Minimal Grammars as Supervision for DSL Generation

Anonymous ACL submission

Abstract

Language models approach saturation on general-purpose code benchmarks but routinely fail on domain-specific languages (DSLs), where paired training data is scarce and regularities learned from mainstream languages transfer poorly. We propose augmenting each training example with a minimal grammar: the subset of DSL productions needed to derive the target program, typically 2–20% of the full grammar. This provides a strong structural prior, and we fine-tune the model to generate the program conditioned on the query and grammar. We predict grammars at inference via retrieval: a frontier model is prompted with (query, grammar) pairs, annotated with a chain-of-thought trace. Across five benchmarks spanning semantic parsing (SMCalFlow, GeoQuery, Overnight) and hardware description languages (HDLs) (Verilog, SPICE), our method improves over grammar-free fine-tuning on three open-weight backbones. Gains reach 36.8pp of program accuracy on SMCalFlow, with consistent improvements on HDLs. Applied to in-context learning, our predicted grammars raise GPT-5.5 accuracy on SMCalFlow from 51.4% (grammar prompting) to 94.3%, with similar gains for Claude Opus 4.7. Ablations rule out the extra training tokens as the source of these gains.

1 Introduction

Recent advances in language models (LMs) have yielded strong performance on general-purpose code generation, saturating benchmarks considered challenging only a few years ago such as HumanEval and MBPP (Zhuo et al., 2025; Jain et al., 2025; Chambon et al., 2025). For languages well represented in pretraining, these models produce correct code at rates approaching those of a competent programmer (Quan et al., 2025). However, progress is uneven. Models that near saturation on general benchmarks routinely produce plausible but incorrect code on domain-specific language (DSL) tasks (Lei et al., 2025; Xu et al., 2025).

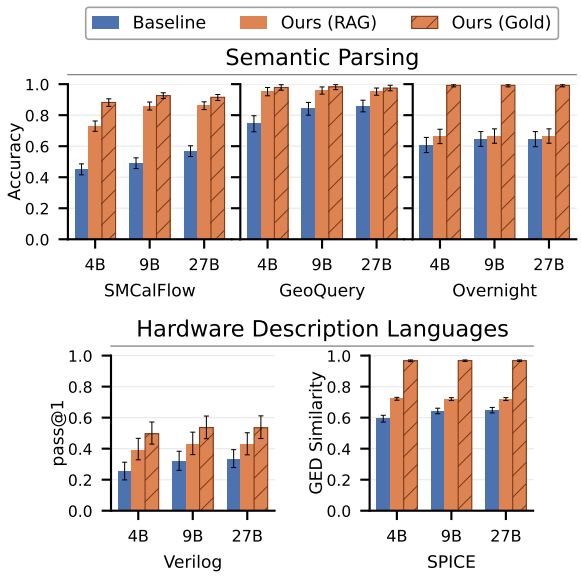


Figure 1: Minimal-grammar supervision (orange) improves over grammar-free fine-tuning (blue) across SP and HDL benchmarks.

Two factors explain the gap. First, targeted fine-tuning data available for DSLs exists at a small fraction of general-purpose-language scale, so a model has had far less exposure to the patterns it needs to reproduce (Liu et al., 2025; Pinckney et al., 2025). Second, DSLs differ structurally from the distribution a model learns, in their syntax and semantics: the regularities a model absorbs from mainstream programming languages do not transfer cleanly (Joel et al., 2024; Cassano et al., 2024).

We propose a method for making the most of existing training data by augmenting each example program with minimal grammars: subsets of DSL rules needed to generate a particular program. They contain the operations, constants, and composition patterns that appear in that example, and omit everything else. This provides a scaffold for the model to acquire the structure of the target DSL.

With ground-truth grammars at inference, the trained model reaches a high performance ceiling.

Since hand-written grammars are rarely available in practice, we develop a method for producing these grammars automatically using retrieval-augmented generation (RAG): the test query is embedded and the k nearest train queries are retrieved. We extract minimal grammars from their associated programs and prompt a frontier model with these query-grammar pairs as in-context examples. Appending chain-of-thought (CoT) reasoning to the in-context examples produces further improvements.

We illustrate our approach on five benchmarks spanning hardware description languages (HDLs: Verilog, SPICE) and semantic parsing (SP: SMCalFlow, GeoQuery, Overnight), comparing against two baselines: training without grammars on the same open-weight models (Qwen-3.5 4B and 9B (Qwen Team, 2026), Gemma-3 27B (Kamath et al., 2025)) and grammar prompting (Wang et al., 2023) on frontier models (GPT-5.5 (OpenAI, 2026) and Claude Opus 4.7 (Anthropic, 2026)). The latter prompts an LM to generate a minimal grammar *and* program from k in-context examples.

Our approach yields substantial improvements in accuracy across several metrics: pass@k and graph edit distance for electrical circuits; exact match and execution accuracy for SP. Our RAG-based method achieves gains of up to +36.8pp on exact match accuracy on SMCalFlow (49.0%→85.8%) over no-grammar training, with improvements on all five benchmarks (Figure 1). Applying our RAG-predicted grammars to frontier in-context learning (ICL) on the same benchmark yields a +42.9pp gain over grammar prompting (51.4%→94.3%). We verify that these gains do not stem from the extra training tokens grammars introduce. Training for additional epochs without grammars does not match the accuracy of training with them.

To summarize, our contributions are as follows:

1. We pair each training program with a minimal grammar capturing only productions it uses, so the model internalizes the DSL's structure rather than inferring it from programs alone.

2. We develop a retrieval-based procedure for predicting these grammars automatically, prompting the model with nearest-neighbor exemplars and reasoning traces.

3. Across five benchmarks spanning SP and HDLs, our method consistently improves over both grammar-free fine-tuning and grammar prompting with frontier models.

```
answer_type ::= num
num         ::= "len(" river ")"
river       ::= "river(" river ")"
             |  "riverid('colorado')"
```

Figure 2: Example minimal grammar for GeoQuery.

2 Background

2.1 Problem Formulation: DSL Generation

Given a natural-language utterance x, the goal is to generate a formal meaning representation $y \in L(G)$ that conforms to grammar G, where $L(G)$ is the set of all well-formed expressions derivable from G.

A prediction $\hat{y}$ is deemed correct if (i) $\hat{y} = y$, or (ii) $[\![\hat{y}]\!] = [\![y]\!]$, where $[\![\cdot]\!]$ denotes the semantic interpretation function mapping expressions to their execution outcomes. Example inputs and outputs are shown for GeoQuery and Verilog in Table 1.

2.2 Minimal Grammar

We adopt the definition of a minimal grammar[1] from Wang et al. (2023). Given G and a program y, a minimal grammar $G(y)$ is a subset of the base grammar with the following properties:

1. $y \in L(G(y))$.

2. $\forall r \in G(y), y \notin L(G(y) \setminus r)$.

Intuitively, removing any rule from $G(y)$ would mean that y is no longer a valid program allowed by the grammar. Only the production rules required for y remain. Figure 2 shows the minimal grammar for the GeoQuery example in Table 1.

3 Method

Our method has two components. During training, we append an automatically extracted minimal grammar to each query and fine-tune the model to generate the corresponding program. At inference, the gold grammar is unavailable, so we predict it via retrieval-augmented prompting. The two components are decoupled: a model fine-tuned with minimal grammars can be paired with any grammar predictor at test time, and a stronger predictor can be swapped in without retraining.

Figure 3 summarizes both components. At inference (panel a), the fine-tuned LM consumes the

[1] Wang et al. use the term "minimal specialized grammar", which we shorten to "minimal grammar".

	x	y	$[\![y]\!]$
GeoQuery	How long is the Colorado River?	`answer(len(river(riverid('colorado'))))`	2333 miles

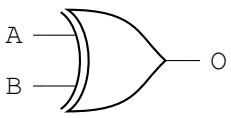
Verilog	An XOR gate over inputs `A` and `B` producing output `O`, implemented as $\overline{A}B + A\overline{B}$.	`module XOR(O, A, B);`
		`input wire A, B;`
		`output wire O;`
		`assign O = (~A)*B + A*(~B);`
		`endmodule`
			
			A B O
			0 0 0
			0 1 1
			1 0 1
			1 1 0

Table 1: Problem formulation illustrated across two DSL families. For Verilog, the schematic next to y visualizes the module that y describes. The denotation $[\![y]\!]$ is the module's input/output behavior, given here as a truth table; for GeoQuery, it is the value returned by executing the FunQL program against the geography database.

query alongside a predicted minimal grammar and emits the program. For the GeoQuery example shown, the two productions for `num` and `state` are enough to pin down `count(state(all))`. The grammar itself is produced by the retrieval-augmented pipeline in panel (b).

3.1 Training with Minimal Grammars

Each training example is a pair (x, y) of natural-language query and gold program. We compute the minimal grammar $G(y)$ by parsing y with the reference grammar G and collecting the productions used in deriving it. The model is trained to autoregressively generate y given the concatenation of x and a serialized form of $G(y)$: $(x,\ G(y)) \ \rightarrow\ y$.

