

CONDITIONAL DEONTICS OVER TERMINALS: A MILDLY CONTEXT-SENSITIVE FORMAL GRAMMAR FOR CONSTRAINED DECODING

Anonymous authors

Paper under double-blind review

ABSTRACT

Constrained decoding researchers have recently thought to extend context-free constrainters to context-sensitive constraints such as matching selected columns to tables in text-to-SQL. This is challenging, because context-sensitivity is achieved by cross referencing different subtrees in the syntactic structure, which is only partially available during LLM generation. Recent frameworks such as IterGen gain some context sensitivity but with drawbacks including the need for expensive backtracking or speculative lookahead of LLMs, overly permissive semantics risking over-constraint, and complications in interfacing with works on improving running speed such as XGrammar. To address these concerns, we propose a new mildly-context-sensitive formal grammar called Conditional Deontics over Terminals (CDoT). Its incremental parser has $O(n^3)$ time complexity each step and allows for GPU acceleration, compared to the $O(n^8)$ of mildly context-sensitive tree-adjoining grammars and the $O(n^2)$ of context-free grammars. This new formal grammar is strong enough to implement the unit propagation algorithm, which we employ to assist an LLM in solving the “Knight and Knave” logical puzzle, achieving substantially improved performance at reduced output token budget. We also evaluate on the traditional constrained decoding task of text-to-SQL.

1 INTRODUCTION

Constrained decoding allows programmatic control of inference-time LLM behavior without modifying model weights or engineering prompts. Prior work has successfully employed it to enforce programming language specifications, molecular structures, and privacy requirements (Ugare et al., 2025; Loula et al., 2025; Scholak et al., 2021; Poesia et al., 2022).

As the technique matures, works have emerged that improve various aspects of it. One line of inquiry pushes for constraints of higher complexity class via speculative look-ahead or LLM backtracking (Ugare et al., 2025; Loula et al., 2025). This is not only computationally expensive—requiring user code to access and modify parser states—but also complicates integration with runtime optimization methods such as XGrammar (Dong et al., 2025). The semantics allowed by such frameworks can also sometimes be overly permissive and risk hazardous over-constraints that are difficult to detect.

To address these concerns, we design a new formal grammar for constrained-decoding, with the aim of shielding users from the complexities of constraider internals while providing researchers with a stable target interface. Our contributions are:

- We design a new formal grammar that offers context-sensitivity in practical use cases and can be efficiently parsed incrementally at $O(n^3)$ each step.
- Our formal grammar allows invocation of Python functions for added flexibility, while eliminating the need for user code to access and modify parser state.
- We implement a GPU-accelerated parser for our grammar by extending an incremental Valiant recognizer.

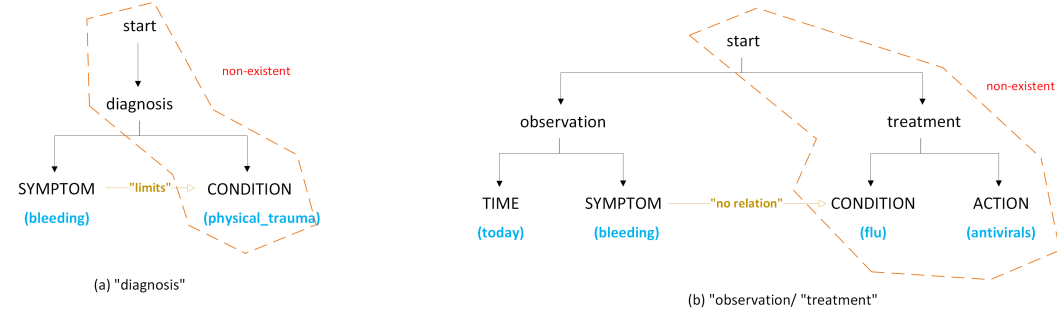


Figure 1: Example: a generated symptom should constrain the associated condition that is generated. But when `CONDITION` is being generated its ancestor structure (dashed lines) has not yet been determined due to the nature of incremental parsing. Knowing the possible ancestor structures is important, as the symptom-condition constraint should fire on the left tree but not the right.

