Conditional Deontics over Terminals: A Mildly Context-Sensitive Formal Grammar for Constrained Decoding

Anonymous ACL submission

Abstract

With the broad adoption of large language models (LLMs), techniques to guide their generation have become increasingly important, with particularly sophisticated patterns found in constrained decoding frameworks expressed via formal grammars. Recent frameworks such as IterGen expand beyond context-free grammars to achieve some context sensitivity but with drawbacks: expensive backtracking or speculative lookahead, permissive semantics that risk spurious constraint activation, and difficult integration with runtime optimizations like XGrammar. To address these concerns, we propose Conditional Deontics over Terminals (CDoT), a mildly context-sensitive grammar paired with a GPU-accelerated $O(n^3)$ parser that can be employed for constrained decoding with LLMs. We show that CDoT can express useful constraints that improve LLM performance in a traditional text-to-SQL task and a complex logic puzzle involving unit propagation.

1 Introduction

As large language models (LLMs) have become more popular, the importance of techniques to guide their generation has also grown. Techniques for steering LLMs target various model states, including activations, attention patterns, MoE router logits, and output logits. These interventions can be applied uniformly, or conditioned on the generation process itself. For example, CAST (Lee et al., 2025) conditions on the cosine similarity between activations and behavior vectors, FASB (Cheng et al., 2025) applies a neural classifier to activations to make steering decisions, and PASTA (Zhang et al., 2024) conditions on user-selected input spans. In such simple conditioning on the generation process, these methods depend on only a fixed window of past steps or inputs.

Constrained decoding is unique among steering methods in that it explicitly models long-range and structurally complex patterns – such as programming language specifications, molecular structures, and privacy requirements (Ugare et al., 2025; Loula et al., 2025; Scholak et al., 2021; Poesia et al., 2022) – that cannot be captured by finite-window or input-fixed conditions. Formal grammars are central to this power.

Our contributions to this research direction are:

- We increase the expressive power of constrained decoders by designing a new formal grammar, Conditional Deontics over Terminals (CDoT), that is strictly more powerful than context-free grammars.
- We demonstrate that constrained decoding with LLMs under this new grammar formalism yields improvements on both a traditional text-to-SQL dataset and on a complex logic puzzle involving unit propagation.

2 Design Statement

2.1 Formal Grammars

Formal grammars operate on any finite sequence where the elements are drawn from a finite set (e.g., text, molecules, or even neural network modules). While our experiments apply our CDoT grammar formalism to constrained decoding with LLMs, we will define it with sufficient abstraction that it could be used on sequences other than text. We lean on two alternative interpretations of formal grammars:

Grammar the proto programming language: Computation theory studies grammar as a fundamental model of computation with the capacity to reach Turing-completeness—meaning any algorithm can be implemented, given an appropriate formal grammar. Following this intuition, we design the "knight-and-knave" experiment, where CDoT is used to implement the unit propagation algorithm to constrain the LLM generation.

Grammar the deduction system: Proposed by Pereira and Warren (1983), and used in PEG (Ford, 2004) and conjunctive grammars (Okhotin, 2001a; Mrykhin and Okhotin, 2023; Barash and Okhotin, 2014; Okhotin, 2013), under this interpretation, instead of saying a sequence $t_0 \dots t_n$ can be parsed into or derived from non-terminal X, we say $t_0 \dots t_n$ has property X. Consequently, grammar rule $X \rightarrow A\ B$ means a sequence of property X can be obtained by concatenating sequences of property A then B. This interpretation inspires us to blend grammar with deontic logic (von Wright, 1951), from which we inherit the name and the symbols ($\Diamond, \Box$). We will also depend on this interpretation to define formal semantics for CDoT.

2.2 Reach for Context-Sensitiveness

Classical constrained decoding use cases, such as JSON schema enforcement, are based on context-free grammars, implying a tree-shaped latent structure. Context-sensitive grammars additionally allow references across subtrees, for example, ensuring that variable references in generated code refer back to valid variable definitions.

