CONDITIONAL DEONTICS OVER TERMINALS: A CROSS-SUBTREE REFERENCE MECHANISM FOR EFFICIENT YET EXPRESSIVE CONSTRAINED DECODING

Anonymous authors
Paper under double-blind review

ABSTRACT

As recent work has improved expressiveness in constrained decoding, a common component has begun to emerge: user-defined functions that read and modify a constrainer's internal state. However, permissive access to a constrainer's internal state makes efficiency optimization difficult. In this work we offer a sharpened contract for such user-defined functions, decomposing them into simpler functions that require minimal access to the constrainer's internal state, which we call *deontic functions*, and providing mechanisms for aggregating and anchoring deontic functions within grammar structures. Using our formalism, we implement the unit-propagation algorithm within a chain of thought, enabling a smaller model to achieve a high solve rate and reduced token consumption on a SAT-style logic puzzle called "Knights and Knaves". Utilizing a GPU-based backend for our formalism, we replicate constrainers such as type checking and e-graph enforcement found in prior works and achieve substantial efficiency gain.

1 INTRODUCTION

Constrained decoders are algorithms that ensure that the output generated by a large language model (LLM) matches a set of formal constraints. For example, constrained decoders can ensure that LLMs generate syntactically valid JSON, Python, or SQL Scholak et al. (2021); Dong et al. (2025); Park et al. (2025); Li et al. (2026). To be practically useful, a constrained decoder must run at the speed of the LLM, and must be able to branch or roll back to accommodate algorithms ranging from beam search to Monte Carlo simulation. Most current constrained decoders achieve these goals by forcing users to submit constraints in restrictive formalisms such as regular expressions or context-free grammars, and delegate efficiency optimization and execution to the constrained decoder.

Recent work has aimed to expand the expressive power of constrained decoders beyond context free grammars Scholak et al. (2021); Ugare et al. (2025); Nagy et al. (2026); Zhang et al. (2026); Li et al. (2025); Albinhassan et al. (2026). Such work frequently makes use of user-defined functions that have well-defined signatures but provide little in the way of restrictions on function internals. For example, decode-time grammar Zhang et al. (2026) requires a user-defined mapping from a program's symbol table to the set of permissible candidates, without restricting the computation performed inside. ChopChop Nagy et al. (2026) provides a domain specific language for implementing mapping functions from program space to program space, but it is permissive enough to allow a potentially untrusted user to submit non-terminating programs whose memory usage explodes.

The complexity of the user-defined function's internals have implications for the efficiency of the constrained decoder. One can offer a set of trusted and optimized mapping functions that are closely coupled to the constrainer, but the programmability of such a system is limited. On the other hand, if programmability is unrestricted (e.g., the user-defined function can arbitrarily modify constrainer-internal data structures) it will incur a heavy communication overhead between the constrainer and the user-defined functions, which is undesirable given that existing work already goes to some length to limit CPU–GPU communication within the constrainer.

To refine this class of user-defined functions with both efficiency and expressiveness in mind, we propose a semantics called *Conditional Deontics over Terminals* (CDoT). We define a class of primitive functions for expressing constraints, which we call *deontic functions*, that can be combined

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

to achieve complex behaviors via efficient aggregation and anchoring mechanisms. The GPU-accelerated aggregation mechanism reduces the number of deontic function calls, and the anchoring mechanism can be directly integrated into a chart parser that benefits from GPU acceleration.

Our technical contributions are as follows.

- We identify a class of computation shared among recent constrained decoders with high expressive power, and capture this class in a semantics named CDoT.
- We show that CDoT is expressive enough to define a variety of useful constraints, such as SAT-style chain-of-thought logical deduction and SQL variable scoping.
- We show that CDoT is efficient and amenable to GPU acceleration, achieving efficiency gains over prior constrained decoders on several different constrained decoding tasks.

2 Related Work

One line of related work has focused on improving the efficiency of constrained decoders to keep up with newer LLM serving hardware and software. XGrammar Dong et al. (2025) and LLGuidance Moskal et al. (2025) are classical CPU-based systems with fast initialization and then-sufficiently fast token-to-token latency. Newer systems such as PSC Li et al. (2026) perform extensive analysis of the constraints and require long initialization times, ranging from minutes to hours, but compress token-to-token latency down to microseconds. Alternatively, STATIC Su et al. (2026) turns to GPU/TPU acceleration. These constrained decoders are limited in their expressiveness (typically to context free grammars), while our approach, CDoT, provides improved expressive power while preserving applicability of GPU acceleration techniques.

Another line of related work extends expressive power beyond context-free grammars. Earlier work such as PICARD Scholak et al. (2021) and IterGen Ugare et al. (2025) requires the practitioner to speculate manually about ancestor structures from incomplete parse trees using user code. ChopChop Nagy et al. (2026) and decode-time grammar Zhang et al. (2026) automate the analysis of incomplete parse trees, but still rely on permissively defined, user-programmed components. Our approach, CDoT, still extends expressive power beyond context-free grammars, but in a managed fashion that allows for automated efficiency optimization.

We also draw inspiration from conjunctive grammars (Okhotin, 2001; Mrykhin & Okhotin, 2023; Barash & Okhotin, 2014), which shows higher expressiveness can be achieved via slightly modified CYK merge operation. We appropriately uses terminology and symbology from deontic logic in our formalism (von Wright, 1951). We improve upon deontic grammars on practicality, as shown in section A,

3 Conditional Deontics over Terminals

We motivate our definition of deontic functions by first reviewing the user-provided mapping functions of prior work and then defining an efficiency-friendly alternative.

ChopChop relies on a user-supplied pruner, a corecursive function that receives the set of abstract syntax trees realizable from the currently generated prefix and returns a pruned set. The pruner thus has the signature $(\texttt{program_space}\colon Set[AST]) \to (\texttt{program_space}\colon Set[AST])$. Such functions may fail to terminate and may allocate ever-growing amounts of memory, and automated analysis of them is difficult. Decode-time grammar requires user-supplied functions that manipulate an environment holding a set of identifiers and their attributes. The environment query thus has the signature $(\texttt{environment}\colon Dict[Name, [Sort, Attributes]]) \to (\texttt{permitted_candidates}\colon Set[Name])$. For example, a query might fetch the set of variable names that have been declared with an appropriate type for the current scope. Notice that both the ChopChop and decode-time grammar functions take inputs that grow with the generated sequence, creating a growing opaque hole that the constrainer can neither access nor optimize. The ChopChop authors report timeouts caused by deeply nested expressions, and decode-time grammar similarly restricts grammar nesting to a finite depth.

In response, we limit the input scope to individual terminal instances (for example, the text `100` is an instance of the grammar terminal type INT), while at the same time enriching the output type so that outputs can be aggregated and integrated with grammar rules that already run on GPU-accelerated chart parsers. Because terminal instances are non-overlapping segments of text, this bounds the growth in cost both within a deontic function call and in the total number of deontic function calls. Given n generated tokens and d defined deontic functions, there are $O(d \cdot n)$ deontic function calls in total, and $O(d)$ calls at each step. Furthermore, because the input type is as simple as a short segment of text, the results of deontic function calls are easy to cache.

3.1 The Deontic Functions

CDoT deontic functions have the type $(\texttt{terminal_instance}: String) \rightarrow (\texttt{necessary_conditions}: Set[Regex], \texttt{permitted_conditions}: Set[Regex])$. That is, the input is a terminal instance and the output is a pair of conditions, necessary ($\Box$) and permitted ($\Diamond$), each of which is a set of regular expressions.

A deontic function is attached to an anchor; together, they form a deontic rule. The anchor specifies the structural relationship between the antecedent, whose terminal instance is passed to the function, and the consequents, whose terminal instances are constrained by its output. The function determines *what* conditions apply, while the anchor determines *where* they apply.

For example, consider the following rules, which enforce part of an SQL schema:

SQL grammar (snippet):

```
query → select_clause from_clause other_clauses
```

Deontic rules:

```
TABLE_NAME ≫ from_clause > query < other_clauses ≪ COLUMN_NAME
def tables_permit_columns(table_name):
    if table_name == "courses":
        return □ {}, ◊ {"course_id", "course_title"}
    ...
COLUMN_NAME ≫ select_clause > query < from_clause ≪ TABLE_NAME
def must_select_tables(column_name):
    if column_name == "course_id":
        return □ {"teachers", "courses"}, ◊ {}
    ...
```

In the first rule, the anchor specifies that the terminal `TABLE_NAME` in a `from_clause` will result in constraints on the terminal `COLUMN_NAME` in a corresponding `other_clauses`. Before such a `COLUMN_NAME` is generated, the corresponding `TABLE_NAME` instance is passed to the function to collect constraints on the `COLUMN_NAME` value. For `courses`, the function returns an empty necessary set indicating that no columns are required to appear, and a non-empty permitted set indicating that the `course_id` and `course_title` columns are allowed to appear. If several `TABLE_NAME`s occur in appropriate `from_clause`s, their permissions are combined by union, thus allowing columns supplied by any of those tables.

The second rule reverses the dependency: its anchor connects column names in `select_clause` to table names in `from_clause`. Each selected `COLUMN_NAME` is passed to `must_select_tables`, whose necessary conditions specify which tables must occur. As written, selecting `course_id` requires both `teachers` and `courses`, because each expression in the necessary set must match at least one consequent table instance. If either table would suffice, the function should instead return the single necessary regular expression $\{$`"teachers|courses"`$\}$. The empty permitted set contributes no additional table permissions.

Notice that a SQL query may select multiple columns from multiple tables, yet a deontic function is limited to a single terminal instance at a time. Their collective effect is obtained by taking the union of their necessary sets and, separately, the union of their permitted sets within the relevant scope. Each selected column contributes requirements on the tables that must appear, while each table contributes permissions for its columns.

In a GPU implementation, these unions need not be materialized explicitly. Instead, we evaluate matches against the individual specifications and combine the results using dynamic programming.

It is instructive to observe that, as we aggregate, the number of regular expressions in both the necessary and the permitted set grows, yet the constraint described tightens in the necessary case and loosens in the permitted case. In a sense, the necessary condition acts as a lower bound and the permitted condition acts as an upper bound.