We considered training the model to jointly generate both the grammar and the program ($x \rightarrow G(y) \cdot y$), the closest training-time analogue to grammar prompting (Wang et al., 2023), but found on SMCalFlow and Verilog that this did not improve over the no-grammar baseline (Section 4.5).

We fine-tune three open-weight models with loss computed only on the y tokens: Qwen-3.5 4B, Qwen-3.5 9B, and Gemma-3 27B. Full hyperparameters appear in Appendix C.

Robustness to imperfect grammars. Predicted grammars at inference may omit productions that y needs or include spurious ones. To prevent the model from relying entirely on the grammar being correct, we drop $G(y)$ from a randomly sampled 10% of training examples, leaving only $x \rightarrow y$.

Generic productions. Some grammars include productions that match open classes of literals, such as `string ::= ESCAPED_STRING` in SMCalFlow, which matches any quoted string. We replace these generic productions with concrete instantiations: when y contains the literal `"Bob"`, the minimal grammar contains `string ::= "Bob"` rather than the generic rule.

3.2 Grammar Prediction

We predict minimal grammars at inference using RAG with a frontier model. We explored some alternatives that underperformed in pilot experiments on SMCalFlow: (i) fine-tuning a model to predict minimal grammars directly, (ii) training a multi-label classifier with a binary include/exclude head per grammar rule, and (iii) k-NN retrieval in which the predicted grammar is formed from the union or intersection of the retrieved neighbors' rules.

Given a test query x, we embed x using BGE-large-en (Xiao et al., 2024) and retrieve the k nearest training queries by cosine similarity. For each retrieved neighbor (x_i, y_i), we compute the minimal grammar G_{y_i} from program y_i. We then prompt a frontier model with these k (query, grammar) pairs as in-context examples and elicit a predicted grammar $\hat{G}_x$ for the test query. We use $k = 16$ throughout – all ICL baselines in Table 4 are matched to this value.

For each training example, we use GPT-5.5 to pre-generate a CoT trace that explains why the gold minimal grammar suffices for the program and why competing productions in the full grammar are not needed. At inference, these traces are appended to the retrieved in-context examples. Prompt templates are available in Appendix D. We provide the generated CoT traces in the supplementary materials and will release them publicly with the paper.

3.3 Datasets

We evaluate on five benchmarks spanning two DSL families. We include SP for comparability with Wang et al. (2023), and hardware verification to test generalization to a substantially different setting: HDLs are full programming languages with

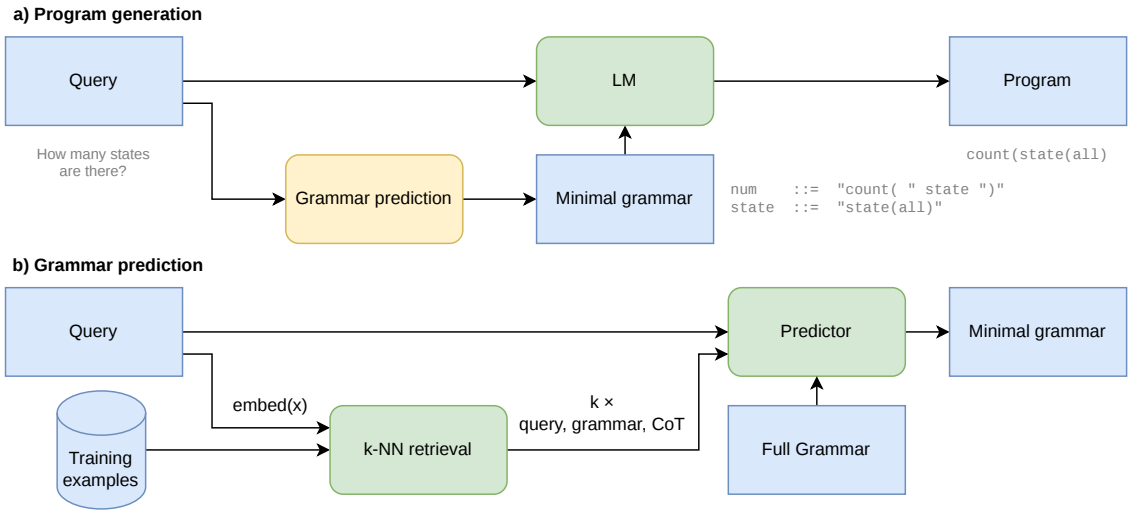


Figure 3: Overview of our method.

Benchmark	Train	Dev	Test	$\lvert G\rvert$	$\lvert G(y)\rvert$
Semantic Parsing					
SMCalFlow	20,962	720	720	291	12.9
GeoQuery	560	60	280	237	4.7
Overnight	1,436	160	399	53	4.6
Hardware Verification					
Verilog	8,573	1,071	156	332	65.2
SPICE	4,848	606	606	185	9.1

Table 2: Dataset statistics. $|G|$ is the number of productions (rule alternatives) in the full reference grammar. $|G(y)|$ is the average number in the minimal grammars.

types, scoping, and hierarchical modules, whereas SP targets are tree-structured logical forms.

Table 2 reports dataset sizes, the size of each benchmark's full reference grammar ($|G|$) and the average size of the per-program minimal grammars ($|G(y)|$). The ratio $|G(y)|/|G|$ varies from roughly 2% (GeoQuery) to 20% (Verilog). A smaller ratio means the minimal grammar prunes away more alternatives, so it carries more information about the target program. All else equal, this would predict larger gains on benchmarks with lower $|G(y)|/|G|$.

Semantic Parsing. SMCalFlow (Andreas et al., 2020) pairs conversational requests to a virtual assistant with dataflow programs. GeoQuery (Zelle and Mooney, 1996) contains questions over a U.S. geography database paired with FunQL logical forms. Overnight-Blocks (Wang et al., 2015) is the Blocks sub-domain of the Overnight benchmark, pairing utterances about stacked-block configurations with λ-DCS logical forms. We follow the splits used in Wang et al. (2023).

Hardware Verification. Verilog is an HDL for digital circuit design. MG-Verilog (Zhang et al., 2024) pairs natural-language module specifications with synthesizable Verilog implementations spanning combinational logic, sequential elements, and finite state machines. The dataset provides descriptions at three granularities (high-level, detailed, and block-level) and proposes a balanced scheme that mixes them. We train on the detailed descriptions only, to keep the effect of our minimal-grammar supervision separable from the gains attributable to balanced mixing. Following Zhang et al. (2024), we evaluate on VerilogEval-Human (Liu et al., 2023), which provides executable testbenches. SPICE (Bhandari et al., 2024) targets analog and mixed-signal circuits. The two HDLs differ in granularity: SPICE programs are netlists specifying topology and component values, whereas Verilog programs are hierarchical with control flow, leading to the larger average minimal grammar reported in Table 2.

3.4 Evaluation Metrics

For all SP datasets, we report program accuracy: the fraction of exact matches ($\hat{y} = y$) after whitespace normalization. For GeoQuery and Overnight-Blocks, we additionally report execution accuracy. SMCalFlow has no execution environment.

For Verilog, we report pass@k (Chen et al., 2021), a sample-based estimator of execution accuracy where the VerilogEval-Human testbench checks input/output behavior. We draw five sam-

ples at temperature 0.8.

SPICE does not admit either Section 2.1 correctness condition cleanly, since netlists have many equivalent textual orderings, and analog simulation yields continuous waveforms that resist exact comparison. Instead, we use two graded structural metrics. GED similarity computes normalized graph edit distance between predicted and gold netlists encoded as labeled graphs. Component F1 is F1 over the sets of component categories present in each netlist, ignoring instance names, multiplicities, parameter values, and connectivity.

4 Results

Figure 1 shows each benchmark's primary metric: program accuracy for SMCalFlow, execution accuracy for GeoQuery and Overnight, pass@1 for Verilog, and GED similarity for SPICE. Error bars are 95% percentile bootstrap confidence intervals over the test set (1,000 resamples). Table 3 reports the full set of metrics.

4.1 Semantic Parsing

Our method yields large gains on SMCalFlow and GeoQuery for all three open-weight models. SMCalFlow program accuracy rises from 49.0 to 85.8 for Qwen-3.5 9B, with comparable gains for the 4B and 27B models, and GeoQuery execution accuracy improves by $\sim$10pp on average. The RAG-predicted grammars close the gap to the gold-grammar ceiling on both benchmarks to within $\sim$7pp on SMCalFlow and $\sim$10pp on GeoQuery.

These gains reflect the $|G(y)|/|G|$ ratios in Table 2: GeoQuery and SMCalFlow sit at the low end (2% and 4%), so the minimal grammar prunes a large fraction of alternatives and acts as a strong structural prior over the derivations that remain.

Overnight is the exception. The predicted-grammar method is similar to baseline ($45.6 \rightarrow 47.4$ for Qwen-3.5 9B), yet the gold-grammar ceiling jumps to 97.2. We attribute this to annotation ambiguity: the same query maps to distinct minimal grammars. Each block stores its own `above` and `below` pointers, populated independently and not constrained to be inverses. The query "blocks above block 1" has two realizations: filtering on each block's `above` field or inverting block 1's `below` field. The two forms would yield identical answers in a physically consistent world, but return different sets of blocks in this dataset. The retrieved neighbors disagree on which form to use for 91% of test queries involving a spatial relation. The gold grammar encodes that choice directly.