Figure 1 illustrates the challenge of handling context-sensitiveness during LLM generation. Assume that we wish to enforce a constraint where the `SYMPTOM` of a `diagnosis` constrains which `CONDITION`s are allowable. Note that constrained decoding is an incremental process, so the ancestor nodes are unavailable at the time a constraint must be applied. Ignoring ancestor structure and using only precedence, as in PiCARD (Scholak et al., 2021) and partially inherited by IterGen (Ugare et al., 2025), will result in the constraint firing incorrectly for the bottom tree in figure 1. An alternative adopted by IterGen (Ugare et al., 2025) and GenLM Control (Loula et al., 2025) is to allow ancestor structures to be considered by invoking expensive backtracking and/or speculative lookahead, where ancestor structures are allowed to emerge before constraints are applied. ChopChop (Nagy et al., 2025) employes more powerful formalism to reason over partial syntax trees, but still cannot produce a complete logit mask and relies on backtracking incorrect token instead. There have also been specialized solutions for particular context-sensitive constraints, such as indentation in Python syntax (Melcer et al., 2024).

We will instead design a cross reference mechanism that can be resolved even when ancestor structure is non-determinant.

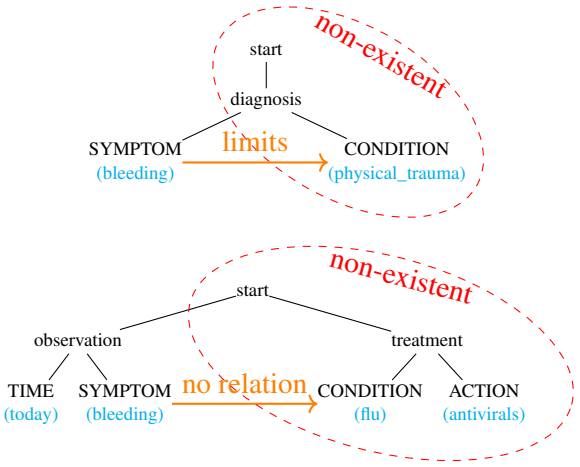

Figure 1: Example of challenges in enforcing a constraint where the `SYMPTOM` of a `diagnosis` constrains which `CONDITION`s are allowable. Because constrained decoding is incremental, the ancestors of `CONDITION` are not known when it is being generated, yet the constraint must fire in the top tree but not the bottom.

2.3 Efficiency

Despite a diverse selection of mildly context-sensitive grammars (Joshi and Schabes, 1997; Vijay-Shanker and Weir, 1993; Okhotin, 2001b; Mrykhin and Okhotin, 2023; Barash and Okhotin, 2014), such grammars are often not designed with the goal of efficient incremental parsing 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 and 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. We design our formalism with the goal of parsing at $O(n^2)$ step with GPU acceleration, inline with general context-free grammars (Azimov and Grigorev, 2017).

2.4 The Solution

In a chart-based recognizer like Valiant(Valiant, 1975), given a subtree root, it is easy to find both the terminals in this subtree and the immediate children of the subtree root. Therefore, we design our cross-subtree references to have the following form: Given a subtree root, terminals under one of its children (consequent) are conditioned on terminals under a child to the left (antecedent). To provide further expressiveness, we deviate from the traditional practice of treating terminals as opaque objects, instead allowing predicates to be placed

on the consequent terminals. Additionally, Azimov and Grigorev (2017) shows that Valiant recognizer can be GPU accelerated and outperform the high performance general context-free parser GLL.

3 CDoT Example

Before formally defining CDoT, we provide some intuition with a concrete example of the formalism.

CDoT extends context free grammars (CFGs; formally defined in appendix A) 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 $(\psi^{\square}, \psi^{\diamond})$, where $\psi^{\square}$ is a set of regular expressions (necessary conditions) and $\psi^{\diamond}$ is a set of regular expressions (permitted conditions).

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

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

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

Example (Balanced Meals). Given the grammar:

meal → *food beverage*

food → `DISH` *food* | `DISH`

beverage → `BEVERAGE` *beverage* | `BEVERAGE`.