3.2 Deontic Semantics for Unit Propagation

The deontic function's output, the necessary ($\Box$) and permitted ($\Diamond$) sets are designed to help compactly express steering conditions of derivation. We demonstrate this aspect by implementing in CDoT the unit propagation algorithm, a common procedure in SAT solvers. A chain of thought for solving a satisfiability problem can be modeled as:

```
start → bad_guess* good_guess
```

Each guess is a sequence of assignments of Boolean values (e.g. a; $\neg b$;), derivable from the rule

```
assignment_seq → assignment_seq SEMI¹ assignment
```

Given a Boolean expression in conjunctive normal form $(a \vee b) \wedge (\neg a \vee c)$, a solution must pick a literal from each clause ($\Box$ {"$a|b$", "$\neg a|c$"}), and may contain any of the literals in the formula ($\Diamond$ {"a", "b", "$\neg a$", "c"}), as expressed by the following set of grammar rules. Notice that the necessary and permitted sets are not wrapped in a deontic function and conditioned on previous terminals; rather, they are attached statically to `sufficient_assignment_seq` and observe all assignments beneath it. Finally, we use semicolons of differing priority to enforce the execution order of cascading if-statements (SEMI^3 takes priority over SEMI^1, and is therefore used in `good_guess` to check whether a solution has already been found before generation continues).

```
good_guess → sufficient_assignment_seq SEMI³
sufficient_assignment_seq → assignment_seq

sufficient_assignment_seq ≪ ASSIGNMENT
□ {"a|b", "¬a|c"}, ◇ {"a", "b", "¬a", "c"}
```

The unit propagation algorithm exploits the fact that when all but one of the literals in a clause of a conjunctive normal form have been rejected, the remaining literal must be accepted. For example, given the clause $a \vee b$, if we commit to $\neg a$ then we must also accept b. This can be expressed by the following rules. Notice that b is not permitted in the condition; otherwise we would risk triggering the same unit propagation rule repeatedly.

```
unit_prop → condition_b SEMI² assign_b
condition_b → assignment_seq
assign_b → ASSIGNMENT

condition_b ≪ ASSIGNMENT
□ {"¬a"}, ◇ {"¬a", "c"}

assign_b ≪ ASSIGNMENT
□ {"b"}, ◇ {"b"}
```

We can likewise express bad guesses, such as those that commit to contradictory literals (e.g. a and $\neg a$) or those that have fallen into a loop. The full implementation of the constrainer is in **??**.

Notice that, without deontic functions, attaching deontic specifications statically to non-terminals does not yield additional expressive power. In fact, the constrainer above can be expressed by a finite state machine. It nevertheless improves runtime complexity by offering a dense representation of the computation. Consider the size of that finite state machine and its implications: a solution to the Boolean formula can be extracted from such a machine in linear time using a depth-first search over its states, so one must either prove $P = NP$ or accept an $O(2^n)$ construction algorithm.

3.3 The Variable Scoping Problem and Union Permission

Efforts to extend the expressive power of constrainers often focus on variable scoping: a program may reference only variables defined in the current scope or an ancestor scope, not those defined in sibling or descendant scopes. We show that variable scoping also underlies other problems in constrained decoding, even when they are not explicitly formulated in these terms.

The SQL example above already shows variable matching within a single scope, but handling nested scopes requires a modified semantics that we call *union-permitted*. That is, a derivation's permitted set is extended with the permitted sets of its ancestors and descendants. This naturally mirrors the syntax tree, since non-overlapping scopes stand in no ancestor/descendant relationship to one another. We make this concrete in the rules below.

Rules for variable scopings

```
stmt_list → stmt stmt_list
stmt_list → scope stmt_list
scope → stmt_list

% union-◊
VAR_DCLR ≫
  stmt > stmt_list < stmt_list
    ≪ VAR_USE

def permission_passthrough(var_dclr):
   return □ {}, ◊ {var_dclr}
```

VAR_DCLR	variable declaration
VAR_USE	variable use
stmt	statement
stmt_list	list of statements

Colored forks: declaration anchor matches.
No scope anchor: no grant from A to B.
Triangle: abbreviated descendant subtree.
Scopes: A and B are siblings within P.

Example derivation tree

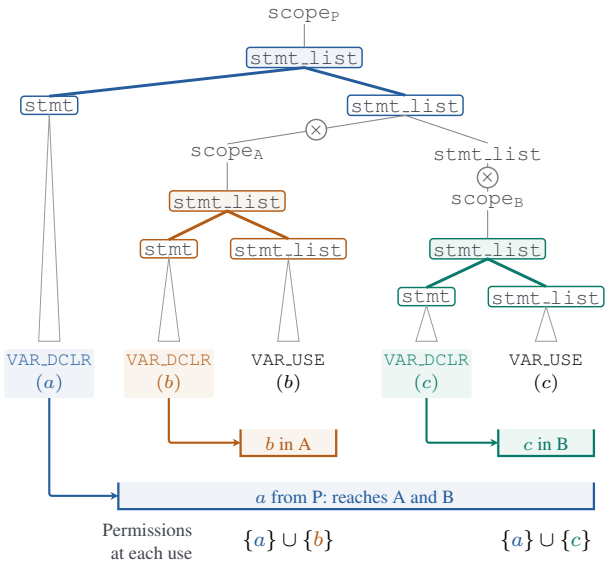


Figure 1: **Permission merging in three scopes.** Scope A and its sibling B share parent scope P. The declaration a in P reaches both children; b in A and c in B remain local. Colored forks match `stmt > stmt_list < stmt_list`. Crossed circles mark scope productions without this anchor, whose antecedent branch must be a `stmt`. No anchor connects declarations in A as antecedents to uses in B as consequents, so A's permissions do not leak into B. Triangles abbreviate the descendant paths to `VAR_DCLR` and `VAR_USE`. Leaf/base productions and empty list tails are omitted.

Rules of the same form can be extended to implement a limited type checker or the e-graph-based equivalence checker described in ChopChop (Nagy et al., 2026). We begin by augmenting the context-free grammar skeleton with type annotations. For example, we specialize the nonterminal `expr` into `expr_number` and `expr_bool`, with corresponding productions such as

$$\texttt{expr_number} \rightarrow \texttt{expr_number "+" expr_number} \mid \mathit{INT}$$

We can then track and enforce the types of variables defined by the LLM using the following rule:

```
stmt_list → let_number stmt_list
let_number → "let" VAR_DCLR ":" "number" "=" expr_number ";"

% attr union-◊
VAR_DCLR ≫ let_number > stmt_list < stmt_list ≪ VAR_USE_NUMBER
def permission_passthrough(var_dclr):
   return □ {}, ◊ {var_dclr}
```

ChopChop also implements an e-graph-based equivalence constrainer, which is useful when the LLM is instructed to rewrite an expression. An e-graph captures the set of subexpressions that are equivalent under a set of rewrite rules, and it can likewise be translated into a context-free grammar.

For example, we can have a context-free non-terminal that generates all known equivalents of `a ^ 2 * a ^ 2`. Again, a problem arises when we need to hoist out common computations and bind them to names. For example, we may want to write `let b = a ^ 2 in b * b` in order to avoid a repeated power-of-two computation. The rules needed to handle this are similar to those for the type constrainer, and we do not repeat them here.

In the cases we have discussed, CDoT has the limitation that it cannot handle scope shadowing. That is, when the same variable name has been declared across nested scopes, we should default to the declaration from the innermost scope.

On the other hand, CDoT provides machinery that is both optimizable and programmable. ChopChop enables the user to implement variable scoping, but is too permissive to admit easy optimization. Decode-time grammar and tree of parsers rely on a specialized scoping mechanism hardcoded into the constrainer.

4 Incremental Parsing and Correctness

Our parser inference rule is included in section H, and we focus on correctness in this section.

Our incremental CYK parser aligns each conditional anchor with a binary rule: a left antecedent and a right consequent. Anchors specify permitted instances and necessary witnesses. Strict permission checks every applicable strict anchor; union permission accepts any applicable permitting anchor. Alive entries track antecedents surviving permission pruning, while necessary witnesses are tracked separately. A merge requires an antecedent alive before commit, a completed consequent, satisfied necessity, and all strict checks. We assume unambiguous terminal instances, exact summaries, and admission and retention of unconstrained paths.

Completeness. Fix a valid tree τ. Maintain that its completed antecedents with unread consequent instances remain alive and encountered witnesses are recorded. For a constrained candidate, upward and downward CYK reachability follow τ from the candidate's leaf and root to a permitting antecedent, collecting its permission. The same tree path enables commit to propagate permission to every needed antecedent above or below that source; validity ensures that all strict guards pass. Recorded witnesses enable closing merges, and newly completed antecedents become alive. Induction therefore completes the root. This argument applies to both permission variants.

Soundness. Reconstruct τ from the accepted root using the completed children from which its alive antecedents originated. Each merge checked its necessary and strict premises, establishing soundness for strict permission.

Union additionally requires *uniform coverage*: all CFG trees compatible with the prefix and candidate type that survive the deontic checks on the committed prefix agree on the candidate's covering anchors and antecedent spans. The parser must establish that each admission has a witness in this class justifying its permission or unconstrained status. Inductively, uniform coverage transfers each admission's justification to τ; necessary and strict premises hold by reconstruction. Thus every instance is free or permitted, and τ is valid. An violating example can be find in section B.

For variable scoping, coverage follows when surviving parses agree on declaration boundaries and visibility, with one eligible type or e-class anchor per declaration. Constructor alternatives must resolve before probe; union types use an anchor representing the whole union rather than competing member-type alternatives.

Prefix extendability. There are cases such as nested permission have disjoint components, necessary condition require more terminal instances than the underlying skeleton can generate. We assume the skeleton would offer an alternative generation path. Otherwise the constrainer would eventually terminate generation.

Model	Constrained		Unconstrained		
	CDoT	*ChopChop*	*Non-thinking*	*Thinking*	
	1024	*1024*	*1024*	*1024*	*2048*
Qwen3.5-0.8B	0.86	0.86	0.03	0.00	0.00
Qwen3.5-2B	0.88	0.88	0.11	0.00	0.02
gemma-4-E2B-it	0.98	0.98	0.14	0.06	0.70

Table 1: Knight and Knave test (700 problems, greedy): success rate (consistent assignment) of constrained vs. unconstrained models. Constrained runs use thinking mode, with CDoT and ChopChop as the constrainer; column subheaders give the new-token limit.

5 Experiments and Results

Our experiments consist of two parts. First, we complete and apply the unit propagation algorithm described above to demonstrate that advanced constrained decoders can actively participate in LLM reasoning through tight integration of symbolic reasoning with chain-of-thought, rather than merely enforcing well-formedness. Second, we evaluate CDoT's efficiency by implementing and benchmarking a broader range of constraints drawn from prior work.

5.1 Knight and Knave

Knight and Knave Xie et al. (2025) is a logic puzzle dataset where each character is either a knight (always tells the truth) or a knave (always lies). The task is to infer each character's type from their statements. For example, if Alice says "Alice and Bob have the same identity", and Bob says "Alice is lying", we must deduce that Alice is a knave and Bob is a knight. The dataset consists of textual and logical forms of puzzles with 2-8 characters as input, and character type assignments as output.

