Gyan: Recursive DSL Reasoning Models for Symbolic Equation Solving

Vihari Kanukollu Collaborators

Abstract

We introduce **Gyan**, a family of recursive reasoning models trained over a structured *Domain-Specific Language* (DSL) rather than byte-pair encoded text. Gyan builds on a recursive transformer architecture originally developed for visual and combinatorial puzzles such as Sudoku, ARC, and mazes, and adapts it to a symbolic equation-solving domain with a rich DSL vocabulary. Instead of predicting the next token autoregressively, Gyan performs iterative non-autoregressive refinement of an entire answer sequence using adaptive computation time (ACT) with nested reasoning cycles.

To make this architecture compatible with a DSL world, we (1) design a unified DSL token space of 457 tokens covering arithmetic, algebra, logical constructs, and code-like primitives; (2) implement a structure-aware dataset builder that converts DSL programs into Gyan-style puzzles, masking the solution span and supervising only the masked answer tokens; and (3) build a large synthetic dataset of 1M equation-solving problems across four algebra modules derived from the DeepMind Mathematics Dataset, with an additional 55-module extension planned.

On this equation-solving benchmark, a 1.9M-parameter Gyan model (hidden size 256, 6 inner reasoning cycles, 128-token sequences) trained for 50 epochs on 1M examples achieves 65% exact answer accuracy, including 70–80% on single-variable equations and 50–60% on coupled two-variable systems. The model learns the correct answer format (e.g., three-token negative numbers `INT_0 INT_N SUB`) and often makes small off-by-one or off-by-few numeric errors, indicating genuine algebraic reasoning rather than pattern memorization. We also compare this configuration to a 7.3M-parameter variant; despite lower language-modeling loss, the larger model does not significantly improve exact match accuracy under comparable compute, suggesting that capacity is not the current bottleneck for this DSL task.

Overall, our results demonstrate that (i) this style of recursive reasoning can be successfully transplanted from perceptual puzzles to symbolic DSL worlds, (ii) task-aligned DSL tokens provide a clean substrate for constraint-satisfaction-style learning, and (iii) compact models can achieve strong performance when data and objective are carefully designed.

1 Introduction

Most modern large language models operate over subword tokens such as byte-pair encodings (BPE). These tokens are syntactic fragments of natural language, often misaligned with the underlying semantic or logical structure. In contrast, symbolic reasoning tasks—such as program synthesis, algebraic manipulation, or puzzle solving—are naturally expressed in structured, typed DSLs.

A compact recursive transformer architecture was proposed as a minimalist testbed for iterative reasoning: it repeatedly refines a latent representation of a candidate solution over multiple outer "H-cycles" and inner "L-cycles" (transformer layers), optionally using Adaptive Computation Time (ACT) to decide how many refinement steps to apply per instance. Originally, this architecture was evaluated on challenging visual and combinatorial puzzles such as Sudoku, ARC, and maze solving, where each "puzzle" corresponds to a structured input grid and a discrete output.

This work explores the hypothesis that the DSL world can be treated analogously to those original puzzle worlds:

- A DSL program plus a masked answer region defines a constraint-satisfaction problem.
- The model's job is to iteratively refine its guesses for the answer tokens until the constraints implied by the DSL are satisfied.
- The DSL tokens themselves carry rich semantic structure (e.g., `EQ`, `REAL_VAR_0`, `INT_7`, `IS_SOLUTION`), making the reasoning problem more direct than when using arbitrary BPE tokens.

We instantiate this idea in **Gyan**, a recursive transformer trained on a math-oriented DSL. The current instantiation focuses on four modules of linear algebra problems:

1. `algebra__linear_1d` – single-variable linear equations,
2. `algebra__linear_1d_composed` – composed single-variable equations with additional structure,
3. `algebra__linear_2d` – coupled two-variable linear systems, and
4. `algebra__linear_2d_composed` – composed two-variable systems.

Input problems are drawn from a modified DeepMind Mathematics Dataset and converted into DSL token sequences by a generator script. We design a Gyan-compatible dataset builder that (i) precisely locates the answer span in the DSL trace using structural markers such as `EQ`, `REAL_VAR_k`, and `IS_SOLUTION`; (ii) masks the answer tokens in the input with `PAD`; and (iii) defines labels that supervise only the answer span while ignoring all context positions. This transforms each algebra problem into a Gyan puzzle: given the full DSL program with a masked solution, fill in the missing answer tokens.