2 RELATED WORK

Incrementally Parsable Mildly Context-sensitive Grammars: Constrained decoding relies on incremental parsers capable of maintaining the valid-prefix property. Despite a diverse selection of mildly context-sensitive grammars (Joshi & Schabes, 1997; Vijay-Shanker & Weir, 1993; Okhotin, 2001; Mrykhin & Okhotin, 2023; Barash & Okhotin, 2014), such grammars are often not designed with the goal of efficient constrained-decoding in mind. For example, Tree-Adjoining Grammars have an $O(n^8)$ step time-complexity (Schabes, 1991). The recently published grammar with one-sided context (Barash & Okhotin, 2014) within the conjunctive grammar family overlaps with the goal of constrained decoding, but is currently a theoretical construct applied only to toy grammars.

Constrained Decoding with Context Sensitivity: Context-sensitivity may reference across subtrees, as in fig. 1 where the `SYMPTOM` subtree must be observed to determine which `CONDITIONS` are allowable. In the case of constrained decoding, an incremental process, the ancestor nodes are unavailable at the time a constraint must be applied. Earlier work such as PiCARD (Scholak et al., 2021) ignored ancestor structure, selecting preceding terminals directly and risking constraint misfires as would be the case in the right side of fig. 1. IterGen (Ugare et al., 2025) and GenLM Control (Loula et al., 2025) allowed ancestor structures to be considered by invoked expensive backtracking and/or speculative look-ahead, where ancestor structures can emerge before constraints are applied (Ugare et al., 2025; Loula et al., 2025). Melcer et al. (2024) implements specialized context-sensitive constraints for some lexing features in programming languages.

3 FORMALISM FOR CONDITIONAL DEONTICS OVER TERMINALS (CDoT)

A *context-free grammar (CFG)* is a 4-tuple $G = (V, \Sigma, R, S)$ where

- V is a finite set of *nonterminal symbols* (also called variables), which represent syntactic categories that can be expanded,
- Σ is a finite set of *terminal symbols*, which are the basic alphabet symbols that appear in the strings of the language, with $V \cap \Sigma = \emptyset$,
- $R \subseteq V \times (V \cup \Sigma)^*$ is a finite set of *production rules* of the form $A \rightarrow \alpha$, where $A \in V$ (a nonterminal) and $\alpha \in (V \cup \Sigma)^*$ (a string of terminals and/or nonterminals),
- $S \in V$ is the designated *start symbol*.

The language generated by G is defined as $L(G) = \{w \in \Sigma^* \mid S \Rightarrow^* w\}$, where $\Rightarrow^*$ denotes the reflexive, transitive closure of the single-step derivation relation $\Rightarrow$ induced by R . A language $L \subseteq \Sigma^*$ is called *context-free* if there exists a CFG G such that $L = L(G)$.

We extend CFGs with *deontic rules*, which constrain not only the structure of derivations, but also the *obligations* and *permissions* of terminals. A *Conditional Deontics over Terminals (CDoT)* grammar is a pair $G^+ = (G, D)$ where

- $G = (V, \Sigma, R, S)$ is a context-free grammar (possibly with regular-expression terminals),
- D is a finite set of deontic rules, which constrain terminal instances generated by G .

Each deontic rule has an *anchor*, identifying the scope of the rule, and a *deontic function*, which maps terminal instances to a pair $(\Box C, \Diamond P)$, where $\Box C$ is a set of regular expressions (necessary conditions), and $\Diamond P$ is a set of regular expressions (permitted conditions).

Non-conditional deontic rules. A non-conditional rule has the form

$$X^{\text{consequent}} \ll t^{\text{consequent}},$$

with $X^{\text{consequent}} \in V$ and $t^{\text{consequent}} \in \Sigma$. When the deontic function of a non-conditional deontic rule returns $(\Box C, \Diamond P)$, the semantics are that all terminals descended from X must *collectively* satisfy every $r \in \Box C$, and each individual terminal must match at least one $r \in \Diamond P$.