5.1.1 Constrainer Design

The implementation intuition is explained in section 3.2, and full grammar can be found in section C.

5.1.2 Experimental Setup and Baselines

We apply LLMs in a 0-shot setting, providing both the textual and logical forms of the puzzle in the input prompt and expecting assignments of characters to types as output. The only constrainer we where able to compare with is ChopChop.

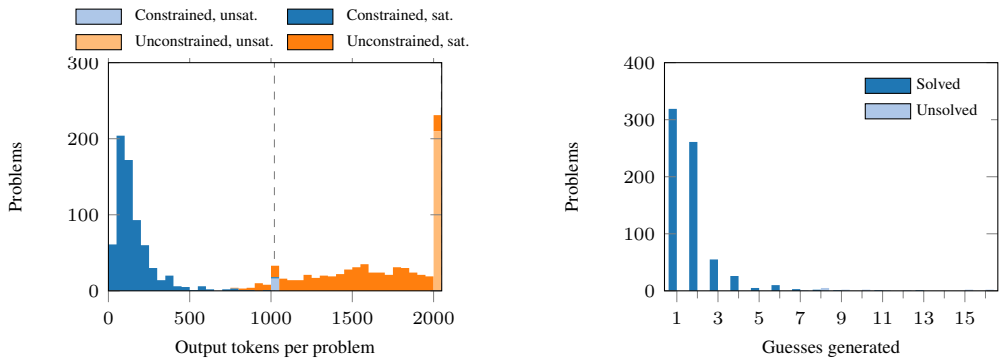
Among alternative baselines, constrainer of lower expressive level would have impractically large grammar. PiCARD is hard-coded to SQL grammar. IterGen can only backtrack an incorrect generation, while our constraints proactively force an assignment once it is fixed, and it is unclear how many assignments should be backtracked once a conflict arises. Tree-of-parsers and decode-time grammar are not publicly available.

5.1.3 Results

Table 1 shows that our CDoT constrainer yields major improvements in LLM generation for the Knight and Knave problem. When LLM output is limited to 1024 tokens, the constrained model success rate is over 80 points higher than that of unconstrained models. When unconstrained models are allowed to generate 2 times more tokens (2048 tokens), constrained models still out performs

Figure 2b shows that the constrained model can often solve the puzzle on the first attempt. This indicates that the constrainer and LLM are organically collaborating. If the constrainer alone was driving success, we would expect the distribution of guesses to be closer to random uniform, and it would be rare to solve the puzzle on the first guess.

We do not expect removing the token cap entirely to reverse the constrained model's solve-rate advantage. Our analysis shows that the unconstrained model leaves 6% of two- and three-person



(a) Output length by outcome.

(b) Guesses per problem.

Figure 2: Knights and Knaves, gemma-4-E2B-it with the pivot constrainer (700 problems, greedy). (a) Token efficiency of satisfied (dark) vs. unsatisfied (light) outputs, constrained (blue) vs. unconstrained (orange). Constrained runs are cut off at 1021 new tokens and unconstrained runs at 2048; 215/700 unconstrained runs hit the cutoff (490/700 satisfied), while 684/700 constrained runs are satisfied and every constrained run that stops on its own does so within 953 tokens. (b) Constrained runs solve 319 problems on the first guess and 261 on the second. The first-guess success rate is far above the expected from a uniformly random assignment, indicating organic LLM/constrainer collaboration rather than constrainer dominance.

puzzles unsolved after 2,048 tokens, with reasoning traces that appear to have stalled, suggesting that additional tokens alone may not resolve these failures.

5.2 Constrainer efficiency

We implement a diverse set of constraints representative of those studied in prior work and compare our system's efficiency against both highly expressive systems, such as ChopChop, and less expressive systems, such as LLGuidance and XGrammar. Our system substantially outperforms ChopChop when enforcing equivalent constraints. Moreover, despite enforcing more complex constraints and not yet being fully optimized to exploit GPU performance, our system achieves efficiency not far from that of less expressive, production-grade libraries.

Spider (Yu et al., 2018) is a text-to-SQL task, where the model is provided with a database schema and a natural text question as input, and it is expected to produce as output the SQL query to the given database that corresponds to the given question. Spider is commonly used as an evaluation dataset in constrained decoding research (Scholak et al., 2021; Jivani et al., 2025, etc.). This task shares the form of library and API specification enforcement that is typically targeted by constrained decoding.

Type-checking and E-graph enforcement as implemented by ChopChop. Performs limited type checking and guaranteed refactoring correctness.

5.2.1 Experimental Setup and Baselines

We implement the same set of constraints in ChopChop and CDoT, with the caveat that CDoT cannot distinguish between identically named variables in parent and child scopes. For less expressive systems, such as XGrammar and LLGuidance, we provide only CDoT's context-free grammar skeletons, omitting the deontic rules.

5.2.2 Results

Table 2 includes the measured mask-to-mask latency for the constrainer. In our measurement, CDoT is only slightly slower than LLGuidiance, despite the later only enforces the context-free skeleton of

	KNK			Spider			e-graph			TypeScript$^{\alpha}$		
Engine	mean	p50	p99	mean	p50	p99	mean	p50	p99	mean	p50	p99
CDoT	0.7	0.2	2.4	2.5	2.1	6.7	2.5	1.5	12.7	1.6	1.6	3.6
LLG (skeleton)	<0.1	<0.1	0.1	1.1	1.1	3.1	0.8	0.5	8.3	0.6	0.5	1.4
XGrammar (skeleton)	<0.1	<0.1	0.1	182.3	45.1	517.0	27.8	2.8	541.1	4.7	5.4	8.9
ChopChop (per-check)	44.0	10.0	295.0	286.0	140.4	2669	32.0	18.2	214.7	63.3	38.2	332.2
ChopChop (per-token)	51.6	13.3	300.7	322.6	153.6	2768	80.3	26.0	845.3	98.5	52.3	372.4

Table 2: Constraint latency in milliseconds on one H100; We include production-grade context-free constrainer (LLGuidiance and XGrammar) for reference. We use LLGuidiance as a pre-filter for ChopChop to reduce rejections. CDoT consistently achieves performance comparable to LLGuidance and sometimes outperforms XGrammar, even when enforcing more complex constraints.

CDoT's constraint. We also verify the output quality of our constrainer and confirms our constrainer performs consistently against ChopChop, with details in section G.

Per-mask vs Per-check time The performance gap between CDoT and ChopChop is larger than the measured latencies alone suggest. Ideally, a constrainer should compute a validity mask over the entire LLM vocabulary rather than validate only individual tokens proposed by the model. Full-mask computation can run concurrently with the LLM forward pass, potentially hiding the constrainer's latency. In contrast, a token-checking approach must wait for the forward pass to produce candidate tokens before validation can begin. To support full-mask computation over LLM vocabularies that typically exceed 100,000 tokens, ChopChop would require substantially higher throughput.

Optimization Potential Our P99 latency measurements on Spider show that the H100 is only 25% faster than the RTX 3090 (6.0 vs 7.5 ms, excluding preprocessing and postprocessing), suggesting that our implementation does not fully exploit the H100's computational capabilities. Fully utilization of GPU hardware requires dedicated effort even for much simpler computation such as attention. Given the size of our parser, we leave this part of work for the future.

Decode-time Grammar is a concurrent work can leverage LLGuidance and XGrammar by reinitializing the constrainer as grammars change, but its unreleased implementation prevents direct comparison. Published results suggest approximately 2 ms of constraint-processing overhead per step, plus a 10 ms LLM forward pass; our entire constraint-processing pipeline averages 2 ms per step without overlap with LLM execution, though these timings are not directly comparable. Repeated reinitialization may also limit gains from preprocessing-heavy optimizations, whose setup costs can range from seconds to an hour (Li et al., 2026). Finally, the approach uses a hard-coded scope stack rather than exposing a programmable constraint interface.

6 Conclusion

We introduced Conditional Deontics over Terminals (CDoT), a programmable semantics for expressing relationships among spans in context-free parse trees. CDoT supports advanced decoding constraints and can be compiled into a CYK-style parser for efficient execution. Our results further demonstrate that constrained decoding can tightly integrate traditional symbolic reasoning algorithms with LLM chain-of-thought reasoning, substantially improving reasoning performance.

CDoT also brings conjunctive grammars closer to practical use. Their fast parsing and ability to express P-complete languages motivate further exploration of expressive, tractable constraints for structured generation and reasoning. An interesting direction is to further integrate the conjunctive grammar primitives such as conjunction and negation operators into CDoT, and explore practical usefulness of the resulting formalism.

Further implementation improvements remain an important direction, although the size of our algorithm makes this a substantial engineering undertaking.

AI USE STATEMENT

In this work, we used generative AI tools for [formulate mathematical claims, provide critical ingredients for proving mathematical claims, assist in the writing of proofs, propose or refine hypotheses, design or provide feedback on research methodology or experiments, implement methods,]. We have not used generative AI tools for [help develop theoretical models or conceptual frameworks, clean and reformat dataset], and [Generate synthetic data sets, assist with translation, support qualitative and thematic data analysis, interpret results] are not applicable to this work. Additionally, we used generative AI tools for [create or modify scientific figures or images, create or edit software code, brainstorming, sourcing/searching for information, edit a research paper to improve readability, identify relevant literature,]. We have reviewed all AI-assisted work. [implement prototype and conduct experiment manually first, before asking AI to provide high performance implementation, manually verify proves]. We take responsibility for the final content of this work, including text, claims or artifacts produced with the aid of generative AI.

REPRODUCIBILITY STATEMENT

Our experiments focus exclusively on decoding and use greedy decoding rather than sampling to control randomness. We use the standard released model weights at standard precision, without quantization. The hardware used for runtime measurements is specified in the experimental section. Our implementation makes assumptions about the LLM tokenizer and user-provided grammars, all of which are addressed in our discussion of correctness. We will release the source code upon publication.

REFERENCES

Mohammad Albinhassan, Pranava Madhyastha, and Alessandra Russo. SEM-CTRL: Semantically controlled decoding, 2026. URL https://arxiv.org/abs/2503.01804.

Mikhail Barash and Alexander Okhotin. An extension of context-free grammars with one-sided context specifications. *Information and Computation*, 237:268–293, 2014. ISSN 0890-5401. doi: https://doi.org/10.1016/j.ic.2014.03.003. URL https://www.sciencedirect.com/science/article/pii/S0890540114000595.

