1”).
 - Compute an “eop” embedding (using the vocab index for the symbol “EOP”).
 - Compute “pos” embeddings (using the vocab index for each position number, e.g., “42”).
 - Initialize the “src” embeddings to all zeros (for all columns and all registers within each column).
 - Finally, we set registers on the src embeddings:
 - * Set the “s” register to the symbols embeddings.
 - * Set the “p” register to the pos embeddings.
 - * Set the “a” register (for the parse flag) to the one embeddings.
 - * Set the “z” register for the last column to the eop embedding.
 - If `input_embeddings` (from processing the prefix columns) have been passed to `forward()`, we set src embeddings to the concatenation of (src embeddings, `input_embeddings`).
- We process each layer of the transformer as follows:
 - If the layer is the first layer of a repeat block, we capture the input value of all of the registers in each column.
 - We process the layer using the input to produce an output.
 - If the layer is the last layer of a repeat block, we compare the register values of the output to the register values of saved-off input, for each column. If any register of any column has changed, we continue processing with the first layer of the repeat block.
 - If the registers have not changed, or if the current layer was not the last layer of a repeat block, we continue processing with the next layer.

F.3 Layer-Level Operation

MHA is performed with the following changes:

- The `softmax()` and `dropout()` of the standardly-computed attention weights are replaced by `DATmax`, i.e., the computation of $\alpha[N]$ given in (70c-i).
- Values of input, query, key, value, MHA output, and `attn_weight` tensors are captured for later diagnostics and visualization (see screenshot in Sec. 7.2.3).
- The standard `dropout()` and `LayerNorm()` used to add the output of MHA to the residual stream are replaced by `DATnorm`, as given in (70c-iii).

G TGT Testing Prompt Prefix

When testing LLMs on our TGT dataset (Sec. 4.2.2), the following `sys_prompt` was prepended to each Q/A string in the dataset:

```
You are a helpful assistant; please complete the following abstract
pattern exactly once. The pattern contains an example question/answer
pair, followed by a second question and a missing answer. Do not output
anything except the final answer. Pay close attention to all special
characters.
```

One of the task variants also appends the TGT grammar to the above system-prompt:

```
The grammar for these patterns can be described as follows:
<tgt> ::= <example> <cue>
<example> ::= Q <question> A <answer>
<cue> ::= Q <question> A
<question> ::= <dc sequence>
<answer> ::= <dc sequence>
<dc sequence> ::= <constituent list> [ <delimiter> ]
<constituent list> ::= <constituent> [ <delimiter> <constituent list> ]
<delimiter> ::= <symbol> [ <delimiter> ]
<constituent> ::= <symbol> [ <constituent> ]
```

H LLM Testing Details

This appendix includes the results of exploratory testing on the GPT-4 LLM Transformer, using variants of the core task: `1_show_rlw`.

To test the effect of different number of constituents in the core task, we varied the constituent count as shown in Table 6. Increasing the count had the most impact on performance of all the variants tested (ranging from 0.96 to 0.23 accuracy). This test shows that even the best LLM starts to fail as the TGT task complexity increases.²²

²²The ‘ood’ in the file names here refers to test items that are out of the training distribution for models trained from scratch, discussed in Online Appendix 5. These same test items are used here to test LLMs, where we of course do not control the training distribution, so ‘ood’ should not be taken literally in this context. However, it is true that except for the `1_shot_eng` split, all the test items use symbols that are random-letter ‘words’ (rlw), and these symbols may never have been encountered during LM training.

Table 6. Experiment results showing accuracy with varying numbers of constituents.

Task	Split	Con Count	Con Length	Prompts	Accuracy
1_shot_rlw	test	1	1	100	0.96
1_shot_rlw	test	2	1	100	0.97
1_shot_rlw	ood_cons_count_3	3	1,2,4	100	0.62
1_shot_rlw	ood_cons_count_5	5	1,2,4	100	0.35
1_shot_rlw	ood_cons_count_7	7	1,2,4	100	0.26
1_shot_rlw	ood_cons_count_10	10	1,2,4	100	0.23

To test the effect of different number of symbols per constituent, we varied the constituent length as shown in Table 7. Here we see a striking contrast with Table 6 – the model performance drops only slowly as this aspect of the task complexity increases. Notice that with the value 2 and 3, the model performed perfectly.

Table 7. Experiment results showing GPT-4 accuracy with varying constituent lengths.

Task	Split	Con Count	Con Length	Prompts	Accuracy
1_shot_rlw	test	1	1	100	0.96
1_shot_rlw	test	1	2	100	1.00
1_shot_rlw	ood_cons_len_3	1,2,4	3	100	1.00
1_shot_rlw	ood_cons_len_5	1,2,4	5	100	0.98
1_shot_rlw	ood_cons_len_7	1,2,4	7	100	0.96
1_shot_rlw	ood_cons_len_10	1,2,4	10	100	0.89

For the next test, we wanted to see if model performance would increase with the number of shots (input/output sample pairs) in each example. Table 8 shows the results. 4 shots seems to be the optimal, with performance dropping on both sides of that number.

Table 8. Experiment results showing accuracy with varying number of shots.

Task	Split	Runs	Accuracy	Notes
1_shot_rlw	test	100	0.7500	baseline core task
3_shot_rlw	test	100	0.8600	
4_shot_rlw	test	100	0.9600	dynamically created from 5_shot_rlw
5_shot_rlw	test	100	0.9500	
7_shot_rlw	test	100	0.9300	dynamically created from 10_shot_rlw
10_shot_rlw	test	100	0.9200	

The next test covers 4 variants that we thought might help the performance of the model (Table 9). Using English words in place of the 2 letter random words improved performance significantly, as did adding the grammar of the TGT dataset to the system prompt. Using uppercase RLW words (vs. lowercase in core task) hurt performance significantly.

Table 9. Experiment results showing the impact of varying task aspects on accuracy.

Task	Split	Runs	Accuracy	Notes
1_shot_rlw	test	100	0.7500	baseline core task
1_shot_eng	test	100	0.8800	uses English words vs. RLW
1_shot_rlw	test	100	0.8000	adding grammar to prompt
1_shot_rlw	ood_lexical	100	0.5400	uses uppercase RLW words

For our final set of tests, we want to test the ability of the model to generalize outside of the N-shot distribution on the cue we trained it with. For this, we used the 5_shot_rlw task. The first row of Table 10 shows the baseline performance of the model using 5 shots. We then tried the 3 out of distribution cues shown in the table. It can be seen that the model performed significantly worse in these cases, but did not fail as typical non-LLM models tend to do in OOD generalization.

Table 10. Experiment results showing how differing cue/answer distributions affect accuracy in N-shot tasks.

Task	Split	Runs	Accuracy	Notes
5_shot_rlw	test	100	0.9500	baseline 5_shot task
5_shot_rlw	dyn_ood_lexical	100	0.8500	N-shots are lowercase rlw; cue is uppercase RLW
5_shot_eng	dyn_ood_cons_len	100	0.8400	N-shots are cons length=1,2,4; cue is cons length=7
5_shot_rlw	dyn_rev_cons_len	100	0.7100	N-shots are cons length=7; cue is cons length=1,2,4

Received 23 October 2024; accepted 2 October 2025