The following deontic rule enforces that every meal includes grains, meat, and vegetables (□), while also allowing optional extras like `soup` or `steamed_egg` (◇).

```
anchor: food ≪ DISH
def enforce_meal():
    return ({rice|bread, chicken|fish,
             salad|roasted_vegetable}□,
            {rice, bread, chicken, fish,
             salad, roasted_vegetable,
             steamed_egg, soup}◇)
```

Conditional deontic rules. A conditional rule has the form

$$t^{ant} \gg X^{ant} B \dots D > X^{par} < X^{con} \ll t^{con}$$

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

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 ≫ food > meal < beverage ≪ BEVERAGE
def drink2food(terminal_instance):
    match terminal_instance.text:
        case 'steak':
            return {red_wine}□, {}◇
        case 'fish':
            return {white_wine}□, {}◇
        case 'spicy_noodle':
            return {soda}□, {}◇
```

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 □ and disjunctive nature of ◇ will still naturally combine so that, if both fish and steak are ordered, both red and white wine are requested.

4 Formal Semantics of CDoT

4.1 Basic Notations

Terminal Instance (t_j) is a subspan of the generation process. Each terminal instance is defined by a sequential index (j), a terminal type, and a user-defined payload. In the case of constrained decoding, a terminal instance might have type "VAR_NAME" and payload of token sequence ["start", "_", "time"].

Type-based Filtering ($[t_i \dots t_j]_b$) extracts the set of all terminal instances within the span that have terminal type b in sequence $t_i \dots t_j$. Formally,

$$[t_i \dots t_j]_b \triangleq \{t_k \mid i \le k \le j \wedge \text{type}(t_k) = b\}$$

Predicate (ρ) We write $\rho(t_i)$ to indicate that the single terminal instance t_i satisfies property ρ. For example, regular expressions are good predicate for constrained decoding (i.e. a variable name must start with a certain prefix). Alternatively, a neural probe or other heuristics might also be predicates.

Predicate Verifier A predicate verifier is needed to confirm a terminal instance satisfies the predicate, for example, matching regular expressions or comparing to a threshold value.

Predicate Enforcer A predicate enforcer produces a new terminal instance that satisfies the predicate, for example, running a regular expression constrainer or cutting off values. An all-permitting enforcer is acceptable and appropriate if permitted is set to universal (e.g., when using a neural probe predicate). Enforcers need to be mergable, with the merged enforcer producing only t_i s.t. $\rho_1(t_i) \vee \rho_2(t_i)$. Enforcers needs to be at least sound, if completeness is not achievable.

4.2 Terminal Specification

CDoT deviates from traditional formal grammars and allows grammar rules to specify properties to terminal instances under their scope.

Terminal Specification (ψ) is a set of predicates. Terminal $t_i \dots t_j$ satisfies a specification ψ is written as $t_i \dots t_j \in \psi$. Notice that the $\in$ operator is overloaded as specification is a set of predicates.

Necessary Specification ($\psi^{\square}$) is the conceptual "lower bound". Each predicate in a necessary specification must be satisfied by at least one consequent terminal instance, formally

$$t_i \dots t_j \in \psi^{\square} \iff \forall \rho \in \psi^{\square} : \exists k \in [i, j] : \rho(t_k)$$

Permitted Specification ($\psi^{\diamond}$) is the conceptual "upper bound". Each consequent terminal instance must satisfy at least one predicate in the permitted specification, formally

$$t_i \dots t_j \in \psi^{\diamond} \iff \forall k \in [i, j] : \exists \rho \in \psi^{\diamond} : \rho(t_k)$$

Obligation Implies Can we stipulate that every predicate in a necessary specification is automatically added to the matching permitted specification.

Aggregating Specifications is achieved by the union $\psi_A \cup \psi_B$. Intuitively, the necessary specification tightens (more predicates to be satisfied), and the permitted specification loosens (more predicates to choose from).

4.3 Deontic Function

A *deontic funtion* maps a single terminal instance to a token specification. This serves a similar role as functions in recursive descent parsers or IterGen. Invoking unrestricted functions during parsing increases both expressiveness and convenience, allowing, for example, a query to an external tool or a recursive LLM call during generation.

We define the function to take a single terminal instance, rather than all antecedent instances to reduce the number of deontic function calls from $O(n^2)$ to $O(n)$. We compute each terminal's deontic result once upon completion, then discard the terminal instance – useful for large payloads such as LLM hidden states. Due to the aggregation behavior of necessary and permitted semantics described in section 4.2, taking a single terminal instance often gets the same result specification.

4.3.1 Notations of deontic function

Concretely, we write a deontic function as a python function that takes a terminal instance and returns a pair of necessary and permitted specifications as a tuple. Given a deontic function, we write $\square t_i$ or $\diamond t_i$ to indicate the necessary or permitted specification obtained by applying such deontic function to t_i. We introduce shorthands:

$$\square(t_i \dots t_j) \triangleq \square(t_i) \cup \square(t_{i+1}) \cup \dots \cup \square(t_j)$$
$$\diamond(t_i \dots t_j) \triangleq \diamond(t_i) \cup \diamond(t_{i+1}) \cup \dots \cup \diamond(t_j)$$

4.4 The Inference Rules

Terminal Instance Partitions (F, P, R) Under a CDoT subtree, each terminal instance belongs to one of three partitions. F are free terminal instances not under deontic constraints (e.g., keyword $SELECT$ in a SQL query is likely unconstrained). P are the permitted terminal instances under deontic constraints and receive permission ($\diamond$) from the current subtree. R are the remaining terminal instances under deontic constraints, but no permission received from the current subtree (e.g., an invocation of variable from outer scope).