Yixin Dong, Charlie F. Ruan, Yaxing Cai, Ziyi Xu, Yilong Zhao, Ruihang Lai, and Tianqi Chen. XGrammar: Flexible and efficient structured generation engine for large language models. In *Eighth Conference on Machine Learning and Systems*, 2025. URL https://openreview.net/forum?id=rjQfX0YgDl.

Smit Jivani, Saravam Maheshwari, and Sunita Sarawagi. Reliable answers for recurring questions: Boosting text-to-sql accuracy with template constrained decoding. *Proc. ACM Manag. Data*, 3(6), December 2025. doi: 10.1145/3769822. URL https://doi.org/10.1145/3769822.

Lingxiao Li, salar rahili, and Yiwei Zhao. Correctness-guaranteed code generation via constrained decoding. In *Second Conference on Language Modeling*, 2025. URL https://openreview.net/forum?id=CYiXNIQegF.

Yongmin Li, Yihong Dong, Jia Li, and Ge Li. Efficient grammar-constrained decoding via parser stack classification, 2026. URL https://arxiv.org/abs/2608.03065.

Michał Moskal, Harsha Nori, Hudson Cooper, and Loc Huynh. Llguidance: Making structured outputs go brrr, 2025. URL https://guidance-ai.github.io/llguidance/llg-go-brrr.

Mikhail Mrykhin and Alexander Okhotin. The hardest language for grammars with context operators. *Theoretical Computer Science*, 958:113829, 2023. ISSN 0304-3975. doi: https://doi.org/10.1016/j.tcs.2023.113829. URL https://www.sciencedirect.com/science/article/pii/S0304397523001421.

Shaan Nagy, Timothy Zhou, Nadia Polikarpova, and Loris D'Antoni. Chopchop: A programmable framework for semantically constraining the output of language models. *Proc. ACM Program.*

Lang., 10(POPL), January 2026. doi: 10.1145/3776708. URL https://doi.org/10.1145/3776708.

Alexander Okhotin. Conjunctive grammars. *J. Autom. Lang. Comb.*, 6(4):519–535, April 2001. ISSN 1430-189X.

Kanghee Park, Timothy Zhou, and Loris D'Antoni. Flexible and efficient grammar-constrained decoding. In *Proceedings of the 42nd International Conference on Machine Learning*, ICML'25. JMLR.org, 2025.

Yves Schabes. The valid prefix property and left to right parsing of Tree-Adjoining Grammar. In Masaru Tomita, Martin Kay, Robert Berwick, Eva Hajicova, Aravind Joshi, Ronald Kaplan, Makoto Nagao, and Yorick Wilks (eds.), *Proceedings of the Second International Workshop on Parsing Technologies*, pp. 21–30, Cancun, Mexico, February 13-25 1991. Association for Computational Linguistics. URL https://aclanthology.org/1991.iwpt-1.4/.

Torsten Scholak, Nathan Schucher, and Dzmitry Bahdanau. PICARD: Parsing incrementally for constrained auto-regressive decoding from language models. In Marie-Francine Moens, Xuanjing Huang, Lucia Specia, and Scott Wen-tau Yih (eds.), *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pp. 9895–9901, Online and Punta Cana, Dominican Republic, November 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.emnlp-main.779. URL https://aclanthology.org/2021.emnlp-main.779/.

Zhengyang Su, Isay Katsman, Yueqi Wang, Ruining He, Lukasz Heldt, Raghunandan Keshavan, Shao-Chuan Wang, Xinyang Yi, Mingyan Gao, Onkar Dalal, Lichan Hong, Ed Chi, and Ningren Han. Vectorizing the trie: Efficient constrained decoding for llm-based generative retrieval on accelerators, 2026. URL https://arxiv.org/abs/2602.22647.

Shubham Ugare, Rohan Gumaste, Tarun Suresh, Gagandeep Singh, and Sasa Misailovic. Itergen: Iterative semantic-aware structured LLM generation with backtracking. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=ac93gRzxxV.

G. H. von Wright. Deontic logic. *Mind*, 60(237):1–15, 1951. ISSN 00264423, 14602113. URL http://www.jstor.org/stable/2251395.

Chulin Xie, Yangsibo Huang, Chiyuan Zhang, Da Yu, Xinyun Chen, Bill Yuchen Lin, Bo Li, Badih Ghazi, and Ravi Kumar. On memorization of large language models in logical reasoning. In Kentaro Inui, Sakriani Sakti, Haofen Wang, Derek F. Wong, Pushpak Bhattacharyya, Biplab Banerjee, Asif Ekbal, Tanmoy Chakraborty, and Dhirendra Pratap Singh (eds.), *Proceedings of the 14th International Joint Conference on Natural Language Processing and the 4th Conference of the Asia-Pacific Chapter of the Association for Computational Linguistics*, pp. 2742–2785, Mumbai, India, December 2025. The Asian Federation of Natural Language Processing and The Association for Computational Linguistics. ISBN 979-8-89176-298-5. URL https://aclanthology.org/2025.ijcnlp-long.148/.

Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, Zilin Zhang, and Dragomir Radev. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-SQL task. In Ellen Riloff, David Chiang, Julia Hockenmaier, and Jun'ichi Tsujii (eds.), *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pp. 3911–3921, Brussels, Belgium, October-November 2018. Association for Computational Linguistics. doi: 10.18653/v1/D18-1425. URL https://aclanthology.org/D18-1425/.

Shuoming Zhang, Ruiyuan Xu, Haofeng Li, Qiuchu Yu, Yangyu Zhang, Chunwei Xia, Xiaobing Feng, Chenxi Wang, Huimin Cui, and Jiacheng Zhao. Decode-time grammars: Constrained llm generation over a refinement order of grammar fragments, 2026. URL https://arxiv.org/abs/2607.18357.

Conjunctive Grammar

var_list	$\rightarrow$ decl var_list $\mid \varepsilon$
	$\rightarrow$ matched_use var_list
decl	$\rightarrow a$ decl $\mid c$
use	$\rightarrow b$ use $\mid c$
matched_use	$\rightarrow$ use $\& \trianglelefteq$ any_var_list pair c
any_var_list	$\rightarrow$ decl var_list
	$\rightarrow$ use var_list $\mid \varepsilon$
pair	$\rightarrow a$ pair $b \mid c$ any_var_list

CDoT

var_list $\rightarrow$ var_list var $\mid$ DECL
var $\rightarrow$ DECL $\mid$ USE
DECL $\rightarrow a^*c$
USE $\rightarrow b^*c$

```
anchor:
  DECL ≫ var_list > var_list < var ≪ USE
def permit_passthrough(decl):
    return □{}, ◊{decl}
```

Figure 3: Side-by-side comparison of a conjunctive grammar from example 2 of Barash & Okhotin (2014) and an equivalent CDoT grammar. The conjunctive grammar implementation is not only more verbose, it also cannot be extended beyond unary encoding of variable name, make it unable to be used in practical case. CDoT on the other hand, uses a simple skeleton grammar, and the deontic function can leverage Python semantics to support more complex interactions between the antecedent and consequent.

A Theoretical Expressiveness Vs Practicalness

Conjunctive grammars achieve beyond-context-free expressiveness while keeping a tractable computation complexity of $O(n)$ to $O(n^3)$. It is also easy to envision a prefix validator with similar complexity (comparing to traditional mildly-context-sensitive such as tree-adjoint grammar at $O(n^9)$ (Schabes, 1991)). However, theoretical expressiveness does not necessarily translate into practical expressive power.

We demonstrate this aspect using example 2 from Barash & Okhotin (2014), which is a simplified task of verifying a variable name is declared before used, formally,

$$L^{var} = \left\{ u_1 \dots u_n \;\middle|\; \begin{array}{l} \text{for every } u_i, \\ \textbf{either } u_i \in a^*c, \\ \textbf{or } u_i = b^k c \text{ and} \\ \text{there exists } j < i, \\ \text{for which } u_j = a^k c \end{array} \right\}$$

Each in-language string is a sequence of variable declaration (of form a^*c) and variable use (of form b^*c). A variable declaration and a variable use matches if and only if the number of a is equal to the number of b. A variable must be declared before first use. For example, $acbcaacbbc$ is a valid sequence that declares variable ac and aac before use, and $aacbc$ is an invalid sequence that uses bc before declaring ac. One can verify this language is beyond context-free by using $\{a^n b^m c^n d^m | m, n \geq 0\}$ for reduction.

Barash & Okhotin (2014) implements a conjunctive grammar for this language, shown on the left of fig. 3. The centerpiece of this grammar is rule "matched_use $\rightarrow$ use $\& \trianglelefteq$ any_var_list pair c", which allows derivation "α matched_use $\Rightarrow \alpha$ use" only if "any_var_list pair $c \Rightarrow^* \alpha$ use". The pair non-terminal generates a pair of matching variable declaration and use.

We implement an equivalent grammar using CDoT on the right of fig. 3. The skeleton grammar is a left-recursive list of variables. And the deontic function simply adds every variable declaration to the permitted set of variable uses.

Besides the obvious difference in verbosity between two grammars, there is a deeper problem in how variable names are encoded. The conjunctive grammar implementation does not offer a clear way to move beyond unary encoded names. CDoT on the other hand, can be switched to encode variable names in full natural language alphabet, we only need to define DECL and USE with appropriate

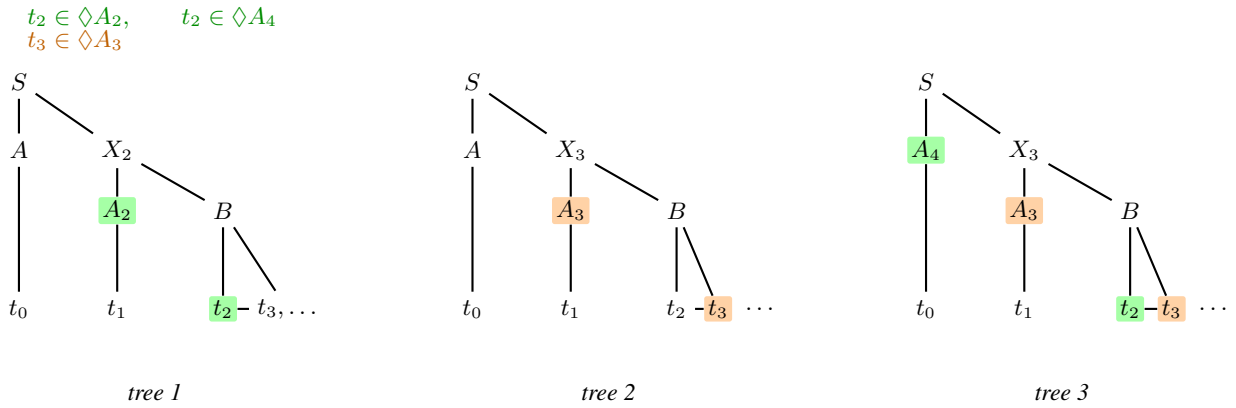


Figure 4: Three trees used in the counter example (highlighted nodes correspond to the committed constraints).

regular expressions. Furthermore, full Python semantics is available in deontic function to allow more complex relationships to be implemented.