4.2 Hardware Description Languages

Verilog pass@1 improves by 9–14pp across the three models, and SPICE GED similarity by 7–13pp, consistently across model sizes. The gap between HDL and SP gains reflects a structural limit of grammar-based supervision on HDLs. HDLs require topological reasoning: identifying how components connect. Topological reasoning is not directly aided by the grammar, which enumerates which components a program uses but not how they are wired together. In the SP domains studied, the target is a tree-structured logical form whose shape the grammar largely determines.

The gold grammar ceiling for Verilog reflects this: even perfect minimal grammars leave pass@1 near 53.6, indicating that richer supervision (e.g. explicit connectivity) could push further.

SPICE behaves differently: all three models converge on a near-ceiling GED of 96.7 with gold grammars. Without them, baseline GED scales mildly with capacity ($59.4 \rightarrow 64.8$) while our method reaches $\sim$72 across all three backbones. SPICE's per-example grammars are smaller than Verilog's ($|G(y)| = 9.1$ vs. 65.2), so a gold grammar largely determines the netlist and leaves little topological work for the model. Predicting these grammars still requires topological reasoning, explaining the persistent gap between RAG and gold.

Moving from pass@1 to pass@5 on Verilog narrows the gap between our method and the baseline. On Gemma-3 27B, the baseline pass@5 even surpasses ours (55.8 vs. 49.4). We suspect grammar conditioning concentrates probability mass on a particular structural skeleton, so the baseline's broader distribution benefits more from additional attempts. For single-shot evaluation, our method remains the stronger choice across all three models.

4.3 In-Context Learning

For zero-shot experiments, we appended the name of the benchmark (e.g. GeoQuery) to the prompt. Both frontier models score near zero on SMCalFlow and Overnight zero-shot but achieve non-trivial accuracy on GeoQuery (26.1 and 40.7 program accuracy for GPT-5.5 and Opus respectively, shown in Table 4). Producing well-formed FunQL without exemplars requires prior exposure to the syntax, so we take this gap as evidence that GeoQuery appears in their training data.

		Semantic Parsing					Hardware Verification				
		SMCalFlow	GeoQuery		Overnight		Verilog			SPICE	
Model	Method	Prog.	Prog.	Exec.	Prog.	Exec.	pass@1	pass@3	pass@5	GED	F1
Qwen-3.5 4B	Baseline	45.0	61.1	74.6	41.1	60.7	25.5	35.1	39.1	59.4	77.4
	Ours	72.9	86.8	95.4	47.4	66.4	39.2	42.5	43.6	72.2	81.8
	Ours + Gold	88.2	96.4	97.9	97.2	99.2	49.6	55.5	57.7	96.7	94.3
Qwen-3.5 9B	Baseline	49.0	75.0	84.3	45.6	64.7	32.2	42.5	46.8	64.3	79.4
	Ours	85.8	87.5	96.1	47.4	66.7	42.8	46.5	48.7	72.0	81.9
	Ours + Gold	92.6	97.5	98.2	97.2	99.2	53.6	59.1	60.9	96.7	94.0
Gemma-3 27B	Baseline	56.8	71.8	86.1	47.9	64.7	33.5	48.4	55.8	64.8	79.6
	Ours	86.1	87.5	95.4	47.4	66.7	42.7	47.4	49.4	72.0	81.8
	Ours + Gold	91.5	96.8	97.5	97.2	99.2	53.5	61.3	63.5	96.7	94.2

Table 3: Main results across SP and hardware verification benchmarks. All values are percentages.

		SMCalFlow	GeoQuery		Overnight	
Model	Method	Prog.	Prog.	Exec.	Prog.	Exec.
Codex-davinci-002	Grammar prompting*	50.8	67.1	87.5	34.8	57.4
GPT-5.5	Zero-shot	0.1	26.1	68.9	0.0	2.5
	Programs only	46.4	61.4	82.9	34.1	52.6
	Grammar prompting	51.4	58.9	82.9	37.6	54.1
	RAG + CoT (ours)	**94.3**	**89.6**	**94.3**	**48.6**	**64.7**
Claude Opus 4.7	Zero-shot	1.4	40.7	76.1	0.0	10.3
	Programs only	51.3	71.8	84.6	34.1	53.1
	Grammar prompting	49.4	72.1	83.9	33.1	51.9
	RAG + CoT (ours)	**91.2**	**85.4**	**95.0**	**50.9**	**68.4**

Table 4: Program and execution accuracy on SMCalFlow, GeoQuery, and Overnight. Within each model's column, the best result is **bold** and the second best is underlined. *Reported by Wang et al. (2023).

The "Programs only" and "Grammar prompting" rows of Table 4 reuse Wang et al. (2023)'s k=16 demonstrations, which for SMCalFlow are curated cross-domain compositional examples from SMCalFlow-CS (Yin et al., 2021).

Grammar prompting does not meaningfully improve over the programs-only baseline on either frontier model, with differences under 6pp across all three benchmarks. It also fails to improve over the code-specialized variant of GPT-3 (Codex-davinci-002) used by Wang et al. (2023): roughly flat on SMCalFlow (50.8 vs. 51.4 / 49.4) and Overnight (34.8 vs. 37.6 / 33.1), with notable regressions on GeoQuery program accuracy for GPT-5.5 (67.1 → 58.9) and on execution accuracy for both models (87.5 → 82.9 / 83.9). Substantial base model improvement between Codex-davinci-002 and these frontier models does not translate into gains under grammar prompting.

We hypothesize that grammar prompting originally served as CoT scaffolding for weaker models. Frontier models now produce intermediate reasoning by default, so the scaffolding is redundant. On pretraining-familiar benchmarks like GeoQuery, it may even interfere with the model's preferred derivation. Most of the remaining value comes from the content of the grammar itself.

Retrieving exemplars by embedding similarity and conditioning on per-example CoT traces (our method) yields large gains over both baselines across all three SP benchmarks and both frontier models, indicating that which grammars are shown as exemplars makes a substantial difference.

4.4 Grammar Prediction Quality

Figure 4 reports grammar quality by category and benchmark. SP benchmarks reach the highest exact match rates (89.6% on GeoQuery, 81.8% on SMCalFlow, 58.6% on Overnight), while Verilog (17.3%) and SPICE (18.0%) sit lowest. Yet on Verilog and SPICE, where fewer than one in five predicted grammars match the gold exactly, our method still outperforms the no-grammar baseline in Table 3. The grammar-dropout of Section 3.1 likely contributes by teaching the model to treat the grammar as a hint rather than a strict specifi-

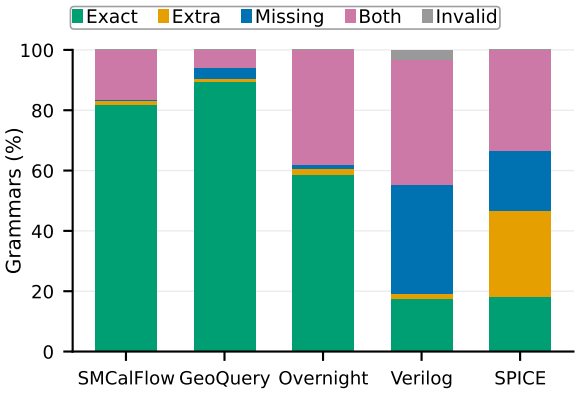


Figure 4: Distribution of predicted grammar quality on each benchmark's test set. *Exact*: the predicted grammar matches the gold minimal grammar. *Extra*: contains every gold rule plus at least one spurious rule. *Missing*: omits at least one gold rule but adds no spurious ones. *Both*: omits at least one gold rule and adds at least one spurious rule. *Invalid*: fails to parse (e.g., references an undefined nonterminal).

Method	SMCalFlow (Prog.)	Verilog (pass@1)
Baseline (1 epoch)	49.0	32.2
Baseline (2 epochs)	62.4	36.0
Joint generation	42.5	21.5
RAG without CoT	67.0	41.0
Ours (RAG)	85.8	42.8

Table 5: Ablation results on SMCalFlow and Verilog for Qwen-3.5 9B.

cation, so spurious or missing productions do not catastrophically degrade generation.

Grammar size and program length shape prediction quality: Verilog's 332-rule grammar and 65.2-rule programs leave more opportunities to miss productions, consistent with missing-heavy errors (77.6% involve at least one missing rule). Pretraining exposure also plays role: Overnight has more training data than GeoQuery (1,436 vs. 560) and a smaller grammar (53 vs. 237 productions), yet exact-match accuracy is 31pp lower. This is consistent with the zero-shot gap between the two benchmarks in Table 4, suggesting GeoQuery is familiar from pretraining.

4.5 Ablations

Table 5 reports ablations on Qwen-3.5 9B for SMCalFlow and Verilog, the largest training sets in each domain.

Token count Appending the minimal grammar lengthens each training example, exposing the model to more tokens per epoch than the baseline. To rule out token count as the source of our gains, we train the baseline for a second epoch. This closes part of the gap but leaves substantial room: $49.0 \rightarrow 62.4$ on SMCalFlow and $32.2 \rightarrow 36.0$ on Verilog, still 23.4pp and 6.8pp short of our method.