Assertion ($A(t_i \dots t_j \mid \langle R, F \rangle)$) is read as $t_i \dots t_j$ has property A, conditioned on the terminal instances in $R \subseteq t_i \dots t_j$ receiving permission from ancestors. We say $t_0 \dots t_n$ is in the language of grammar if and only if $START(t_0 \dots t_n \mid \varnothing)$.

4.4.1 Context-Free Rules

We rewrite context-free rules into Chomsky normal form to include the deontic condition.

Terminal Rule $X \to a$ generates a single terminal from a non-terminal. Because it is not under deontic constraint yet, t_i belongs to partition F.

$$\frac{\text{type}(t_i) = a}{X(t_i \mid \langle \emptyset, \{t_i\} \rangle)} \quad \text{(Terminal)}$$

Context-free Rule $X \to A\ B$ simply aggregates and passes on the deontic condition.

$$\frac{A(t_i \dots t_j \mid \langle R_A, F_A \rangle) \quad B(t_{j+1} \dots t_k \mid \langle R_B, F_B \rangle)}{X(t_i \dots t_k \mid \langle R_A \cup R_B, F_A \cup F_B \rangle)} \quad \text{(Context-free)}$$

Since neither rule is conditioned on deontic premises nor adds terminals to the R partition, CDoT behaves identically to a context-free grammar in the absence of deontic rules.

4.4.2 Strict Permitted Rule

The crux of the problem is defining behavior when deontic rules are nested. We begin with a naive but useful semantics termed "strict permitted."

Strict-$\Diamond$ rule requires the consequent to strictly satisfy the terminal specification returned by the antecedent's deontic function, disregarding any constraints from ancestor rules or rules nested within the consequent. As a result, all terminals of consequent type (b) under the consequent non-terminal (B) belong to partition P for parent X, and therefore do not appear in F_x or R_x, as shown in the inference rule.

$$\frac{\begin{array}{c} A(t_i \dots t_j \mid \langle R_A, F_A \rangle)\ B(t_{j+1} \dots t_k \mid \langle R_B, F_B \rangle) \\ [t_{j+1} \dots t_k]_b \in \Box([t_i \dots t_j]_a) \\ [t_{j+1} \dots t_k]_b \in \Diamond([t_i \dots t_j]_a) \end{array}}{\begin{array}{c} X(t_i \dots t_k \mid \langle R_X, F_X \rangle) \\ R_X = (R_A \cup R_B) \setminus [t_{j+1} \dots t_k]_b \\ F_X = (F_A \cup F_B) \setminus [t_{j+1} \dots t_k]_b \end{array}} \quad \text{(Deontic - Strict } \Diamond\text{)}$$

Upon closer examination, one may realize that in a system with only context-free and strict-$\Diamond$ again keeps $R = \varnothing$ and never conditions on $\langle R, F \rangle$. Thus, a CYK-like parser can be easily implemented if the entire terminal instance sequence is available.

However, decidability issues arise when we seek to parse incrementally while maintaining the valid-prefix property. Because each consequent terminal instance can further serve as an antecedent in descendant subtrees—used as inputs to another deontic function—the parser must determine whether future terminals could yield a terminal specification incompatible with the current one.

4.4.3 Union Permitted Rule

Beside tractability, the strict-$\Diamond$ semantic sometimes also does not capture desired expressions when we need to aggregate specifications from multiple antecedent subtrees, for example, variables from different scopes.

Union-$\Diamond$ rule While keeping the necessary($\Box$) behavior unchanged, we allow subtrees to put terminal instances under constraint but defer permission giving to ancestors. This is reflected in removal of $\Diamond$ condition in the premise and updated R_X definition. Notice that we are overloading the $\setminus$ operator between terminal instance collection and specification, $(R_B \cup F_B) \setminus \Diamond[t_i \dots t_j]_a \triangleq \{t \mid t \in [R_B \cup F_B]_b \wedge t \notin \Diamond[t_i \dots t_j]_a\}$.

$$\frac{\begin{array}{c} A(t_i \dots t_j \mid \langle R_A, F_A \rangle)\ B(t_{j+1} \dots t_k \mid \langle R_B, F_B \rangle) \\ [t_{j+1} \dots t_k]_b \in \Box([t_i \dots t_j]_a) \end{array}}{\begin{array}{c} X(t_i \dots t_k \mid \langle R_X, F_X \rangle) \\ R_X = R_A \cup (R_B \cup F_B \setminus \Diamond[t_i \dots t_j]_a) \\ F_X = (F_A \cup F_B) \setminus [t_{j+1} \dots t_k]_b \end{array}} \quad \text{(Deontic - Union } \Diamond\text{)}$$

5 Parser

Building on past work on GPU accelerated parsers, we construct our parser using two waves of extended Incremental Valiant Recognizers. The detail of parser is shown in appendix C.

Parser Behavior The constrainer is complete, meaning is would not falsely deny a valid token.

However, due to the decidability issue discussed in section 4.4.2, we only achieve eventual soundness: a strict-$\Diamond$ deontic rule instance is pruned only upon its first direct violation. At that point, if the grammar does not provide an alternative path, generation terminates. Our Knight-and-Knave grammar (figure 7) demonstrates how to provide such a fallback. This pattern commonly arises when constraining a thinking trace, where deontic rules are used for early termination of an attempt.

Furthermore, 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 and looser than

specified constraint. We demonstrate this behavior in appendix B.

Efficiency By preserving the loop structure of incremental Valiant recognizer and only modify the constant-time loop body, we achieve $O(n^2)$ step complexity. Our current implementation is CPU bound and the token-to-token time is similar to HuggingFace implementation of Llama3 3B models. However, an efficient implementation of incremental Valiant recognizer steps at around 100 µs with $n = 100$ and $|G^{CNF}| = 3000$ using a single stream multiprocessor (out of > 100) on a Nvidia H100 GPU.

6 Experiments and Results

6.1 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.

6.1.1 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 (Tseitin, 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.

6.1.2 Constrainer Design

Figure 7 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 boolean values to auxiliary variables. The grammar alone will not ensure the desired behavior, so we add the following deontic rules.

To ensure that *sufficient_assignment_seq* is recognized iff an $assignment\_seq$ commits to at least one literal from each clause is, we add the deontic rule:

```
anchor: sufficient_assignment_seq ≪ ASSIGNMENT
% attr: hidden\_permitted
def correct_guess():
    return (□{"x_i|...|x_j"
                for x_i ∨ ··· ∨ x_j
                in input_clauses},
            ◇{})
```

We add a similar rule for constraining *contradictory_assignment_seq*.

To ensure that a REPEATED_ASSIGNMENT is recognized iff it replicates a previous ASSIGNMENT within this guess, we add the deontic rule:

```
anchor: ASSIGNMENT >> assignment_seq_semi_1
        > repetition_assignment_seq
        < repeating_assignment
        ≪ REPEATED_ASSIGNMENT
def repetition(terminal):
    return (□{}, ◇{terminal})
```

Deontic rules for implementing the unit propogation algorithm are defined for each x_k in the input clauses. We add context-free rules of the form:

$$\textit{unit\_prop} \rightarrow \textit{x\_k\_condition semi\_2}\ \textit{x\_k\_assignment}$$

We add deontic rules of the form:

```
anchor: x_k_assignment ≪ ASSIGNMENT
% attr: self_sufficient_necessary
def x_k_condition():
    return (□{"x_k"}, ◇{})
```

For each input clause c that contains x_k, we add a deontic rule:

```
anchor: x_k_condition ≪ ASSIGNMENT
% attr: hidden_permitted
def x_k_assignment():
    return (□{"NEGATE_LITERAL(x_m)"
                for x_m in c if m ≠ k},
            ◇{"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

Model	Constrained	Unconstrained		
	Thinking	*Non-thinking*	*Thinking*	
	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 does exhibit chain of thought-like behavior. Xie et al. (2024) reported 0.14 for Llama-3-8B-instruct in the same setting.