Example (Balanced Meals). Given the grammar:

$$\begin{aligned} \text{meal} &\rightarrow \text{food beverage} \\ \text{food} &\rightarrow \text{DISH food} \mid \text{DISH} \\ \text{beverage} &\rightarrow \text{BEVERAGE beverage} \mid \text{BEVERAGE}. \end{aligned}$$

The following deontic rule enforces that every meal includes grains, protein, and vegetables ($\Box$), while also allowing optional extras like soup or steamed_egg ($\Diamond$).

```

anchor: food  $\ll$  DISH
def enforce_meal():
    return ( $\Box$ {rice|bread, chicken|fish,
               salad|roasted_vegetable},
            $\Diamond$ {rice, bread, chicken, fish,
               salad, roasted_vegetable, steamed_egg, soup})

```

Conditional deontic rules. A conditional rule has the form

$$t^{\text{antecedent}} \gg X^{\text{antecedent}} B \dots D > X^{\text{parent}} < X^{\text{consequent}} \ll t^{\text{consequent}},$$

where $t^{\text{antecedent}}, t^{\text{consequent}} \in \Sigma$, and $B, D, X^{\text{antecedent}}, X^{\text{consequent}} \in V$ are variables appearing in a parent production $X^{\text{parent}} \rightarrow \dots X^{\text{antecedent}} B \dots D X^{\text{consequent}} \dots$. The deontic function of a non-conditional deontic rule takes an antecedent terminal instance $t^{\text{antecedent}}$ as input, and uses it to determine and return constraints $(\Box C, \Diamond P)$, on the consequent terminal $t^{\text{consequent}}$.

Example (Food-Drink Pairings). Given the same grammar as the previous example, the following deontic rule enforces that when a food and a beverage are paired within a meal, steaks require red wine, fish requires white wine, and spicy noodles require soda.

```

anchor: DISH  $\gg$  food > meal < beverage  $\ll$  BEVERAGE.
def match_drink_to_food(terminal_instance):
    match terminal_instance.text:
        case 'steak':
            return ( $\Box$ {'red_wine'},  $\Diamond$ {})
        case 'fish':
            return ( $\Box$ {'white_wine'},  $\Diamond$ {})
        case 'spicy_noodle':
            return ( $\Box$ {'soda'},  $\Diamond$ {})

```

Notice that, while a single antecedent variable *food* can cover multiple antecedent DISH variables, each invocation of the deontic function *match_drink_to_food* sees only a single instance of DISH. This prevents repetitive calling of the deontic function over different combinations. At the same time, the conjunctive nature of $\Box$ and disjunctive nature of $\Diamond$ will still naturally combine so that, if both fish and steak are ordered, both red and white wine are requested.

4 PARSER

CYK-like parsers use **TABLE** $[row, col, nt]$ to denote $nt \Rightarrow *t_{row} \dots t_{row+col}$. To also track deontic rules we introduce a new state type **LF** $[row, col, cnf]$, which indicates a property of a CNF context-free rule with its left foot covering $t_{row} \dots t_{row+col}$, and the CNF rule itself covering t_{row} to any possible future terminal.

Our parser consists close to 20 slightly modified incremental valiant recognizers computing different properties. For simplicity, we focus on **LF**^{ALIVE}, **LF**^{REACHABLE $\uparrow$} , **LF**^{REACHABLE $\downarrow$} .

LF^{ALIVE} tracks left feet whose has $\diamond$ has not yet being violated. All $\square$ underneath the left foot is satisfied, although the left foot itself might not. By tracking invalidated $\diamond$ deontics, we prevent comingling of deontics from incompatible parse trees. We compute this attribute when a new terminal t_n is being committed.

LF^{ALIVE} $[row, col, cnf] = True$ if and only if any of

- it is deontic that permits the new terminal.
- it is a descendant or ancestor of a deontic that permits the new terminal.
- reachable from both top and bottom without going pass a CNF rules associated with deontic rule has $type(t_n)$ as consequent. (unconstrained version of t_n)
- is a newly formed left foot, cannot be exceeded yet

LF^{REACHABLE $\uparrow$} , **LF**^{REACHABLE $\downarrow$} is computed when probing for prospective next terminal t_n . A left foot need to be both *REACHABLE* $\uparrow$ and *REACHABLE* $\downarrow$ for it to be in one of the completable parse tree of $t_0 \dots t_n$. Both attributes can be computed by logging successful triggering of CNF rule during execution of incremental Valiant recognizer.

Permitted deontics are actualized by only allowing deontics of antecedent terminals beneath deontic left feet that are *ALIVE*, *REACHABLE* $\uparrow$ and *REACHABLE* $\downarrow$ at the same time, conditioned on matching consequent terminal type to the prospective terminal.

Necessary deontics enforcement simply happens at commit-time by preventing underlying CNF rules from finishing until the right foot is grown enough to satisfy all required regular expressions.