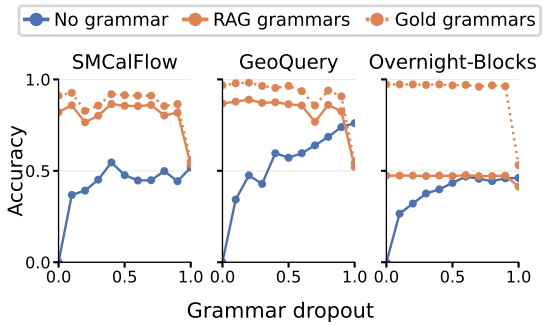


Figure 5: Effect of grammar dropout during fine-tuning. We vary the number of training examples that omit the minimal grammar, then evaluate the resulting model with and without grammars at test time.

Joint generation We next consider the training-time analogue of grammar prompting: fine-tuning the model to emit the grammar followed by the program, $x \rightarrow G(y) \cdot y$. This degrades performance below the no-grammar baseline, dropping to 42.5 on SMCalFlow and 21.5 on Verilog. We attribute the failure to teacher forcing: gradient updates always condition the program continuation on the exact gold grammar, welding the two in weight space so that errors in sampled grammars propagate into the program at inference. In-context grammar prompting (Section 4.3) presents the same two tasks without this coupling and is roughly flat rather than harmful relative to its baseline.

Grammar-free fallback. Figure 5 sweeps the grammar dropout rate from Section 3.1 on the three SP benchmarks. With-grammar accuracy, under both RAG-predicted and gold grammars, remains nearly flat from dropout 0 to 0.8, while no-grammar accuracy rises steeply and matches the grammar-free baseline by 0.9. A model trained at dropout 0.9 performs near peak when a grammar is supplied and recovers baseline accuracy when one is not.

Retrieval vs. retrieval+CoT We isolate the contribution of exemplar retrieval by running the same pipeline without CoT traces. Retrieval alone improves over the no-grammar baseline by 18.0pp on SMCalFlow and 8.8pp on Verilog, confirming that selecting relevant exemplars is the dominant source

of gains. Adding CoT traces further improves SMCalFlow (+18.8pp), where grammar prediction has substantial headroom relative to the gold-grammar ceiling of 92.6. On Verilog, where the gold ceiling is 53.6 pass@1, CoT adds a smaller +1.8pp.

5 Related Work

Structure as training signal A long line of work injects formal structure into models for SP, either at training time or via ICL. Wang et al. (2023) introduce minimal grammars for ICL, using them as a proxy for CoT reasoning. We build on this approach in three ways: (i) We treat minimal grammars as a fine-tuning signal, supervising open-weight models to internalize DSL structure; (ii) We decouple grammar and program generation – the natural training analogue of grammar prompting (jointly generating both) does not improve over the no-grammar baseline; (iii) Our grammar predictor retrieves exemplars via embedding similarity and conditions on per-example CoT traces. Gong et al. (2024) inject structure at pretraining by using abstract syntax trees (ASTs) to shape the masking objective. Herzig et al. (2021) propose training with reversible and lossy intermediate representations for SP and Fuchs et al. (2024) extend it to a new formalism. Both require the intermediate representations to be designed by hand.

Retrieval-augmented generation for DSLs Lewis et al. (2020) introduce RAG and Liu et al. (2022) show that, for ICL specifically, selecting exemplars by embedding similarity to the test input substantially outperforms random sampling. Li et al. (2025b) argue that embedding similarity alone misses structural alignment in SP and propose a two-stage retriever combining semantic and structural signals. Zhao and Chen (2025) retrieve structural graph embeddings of relevant Verilog modules to generate structure-aware prompts. Drozdov et al. (2022) decompose the test utterance and retrieve exemplars matched to each sub-utterance. Building on this approach, Kothyari et al. (2025) decompose the training pool into AST fragments and retrieve diverse fragment-level exemplars.

Fine-tuning for DSLs DSL-specific fine-tuning is limited by data scarcity (Joel et al., 2024; Cassano et al., 2024). Recent work addresses this via synthetic data. Li et al. (2025a) generate a million-scale dataset for text-to-SQL and fine-tune open-source models that match closed-source frontier models. Guo et al. (2025) synthesize text-to-SQL data by extracting syntactic templates from parsed SQL queries, reusing them to generate new training pairs. Liu et al. (2025) target Verilog with synthetic data tailored to known failure modes. MG-Verilog (Zhang et al., 2024), which we use for training, pairs each Verilog module with LLM-generated descriptions at three granularity levels and proposes a balanced fine-tuning scheme that mixes them. Similarly, Masala-CHAI (Bhandari et al., 2024) pairs SPICE netlists with LLM-generated captions extracted from textbook schematics. These approaches expand the training set. We instead add structural signal to each existing example.

6 Conclusion

We introduced minimal grammars as a per-example supervision signal for DSL generation. By appending to each training query the small subset of productions actually used by its program, we expose the model to the DSL's structure example by example rather than asking it to infer that structure from programs alone. At inference, we predict these grammars via retrieval-augmented prompting with a frontier model, conditioned on chain-of-thought traces over the retrieved neighbors. Training and prediction are decoupled, so either component can be improved in isolation.

Across five benchmarks spanning SP and HDLs, the method consistently improves over grammar-free fine-tuning on three open-weight backbones. Predicted grammars need not be exact: on Verilog and SPICE, fewer than one in five match the gold exactly, yet the method improves over the no-grammar baseline. Applied to ICL with frontier models, the same predicted grammars substantially outperform standard grammar prompting, which we find no longer improves over a programs-only baseline on GPT-5.5 and Claude Opus 4.7. Ablations rule out extra training tokens as the source of these gains and show that the training-time analogue of grammar prompting (jointly generating grammar and program) underperforms the no-grammar baseline.

Future work might refine predicted grammars iteratively using parser or simulator feedback to recover missing productions. Minimal grammars could also be extended to encode connectivity between components, addressing the topological ceiling we observe on HDLs.

Limitations

Three limitations are worth noting. First, minimal grammars enumerate the productions a program uses but not how its symbols compose into structure. Where correctness depends on connectivity rather than symbol choice, as in the case of HDLs, a fraction of the problem lies outside what the grammar constrains. The gold-grammar ceiling on Verilog (53.6 pass@1) reflects this bound on our method.

Second, computing minimal grammars at training time presupposes a parseable reference grammar. The DSLs studied here all admit one, but some practical targets (informally specified configuration formats, evolving internal DSLs) do not, which bounds applicability beyond the breadth of our experiments.

Third, our inference predictor uses a frontier model. The supervision signal is independent of this choice: the "Ours + Gold" rows of Table 2 show the fine-tuned models exploit oracle grammars effectively without any frontier predictor in the loop, the model is robust to imperfect predictions where the frontier predictor is wrong (§5.4), and Figure 4 shows the trained model recovers baseline accuracy when grammars are absent. Cheaper predictors are an open research direction but the training method itself does not depend on this choice.

Potential risks. Like other code-generation research, improvements to DSL generation can in principle be misused (e.g., to lower the cost of writing malicious code in any of the studied DSLs). The benchmarks we use target benign domains (calendar dialogues, geography, blocks, digital and analog circuits), and we see no specific new risks introduced by this work.

References

Jacob Andreas, John Bufe, David Burkett, Charles Chen, Josh Clausman, Jean Crawford, Kate Crim, Jordan DeLoach, Leah Dorner, Jason Eisner, Hao Fang, Alan Guo, David Hall, Kristin Hayes, Kellie Hill, Diana Ho, Wendy Iwaszuk, Smriti Jha, Dan Klein, and 26 others. 2020. Task-oriented dialogue as dataflow synthesis. *Transactions of the Association for Computational Linguistics*, 8:556–571.

Anthropic. 2026. Claude Opus 4.7 System Card. https://www.anthropic.com/claude-opus-4-7-system-card.

Jitendra Bhandari, Vineet Bhat, Yuheng He, Hamed Rahmani, Siddharth Garg, and Ramesh Karri. 2024. Masala-CHAI: A Large-Scale SPICE Netlist Dataset for Analog Circuits by Harnessing AI. *arXiv preprint arXiv:2411.14299*.

Federico Cassano, John Gouwar, Francesca Lucchetti, Claire Schlesinger, Anders Freeman, Carolyn Jane Anderson, Molly Q Feldman, Michael Greenberg, Abhinav Jangda, and Arjun Guha. 2024. Knowledge Transfer from High-Resource to Low-Resource Programming Languages for Code LLMs. *Proceedings of the ACM on Programming Languages*, 8(OOPSLA2):677–708.

Pierre Chambon, Baptiste Roziere, Benoit Sagot, and Gabriel Synnaeve. 2025. BigO (Bench)–Can LLMs Generate Code with Controlled Time and Space Complexity? *arXiv preprint arXiv:2503.15242*.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, and 1 others. 2021. Evaluating Large Language Models Trained on Code. *arXiv preprint arXiv:2107.03374*.