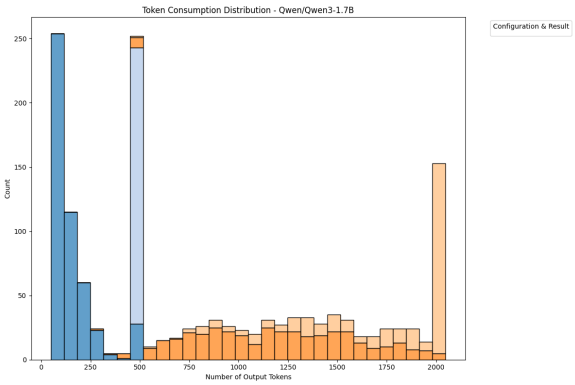

Figure 2: Knight and Knave test: token efficiency of successes (dark) vs failures (light) of constrained (blue) vs. unconstrained (orange) models. Constrained models were cut off at 512 tokens, unconstrained at 2048 tokens.

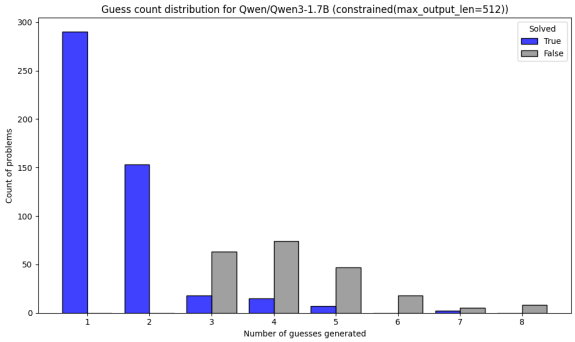

Figure 3: Knight and Knave test: constrained models solve many problems in the first guess. Notice that the solve rate at first guess is significantly higher than random guess success rate, indicating organic LLM/Constrainer collaboration instead of constrainer dominance.

guess. Good guesses will be detected immediately once the formula is solved and generation will be immediately terminated.

6.1.3 Baselines

We compare against only unconstrained models for the lack of alternatives. PiCARD is hard-coded to SQL grammar. IterGen can only backtrack an incorrect generation, while our constraints proactively forces an assignment once it is fixed. Furthermore, it is unclear how many assignments should be backtracked once conflict arises. ChopChop has similar backtracking issue and is not publicly released.

6.1.4 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-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 figure 2. 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 observe that the constrained model can often solve the puzzle on the first attempt (figure 3). 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.

When we break down the solve rate by the size of the puzzle as in figure 6 in appendix D, 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.

6.2 Text-to-SQL

Spider (Yu et al., 2019) is a text-to-sql task, where 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 gram-