We further illustrate the different between a CDoT parser states and ordinary CYK parser state in figure 2.

5 EXPERIMENTS AND RESULTS

5.1 KNIGHT AND KNAVE

Knight and Knave is a logic puzzle 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. Xie et al. (2024) provides a readily usable dataset containing both textual and logical form of puzzles with 2-8 characters.

5.1.1 PREPARATION: CONJUNCTIVE NORMAL FORM AND UNIT PROP

Conjunctive Normal Form is a conjunction of *clauses*, each clause being a disjunction of *literals* (a variable or its negation). For example, $(x_1 \vee \neg x_3 \vee x_4) \wedge (\neg x_1 \vee x_3) \wedge (x_1 \vee x_5)$. An arbitrary boolean expression can be converted into equalsatisfiable conjunctive normal form efficiently via the Tseytin transformation (Tseytin, 1983) with a caveat of adding $O(n)$ auxiliary variables.

Unit Propagation is an algorithm that deduces assignments based on committed assignments of other literals. It exploits the fact that each clause in conjunctive normal form must have at least one literal to be true. Therefore, if all but one literal in a clause are false, the last remaining literal must be true. In the example above, if we assume x_1 is true, then x_3 must also be true, because it is the last remaining option to keep the second clause satisfied.

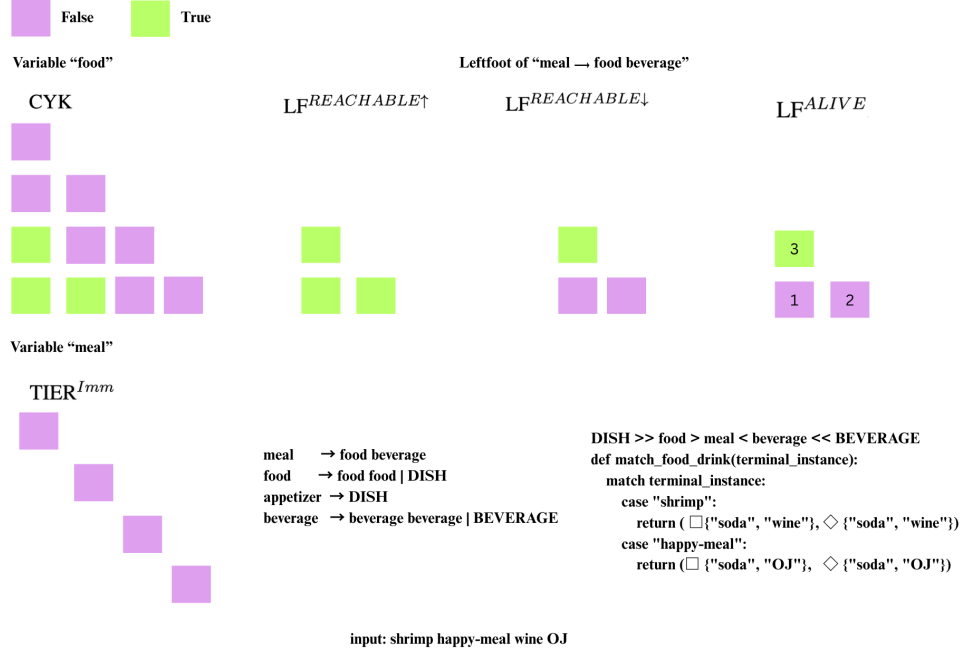


Figure 2: A CYK parser (context-free) and partial states of a CDoT parser after consuming input sequence "shrimp happy-meal wine OJ". There is no *Box* guarding *food*, therefore $\mathbf{LF}^{REACHABLE\uparrow}$ for *meal* $\rightarrow$ *food beverage* behaves the same as CYK states for *food*. Position 3 in $\mathbf{LF}^{ALIVE}$ is true because it covers both "shrimp" and "happy-meal" to permit both "wine" and "OJ". Both position 1 and 2 are cannot permit both drinks by themselves. Although the left feet of both positions are in the same tree as 3, the left feet they represent are not. Therefore 1 and 2 cannot "inherent" aliveness by reachability. $\mathbf{TIER}^{Imm}$ does not form any *meal* instances because $\square$ are not yet satisfied.