B Union-$\Diamond$ Counter Example

Under union-$\Diamond$ semantics, consider a prefix where two deontic rule instances satisfy the following conditions: (1) they share at least one potential derivation tree, and (2) each can also belong to derivation trees that exclude the other. In such cases, permission granted in one derivation tree may incorrectly propagate to another, causing cross contamination of permissions.

We illustrate this with two examples: one demonstrating correct pruning behavior, and one demonstrating a failure case.

B.1 Case 1: Sound and complete (as strict as specified)

Consider trees in fig. 4. Assume a grammar allows tree 1 and tree 2, but does not have tree 3. Although node A is used in two separate trees, it does not cause problem. Tree 2 is pruned after committing t_2 for lack of permission. Tree 1 is kept alive for having A_2, but pruned after committing t_3 for lack of permission. This is illustrated in table 3.

	t_2	t_3
A	alive (prop. from A_2)	pruned
A_2	alive ($t_2 \in \Diamond A_2$)	pruned
A_3	pruned	—
tree 1	alive ($t_2 \in \Diamond A_2$)	pruned
tree 2	pruned	—

Table 3: Case 1 (sound and complete): after committing, both trees are pruned.

B.2 Case 2: Complete but not sound (looser than specified)

Now assume a grammar allows tree 1, tree 2, and tree 3. Tree 1 and tree 2 as we know should have been pruned after committing t_2 and t_3, but instead tree 2 is incorrectly kept alive. This is because node A is kept alive by being tree 1 and 2, node A_3 is kept alive by being in tree 3, collectively all components of tree 2 survives, as shown in table 4

ERROR: no tree contains A, A_2, and A_3.

	t_2	t_3
A	alive (prop. from A_2)	alive (prop. from A_3)
A_2	alive ($t_2 \in \Diamond A_2$)	pruned
A_3	alive (prop. from A_4)	alive ($t_3 \in \Diamond A_3$)
A_4	alive (prop. from A_3)	alive ($t_3 \in \Diamond A_4$)
tree 1	alive ($t_2 \in \Diamond A_2$)	pruned
tree 2	alive	alive
tree 3	alive ($t_2 \in \Diamond A_4$)	alive ($t_3 \in \Diamond A_3$)

Table 4: Case 2 (complete but not sound): propagation keeps some combinations alive that cannot co-occur in any single grammar-valid tree. Red boxes mark components falsely kept alive.

C KNK

The KNK constrainer forces the model to write its reasoning as a sequence of guesses. Each guess is a list of literals. The grammar fixes the shape of a guess, and the deontic rules tie each guess's closing verdict to the puzzle's CNF clauses: `<<DONE>>` only when every clause is covered, *contradiction* only when some x and $\neg x$ are both written, and *Bad guess* only on a repeated literal. The rules also force unit-propagated literals, and they make every new guess begin by flipping a literal of an earlier failed guess (the *pivot*). The CNF, its literals and hence all $\Box/\Diamond$ sets are per puzzle. We show them schematically over the puzzle's literal set $\mathcal{L}$ (its person literals $\mathcal{P} \subseteq \mathcal{L}$) and its simplified clause list $C_1, \ldots, C_m$. For one concrete instance we use the two-person dev puzzle 0 (Zoey, Oliver): z = "Zoey-¿knight", o = "Oliver-¿knight", s = "statement 4 is True", and $\neg z$, $\neg o$, $\neg s$ are the knave/False forms. Its clauses are $o \vee z$, $\neg o \vee \neg z$, $\neg o \vee \neg z \vee \neg s$, $o \vee z \vee \neg s$, $s \vee \neg o \vee z$, $s \vee o \vee \neg z$, $\neg s \vee o$, $s \vee \neg o$.

C.1 Grammar

The unit-propagation productions are instantiated for $i = 1, \ldots, 31$ and both polarities. The per-puzzle part of the grammar is ASSIGNMENT, whose regex is replaced by the alternation of the puzzle's literals $\mathcal{L}$. No whitespace is allowed between terminals. A superscript on a terminal is its lexer priority. We omit the unused productions (`answer`, the thinking and non-existent-name terminals).

```
start → good_first | history good_guess
history → bad_first | history bad_guess
good_first → piv_good_first semi_3 GOOD_GUESS_TAIL
bad_first → piv_contra_first semi_4 CONTRADICTORY_GUESS_TAIL
       | piv_rep_first semi_3 REPETITION_GUESS_TAIL
good_guess → piv_good semi_3 GOOD_GUESS_TAIL
bad_guess → piv_contra semi_4 CONTRADICTORY_GUESS_TAIL
       | piv_rep semi_3 REPETITION_GUESS_TAIL
piv_good_first → FIRST_HEAD sufficient_assignment_seq
piv_contra_first → FIRST_HEAD contradictory_assignment_seq
piv_rep_first → FIRST_HEAD repetition_assignment_seq
piv_good → GUESS_HEAD sufficient_assignment_seq
piv_contra → GUESS_HEAD contradictory_assignment_seq
piv_rep → GUESS_HEAD repetition_assignment_seq
sufficient_assignment_seq → assignment_seq
contradictory_assignment_seq → assignment_seq
repetition_assignment_seq → assignment_seq_semi_1
repeating_assignment
assignment_seq_semi_1 → assignment_seq semi_1
assignment_seq → assignment_seq semi_1 assignment | HEAD_SP
pivot_first | unit_prop_rule
pivot_first → pivot_decl open_first
pivot_decl → PIVOT
open_first → GUESS_OPEN assignment
unit_prop_rule → var_1_is_true | var_1_is_false | ⋯ | var_31_is_false
var_i_is_true → var_i_true_condition semi_2 assignment
```

```
var_i_is_false → var_i_false_condition semi_2 assignment
var_i_true_condition → assignment_seq
var_i_false_condition → assignment_seq
assignment → ASSIGNMENT
repeating_assignment → REPEATING_ASSIGNMENT
semi_k → SEMI^k "\n"                                        (k = 1, ..., 4)
SEMI^k → ";"
ASSIGNMENT → NAME "->" KK | AUX_VAR                  (per puzzle: L)
REPEATING_ASSIGNMENT^2 → ASSIGNMENT
PIVOT^3 → NAME "->" KK
NAME → "Jacob" | "Noah" | ··· | "Ava"                        (51 names)
KK → "knight" | "knave"
AUX_VAR → "statement N is True" | "statement N is False"
(N = 1, ..., 31)
FIRST_HEAD → "Here is my first guess, starting with"
GUESS_HEAD → "Here is a different and improved guess, flipping an
earlier starting identity to"
HEAD_SP → " "
GUESS_OPEN → ": {\n"
GOOD_GUESS_TAIL → "} <<DONE>>"
CONTRADICTORY_GUESS_TAIL → "} This guess leads to a
contradiction\n\n"
REPETITION_GUESS_TAIL → "} Bad guess\n\n"
```

All four $\texttt{SEMI}^k$ lex the same ";". The priorities make ";" commit to the first viable reading, in this order: contradiction closure, good or repetition closure, forced unit literal, free literal.

C.2 DEONTIC RULES

Unless marked otherwise, the obligations of different rules on the same head are conjunctive, and the permissions of all activations that reach one output terminal are unioned. We use three further markers. `% attr any-`$\Box$ marks a rule whose obligations are *self-sufficient*: the head is satisfied once any one such rule on it is satisfied. `% attr strict-`$\Diamond$ marks a permission that every output terminal under the head must meet. `% attr hidden-`$\Diamond$ marks a permission that is enforced when a terminal is committed but is not used to shape the token mask. Every $\Diamond$ set contains its rule's $\Box$ set (the engine folds obligations into permissions).

Good guess: every clause is covered. There is one obligation per clause C_j, written as the alternation of its literals.

```
piv_good → GUESS_HEAD sufficient_assignment_seq
sufficient_assignment_seq → assignment_seq
% attr hidden-◇
sufficient_assignment_seq ≪ ASSIGNMENT
□ {"C_1", ..., "C_m"}, ◇ {"C_1", ..., "C_m"}
% instance:  □ {"o|z", "¬o|¬z", "¬o|¬z|¬s", "o|z|¬s", "s|¬o|z",
"s|o|¬z", "¬s|o", "s|¬o"},
%          ◇ {"z", "¬z", "o", "¬o", "s", "¬s"}
```

Contradiction: some x and $\neg x$ are both written. There is one rule per variable x with $x, \neg x \in \mathcal{L}$.

```
piv_contra → GUESS_HEAD contradictory_assignment_seq
contradictory_assignment_seq → assignment_seq
% attr any-□, hidden-◇
contradictory_assignment_seq ≪ ASSIGNMENT
□ {"x", "¬x"}, ◇ {"x", "¬x"}
% instance:  x ∈ {z, o, s}
```

Repetition: the closing literal repeats one already in the guess.

```
repetition_assignment_seq → assignment_seq_semi_1
repeating_assignment
repeating_assignment → REPEATING_ASSIGNMENT

% attr union-◊
ASSIGNMENT ≫ assignment_seq_semi_1 > repetition_assignment_seq <
repeating_assignment ≪ REPEATING_ASSIGNMENT
def permission_repeat(assignment):
    return □ {}, ◊ {assignment}
```

Unit propagation, condition. For each clause C with $|C| \geq 2$ and each literal $\ell \in C$, the head is `var_i_true_condition` if $\ell = x_i$ and `var_i_false_condition` if $\ell = \neg x_i$. Writing the negations of C's other literals licenses ℓ. In the 13 of 700 puzzles where unit propagation empties the formula, the raw clauses are kept, and a unit clause $\{u\}$ yields the same rule with □ {"$\neg u$"}.

```
var_i_true_condition → assignment_seq

% attr any-□, strict-◊, hidden-◊
var_i_true_condition ≪ ASSIGNMENT
□ {"¬ℓ′" | ℓ′ ∈ C, ℓ′ ≠ ±ℓ}, ◊ ℒ \ {"ℓ"}
% instance (C = o∨z, ℓ = o, head var_2_true_condition):
%     □ {"¬z"}, ◊ {"z", "¬z", "¬o", "s", "¬s"}
```