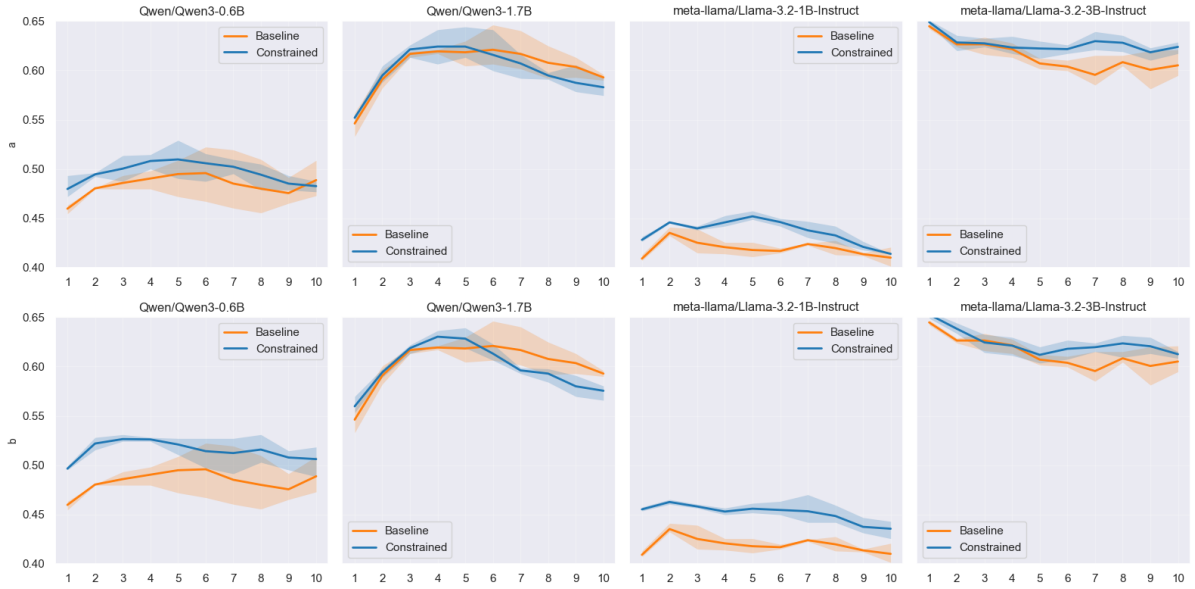

Figure 4: Text-to-SQL test: Execution accuracy of constrained vs non-constrained models over 3 runs and 10 epochs. Row (a) uses selected columns in "SELECT" clause to limit the remaining identifiers to that of databases containing them. Row (b) additionally constraints the "SELECT" clause to identifiers of all databases. Constrained decoding helps three of the four models.