5.1.2 CONSTRAINER DESIGN

Figure 3 shows the context free grammar we employ for the output of the LLM on the Knight and Knave problem. It ensures that the model outputs a sequence of guesses. Each guess assigns roles to characters and booleans value to auxiliary variables. The grammar alone will not ensure the desired behavior, so we add the following deontic rules:

- The deontic rule for ensuring that *sufficient_assignment_seq* is recognized iff an *assignment_seq* commits to at least one literal from each clause is:

```

anchor: sufficient_assignment_seq << ASSIGNMENT
% attr: hidden\_permitted
def correct_guess():
    return ( $\square$ { "xi | ... | xj" for xi  $\vee$  ...  $\vee$  xj in input_clauses},  $\diamond$ {})

```

- The deontic rule for constraining *contradictory_assignment_seq* is defined similarly.
- The deontic rule for ensuring that a REPEATED_ASSIGNMENT is recognized iff it replicates a previous ASSIGNMENT within this guess is defined as:

```

anchor: ASSIGNMENT >> assignment_seq.semi.1 > repetition_assignment_seq <
    repeating_assignment << REPEATED_ASSIGNMENT
def repetition(terminal_instance):
    return ( $\square$ {},  $\diamond$ {terminal_instance})

```

```

270
271      start → bad_guess* good_guess
272      good_guess → sufficient_assignment_seq semi_3 GOOD_GUESS_TAIL
273      sufficient_assignment_seq → assignment_seq
274      bad_guess → contradictory_assignment_seq semi_3 CONTRADICTORY_GUESS_TAIL
275                  | repetition_assignment_seq semi_3 REPETITION_GUESS_TAIL
276      contradictory_assignment_seq → assignment_seq
277      repetition_assignment_seq → assignment_seq semi_1 repeating_assignment
278      assignment_seq semi_1 → assignment_seq semi_1
279      assignment_seq → assignment_seq semi_1 assignment
280                  | GUESS_HEAD assignment
281      assignment → unit_prop
282      ASSIGNMENT → ASSIGNMENT
283      (NAME "->" KK) → (NAME "->" KK)
284      AUX_VAR → AUX_VAR
285      REPEATING_ASSIGNMENT → REPEATING_ASSIGNMENT
286      REPEATING_ASSIGNMENT2 → ASSIGNMENT
287      semi_3 → SEMI_3 "\n"
288      semi_2 → SEMI_2 "\n"
289      semi_1 → SEMI_1 "\n"
290      SEMI_33 → ";"
291      SEMI_22 → ";"
292      SEMI_11 → ";"
293      GUESS_HEAD → "Here is a different and improved guess:  {{\n"
294      GOOD_GUESS_TAIL → "}} <<DONE>>"
295      CONTRADICTORY_GUESS_TAIL → "}} contradiction\n\n"
296      REPETITION_GUESS_TAIL → "}} Bad guess\n\n"
297      NAME → / (Jacob|Noah|Michael|Liam|Ella|...|Ava) /
298      KK → "knight" | "knave"
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323

```

Figure 3: Context free grammar for LLM generations in the Knight and Knave problem.

- Deontic rules for implementing the unit propagation algorithm are defined for each x_k in the input clauses:

- Add context-free rule “ $unit_prop \rightarrow x.k_condition \text{ semi_2 } x.k_assignment$ ”
- Add deontic rule

```

303      anchor: x.k_assignment << ASSIGNMENT
304      % attr: self\_sufficient\_necessary
305      def x.k_condition():
306          return (□{"x_k"}, ◇{})

```

- For each input clause c that contains x_k , add deontic rule

```

308      anchor: x.k_condition << ASSIGNMENT
309      % attr: hidden\_permitted
310      def x.k_assignment():
311          return (□{"NEGATE_LITERAL(x_m)" for x_m in c if m ≠ k},
312                  ◇{"x_m" if x_m exists in any clause and m ≠ k})

```

These deontic rules result in the real-time triggering of unit propagation and the real-time detection of good and bad guesses. As the LLM commits to literals, unit propagation will trigger immediately upon conditions being met, and will constrain generation to only entailed literals. Bad guesses will be detected immediately upon their first contradiction and generation will be forced to start a new guess. Good guesses will be detected immediately once the formula is solved and generation will be immediately terminated.