Unit propagation, conclusion. There is one rule for each literal ℓ that some condition rule can force.

```
var_i_is_true → var_i_true_condition semi_2 assignment

% attr any-□
SEMI² ≫ semi_2 > var_i_is_true < assignment ≪ ASSIGNMENT
def obligation_force_ℓ(semi):
    return □ {"ℓ"}, ◊ {"ℓ"}
% instance:  ℓ ∈ {z, ¬z, o, ¬o, s, ¬s}
```

Unit propagation, unused heads. The grammar has condition heads for 31 variables in both polarities. Every head that no rule above can force gets an obligation that no terminal can meet, which removes its production.

```
% attr hidden-◊
var_i_true_condition ≪ ASSIGNMENT
□ {"disabled"}, ◊ {"disabled"}
% instance:  x_i for i ∈ {3, 5, ..., 31}, both polarities
```

Pivot: the declared pivot is the guess's first literal.

```
pivot_first → pivot_decl open_first
pivot_decl → PIVOT
open_first → GUESS_OPEN assignment

% attr union-◊
PIVOT ≫ pivot_decl > pivot_first < open_first ≪ ASSIGNMENT
def permission_passthrough(pivot):
    return □ {}, ◊ {pivot}
```

Pivot: a later guess flips a person literal of an earlier failed guess. There are two rules, one with the right child `bad_guess` and one with `good_guess`.

```
history → history bad_guess
start → history good_guess

% attr union-◊
ASSIGNMENT ≫ history > history < bad_guess ≪ PIVOT
ASSIGNMENT ≫ history > start < good_guess ≪ PIVOT
```

```
def permission_flip(assignment):
    if assignment ∉ 𝒫:  return □ {}, ◊ {}
    return □ {}, ◊ {¬assignment}
```

Pivot: the first guess pivots on any person literal. There is one rule per first-guess head.

```
piv_good_first → FIRST_HEAD sufficient_assignment_seq

piv_good_first ≪ PIVOT
piv_contra_first ≪ PIVOT
piv_rep_first ≪ PIVOT
□ {}, ◊ 𝒫
% instance:  ◊ {"z", "¬z", "o", "¬o"}
```

D SPIDER

For Spider (text-to-SQL) the constrainer is the SQLite `SELECT` grammar plus four deontic rules (six anchors, since one rule is copied onto two folded parents). The rules tie each identifier in a query to the schema: a selected column permits, and obliges, a `FROM` table that owns it; a `FROM` table permits its own columns in the `WHERE`/`GROUP BY`/`HAVING`/`ORDER BY` clauses; and $\star$ or `DISTINCT` lifts the restriction. The schema is the union of all 166 Spider databases. The constrainer receives no `db_id`, so the rules are the same for every question.

Grammar. The grammar is SQLite's `parse.y` converted to Lark (740 lines). Only the productions the rules anchor on, and the identifier terminals, are shown here. The rest is standard SQLite `SELECT` syntax: `expr`, joins, subqueries (`"(" select ")"` in `FROM`, `IN`, `EXISTS` and scalar positions), `GROUP BY`/`HAVING`/`ORDER BY`/`LIMIT`, and window clauses. $[x]$ marks an optional item. Every keyword except `SELECT` is case-insensitive (`/KW\b/i`), and whitespace is ignored.

```
start → selectnowith SEMI
select → selectnowith | "WITH" wqlist selectnowith | "WITH"
"RECURSIVE" wqlist selectnowith
selectnowith → oneselect | selectnowith multiselect_op oneselect
oneselect → values | select_kw distinct selcollist from
other_clauses
select_kw → SELECT
distinct → ["DISTINCT" | "ALL"]
selcollist → sclp STAR | sclp expr as | sclp nm "." STAR
sclp → [selcollist ","]
as → [ids | "AS" nm]
from → ["FROM" seltablist]
seltablist → stl_prefix nm dbnm as on_using | stl_prefix "(" select
")" as on_using | ...
other_clauses → where_opt groupby_opt having_opt orderby_opt
limit_opt
   | where_opt groupby_opt having_opt window_clause orderby_opt
limit_opt
nm → idj | STRING
idj → ID | INDEXED | JOIN_KW
ids → ID | STRING

SELECT → /SELECT\b/   % case-sensitive
ID → /[^\W\d][\w\d]*\b/   % priority −100:  a keyword reading wins
STAR → "*"
SEMI → ";"
```

Schema maps and name tests. Let $\mathcal{D}$ be the 166 databases. Names are compared after `strip().lower()`. For a column name c and a table name t: $\mathrm{Own}(c)$ is the set of tables (in any database) that have a column c; $\mathrm{DbTabs}(c)$ is the set of all tables of every database that has a column c; $\mathrm{Cols}(t)$ is the set of columns of every table named t (in any database); $\mathrm{SameDb}(t)$ is the set of all tables of every database that has a table t; and Tabs is the set of all table names. Every

regex a rule returns is matched case-insensitively against the whole lexeme. A table or column name is a literal. Two regexes are shared:

```
AGG = "avg|count|max|min|sum"
ALIAS = "T\d | L | LL | L\d | LL\d"   % L a letter; LL a letter pair
that is not one of the
   % 2-letter keywords AS BY DO IF IN IS NO OF ON OR TO
```

The engine adds every $\square$ member to $\Diamond$ automatically ($W \subseteq P_d$), so the functions below list only the additional permitted names.

Deontic rules. All four rules are conditional union-$\Diamond$ anchors on `oneselect`. An antecedent (consequent) is any terminal of that type anywhere under the antecedent (consequent) child, including inside nested subqueries. Several covering anchors therefore combine by union: an `ID` in a subquery's `FROM` may be permitted by its own select list or by the enclosing select. Obligations are conjunctive: each selected column's clause needs its own witness. They are checked when the anchor's `from` child completes. An `ID` that lies in no consequent span is unconstrained. Examples are a select list without `DISTINCT` that is not nested in another select's clause, and the clauses of a select without `FROM`.

```
oneselect → select_kw distinct selcollist from other_clauses

% attr union-◇
ID ≫ selcollist > oneselect < from ≪ ID
def selected_col_necessitates_table(c):
   T = DbTabs(c) ∪ ({c} if c ∈ Tabs else {})
   O = Own(c) ∪ ({c} if c ∈ Tabs else {}) if Own(c) else {}
   P = T ∪ ⋃_{t∈T} Cols(t) ∪ {ALIAS, AGG}
   if c matches "T\d|AGG" or O = {}:  return □ {}, ◇ P
   if c matches ALIAS: return □ {"o1|⋯|ok|ALIAS"}, ◇ P   % O =
{o1, ..., ok}
   return □ {"o1|⋯|ok"}, ◇ P

% attr union-◇
STAR ≫ selcollist > oneselect < from ≪ ID
□ {}, ◇ {".+"}

% attr union-◇
ID ≫ from > oneselect < other_clauses ≪ ID
def selected_table_permits_col_in_other_clause(t):
   return □ {}, ◇ {t} ∪ Cols(t) ∪ SameDb(t) ∪ {ALIAS, AGG}

% attr union-◇; child sequence select_kw distinct selcollist
(distinct non-empty)
SELECT ≫ select_kw > oneselect < selcollist ≪ ID
SELECT ≫ select_kw > selectnowith < selcollist ≪ ID
SELECT ≫ select_kw > select < selcollist ≪ ID
□ {}, ◇ {".+"}
```

The last rule makes a `DISTINCT`/`ALL` select list unrestricted. It fires only on the expansions of `oneselect` whose `distinct` child is non-empty. The two extra anchors, with parents `selectnowith` and `select`, exist because CNF folds the unit rules `select` → `selectnowith` → `oneselect` into copies of the production headed by those parents. Without them, the first select and every subquery would not be covered.

Instance (`concert_singer`). The database has the tables `stadium`(Stadium_ID, Location, Name, Capacity, Highest, Lowest, Average), `singer`(Singer_ID, Name, Country, Song_Name, Song_release_year, Age, Is_male), `concert`(concert_ID, concert_Name, Theme, Stadium_ID, Year) and `singer_in_concert`(concert_ID, Singer_ID). The functions return (over the union of all databases):

```
selected_col_necessitates_table(concert_Name) =
   □ {"concert"}, ◇ {"stadium", "singer", "concert",
```

```
    "singer_in_concert", Cols of those tables, ALIAS, AGG}
    selected_col_necessitates_table(Song_Name) =
       □ {"singer|song"}, ◊ {the tables of concert_singer and music_1,
    their columns, ALIAS, AGG}
    selected_table_permits_col_in_other_clause(singer) =
       □ {}, ◊ {"singer", Cols(singer), "stadium", "concert",
    "singer_in_concert", "song", ALIAS, AGG}
```

So `SELECT concert_Name FROM concert WHERE Year = 2014;` is accepted. `concert` discharges the obligation of `concert_Name`, and `Year` ∈ Cols(concert) is permitted in `WHERE`. `SELECT Song_Name FROM stadium;` is rejected: `stadium` is permitted, but the obligation □{"singer|song"} is unmet when `from` closes.

E E-GRAPH

For each benchmark program, ChopChop's e-graph task saturates the program under algebraic rewrites. The constrainer then accepts exactly the `let`-programs whose value is still an e-class of that saturated e-graph. We compile each saturated e-graph into a class-indexed grammar: the grammar keeps the productive e-classes q that are reachable from the root class r, one nonterminal family per class. Deontic rules then control the identifiers. A name may be used as q only after it has been bound (`let x = ...`) to an expression of class q. Each new binding permits every identifier except the name it binds. The grammar is built separately for each program, and each program is run with and without a closing fence, giving 20 constrainers. In the no-fence variant, the opening fence is teacher-forced and the program ends at EOS. Below, q, q_1, q_2 range over the retained e-classes.

```
start → "```" let_expr ["```"]
let_expr → add_r | binding_q body
binding_q → "let" DECL "=" add_q
body → "in" let_expr
add_q → mul_q
mul_q → app_q
app_q → atom_q
atom_q → nna_q
nna_q → "(" add_q ")" | USE_q
add_q → add_q1 "+" mul_q2                  for each e-node Add(q1,q2) ∈ q
add_q → add_q1 "-" mul_q2                  for each e-node Sub(q1,q2) ∈ q
mul_q → mul_q1 "*" app_q2                  for each e-node Mul(q1,q2) ∈ q
mul_q → mul_q1 "/" app_q2                  for each e-node Div(q1,q2) ∈ q
app_q → app_q1 nna_q2                      for each e-node App(q1,q2) ∈ q
atom_q → "-" atom_q1                         for each e-node Neg(q1) ∈ q
nna_q → "x"                          for each e-node Var(x) or Num(x) ∈ q
DECL, USE_q → /[A-Za-z_][A-Za-z_0-9]*/ \ (original_names ∪ {"let",
"in"})
%ignore /[ \t\r\n\x0b\x0c]+/

start ≪ USE_q
□ {}, ◊ {}

% attr union-◊
DECL ≫ binding_q > let_expr < body ≪ USE_q
def identity(decl):
    return □ {}, ◊ {decl}

% attr union-◊
DECL ≫ binding_q > let_expr < body ≪ DECL
def exclusion(decl):
    return □ {}, ◊ {/[A-Za-z_][A-Za-z_0-9]*/ \ {decl}}
```

Each retained class q contributes these three rules. The first rule makes every use of q need a permission: its permission set is empty and it has no obligations. The second and third rules grant permissions conditionally, and their grants are unioned. In both transfer functions, `decl` is the committed `DECL` text with the surrounding whitespace removed. The identity grant permits exactly that name, as an escaped literal. The exclusion grant permits every identifier other than that name.