Andrew Drozdov, Nathanael Schärli, Ekin Akyürek, Nathan Scales, Xinying Song, Xinyun Chen, Olivier Bousquet, and Denny Zhou. 2022. Compositional Semantic Parsing with Large Language Models. *arXiv preprint arXiv:2209.15003*.

Stefan Fuchs, Johannes Dimyadi, Michael Witbrock, and Robert Amor. 2024. Intermediate Representations to Improve the Semantic Parsing of Building Regulations. *Advanced Engineering Informatics*, 62:102735.

Linyuan Gong, Mostafa Elhoushi, and Alvin Cheung. 2024. AST-T5: Structure-Aware Pretraining for Code Generation and Understanding. *arXiv preprint arXiv:2401.03003*.

Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, and 1 others. 2024. The Llama 3 Herd of Models. *arXiv preprint arXiv:2407.21783*.

Yu Guo, Dong Jin, Shenghao Ye, Shuangwu Chen, Jian Yang, and Xiaobin Tan. 2025. SQLForge: Synthesizing reliable and diverse data to enhance text-to-SQL reasoning in LLMs. In *Findings of the Association for Computational Linguistics: ACL 2025*, pages 8441–8452, Vienna, Austria. Association for Computational Linguistics.

Jonathan Herzig, Peter Shaw, Ming-Wei Chang, Kelvin Guu, Panupong Pasupat, and Yuan Zhang. 2021. Unlocking Compositional Generalization in Pre-trained Models Using Intermediate Representations. *arXiv preprint arXiv:2104.07478*.

John E Hopcroft, Rajeev Motwani, and Jeffrey D Ullman. 2006. *Introduction to Automata Theory, Languages, and Computation*, 3rd edition. Pearson, Harlow, Essex, UK.

Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2022. LoRA: Low-rank adaptation of large language models. In *International Conference on Learning Representations*.

Naman Jain, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. 2025. LiveCodeBench: Holistic and Contamination Free Evaluation of Large Language Models for Code. In *International Conference on Learning Representations*, volume 2025, pages 58791–58831.

Sathvik Joel, Jie Wu, and Fatemeh Fard. 2024. A Survey on LLM-based Code Generation for Low-Resource and Domain-Specific Programming Languages. *ACM Transactions on Software Engineering and Methodology*.

Aishwarya Kamath, Johan Ferret, Shreya Pathak, Nino Vieillard, Ramona Merhej, Sarah Perrin, Tatiana Matejovicova, Alexandre Ramé, Morgane Rivière, and 1 others. 2025. Gemma 3 Technical Report. *arXiv preprint arXiv:2503.19786*.

Mayank Kothyari, Sunita Sarawagi, Soumen Chakrabarti, Gaurav Arora, and Srujana Merugu. 2025. Diverse in-context example selection after decomposing programs and aligned utterances improves semantic parsing. In *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pages 5611–5629, Albuquerque, New Mexico. Association for Computational Linguistics.

Fangyu Lei, Jixuan Chen, Yuxiao Ye, Ruisheng Cao, Dongchan Shin, Hongjin Su, Zhaoqing Suo, Hongcheng Gao, Wenjing Hu, Pengcheng Yin, and 1 others. 2025. Spider 2.0: Evaluating Language Models on Real-World Enterprise Text-to-SQL Workflows. In *International Conference on Learning Representations*, volume 2025, pages 28691–28735.

Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, and 1 others. 2020. Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks. *Advances in neural information processing systems*, 33:9459–9474.

Haoyang Li, Shang Wu, Xiaokang Zhang, Xinmei Huang, Jing Zhang, Fuxin Jiang, Shuai Wang, Tieying Zhang, Jianjun Chen, Rui Shi, and 1 others. 2025a. OmniSQL: Synthesizing High-Quality Text-to-SQL Data at Scale. *arXiv preprint arXiv:2503.02240*.

Jiaqian Li, Qisheng Hu, Jing Li, and Wenya Wang. 2025b. STARE at the structure: Steering ICL exemplar selection with structural alignment. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pages 14765–14782, Suzhou, China. Association for Computational Linguistics.

Jiachang Liu, Dinghan Shen, Yizhe Zhang, Bill Dolan, Lawrence Carin, and Weizhu Chen. 2022. What makes good in-context examples for GPT-3? In *Proceedings of Deep Learning Inside Out (DeeLIO 2022): The 3rd Workshop on Knowledge Extraction and Integration for Deep Learning Architectures*, pages 100–114, Dublin, Ireland and Online. Association for Computational Linguistics.

Mingjie Liu, Nathaniel Pinckney, Brucek Khailany, and Haoxing Ren. 2023. VerilogEval: Evaluating Large Language Models for Verilog Code Generation. In *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, pages 1–8. IEEE.

Mingjie Liu, Yun-Da Tsai, Wenfei Zhou, and Haoxing Ren. 2025. CraftRTL: High-quality Synthetic Data Generation for Verilog Code Models with Correct-by-Construction Non-Textual Representations and Targeted Code Repair. In *International Conference on Learning Representations*, volume 2025, pages 90377–90422.

OpenAI. 2026. GPT-5.5 System Card. https://openai.com/index/gpt-5-5-system-card/.

Nathaniel Pinckney, Christopher Batten, Mingjie Liu, Haoxing Ren, and Brucek Khailany. 2025. Revisiting VerilogEval: A Year of Improvements in Large-Language Models for Hardware Code Generation. *ACM Transactions on Design Automation of Electronic Systems*, 30(6):1–20.

Shanghaoran Quan, Jiaxi Yang, Bowen Yu, Bo Zheng, Dayiheng Liu, An Yang, Xuancheng Ren, Bofei Gao, Yibo Miao, Yunlong Feng, and 1 others. 2025. CodeElo: Benchmarking Competition-level Code Generation of LLMs with Human-comparable Elo Ratings. *arXiv preprint arXiv:2501.01257*.

Qwen Team. 2026. Qwen3.5: Towards native multimodal agents. https://qwen.ai/blog?id=qwen3.5.

Bailin Wang, Zi Wang, Xuezhi Wang, Yuan Cao, Rif A Saurous, and Yoon Kim. 2023. Grammar Prompting for Domain-Specific Language Generation with Large Language Models. *Advances in Neural Information Processing Systems*, 36:65030–65055.

Yushi Wang, Jonathan Berant, and Percy Liang. 2015. Building a semantic parser overnight. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 1332–1342, Beijing, China. Association for Computational Linguistics.

Shitao Xiao, Zheng Liu, Peitian Zhang, Niklas Muennighoff, Defu Lian, and Jian-Yun Nie. 2024. C-Pack: Packed Resources for General Chinese Embeddings. In *Proceedings of the 47th international ACM SIGIR*

conference on research and development in information retrieval, pages 641–649.

Xu Xu, Xin Li, Xingwei Qu, Jie Fu, and Binhang Yuan. 2025. Local Success Does Not Compose: Benchmarking Large Language Models for Compositional Formal Verification. *arXiv preprint arXiv:2509.23061*.

An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, and 1 others. 2024. Qwen2.5 Technical Report. *arXiv preprint arXiv:2412.15115*.

Pengcheng Yin, Hao Fang, Graham Neubig, Adam Pauls, Emmanouil Antonios Platanios, Yu Su, Sam Thomson, and Jacob Andreas. 2021. Compositional generalization for neural semantic parsing via span-level supervised attention. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 2810–2823, Online. Association for Computational Linguistics.

John M Zelle and Raymond J Mooney. 1996. Learning to Parse Database Queries Using Inductive Logic Programming. In *Proceedings of the National Conference on Artificial Intelligence*, pages 1050–1055.

Yongan Zhang, Zhongzhi Yu, Yonggan Fu, Cheng Wan, and Yingyan Celine Lin. 2024. MG-Verilog: Multi-Grained Dataset Towards Enhanced LLM-Assisted Verilog Generation. In *2024 IEEE LLM Aided Design Workshop (LAD)*, pages 1–5. IEEE.

Jiayu Zhao and Song Chen. 2025. VeriGRAG: Enhancing LLM-Based Verilog Code Generation with Structure-Aware Soft Prompts. *arXiv preprint arXiv:2510.15914*.

Terry Yue Zhuo, Minh Chien Vu, Jenny Chim, Han Hu, Wenhao Yu, Ratnadira Widyasari, Imam Nur Bani Yusuf, Haolan Zhan, Junda He, Indraneil Paul, and 1 others. 2025. BigCodeBench: Benchmarking Code Generation with Diverse Function Calls and Complex Instructions. In *International Conference on Learning Representations*, volume 2025, pages 66602–66656.