5.2 RESULTS

We set LLMs to solve puzzles in a 0-shot setting. Models are provided with both the textual and logical forms of the puzzle in the prompt. The results are shown in table 1. When the output is limited to 512 tokens, constrained models achieve about a 50-point higher solve rates than non-

Model	Constrained		Unconstrained	
	<i>Thinking</i>	<i>Non-thinking</i>	<i>Thinking</i>	
	512	512	512	2048
Llama-3.2-1B-Instruct	0.52	0.04	0.04	OOM
Llama-3.2-3B-Instruct	0.65	0.13	0.13	OOM
Qwen/Qwen3-0.6B	0.61	0.07	0.07	0.36
Qwen/Qwen3-1.7B	0.69	0.03	0.03	0.55

Table 1: Knight and Knave test: solve rate of constrained vs. unconstrained models with 512 or 2048 token limits. Llama-3 models do not have a thinking switch, but do exhibit chain of thought-like behavior. Xie et al. (2024) reported 0.14 for Llama-3-8B-instruct in the same setting.

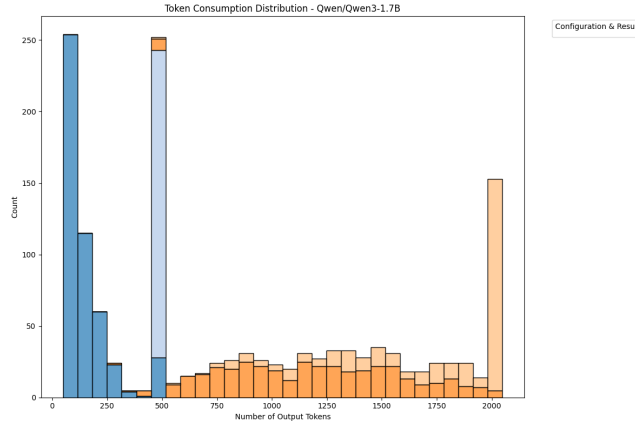


Figure 4: Knight and Knave test: token efficiency of successes (dark) vs failures (light) of constrained (blue) vs. unconstrained (orange) models. Constrained models were cut off at 512 tokens, unconstrained at 2048 tokens.

constrained models, and even outperform non-constrained models that are allowed to generate 4 times more tokens.

We also plot the number of output tokens for successful (dark) and failed (light) LLM responses across constrained (blue) and unconstrained (orange) Qwen3-1.7B, shown in fig. 4. The constrained model shows a skewed distribution, where a larger portion of puzzles are solved with lesser tokens, and failures don’t show up until around 512 tokens. (Qualitatively, we observe that at this point, the constrained model gets stuck in loops.) The non-constrained model forms a flat bell curve distribution, and requires many more output tokens before starting to solve puzzles.

We also observe that the constrained model can often solve puzzle on the first attempt (fig. 5). 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. If the LLM alone was driving success, we would expect to see the non-constrained model having a similar number of solutions on the first attempt.

When we break down the solve rate by the size of the puzzle in fig. 6, we observe that the constrained models scale much better and still achieve solve rate of around 0.50, in contrast to under 0.20 for non-constrained models.

5.3 TEXT-TO-SQL

Spider (Yu et al., 2019) is a text-to-sql task, the model is provided with a database schema and a natural text question, and it is expected to answer the question with an generated SQL query.

We finetune an LLM with the constrainer on during both finetuning and inferencing. For the constrainer, we use a grammar extracted from the SQLite source code as the base context-free grammar, then add constraints that allow only identifiers that come from databases containing the columns in

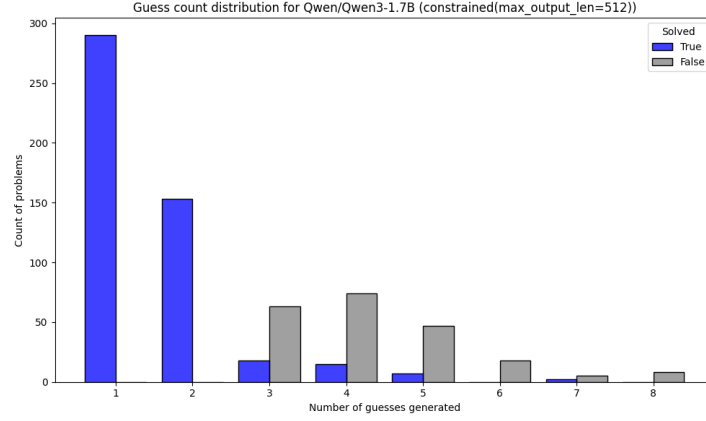


Figure 5: Knight and Knave test: constrained models solve many problems in the first guess, while unconstrained models require more guesses.