Instance: `positives`. The reference program is `(sum (filter positive xs)) / (length (filter positive xs))`. After saturation, 13 of its 23 e-classes are retained, with 26 e-nodes, 133 productions and 39 deontic rules. The fragment below shows the root class `math_11`, the constant class `math_12` (the numeral `1`, closed under `1*1`) and the class `math_8` (`filter positive xs`). It also shows the three rules of `math_8`; the other 12 classes follow the same pattern. The unit chains $\text{add}_q \to \cdots \to \text{nna}_q$ of the schema above are omitted, except where a level has further alternatives.

```
start → "```" let_expr
let_expr → add_math_11 | binding_math_2 body | ... | binding_math_14
body
binding_math_8 → "let" DECL "=" add_math_8
body → "in" let_expr
mul_math_11 → app_math_11 | mul_math_14 "/" app_math_10 | mul_math_9
"/" app_math_10
   | mul_math_11 "*" app_math_12 | mul_math_12 "*" app_math_11 |
mul_math_13 "*" app_math_14
   | mul_math_13 "*" app_math_9 | mul_math_14 "*" app_math_13 |
mul_math_9 "*" app_math_13
nna_math_11 → "(" add_math_11 ")" | USE_MATH_11
mul_math_12 → app_math_12 | mul_math_12 "*" app_math_12
nna_math_12 → "(" add_math_12 ")" | USE_MATH_12 | "1"
app_math_8 → atom_math_8 | app_math_7 nna_math_2
nna_math_8 → "(" add_math_8 ")" | USE_MATH_8
app_math_7 → atom_math_7 | app_math_4 nna_math_3
nna_math_2 → "(" add_math_2 ")" | USE_MATH_2 | "xs"
DECL, USE_MATH_q → /[A-Za-z_][A-Za-z_0-9]*/ \ {"filter", "in",
"length", "let", "positive", "pow", "sqrt", "sum", "xs"}

start ≪ USE_MATH_8
□ {}, ◊ {}

% attr union-◊
DECL ≫ binding_math_8 > let_expr < body ≪ USE_MATH_8
def identity(decl):
   return □ {}, ◊ {decl}

% attr union-◊
DECL ≫ binding_math_8 > let_expr < body ≪ DECL
def exclusion(decl):
   return □ {}, ◊ {/[A-Za-z_][A-Za-z_0-9]*/ \ {decl}}
```

F TypeScript

The TypeScript constraint encodes ChopChop's type-pruning fragment: typed `let`/`const` bindings, functions over a fixed menu of ten signatures, the `Math` environment, and ChopChop's return flow. Each declaration permits its own name, for its type, over the rest of its scope, and every use, assignment target and call site must be permitted by some declaration. The lowered grammar has 268 nonterminals, 1,001 productions and 364 rules, so we give it as schemas over four metavariables: $T, R \in \{\texttt{number}, \texttt{boolean}\}$; the statement-list context $C \in \{\texttt{top}, \texttt{t}, \texttt{number}, \texttt{boolean}, \texttt{numberv}, \texttt{booleanv}\}$, with C' the context of the rest of the list ($C' = \texttt{t}$ for $C \in \{\texttt{numberv}, \texttt{booleanv}\}$, otherwise $C' = C$) and $\beta(C)$ the context of a nested body ($\texttt{number} \mapsto \texttt{numberv}$, $\texttt{boolean} \mapsto \texttt{booleanv}$, otherwise $\beta(C) = C$); the body context $X \in \{\texttt{top}, \texttt{t}, \texttt{numberv}, \texttt{booleanv}\}$; and the signature $s = (\vec{A}) \Rightarrow R$ with $\vec{A} \in \{(), (n), (b), (n, n), (n, n, n)\}$. Context `t` is ChopChop's Top type. A `number` or `boolean` body must end in a definite return; inside it, the first statement of a nested body is typed at $R|\texttt{void}$ and the rest at Top. At decoding time, the fence and the prompt's function header are forced as a fixed prefix.

```
start → "```" "typescript" stmts_top "```" | "```" "typescript"
stmts_top "```" trail_comments
trail_comments → COMMENT | COMMENT trail_comments
stmts_C → letdecl_T stmts_C′ | constdecl_T stmts_C′ | letdeclfn_s
```

```
stmts_C′ | constdeclfn_s stmts_C′
    | funchead funccont_s_C | other_β(C) stmts_C′ | COMMENT stmts_C′
stmts_C → letdecl_T | constdecl_T | letdeclfn_s | constdeclfn_s |
other_β(C) | COMMENT                                (C ∉ {number, boolean})
stmts_R → ret_R stmts_t | ret_R
funccont_s_C → funcrest_s stmts_C′ | funcrest_s  (second alternative
only for C ∉ {number, boolean})
ret_R → "return" expr_R ";" | "if" "(" expr_boolean ")" ret_R
"else" ret_R | "{" stmts_R "}"
other_X → reassign ";" | exprstmt | block_X | if_X | while_X |
for_X
other_t → "return" expr_top ";"
other_Rv → "return" expr_R ";"
block_X → "{" stmts_X "}" | "{" "}"
cmd_X → other_X | letdecl_T | constdecl_T | letdeclfn_s |
constdeclfn_s | branchfunc_s
if_X → "if" "(" expr_boolean ")" cmd_X "else" cmd_X | "if" "("
expr_boolean ")" cmd_X
while_X → "while" "(" expr_boolean ")" block_X
for_X → "for" forparen_X
forparen_X → forlet_T fortail_X | forconst_T fortail_X |
forletfn_s fortail_X | forconstfn_s fortail_X | forasg fortail_X
fortail_X → expr_boolean ";" reassign ")" block_X
letbody_T → "let" LET ":" "T" "=" expr_T | "let" LET "=" expr_T
constbody_T → "const" CONST ":" "T" "=" expr_T | "const" CONST
"=" expr_T
letdecl_T → letbody_T ";"   constdecl_T → constbody_T ";"
forlet_T → "(" letbody_T ";"   forconst_T → "(" constbody_T ";"
letbodyfn_s → "let" LET "=" fnval_s   constbodyfn_s → "const"
CONST "=" fnval_s
letdeclfn_s → letbodyfn_s ";"   constdeclfn_s → constbodyfn_s ";"
forletfn_s → "(" letbodyfn_s ";"   forconstfn_s → "(" constbodyfn_s
";"
forasg → "(" reassign ";"
reassign → target_T "=" expr_T | target_number incop | incop
target_number | target_number compop expr_number
target_number → TARGET_number | "Math.PI"   target_boolean →
TARGET_boolean
exprstmt → expr_top ";"
incop → "++" | "--"   compop → "+=" | "-=" | "*=" | "/=" | "%="
intop → "+" | "-" | "*" | "/" | "%" | "**"   boolop → "&&" | "||"
cmp → "<" | "<=" | ">" | ">=" | "==" | "===" | "!=" | "!=="
base_number → NUMBER | USE_number | "(" expr_number ")" | "-"
base_number | "Math.PI"
    | mathunary "(" expr_number ")" | "Math.pow" "(" expr_number
"," expr_number ")"
    | mathvar "(" expr_number ")" | mathvar "(" expr_number
moreargs ")"
base_boolean → "true" | "false" | USE_boolean | "(" expr_boolean
")"
base_R → fnval3_s "(" args_A⃗ ")"    (s = (A⃗) ⇒ R; "(" ")" for A⃗ = ())
p1_number → base_number | base_number intop p1_number
p2_boolean → base_boolean | p1_number cmp p1_number
p3_boolean → p2_boolean | p2_boolean boolop p3_boolean
expr_number → p1_number | p3_boolean "?" p1_number ":" expr_number
expr_boolean → p3_boolean | p3_boolean "?" p3_boolean ":"
expr_boolean
expr_top → expr_number | expr_boolean | fnval_s | "(" expr_top ")"
| p3_boolean "?" then_top ":" expr_top
then_top → p1_number | p3_boolean | fnval3_s | "(" expr_top ")"
mathunary → "Math.log2" | "Math.sqrt" | "Math.floor" |
"Math.round" | "Math.ceil" | "Math.clz32"
mathvar → "Math.min" | "Math.max"   moreargs → "," expr_top | ","
expr_top moreargs
args_A1...Ak → expr_A1 "," args_A2...Ak   args_A → expr_A
```

```
funchead → "function" FUNC   funcname → FUNC
funcrest_s → "(" params_A⃗_R                ("(" funcend_R for A⃗ = ())
params_A_1 ... A_k_R → param_A_1 paramsep_A_2 ... A_k_R   params_A_R →
param_A funcend_R
paramsep_A⃗_R → "," params_A⃗_R
param_T → PARAM ":" "T"
funcend_R → ")" ":" "R" "{" stmts_R "}"
fnval3_s → CALL_s | "(" fnval_s ")"   fnval3_(n)   ⇒   n → mathunary
fnval3_(n, n) ⇒ n → "Math.pow"
fnval_s → fnval3_s | p3_boolean "?" fnval3_s ":" fnval_s
branchfunc_s → "function" branchfunctail_s   branchfunctail_s →
funcname funcrest_s
LET, CONST, PARAM, FUNC, TARGET_T, USE_T, CALL_s →
/[A-Za-z][A-Za-z0-9_]*/ minus {let, const, function, return,
if, else, while, for, true, false, number, boolean, string,
typescript}
NUMBER → /[0-9]+(\.[0-9]+)?/   COMMENT → "//" /[^\n]*/   %ignore
/[ \t\r\n\x0b\x0c]+/

% coverage:  every use, assignment target and call must be
permitted
start ≪ USE_T □ {}, ◊ {}
start ≪ TARGET_T □ {}, ◊ {}
start ≪ CALL_s □ {}, ◊ {}
% declaration names:  any identifier
start ≪ LET □ {}, ◊ {IDENT}   (likewise CONST, PARAM, FUNC)

% attr union-◊
LET ≫ letdecl_T > stmts_C < stmts_C′ ≪ USE_T
LET ≫ letdecl_T > stmts_C < stmts_C′ ≪ TARGET_T
CONST ≫ constdecl_T > stmts_C < stmts_C′ ≪ USE_T
LET ≫ letdeclfn_s > stmts_C < stmts_C′ ≪ CALL_s
CONST ≫ constdeclfn_s > stmts_C < stmts_C′ ≪ CALL_s
FUNC ≫ funchead > stmts_C < funccont_s_C ≪ CALL_s
FUNC ≫ funcname > branchfunctail_s < funcrest_s ≪ CALL_s
LET ≫ forlet_T > forparen_X < fortail_X ≪ USE_T
LET ≫ forlet_T > forparen_X < fortail_X ≪ TARGET_T
CONST ≫ forconst_T > forparen_X < fortail_X ≪ USE_T
LET ≫ forletfn_s > forparen_X < fortail_X ≪ CALL_s
CONST ≫ forconstfn_s > forparen_X < fortail_X ≪ CALL_s
PARAM ≫ param_A > params_AA⃗_R < paramsep_A⃗_R ≪ USE_A
(funcend_R for A⃗ = ())
PARAM ≫ param_A > params_AA⃗_R < paramsep_A⃗_R ≪ TARGET_A
(funcend_R for A⃗ = ())
def permission_passthrough(decl):
   return □ {}, ◊ {decl}
```