Our experiments show that a 1.9M-parameter Gyan model trained on 1M synthetic examples achieves strong accuracy (65% overall exact match across modules). Errors are predominantly small numeric deviations rather than structural failures, and the model generalizes well to both 1D and 2D equation families. A larger 7.3M-parameter variant improves cross-entropy but does not meaningfully improve exact match accuracy under our current training budget, implying that data and objective design matter more than sheer parameter count in this regime.

2 Background

2.1 Recursive transformer models

The underlying architecture is a minimalist transformer-style network designed for iterative refinement. Instead of generating tokens autoregressively, the model maintains a full sequence of token logits and refines them over multiple reasoning cycles. Key ideas include:

- **H-cycles**: outer iterations that take the current hidden state of the entire sequence and transform it through a sequence of L-level blocks;
- **L-cycles**: inner transformer layers (self-attention + MLP) that implement one "step" of reasoning within an H-cycle;
- **Adaptive Computation Time (ACT)**: a scalar halting probability is predicted at each step, allowing the model to use more steps for harder instances; and
- **Puzzle embeddings**: each puzzle instance belongs to a "puzzle family" (e.g., a specific Sudoku instance, ARC pattern, or module), and the model learns a low-dimensional embedding per identifier.

Originally, this style of model was applied to synthetic puzzles where each training example is a grid or pattern and the objective is to reconstruct a target configuration. The architecture's small size (on the order of a few million parameters) made it feasible to study algorithmic behavior rather than rely on scale.

2.2 Why a DSL world?

In the original puzzle setting, each instance is defined over a bounded, discrete world (for example, a 9×9 Sudoku grid with digits 1–9). A DSL world offers a similarly discrete, bounded representation:

- tokens represent typed operations (e.g., addition, multiplication), variables (`REAL_VAR_k`), constants (`INT_n`, `INT_NEGn`), and problem delimiters (`BOS`, `EOS`, `IS_SOLUTION`);
- the resulting sequences have clear semantic roles: problem statement, intermediate symbolic manipulations, and final answer.

Compared to generic BPE tokens, this design provides:

- direct access to algebraic structure (e.g., identifying `EQ` vs. `ADD` vs. `MUL` tokens),
- the ability to reconstruct numbers exactly from token patterns (sign + magnitude), and
- a clean way to mask and supervise specific semantic spans (the answer) without ambiguity.

This makes the DSL world an appealing testbed for neuro-symbolic reasoning: the network learns over a symbolic surface that closely mirrors the underlying mathematical structure.

3 The Gyan Architecture

3.1 Base model configuration

Gyan builds on the `RecursiveReasoningModel_ACTV1` implementation from our recursive-transformer codebase. We adapt the configuration for DSL reasoning as follows for the small 1.9M-parameter variant:

- **Vocabulary size**: 457 DSL tokens (from `GyanDSLToken`).
- **Sequence length**: 128 tokens per example.
- **Embedding and hidden size**: `hidden_size = 256`.
- **Heads and expansion**: `num_heads = 8`, MLP expansion factor 4.
- **Reasoning cycles**: `H_cycles = 3`, `L_cycles = 6` (inner iterations per H-cycle).
- **Layers**: `L_layers = 2` transformer blocks in the L-level; `H_layers = 0` (unused).
- **Position encoding**: rotary position embeddings (RoPE).
- **ACT settings**: `halt_max_steps = 16` (we effectively use 16 refinement steps), `halt_exploration_prob = 0.1`.
- **Puzzle embeddings**: `puzzle_emb_ndim = hidden_size`, `puzzle_emb_len = 8` or 16 synthetic tokens prepended to encode the module identity.

Even with this compact configuration (∼1.9M parameters), Gyan can solve a substantial fraction of algebra problems when trained with the right dataset and objective.

The large 7.3M-parameter variant increases:

- `hidden_size` from 256 to 512,

- num_heads from 4 to 8, and
- L_cycles from 4 to 6,

while keeping the rest of the design identical. This model is more expressive but also more computationally demanding.

3.2 DSL vocabulary and tokenization

The DSL token set is implemented as an Enum (GyanDSLToken) that assigns a dense integer ID to each token and captures metadata about its kind. The vocabulary includes:

- structural tokens: BOS, EOS, PAD, separators, brackets;
- variables: REAL_VAR_0, REAL_VAR_1, REAL_VAR_2, REAL_VAR_3, etc.;
- integer constants: INT_0, INT_1, ..., INT_99, plus negative encodings such as INT_0 INT_N SUB or specialized INT_NEGn tokens;
- operators: ADD, SUB, MUL, DIV, EQ, comparison operators, and boolean logic;
- higher-level constructs: placeholders for control flow, code IR, and structured data (used by future modules).