A Results for Additional Models

		Semantic Parsing					Hardware Verification				
		SMCalFlow	GeoQuery		Overnight		Verilog			SPICE	
Model	Method	Prog.	Prog.	Exec.	Prog.	Exec.	pass@1	pass@3	pass@5	GED	F1
Qwen-2.5 7B	Baseline	55.1	83.9	91.4	47.1	63.7	28.6	40.3	44.9	70.5	80.5
	Ours	84.9	87.5	96.4	47.4	66.7	40.5	43.4	44.9	71.9	81.8
	Ours + Gold	90.6	96.8	98.2	97.2	99.2	51.2	57.7	59.6	96.7	93.8
Llama-3.1 8B	Baseline	47.2	45.7	50.4	47.1	62.4	27.4	37.4	41.7	69.7	81.3
	Ours	76.7	68.6	75.7	47.6	64.4	40.9	46.5	48.7	72.3	81.8
	Ours + Gold	80.8	75.4	75.7	97.5	97.0	50.4	57.4	60.3	96.8	93.8
Gemma-3 12B	Baseline	50.1	76.4	86.8	46.6	63.9	25.9	36.4	40.4	64.3	79.0
	Ours	75.6	87.5	95.4	47.1	66.4	41.3	44.9	46.2	71.9	81.7
	Ours + Gold	86.4	96.8	97.5	97.0	99.0	51.3	59.2	62.8	96.7	94.0

Table 6: Results for three additional open-weight backbones across the five paper benchmarks. All values are percentages.

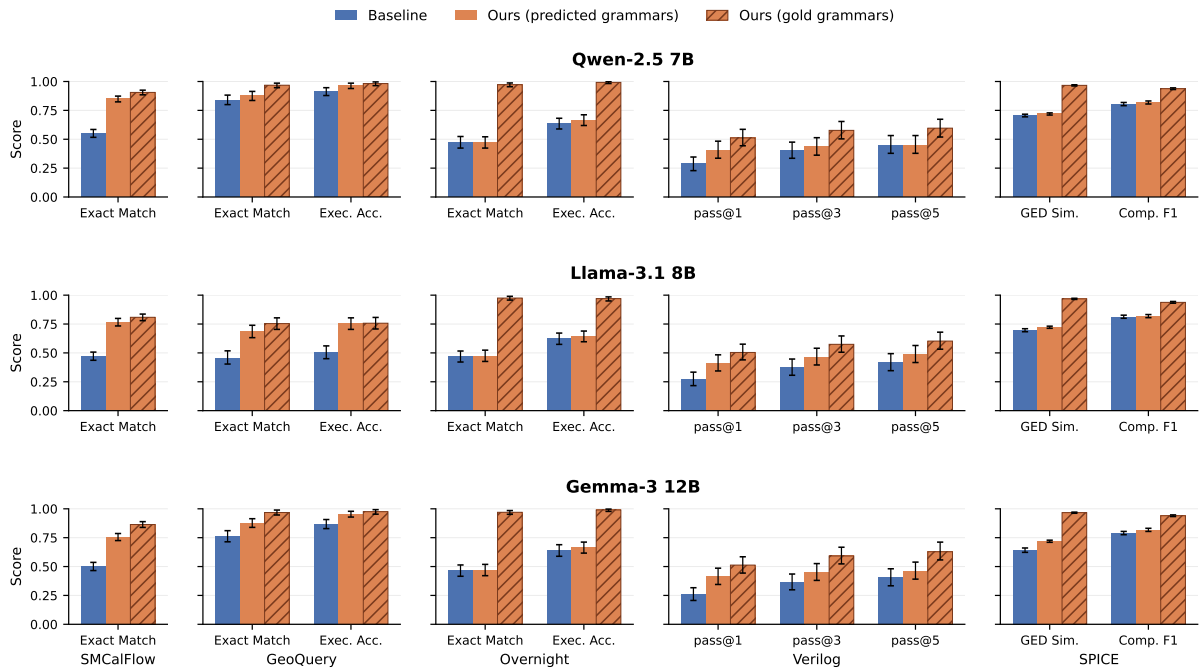


Figure 6: Per-domain results for the three additional open-weight backbones. Each row corresponds to one model (Qwen-2.5 7B, Llama-3.1 8B, Gemma-3 12B). Bars show Baseline (blue), Ours with RAG+CoT-predicted grammars (orange), and Ours + Gold with oracle grammars (orange, hatched). Error bars are 95% percentile bootstrap CIs (1,000 resamples).

We present results for Qwen-2.5 7B (Yang et al., 2024), Llama-3.1 8B (Grattafiori et al., 2024), and Gemma-3 12B (Kamath et al., 2025). All three follow the same overall pattern as Table 3: large gains on SMCalFlow program accuracy (25–30pp), HDL pass@1 and GED gains in the same 9–14pp / 7–13pp ranges as the main models, and oracle grammars near the ceiling on GeoQuery, Overnight, and SPICE. Llama-3.1 8B is the exception: on Overnight, predicted grammars yield no improvement over the baseline (47.6 vs. 47.1 program accuracy) while oracle grammars reach 97.5%, indicating the RAG+CoT predictor for this backbone is leaving most of the structural information on the table.

B Training on Predicted Grammars

Our main-text training pipeline fine-tunes on (query, gold minimal grammar, program) triples while inference supplies a RAG-predicted grammar. A natural question is whether matching the training distribution to inference – training on the

Training grammar	Prog.	Exec.
None (Baseline)	75.0	84.3
RAG-predicted	85.0	96.1
Gold (Ours)	87.5	96.1

Table 7: Effect of replacing gold minimal grammars with RAG-predicted grammars at training time. Qwen-3.5 9B on GeoQuery; all rows use RAG-predicted grammars at inference. Baseline and Gold rows are reproduced from Table 3.

same noisy predicted grammars the model will see at test time – would make it more robust to that noise.

We tested this on GeoQuery with Qwen-3.5 9B by replacing each training example's gold minimal grammar with the RAG+CoT prediction (using the same predictor as inference) and keeping all other settings from Appendix C. Table 7 reports program and execution accuracy with RAG-predicted grammars at inference.

Training on predicted grammars improves over the no-grammar baseline ($75.0 \rightarrow 85.0$ program accuracy) but does not match training on gold (87.5). Execution accuracy is tied at 96.1. We did not extend this experiment beyond GeoQuery: the gold-trained model already absorbs noisy predicted grammars effectively, which we attribute to the grammar-dropout schedule of Section 3.1 teaching the model to treat the supplied grammar as a hint rather than a strict specification.

C Training Parameters

We fine-tuned one LoRA adapter (Hu et al., 2022) for each reported model, dataset, and training condition. The reported open-weight backbones were `Qwen/Qwen3.5-4B`, `Qwen/Qwen3.5-9B`, and `google/gemma-3-27b-it`. Main-result runs used one epoch. Table 8 gives the shared training parameters. The two ablations in Table 5 used the same settings except that the token-count baseline trained for two epochs, and the joint generation ablation changed the target from program-only generation to grammar-then-program generation.

SMCalFlow, GeoQuery, and Overnight used a maximum sequence length of 1024 tokens. Verilog and SPICE used 2048 tokens because examples in those domains contain longer programs and grammars. Qwen-3.5 runs used the chat template with thinking disabled. Baseline adapters were trained without grammars. Our adapters were trained with the mixed-ratio setting `0.1`: 90% of training examples included the minimal grammar and 10% omitted it, sampled with seed 42. The same ratio was used for the validation set. All adapters were trained on a single NVIDIA A100 80GB GPU.

Parameter	Value
LoRA rank / alpha / dropout	64 / 128 / 0.05
LoRA target modules	`all-linear`
Optimizer	AdamW fused
Adam betas / epsilon	0.9, 0.999 / 10^{-8}
Weight decay / max grad norm	0 / 1.0
Learning rate	10^{-4}
LR schedule / warmup	cosine / 0.05 warmup ratio
Batching	4 examples per device, 4 gradient-accumulation steps
Precision	bfloat16
Gradient checkpointing	enabled
Attention implementation	`flash_attention_2`

Table 8: Shared training parameters for fine-tuned adapters.

Inference and evaluation. For SMCalFlow, GeoQuery, and Overnight, open-weight model evaluation used greedy decoding with batch size 32 and at most 512 new tokens. SPICE used greedy decoding with batch size 32 and at most 2048 new tokens. Verilog used 5 samples per problem with temperature 0.8, top-$p = 0.95$, at most 1024 new tokens, and reports pass@1, pass@3, and pass@5.

The RAG grammar files used for the open-weight "Ours" rows were generated with BGE-large-en-v1.5 embeddings, cosine nearest neighbors, $k = 16$, CoT-style retrieved examples, temperature 0, and a 4096-token output limit. The frontier ICL experiments in Table 4 used $k = 16$. Programs-only and grammar-prompting ICL used the k=16 demonstration set released by Wang et al. (2023); RAG+CoT ICL used the 16 nearest neighbors under the same BGE-large-en-v1.5 retriever. Frontier model decoding used temperature 0 and a 4096-token output limit for the SP domains.

D Prompt Templates

D.1 In-Context Learning Prompts

The four in-context learning conditions reported in Table 4 (zero-shot, programs only, grammar prompting, and RAG+CoT) share a chat-style message format: a system message, alternating demo `user`/`assistant` turns, and a final test `user` turn. They differ in (i) the system prompt, (ii) the contents of the demo turns, and (iii) how demos are selected (first-k training examples vs. k

nearest neighbours under BGE-large-en-v1.5, with k=16). The optional `Module header` line appears only for MG-Verilog, where the canonical module header from the benchmark prompt is appended.