Every conditional rule uses `permission_passthrough`. Its argument `decl` is the declared identifier with leading whitespace removed, and it is permitted as an exact literal. Nothing in this grammar ever becomes obligatory. IDENT is the identifier regex above. For clarity, `"--"`, `"<"` and `">"` stand for the literal operators.

G REPLICATION OF VARIOUS CONSTRAINER

We measure the improvements of constrained model's output quality on varies constrainer and tasks. KNK is evaluated by solve rate, spider is evaluated by execution accuracy, TypeScript is evaluated by compilation rate, e-graph is evaluated by equivalency. We see constrained model outperform unconstrained consistently/ CDot and ChopChop behave similarly.

Task (n)	Model	Unconstr.	CDoT	ChopChop
KNK (700)	Qwen3.5-0.8B	1	601	601
	Qwen3-1.7B	3	620	620
	Qwen3.5-2B	2	613	613
	Gemma-4-E2B	44	684	684
Spider (1034)	Qwen3.5-0.8B	275	286	– [a]
TypeScript (74)	Qwen3.5-0.8B	36	61	48
e-graph (20)	Qwen3.5-0.8B	0	14	14

Table 5: – = not run. [a]Didn't run fully due to slowness. Step-by-step comparison of 122 data points was done, all identical to CDoT.

H PARSER INFERENCE RULES

Let $h = n + 1$, $\ell = r[i : k]$, and $r = (x \to a\, b)$. Probe fixes $\boldsymbol{t}$ of type T; its facts are local to that invocation. Arrows and fractions both denote inference rules.

Table 6: Probe from state n for position h.

Preprocessing

$$H^{\downarrow}_{\mathrm{p}}(s[1:\cdot])\ \textsc{Root}\quad x \to \mathsf{T} \implies H^{\uparrow}_{\mathrm{p}}(x[h:\cdot])\ \textsc{Frontier}$$

$$\big(\forall d \in D_{\mathrm{s}}(r) : Q_{d,n}(\ell) \subseteq P_d(\alpha_\ell)\big) \implies \mathsf{strict}^{\mathrm{p}}_n(\ell)\ \textsc{Strict-Prefix}$$

CYK-like *Structural reachability with strict guards*

$$\frac{H^{\uparrow}_{\mathrm{p}}(b[k+1:\cdot])\quad \mathsf{alive}_n(\ell)}{H^{\uparrow}_{\mathrm{p}}(x[i:\cdot]), X^{\uparrow}_{\mathrm{p}}(\ell)}\ \textsc{Up} \qquad \frac{H^{\downarrow}_{\mathrm{p}}(x[i:\cdot])\quad \mathsf{alive}_n(\ell)\quad \mathsf{strict}^{\mathrm{p}}_n(\ell)}{H^{\downarrow}_{\mathrm{p}}(b[k+1:\cdot]), X^{\downarrow}_{\mathrm{p}}(\ell)}\ \textsc{Down}$$

$$\frac{H^{\uparrow}_{\mathrm{p}}(a[i:\cdot])\quad a \preceq x}{H^{\uparrow}_{\mathrm{p}}(x[i:\cdot])}\ \textsc{LC-Up} \qquad \frac{H^{\downarrow}_{\mathrm{p}}(x[i:\cdot])\quad a \preceq x}{H^{\downarrow}_{\mathrm{p}}(a[i:\cdot])}\ \textsc{LC-Down}$$

Postprocessing

$$\frac{d \in r\quad X^{\uparrow}_{\mathrm{p}}(\ell)\quad X^{\downarrow}_{\mathrm{p}}(\ell)}{\mathsf{relevant}_n(\ell)}\ \textsc{Relevant} \qquad \frac{H^{\downarrow}_{\mathrm{p}}(x[h:\cdot])\quad x \to \mathsf{T}}{\mathsf{next}_n(\mathsf{T})}\ \textsc{Next}$$

$$\mathsf{relevant}_n(\ell),\ \mathsf{next}_n(\mathsf{T}),\ \boldsymbol{t} \in \widehat{P}(\ell) \implies \boldsymbol{t} \in \mathsf{permit}_n(\mathsf{T})\ \textsc{Collect}$$

Table 7: Commit of $\boldsymbol{t}_h$: CFG and strict permission, then the union extension.

Preprocessing

$$\frac{x \to \mathsf{T} \quad (\boldsymbol{t}_h) = \mathsf{T}}{\mathsf{chart}(x[h:h]),\ \mathsf{done}(x[h:h]),\ H_{\mathrm{p}}^{\uparrow}(x[h:\cdot])}\ \textsc{Scan} \qquad \frac{\boldsymbol{t}_h \in \widehat{P}(\ell)}{\mathsf{direct}_h(\ell)}\ \textsc{Direct}$$

$$\frac{\mathsf{satisfier}_n(\ell, q)}{\mathsf{satisfier}_h(\ell, q)}\ \textsc{Remember} \qquad \frac{\boldsymbol{t}_h \in W_q \quad (\boldsymbol{t}_h) = \mathsf{U}_q}{\mathsf{satisfier}_h(\ell, q)}\ \textsc{Witness}$$

$$\frac{\forall d \in D_{\mathrm{s}}(r) : Q_{d,h}(\ell) \subseteq P_d(\alpha_\ell)}{\mathsf{strict}_h(\ell)}\ \textsc{Strict} \qquad \frac{\exists B \in \mathcal{B}(\ell)\ \forall q \in B :\ \mathsf{satisfier}_h(\ell, q)}{\mathsf{need}_h(\ell)}\ \textsc{Necessary}$$

CYK-like *CFG and strict permission*

$$\mathsf{chart}(a[i:k]),\ \mathsf{chart}(b[k+1:h]),\ r = (x \to a\, b) \implies \mathsf{chart}(x[i:h])\ \textsc{Syntax}$$

$$\frac{H_{\mathrm{p}}^{\uparrow}(b[k+1:\cdot]) \quad \mathsf{alive}_n(\ell) \quad \mathsf{strict}_h(\ell)}{H_{\mathrm{p}}^{\uparrow}(x[i:\cdot]), X_{\mathrm{p}}^{\uparrow}(\ell)}\ \textsc{Up-p} \qquad \frac{H_{\mathrm{p}}^{\uparrow}(a[i:\cdot]) \quad a \preceq x}{H_{\mathrm{p}}^{\uparrow}(x[i:\cdot])}\ \textsc{LC-Up-p}$$

$$\mathsf{alive}_n(\ell),\ \mathsf{done}(b[k+1:h]),\ \mathsf{strict}_h(\ell),\ \mathsf{need}_h(\ell) \implies \mathsf{done}(x[i:h])\ \textsc{Complete}$$

Union-permission extension

$$\frac{\mathsf{alive}_n(\ell) \quad \mathsf{strict}_h(\ell) \quad H_{\mathrm{p}}^{\uparrow}(b[k+1:\cdot]) \quad \mathsf{direct}_h(\ell)}{H_{\mathrm{c}}^{\uparrow}(x[i:\cdot]), X_{\mathrm{c}}^{\uparrow}(\ell)}\ \textsc{Permit-Up}$$

$$\mathsf{alive}_n(\ell),\ \mathsf{direct}_h(\ell) \implies H_{\mathrm{c}}^{\downarrow}(b[k+1:\cdot]),\ X_{\mathrm{c}}^{\downarrow}(\ell)\ \textsc{Permit-Down}$$

$$\frac{H_{\mathrm{c}}^{\uparrow}(b[k+1:\cdot]) \quad \mathsf{alive}_n(\ell) \quad \mathsf{strict}_h(\ell)}{H_{\mathrm{c}}^{\uparrow}(x[i:\cdot]), X_{\mathrm{c}}^{\uparrow}(\ell)}\ \textsc{Up-c} \qquad \frac{H_{\mathrm{c}}^{\downarrow}(x[i:\cdot]) \quad \mathsf{alive}_n(\ell)}{H_{\mathrm{c}}^{\downarrow}(b[k+1:\cdot]), X_{\mathrm{c}}^{\downarrow}(\ell)}\ \textsc{Down-c}$$

$$\frac{H_{\mathrm{c}}^{\uparrow}(a[i:\cdot]) \quad a \preceq x}{H_{\mathrm{c}}^{\uparrow}(x[i:\cdot])}\ \textsc{LC-Up-c} \qquad \frac{H_{\mathrm{c}}^{\downarrow}(x[i:\cdot]) \quad a \preceq x}{H_{\mathrm{c}}^{\downarrow}(a[i:\cdot])}\ \textsc{LC-Down-c}$$

Postprocessing

$$X_{\mathrm{c}}^{\uparrow}(\ell) \vee X_{\mathrm{c}}^{\downarrow}(\ell) \implies \mathsf{keep}_h(\ell)\ \textsc{Union-Witness} \quad \mathsf{keep}_h(\ell) \implies \mathsf{alive}_h(\ell)\ \textsc{Retain}$$

$$\mathsf{done}(a[i:h]),\ r = (x \to a\, b) \implies \mathsf{alive}_h(r[i:h])\ \textsc{Activate}$$