The vocabulary size is exposed programmatically via get_vocab_size() and used directly by Gyan's embedding layer. Utility functions token_to_id, id_to_token, and helpers such as get_int_const_token ensure that the token space remains the single source of truth for all DSL operations.

3.3 Input representation and puzzle embeddings

Given an example DSL program, we construct an input sequence as:

1. optional puzzle embedding tokens encoding the module identifier (e.g., which algebra module generated the problem);
2. the BOS token;
3. the full sequence of DSL tokens produced by the generator script, with the answer span masked by PAD (see Section 4);
4. the EOS token.

All sequences are padded or truncated to a fixed length of 128 tokens. Puzzle embeddings allow the model to specialize its reasoning per module while sharing a common backbone. In the current experiments, puzzle embeddings are indexed by module name rather than by individual problem instance, which keeps the number of embeddings small.

3.4 Recursive reasoning and ACT

At each refinement step, Gyan maintains a hidden state for every position in the sequence. An L-level transformer block updates these states using self-attention and MLP layers. After each inner cycle, an ACT head predicts a halting logit for the current step.

In practice, for the equation-solving DSL tasks we observe that:

- the halting probability tends to increase monotonically over steps;
- most examples end up using the maximum 16 steps, effectively turning ACT into a fixed-depth iterative process;
- the primary benefit of the recursive design is thus the multi-step refinement, not early stopping.

Further analysis of the halting head suggests that learning a meaningful early-exit policy is challenging in this regime and may require additional regularization or auxiliary objectives.

3.5 Loss function and training objective

Gyan uses an `ACTLossHead` that combines:

1. a language-modeling loss over tokens, implemented as a numerically stable cross-entropy variant (`stablemax_cross_ent` and
2. a binary cross-entropy loss for the halting decision at each step.

Crucially, our dataset builder sets labels to `-100` (ignore index) for all non-answer tokens. As a consequence:

- the LM loss is non-zero only on the masked answer span;
- the model is explicitly trained to predict the correct answer tokens given the full DSL context, rather than to reconstruct the entire program;
- this prevents trivial "copying" behavior where the model simply reproduces its input.

We found this design to be essential: earlier attempts that used full-sequence supervision led to models that mostly copied the input and achieved misleadingly high token accuracy without solving the underlying equations.

4 Dataset Construction

4.1 From DeepMind Mathematics to GyanDSL

We start from the DeepMind Mathematics Dataset, specifically the algebra modules related to linear equations. Using a custom generator script (`dev/gen_full_math.py`), we:

1. select four modules:
   - `algebra__linear_1d`,
   - `algebra__linear_1d_composed`,
   - `algebra__linear_2d`,
   - `algebra__linear_2d_composed`;
2. for each module, configure `train_per_module = 250,000` and `test_per_module = 10,000`;
3. ensure that both train and test splits are sampled from the same distribution by using `algebra.train(entropy_fn)` for both, rather than the original `train/test` generator pair.

This last step is critical. In the original DeepMind setup, the `train()` and `test()` generators for many modules produce different difficulty distributions (e.g., smaller magnitudes in train, larger in test), which can cause severe distribution shift. By switching both to `train(entropy_fn)`, we obtain an i.i.d. train/test split over the same underlying problem family.

The generator outputs JSONL files with fields such as module name, natural-language question, string answer, DSL token IDs, and token names.

4.2 Structure-aware answer masking

The core of our data pipeline is `dataset/build_dsl_dataset.py`, a structure-aware dataset builder that converts JSONL records into the numpy arrays expected by Gyan's training loop. Key design decisions include:

1. **Module filtering.**

   We currently support an explicit set of equation modules:

   ```
   EQ_SOLUTION_MODULES = {
       "algebra__linear_1d",
       "algebra__linear_1d_composed",
       "algebra__linear_2d",
       "algebra__linear_2d_composed",
   }
   ```

   Examples from other modules are skipped (but the pipeline is extensible).

2. **Span detection via DSL markers.**

   For supported modules, the DSL generator emits a canonical tail pattern:

   ```
   ... EQ REAL_VAR_k <answer_tokens...> IS_SOLUTION EOS
   ```

   We deterministically locate the answer span as the tokens between the last `EQ` followed by `REAL_VAR_k` and the subsequent `IS_SOLUTION` marker. If this pattern is not found or is malformed, the example is skipped.