Zero-shot

System prompt:

```
You are a semantic parser for {benchmark}. Given a
user query, produce the {benchmark} program that
parses the query. Output only the program, nothing
else.
```

The placeholder `{benchmark}` is instantiated as one of `SMCalFlow`, `GeoQuery FunQL`, `Overnight Lambda DCS`, `Verilog`, or `SPICE`.

Test user message:

```
Query: {x}

[Module header:
{module_header}]
```

Programs only

System prompt:

```
You are a semantic parser. Given a user query, produce
the program that parses the query. Output only the
program, nothing else.
```

Demo user message:

```
Query: {x_i}

[Module header:
{module_header_i}]
```

Demo assistant message:

```
{y_i}
```

Test user message:

```
Query: {x}

[Module header:
{module_header}]
```

Grammar prompting

System prompt:

```
You are a semantic parser. The in-context examples
show, for each query, the minimal grammar wrapped in
<grammar>...</grammar> tags followed by a "Program:"
label and the program.

Given a new query, first emit the minimal grammar in
<grammar>...</grammar> tags, then emit a blank line,
then 'Program:', then the program. Output nothing else.
```

Demo user message:

```
Query: {x_i}

[Module header:
{module_header_i}]
```

Demo assistant message:

```
<grammar>
{G_i}
</grammar>

Program:
{y_i}
```

Test user message:

```
Query: {x}

[Module header:
{module_header}]
```

RAG + CoT (ours)

System prompt:

```
You are a semantic parser. The in-context examples
show, for each query, step-by-step reasoning about
which grammar rules are needed, the minimal grammar
wrapped in <grammar>...</grammar> tags, and then a
'Program:' label followed by the program.

Given a new query, follow the same pattern: produce
the reasoning, then the grammar in
<grammar>...</grammar> tags, then a blank line, then
'Program:', then the program. Output nothing else.
```

Demo user message:

```
Query: {x_i}

[Module header:
{module_header_i}]
```

Demo assistant message:

```
{reasoning_i}

<grammar>
{G_i}
</grammar>

Program:
{y_i}
```

Test user message:

```
Query: {x}

[Module header:
{module_header}]
```

The block `{reasoning_i}` is the pre-computed CoT trace produced for each training example using the prompt in Appendix D.2 below. Demos are the 16 nearest training queries under BGE-large-en-v1.5 cosine similarity.

D.2 Chain-of-Thought Generation Prompt

Each training example is paired with a CoT trace pre-generated by GPT-5.5 using the prompt below; the resulting reasoning text is concatenated with a `<grammar>...</grammar>` block and used as the reasoning block in the RAG+CoT mode above.

CoT generation

System prompt:

```
You are an expert grammar analyst. You are given a
context-free grammar for {domain_description}, along
with a natural-language query, its gold program, and
the minimal grammar needed to parse that program.

Your task: explain step-by-step WHY these specific
grammar rules are needed for this query/program.
Reason about:
- What the query is asking for
- What top-level construct the program uses
- Which rules are needed to build each part of the
program
- Why certain alternatives were chosen over others in
the full grammar

Output ONLY the reasoning. Do NOT output the grammar
itself.

Here is the full reference grammar:

{full_grammar}
```

User message:

```
Query: {x}

[Module header:
```

```
{module_header}]

Gold program:
{y}

Minimal grammar:
{G}
```

Domain descriptions substituted for `{domain_description}`:
SMCalFlow: `a calendar-domain semantic parsing language.`
GeoQuery: `a FunQL geographic query language for querying US geography facts (states, cities, rivers, mountains, populations, areas).`
Overnight: `a Lambda DCS language for querying block objects with spatial relationships (left, right, above, below) and properties (shape, color, length, width, height).`
MG-Verilog: `a Verilog hardware description language.`
SPICE: `a SPICE circuit simulation language.`

E Example Chain-of-Thought Traces

This section presents one example RAG-CoT trace per domain. Each box shows the natural-language input x followed by the model's chain-of-thought reasoning and the generated grammar produced for that query.

SMCalFlow

Query: Make a appointment for tomorrow
Chain-of-thought trace:

```
The query is a creation request: "Make an appointment
for tomorrow."

This is identical to all the examples. The user wants
to create a new event for tomorrow.

1. **call**: We need `(Yield event)` to return the
result of creating an event.
2. **event**: We need `(CreateCommitEventWrapper
(CreatePreflightEventWrapper event_constraint))`
because "make" means create.
3. **event_constraint**: We need `(Event.start_?
datetime_constraint)` since the only specification is
when -- "for tomorrow."
4. **datetime_constraint**: We need `(DateTime.date_?
datetime_constraint)` to project onto the date part,
and `(?= datetime)` for equality.
5. **datetime**: We need `date` since tomorrow is a
date expression.
6. **date**: We need `day` since tomorrow is a
day-level primitive.
7. **day**: We need `(Tomorrow)`.

<grammar>
call ::= "(Yield" event ")"
event ::= "(CreateCommitEventWrapper
(CreatePreflightEventWrapper" event_constraint "))"
event_constraint ::= "(Event.start_?"
datetime_constraint ")"
datetime_constraint ::= "(DateTime.date_?"
datetime_constraint ")" | "(?=" datetime ")"
datetime ::= date
date ::= day
day ::= "(Tomorrow)"
</grammar>
```

GeoQuery

Query: what is the area of wisconsin ?
Chain-of-thought trace:

```
The query asks for a numeric property: the area of the
state Wisconsin. So the program must return a number,
not a state, city, river, place, or country.

At the top level, every complete FunQL query is
wrapped in `answer(...)`. Since the result is numeric,
the inside of `answer(...)` must be an `answer_type`
that resolves to `num`.

To express "the area of wisconsin," the program uses
the numeric constructor for area applied to a state.
Among the `num` alternatives, the needed one is
`area_1(state)`, because Wisconsin is a state and the
question asks for its area. Other numeric options like
population, density, size, count, or area of a
city/country do not match.

Then we need a way to denote the specific state
Wisconsin. The program uses `stateid('wisconsin')`, so
the `state` category needs the lexical rule that
directly produces that state identifier. This is the
correct choice because the query names a specific
state directly.

<grammar>
answer_type ::= num
num ::= "area_1(" state ")"
state ::= "stateid('wisconsin')"
</grammar>
```

Overnight-Blocks

Query: find a pyramidshaped block
Chain-of-thought trace:

```
The query asks for a set of blocks: specifically,
blocks whose shape is pyramid. So the program must
denote a collection of entities, not a number or
property value.

The top-level construct is a list-valued answer,
because the query says "find a ... block," meaning we
want the resulting block or set of blocks. That is why
the outermost rule is the one wrapping a list
expression as a list value.

Inside that, the main operation is filtering. The
query imposes a shape constraint on blocks, so we need
the filter rule that takes a source list, the property
shape, equality, and a target value.

To apply that filter, we need a base set to filter.
The program uses the set of all blocks, obtained by
taking the type extension of the singleton type entity
`en.block`. So the rule for obtaining the block set is
necessary.

The value compared against the shape property is
`en.shape.pyramid`, because the query specifies
"pyramidshaped." So that entity constant is required.

No other rules from the full grammar are needed
because the query does not involve spatial relations,
numeric properties, aggregation, counting,
superlatives, comparatives, other properties, or other
entity constants.

<grammar>
list_value ::= "(listValue" list_value ")" | "(filter"
list_value "shape =" list_value ")" | "(getProperty
(singleton en.block) !type)" | "en.shape.pyramid"
</grammar>
```

MG-Verilog

Query: Create a module that implements a NOR gate.
Chain-of-thought trace:

```
I need to figure out what a NOR gate module body looks
like based on the query and similar examples.

The query asks for a NOR gate implementation. Looking
at the similar examples, NOR gates are typically
implemented with a continuous `assign` statement using
either:
- `~(A | B)` (bitwise NOT of OR) - Examples 1, 3
- `!(A | B)` (logical NOT of OR) - Example 2

The most common pattern in the examples for NOR gates
uses `assign` with `~` and `|`. Since the query is
very simple ("Create a module that implements a NOR
gate"), I expect a simple 2-input NOR with inputs like
`a`, `b` and output `out`.
```

```
Looking at the simplest NOR examples (Examples 1, 3),
the typical program would be:
```
assign out = ~(a | b);
```

This requires:
1. **continuous_assign** - for the `assign` statement
2. **list_of_assignments** - containing one assignment
3. **assignment** - `out = ~(a | b)` where LHS is
identifier, RHS is unary expression
4. **identifier_ref** / **hier_identifier** - for
signal names `out`, `a`, `b`
5. **unary** - for `~` operator applied to a primary
6. **primary** - for the parenthesized `(a | b)`
7. **bitwise_or** - for `a | b`

The expression structure is:
- outer: `~(...)` -> unary negation of a primary
- primary: `(a | b)` -> parenthesized bitwise OR
- bitwise_or: `a | b` -> two identifier refs joined by
`|`

<grammar>
continuous_assign ::= "assign" list_of_assignments ";"
list_of_assignments ::= assignment
assignment ::= identifier_ref "=" unary
identifier_ref ::= hier_identifier
hier_identifier ::= "out" | "a" | "b"
unary ::= "~" primary
primary ::= "(" bitwise_or ")"
bitwise_or ::= identifier_ref "|" identifier_ref
</grammar>
```