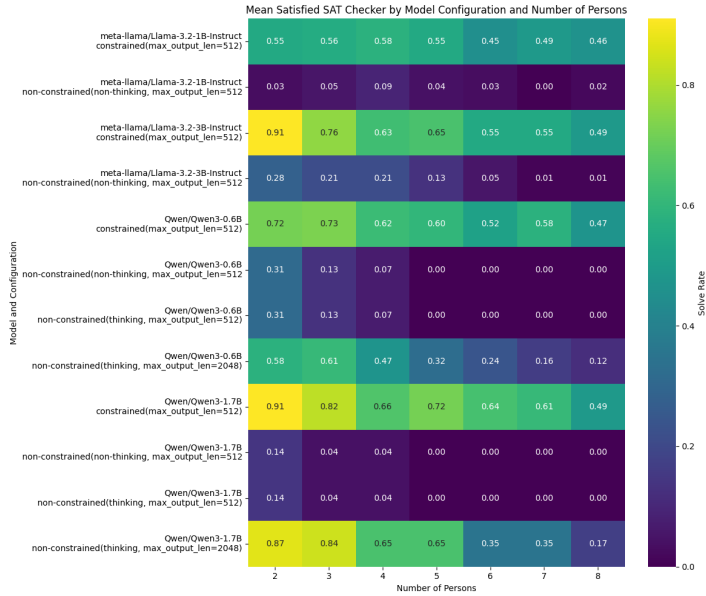


Figure 6: Knight and Knave test: Constrained models outperform unconstrained models across all puzzle sizes.

the “SELECT” clause. We also run experiments with the additional constraint of limiting identifier in the “SELECT” clause to the scope of all databases. Formally:

```

421 anchor: ID >> select_clause > select < from_clause << ID
422 def allow_containing_database(terminal_instance):
423     return (□{}, ◇{name if name is an identifier of a database containing terminal_instance})
424
425 anchor: select_clause << ID
426 def correct_guess():
427     return (□{}, ◇{name if is a table or column of any database })

```

For both constrained and unconstrained models, we remove tokens from the tokenizer that do not align with SQL grammar terminals, such as “.Name” and “);”. LLM performance is known to suffer when such tokens are not handled by constrainer (Beurer-Kellner et al., 2024). For the unconstrained model, we add the target database schema to the model input, so the model has the same information on the target database that is available to the constrained model.

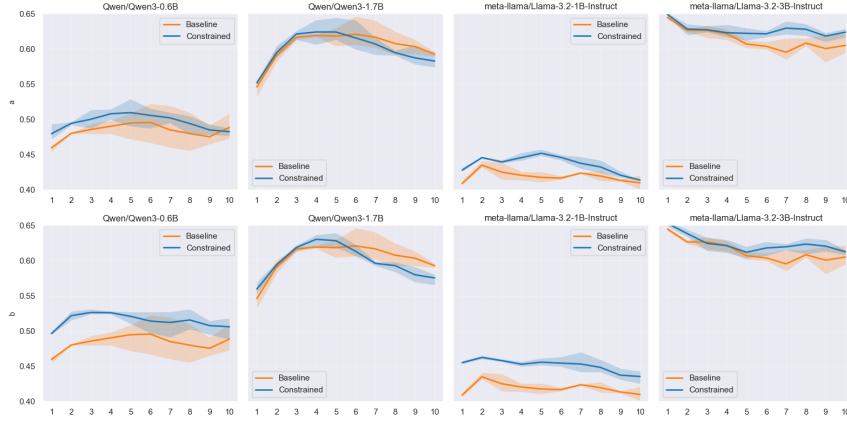


Figure 7: Text-to-SQL test: Execution accuracy of constrained vs non-constrained models over 3 runs and 10 epochs. Row (a) uses selected columns in “SELECT” clause to limit the remaining identifiers to that of databases containing them. Row (b) additionally constraints the “SELECT” clause to identifiers of all databases. Constrained decoding helps three of the four models.

The results are shown in fig. 7. “Qwen3-0.7B”, “Llama-3.2-1B-Instruct”, and “Llama-3.2-3B-Instruct” benefit from the constrainer with either constraint configuration. However, “Qwen3-1.7B” does not benefit from constrainer.