3. **Input and label construction.**

   We create an input vector `inputs` of length 128, initialized to `PAD`, and copy the `token_ids` sequence into its prefix. We create a label vector `labels` of length 128, initialized to `IGNORE_LABEL_ID = -100`. For the identified answer span `[ans_start:ans_end)`, we set `labels[ans_start:ans_end] = inputs[ans_start:ans_end]` and `inputs[ans_start:ans_end] = PAD_ID`. Thus, the model sees the full problem statement but with the answer masked, and is trained only on the answer positions.

4. **Puzzle identifiers and grouping.**

   Each example is treated as its own puzzle and group: `puzzle_indices = [0, 1, 2, ..., N]` and `group_indices = [0, 1, 2, ..., N]`, where N is the number of usable examples. Puzzle identifiers are derived from module names via a simple mapping from module names to integers.

5. **Metadata.**

   We store a `dataset.json` file per split that records sequence length, vocabulary size, padding and ignore IDs, number of puzzle identifiers, total groups and puzzles, and an `identifiers.json` mapping puzzle IDs back to module names.

This pipeline ensures that supervision is perfectly aligned with the semantic answer and that no heuristics based on natural-language answers are required.

4.3 Dataset statistics

For the 4-module equation dataset built in this work, we obtain approximately:

- train set: 1,000,000 examples (250k per module);
- test set: 40k examples (10k per module);
- most sequences use fewer than 64 tokens but are padded to 128;
- answer lengths are predominantly 1 token (e.g., `INT_7`) or 3 tokens (e.g., `INT_0 INT_37 SUB` for negative integers).

Empirical analysis reveals a roughly balanced sign distribution (about 50% positive vs. 50% negative answers) and a magnitude distribution where most answers have small absolute value but the training set includes a long tail (up to about ±150) while the test set typically covers ±50.

4.4 Extensions to 55 modules

A larger dataset with about 2M examples across 55 mathematical modules is available as a future extension. Our current dataset builder only supports modules that follow the `EQ REAL_VAR_k ... IS_SOLUTION` pattern. Extending Gyan to this broader curriculum requires per-module or per-family answer span rules, careful handling of different DSL idioms (inequalities, multi-step solutions), and potentially new evaluation metrics beyond simple scalar equation solving.

5 Experiments

5.1 Experimental setup

All experiments use our training script (`pretrain.py`) with configuration `cfg_pretrain_dsl.yaml`, which specifies:

- `global_batch_size = 768` (aggregated across GPUs),
- `lr = 1e-4`, `beta1 = 0.9`, `beta2 = 0.95`, `weight_decay = 0.1`,
- cosine or flat learning-rate schedules with 200–2000 warmup steps,
- optional EMA,
- evaluation on a held-out test split at regular intervals.

We train on GPU clusters (e.g., H200) using `torchrun` for distributed data-parallel training. Unless otherwise noted, we treat each module identically and focus on exact match accuracy (EM) over the answer tokens as our primary metric.

5.2 Baseline: small model on 100k examples

As an initial baseline, we trained the small Gyan model (`hidden_size = 256`, ∼1.9M parameters) on a 100k-example dataset (25k per module) for 500 epochs: about $100\text{k} \times 128 \times 500 \approx 6.4\text{B}$ training tokens. This aggressive reuse of a small dataset leads to high training accuracy (approaching 90% EM) but evaluation EM around 50% on the test split and clear signs of overfitting. This experiment validated the correctness of the masking pipeline, the ability of Gyan to learn non-trivial algebraic rules from DSL tokens, and the approximate scale at which convergence occurs for this architecture.

5.3 Scaling data: 1M examples

To mitigate overfitting and test the impact of data diversity, we scaled the dataset to 1M examples (250k per module) while keeping the total training tokens comparable ($\approx 6.4\text{B}$, via 50 epochs). Thus each example is seen about 50 times instead of 500.

For the small model (`hidden_size = 256`), at around step 52k (about 40 of 50 epochs), evaluation on a 40-example batch yields: 70% EM on `algebra__linear_1d`, 80% on `algebra__linear_1d_composed`, 60% on `algebra__linear_2d`, 50% on `algebra__linear_2d_composed`, and 65% overall EM. Errors are dominated by small numeric deviations and occasional sign mistakes, while the model nearly always maintains the correct structural form of the answer.

Unlike the 100k-example run, the 1M-example training curve continues to improve through 50 epochs, indicating that data rather than compute was the primary bottleneck in the smaller-dataset regime.

5.4 Scaling model size: 1.9M vs. 7.3M parameters

We next compared the small Gyan model to a larger variant with approximately 7.3M parameters (`hidden_size = 512`, `L_cycles = 6`, `num_heads = 8`) on the same 1M-example dataset. Under a modest three-epoch schedule (∼150M tokens), the larger model achieves lower LM loss but only about 17% EM, consistent with an undertrained regime.