SPICE

Query: The circuit is a BJT differential amplifier with two NPN transistors (Q1 and Q2) sharing a common emitter resistor (REE) and each transistor having a collector resistor (RC). The input is differential with vIN applied between the two bases, and the output is taken single-ended from the collector of Q2. The differential half-circuit model is shown, where REE is replaced with 2rpi and the transconductance gm, transistor base resistance rpi, and collector resistance RC are considered for small-signal analysis.
Chain-of-thought trace:

```
I need to analyze the query and determine what SPICE
netlist would represent this circuit, then derive the
minimal grammar.

The query describes a BJT differential amplifier with:
- Two NPN transistors Q1 and Q2
- A shared emitter resistor REE
- Collector resistors RC
- Differential input vIN
- Small-signal model with rpi, gm, RC, and 2REE

Looking at the circuit description, this appears to be
a small-signal equivalent model representation. The
program would likely contain:
- Resistors for rpi, RC, and 2REE
- A voltage-controlled current source for gm
- A voltage source for the input

Based on the pattern of similar examples (especially
Examples 10, 16, 40 which are small-signal
differential pair models), the program would use
resistors, VCCS elements, and voltage sources.

The expected program would be:
```
spice
Vin 1 0 DC vid/2
Rpi 1 2 rpi
Gm 3 0 1 2 gm
RC 4 3 RC
R2REE 2 0 2REE
Vout 3 0 DC 0
.end
```

Let me reason about the specific components:

- `Vin 1 0 DC vid/2` - voltage source with DC spec for
the half-circuit input
- `Rpi 1 2 rpi` - resistor for base-emitter resistance
- `Gm 3 0 1 2 gm` - VCCS for transconductance (4-node
form with value)
- `RC 4 3 RC` - collector resistor
- `R2REE 2 0 2REE` - emitter resistor (2xREE for
half-circuit)
- `Vout 3 0 DC 0` - output sensing voltage source

The rules needed:
1. **voltage_source** - for Vin and Vout with DC specs
2. **dc_spec** - for "DC vid/2" and "DC 0"
3. **resistor** - for Rpi, RC, R2REE
4. **vccs** - for Gm (4-node plus value form)

<grammar>
voltage_source ::= "Vin 1 0" dc_spec | "Vout 3 0"
dc_spec
dc_spec ::= "DC vid/2" | "DC 0"
resistor ::= "Rpi 1 2 rpi" | "RC 4 3 RC" | "R2REE 2 0
2REE"
vccs ::= "Gm 3 0 1 2 gm"
</grammar>
```

F Worked Example of a Minimal Grammar

Formally, a context-free grammar is a tuple $G = (N, \Sigma, P, S)$ (Hopcroft et al., 2006), where:

- N is a finite set of non-terminal symbols;
- Σ is a finite set of terminal symbols (the alphabet of the language);
- P is a finite set of production rules of the form $A \to \alpha$, with $A \in N$ and $\alpha \in (N \cup \Sigma)^*$;
- $S \in N$ is the start symbol.

Context-free grammars are a flexible formalism that captures a range of domains including code generation, formal reasoning, and symbolic question answering.

To make the concept of a minimal grammar concrete, consider the following toy grammar:

$$G = (\{S, A, B\},\ \{a, b\},\ P,\ S)$$

with production rules

$$P = \begin{cases} S \to AB \\ A \to a \mid \epsilon \\ B \to b \mid \epsilon. \end{cases}$$

Suppose we are given the program $y = ab$. The minimal grammar $G(y)$ is the smallest subset of G that still admits y as a valid derivation. The ϵ-productions are not required to derive y, so they are excluded:

$$G(y) = (\{S, A, B\},\ \{a, b\},\ \{S \to AB,\ A \to a,\ B \to b\},\ S).$$

No rule in $G(y)$ is extraneous: removing any one of them would make y ungrammatical. This illustrates how minimal grammars capture just the structural components needed to generate a given program.

G Example Query, Minimal Grammar, and Program for Each Domain

This section presents example queries, their corresponding minimal grammars, and target programs for each domain. Line breaks are added for readability. Each box shows the natural-language input x, the example-specific minimal grammar $G(y)$, and the target program y.

SMCalFlow

Query: Schedule a lunch with Tom at noon today.
Minimal grammar:

```
call ::= "(Yield" event ")"
event ::= "(CreateCommitEventWrapper
(CreatePreflightEventWrapper"
          event_constraint "))"
event_constraint ::= "(Event.subject_? (?=" string "))"
                   | "(Event.attendees_?"
attendee_constraint ")"
                   | "(Event.start_?"
datetime_constraint ")"
                   | "(&" event_constraint
event_constraint ")"
string ::= ""lunch"" | ""Tom""
datetime_constraint ::= "(?=" datetime ")"
datetime ::= "(DateAtTimeWithDefaults" date time ")"
date ::= day
day ::= "(Today)"
time ::= "(Noon)"
attendee_constraint ::= "(AttendeeListHasRecipient"
person ")"
person ::= recipient
recipient ::= "(Execute (refer (extensionConstraint"
recipient ")))"
            | "(RecipientWithNameLike (^(Recipient)
EmptyStructConstraint)
                (PersonName.apply" string "))"
```

Program:

```
(Yield
 (CreateCommitEventWrapper
  (CreatePreflightEventWrapper
   (&
    (& (Event.subject_? (?= "lunch"))
       (Event.start_?
        (?= (DateAtTimeWithDefaults (Today) (Noon)))))
    (Event.attendees_?
     (AttendeeListHasRecipient
      (Execute
       (refer
        (extensionConstraint
         (RecipientWithNameLike
          (^(Recipient) EmptyStructConstraint)
          (PersonName.apply "Tom")))))))))))
```

GeoQuery

Query: Give me the states that border Utah.
Minimal grammar:

```
answer_type ::= state
state ::= "state(" state ")"
        | "next_to_2(" state ")"
        | "stateid('utah')"
```

Program:

```
answer(state(next_to_2(stateid('utah'))))
```

Overnight-Blocks

Query: Find me a pyramid block that is 3 inches tall.
Minimal grammar:

```
list_value ::= "(listValue" list_value ")"
             | "(filter" list_value "height ="
list_value ")"
             | "(filter" list_value "shape ="
list_value ")"
             | "(getProperty (singleton en.block)
!type)"
             | "en.shape.pyramid"
             | "3 en.inch"
```

Program:

```
(listValue
 (filter
  (filter (getProperty (singleton en.block) !type)
          shape = en.shape.pyramid)
  height = 3 en.inch))
```

MG-Verilog

Query: An XOR gate over inputs `A` and `B` producing output `O`, implemented as $\overline{A}B + A\overline{B}$.
Minimal grammar:

```
port_decl_stmt ::= port_dir net_or_reg
list_of_port_variables ";"
net_or_reg ::= "wire"
list_of_port_variables ::= port_variable ","
port_variable
                         | port_variable
port_variable ::= "A" | "B" | "O"
continuous_assign ::= "assign" assignment ";"
assignment ::= identifier_ref "=" addition
identifier_ref ::= hier_identifier
hier_identifier ::= "O" | "A" | "B"
addition ::= multiplication "+" multiplication
multiplication ::= primary "*" identifier_ref
                 | identifier_ref "*" primary
primary ::= "(" unary ")"
unary ::= "~" identifier_ref
```

Program:

```
module XOR(O, A, B);
  input  wire A, B;
  output wire O;
  assign O = (~A)*B + A*(~B);
endmodule
```

SPICE

Query: A network with two voltage sources $V_{S1} = 20\text{V}$ and $V_{S2} = 15\text{V}$, with $R_1 = R_2 = 1\,\text{k}\Omega$ tying their nodes together through a common node, and $R_3 = 2.2\,\text{k}\Omega$ feeding an open terminal.
Minimal grammar:

```
voltage_source ::= "VS1 5 0" dc_spec
                 | "VS2 3 0" dc_spec
dc_spec ::= "DC 20V" | "DC 15V"
resistor ::= "R1 5 2 1k"
           | "R2 2 3 1k"
           | "R3 2 4 2.2k"
```

Program:

```
VS1 5 0 DC 20V
R1  5 2 1k
R2  2 3 1k
R3  2 4 2.2k
VS2 3 0 DC 15V
.end
```

H Artifact Licenses

All datasets are used under their respective licenses: SMCalFlow and Overnight under CC BY-SA 4.0, GeoQuery under GPL 2.0, MG-Verilog under CC BY-NC-SA 4.0, VerilogEval under MIT, and Masala-CHAI under its open-source release. Open-weight backbones (Qwen, Gemma, Llama families) and the BGE-large-en-v1.5 retriever are used under their respective model and code licenses. Frontier models (GPT-5.5, Claude Opus 4.7) are accessed via provider APIs under the respective terms of service.

Intended use All datasets and models used in this work are research artifacts, and we use them for research purposes consistent with their stated intended use. Artifacts we release accompanying this paper are intended for research use only.