6 CONCLUSION

We introduced a new context-sensitive grammar formalism, conditional deontics over terminals (CDoT), designed for use with constrained decoding, and a new $O(n^3)$ incremental CYK-like parser for the grammar. This constrained decoding framework is strong enough to implement the unit propagation algorithm, which we demonstrate on “Knight and Knave” logical puzzles, finding that our framework dramatically improves LLM accuracy on this dataset while simultaneously reducing the number of tokens the LLMs generate. We also apply our framework to constrain SQL table column names in a text-to-SQL task and find improved accuracy for most LLMs tested.

REFERENCES

- 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>.
- Luca Beurer-Kellner, Marc Fischer, and Martin Vechev. Guiding llms the right way: Fast, non-invasive constrained generation, 2024. URL <https://arxiv.org/abs/2403.06988>.
- Yixin Dong, Charlie F. Ruan, Yaxing Cai, Ruihang Lai, Ziyi Xu, Yilong Zhao, and Tianqi Chen. Xgrammar: Flexible and efficient structured generation engine for large language models, 2025. URL <https://arxiv.org/abs/2411.15100>.
- Aravind K. Joshi and Yves Schabes. *Tree-Adjoining Grammars*, pp. 69–123. Springer Berlin Heidelberg, Berlin, Heidelberg, 1997. ISBN 978-3-642-59126-6. doi: 10.1007/978-3-642-59126-6_2. URL https://doi.org/10.1007/978-3-642-59126-6_2.
- João Loula, Benjamin LeBrun, Li Du, Ben Lipkin, Clemente Pasti, Gabriel Grand, Tianyu Liu, Yahya Emara, Marjorie Freedman, Jason Eisner, Ryan Cotterell, Vikash Mansinghka, Alexander K. Lew, Tim Vieira, and Timothy J. O’Donnell. Syntactic and semantic control of large language models via sequential monte carlo, 2025. URL <https://arxiv.org/abs/2504.13139>.

- Daniel Melcer, Nathan Fulton, Sanjay Krishna Gouda, and Haifeng Qian. Constrained decoding for fill-in-the-middle code language models via efficient left and right quotienting of context-sensitive grammars, 2024. URL <https://arxiv.org/abs/2402.17988>.
- 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>.
- Alexander Okhotin. Conjunctive grammars. *J. Autom. Lang. Comb.*, 6(4):519–535, April 2001. ISSN 1430-189X.
- Gabriel Poesia, Oleksandr Polozov, Vu Le, Ashish Tiwari, Gustavo Soares, Christopher Meek, and Sumit Gulwani. Synchromesh: Reliable code generation from pre-trained language models. *arXiv preprint arXiv:2201.11227*, 2022.
- 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/>.
- G. S. Tseitin. *On the Complexity of Derivation in Propositional Calculus*, pp. 466–483. Springer Berlin Heidelberg, Berlin, Heidelberg, 1983. ISBN 978-3-642-81955-1. doi: 10.1007/978-3-642-81955-1_28. URL https://doi.org/10.1007/978-3-642-81955-1_28.
- 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>.
- K Vijay-Shanker and David J. Weir. Parsing some constrained grammar formalisms. *Computational Linguistics*, 19(4):591–636, 1993. URL <https://aclanthology.org/J93-4002/>.
- 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. 2024. URL <https://arxiv.org/abs/2410.23123>.
- 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, 2019. URL <https://arxiv.org/abs/1809.08887>.

A APPENDIX

A.1 ADDITIONAL DETAILS OF KNIGHT AND KNAVE CONSTRAINER DESIGN

In the grammar for the Knight and Knave Constrainer Design, any special assignment sequence such as *x.k_condition* is also acceptable as a plain *assignment_seq*. This undermines the new assignment we intend to force through our deontic rules. So we introduce the same delimiter at different priorities, defined as *semi.p* $\rightarrow$ SEMI_P NEWLINE in the grammar. Two terminals are needed to handle parser/LLM interfacing technicalities.

When a special assignment sequence is not complete, the following higher priority delimiter terminal will not be acceptable, consequently, the remaining part of special rule is not activated. On the other hand, when the conditions of a special assignment are met, the delimiter would be recognized as the higher priority version, and therefore deny the more general version of *assignment_seq*.