When training both models for comparable total tokens (on the order of 6.4B), the larger model continues to win on cross-entropy, but the smaller model reaches higher or comparable EM at the same number of optimization steps and converges faster. This suggests that, for this DSL equation-solving task, capacity is not the limiting factor: 1.9M parameters already suffice to capture the necessary algebraic structure, and extra capacity mainly helps fit token distributions rather than discrete reasoning.

5.5 Error analysis and ACT behavior

Across modules, we observe consistent error patterns: off-by-one or off-by-few magnitude errors (especially for larger integers), occasional sign flips, and very few structural errors. The model almost always obeys the expected answer format (single `INT_k` for positive k, `INT_0 INT_k SUB` for negative k), indicating that it has largely internalized the symbolic structure and is focusing its remaining capacity on fine-grained numeric precision.

The halting mechanism behaves similarly across runs: the halting probability `q_halt` increases gradually with each step, and the model typically uses all 16 steps even on easy problems. Over training, `q_halt` accuracy decreases slightly while its loss increases, suggesting that the halting head is not yet learning a strong early-exit policy; ACT effectively acts as a fixed-depth unrolled recurrent computation.

6 Discussion

6.1 DSL vs. BPE tokenization for reasoning

Gyan's performance supports the hypothesis that DSL tokens aligned with task semantics are more effective for reasoning than generic BPE tokens. The model never has to infer that a minus sign followed by digits represents a negative number; the DSL encodes sign and magnitude explicitly. Algebraic operations (`ADD`, `SUB`, `MUL`, `EQ`) are explicit tokens rather than dispersed over multiple subwords, and markers such as `IS_SOLUTION` let us pinpoint supervision exactly on the answer span. This clarity likely contributes to strong performance at relatively modest scale (1.9M parameters, 1M examples).

6.2 Gyan vs. standard transformers

Standard autoregressive transformers can learn algebraic reasoning but typically require much larger parameter counts, careful prompt engineering, and large-scale pretraining on heterogeneous text. Gyan instead uses a non-autoregressive, iterative refinement architecture trained from scratch on a narrow, synthetic DSL corpus and focuses exclusively on producing correct answers given full context. These results suggest that specialized recursive reasoning architectures like Gyan remain competitive for targeted domains when the input representation is well aligned with the problem structure.

7 Limitations and Future Work

Despite promising results, several limitations remain. First, the current experiments focus on four algebra modules from the DeepMind Mathematics Dataset. Extending Gyan to the full 55-module DSL curriculum—or to other domains such as logic, program synthesis, or ARC-style tasks—will require additional work on dataset construction and curriculum design.

Second, ACT does not yet yield meaningful early exits: most examples run for the maximum number of steps. Future work could add regularizers that penalize unnecessary computation, design auxiliary tasks that encourage earlier halting on easy problems, or experiment with alternative recurrent scheduling schemes.

Third, evaluation is currently limited to exact match accuracy over answer tokens. Larger-scale evaluation—including robustness to distribution shift (e.g., larger magnitudes, more complex compositions) and interpretability analyses of intermediate refinement steps—is left for future work.

Finally, while we conceptually compare Gyan to standard transformers, a full empirical comparison (e.g., training a same-size autoregressive transformer on the same DSL dataset) remains to be done.

Future work will explore integrating additional modules and symbolic domains into the DSL pipeline, scaling up the data generator to richer forms of constraint satisfaction (SAT-like problems, logical puzzles), experimenting with larger Gyan variants when the task demands more capacity, and using Gyan as a neuro-symbolic module inside larger systems where it can act as a specialized solver for DSL-encoded subproblems.

8 Conclusion

We presented Gyan, a recursive reasoning model that operates over a structured DSL to solve equation-solving problems as constraint-satisfaction tasks. By designing a recursive architecture tailored to the DSL world, building a structure-aware dataset builder that masks answer spans, and training on a large synthetic corpus of algebra problems, we show that:

- a 1.9M-parameter model can achieve 65% exact match accuracy across four linear equation modules;

- the model learns robust structural understanding of the DSL and primarily fails through small numeric errors;
- simply increasing model size to 7.3M parameters does not guarantee better discrete reasoning performance under comparable compute.

These results underscore the importance of representation, objective design, and recursive reasoning in building compact yet effective neuro-symbolic systems. Gyan demonstrates that, with an appropriate DSL and dataset, small recursive models can be powerful equation solvers, providing a promising direction for future work at the intersection of symbolic reasoning and deep learning.