mar, then add constraints that allow only identifiers that come from databases containing the columns in the "SELECT" clause. We also run experiments with the additional constraint of limiting identifiers in the "SELECT" clause to the scope of all databases. Formally:

```
anchor: ID ≫ select_clause>select<from_clause ≪ ID
def allow_containing_database(terminal):
    return (□{},
            ◇{name if name is an
              identifier of a database
              containing terminal})


anchor: select_clause ≪ ID
def correct_guess():
    return (□{},
            ◇{name if name 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 the 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.

The results are shown in figure 4. "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 the constrainer.

Baseline is again restricted to unconstrained models. PiCARD was implemented for T5 and is now defunct. Backtracking systems such as IterGen cannot be used during finetuning for its inability to produce logit mask, while our system requires finetuning in SQL generation due to token/terminal alignment issue mentioned above. For reference, IterGen reports improvement from 25.5 to 30.9 in accuracy in a non-finetuned Llama-3.2-1B model, while our finetuned model shows similar delta but at a higher starting point in the 40s.

7 Conclusion

We exploit chart-based parsing to design a novel cross-subtree reference mechanism that achieves mild context-sensitivity. By allowing predicates to be attached to terminals, we expand the expressiveness. Attaching deontic functions to grammar rules improves both expressiveness and usability. Our parser's efficiency builds on prior work in GPU-accelerated parsing.

We use our formalism to implement the unit propagation algorithm, guiding reasoning traces on Knight-and-Knave puzzles and achieve significant performance gain at reduced token budget. Our constrainer also proves helpful in text-to-SQL.

Limitations

Although our constrainer is efficient with GPU acceleration, there are cases where it fails to reject incorrect tokens, resulting in constraints that are looser than specified. Future work could identify restrictions on the underlying grammar or deontic functions that guarantee tightness of constraints. Additionally, our constrainer has difficulty handling tokens that span multiple terminals, limiting its usefulness in code generation tasks.

References

Rustam Azimov and Semyon V. Grigorev. 2017. Graph parsing by matrix multiplication. *CoRR*, abs/1707.01007.

Mikhail Barash and Alexander Okhotin. 2014. An extension of context-free grammars with one-sided context specifications. *Information and Computation*, 237:268–293.

Luca Beurer-Kellner, Marc Fischer, and Martin Vechev. 2024. Guiding llms the right way: Fast, non-invasive constrained generation. *Preprint*, arXiv:2403.06988.

Zifeng Cheng, Jinwei Gan, Zhiwei Jiang, Cong Wang, Yafeng Yin, Xiang Luo, Yuchen Fu, and Qing Gu. 2025. Steering when necessary: Flexible steering large language models with backtracking. *Preprint*, arXiv:2508.17621.

Bryan Ford. 2004. Parsing expression grammars: a recognition-based syntactic foundation. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '04, page 111–122, New York, NY, USA. Association for Computing Machinery.

Aravind K. Joshi and Yves Schabes. 1997. *Tree-Adjoining Grammars*, pages 69–123. Springer Berlin Heidelberg, Berlin, Heidelberg.

Bruce W. Lee, Inkit Padhi, Karthikeyan Natesan Ramamurthy, Erik Miehling, Pierre Dognin, Manish Nagireddy, and Amit Dhurandhar. 2025. Programming refusal with conditional activation steering. *Preprint*, arXiv:2409.05907.

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. 2025. Syntactic and semantic control of large language models via sequential monte carlo. *Preprint*, arXiv:2504.13139.

Daniel Melcer, Nathan Fulton, Sanjay Krishna Gouda, and Haifeng Qian. 2024. Constrained decoding for fill-in-the-middle code language models via efficient left and right quotienting of context-sensitive grammars. *Preprint*, arXiv:2402.17988.

Mikhail Mrykhin and Alexander Okhotin. 2023. The hardest language for grammars with context operators. *Theoretical Computer Science*, 958:113829.

Shaan Nagy, Timothy Zhou, Nadia Polikarpova, and Loris D'Antoni. 2025. Chopchop: a programmable framework for semantically constraining the output of language models. *Preprint*, arXiv:2509.00360.

Alexander Okhotin. 2001a. Conjunctive grammars. *Journal of Automata, Languages and Combinatorics*, 6(4):519–535.

Alexander Okhotin. 2001b. Conjunctive grammars. *J. Autom. Lang. Comb.*, 6(4):519–535.

Alexander Okhotin. 2013. Conjunctive and boolean grammars: The true general case of the context-free grammars. *Computer Science Review*, 9:27–59.

Fernando C. N. Pereira and David H. D. Warren. 1983. Parsing as deduction. In *21st Annual Meeting of the Association for Computational Linguistics*, pages 137–144, Cambridge, Massachusetts, USA. Association for Computational Linguistics.

Gabriel Poesia, Oleksandr Polozov, Vu Le, Ashish Tiwari, Gustavo Soares, Christopher Meek, and Sumit Gulwani. 2022. Synchromesh: Reliable code generation from pre-trained language models. *arXiv preprint arXiv:2201.11227*.

Yves Schabes. 1991. The valid prefix property and left to right parsing of Tree-Adjoining Grammar. In *Proceedings of the Second International Workshop on Parsing Technologies*, pages 21–30, Cancun, Mexico. Association for Computational Linguistics.

Torsten Scholak, Nathan Schucher, and Dzmitry Bahdanau. 2021. PICARD: Parsing incrementally for constrained auto-regressive decoding from language models. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 9895–9901, Online and Punta Cana, Dominican Republic. Association for Computational Linguistics.

G. S. Tseitin. 1983. *On the Complexity of Derivation in Propositional Calculus*, pages 466–483. Springer Berlin Heidelberg, Berlin, Heidelberg.

Shubham Ugare, Rohan Gumaste, Tarun Suresh, Gagandeep Singh, and Sasa Misailovic. 2025. Itergen: Iterative semantic-aware structured LLM generation with backtracking. In *The Thirteenth International Conference on Learning Representations*.

Leslie G. Valiant. 1975. General context-free recognition in less than cubic time. *J. Comput. Syst. Sci.*, 10:308–315.

K Vijay-Shanker and David J. Weir. 1993. Parsing some constrained grammar formalisms. *Computational Linguistics*, 19(4):591–636.

G. H. von Wright. 1951. Deontic logic. *Mind*, 60(237):1–15.

Chulin Xie, Yangsibo Huang, Chiyuan Zhang, Da Yu, Xinyun Chen, Bill Yuchen Lin, Bo Li, Badih Ghazi, and Ravi Kumar. 2024. On memorization of large language models in logical reasoning.

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. 2019. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-sql task. *Preprint*, arXiv:1809.08887.

Qingru Zhang, Chandan Singh, Liyuan Liu, Xiaodong Liu, Bin Yu, Jianfeng Gao, and Tuo Zhao. 2024. Tell your model where to attend: Post-hoc attention steering for llms. *Preprint*, arXiv:2311.02262.

A Context-Free Grammars and Languages

A *context-free grammar (CFG)*, which 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 $V \cap \Sigma = \varnothing$,
- $R \subseteq V \times (V \cup \Sigma)^*$ is a finite set of *production rules* of the form $A \to \alpha$, 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 $L = L(G)$.

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.

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

	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 3: 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.

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

Consider trees in figure 5. 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 2.

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

C Parser Details

Data Structure ($\mathsf{LF}[tier, row, X \to A\ B]$, $\boldsymbol{TIER}[row, X]$) Chart-like parsers such as CYK use a table indexed by row, $column$ and non-terminal X. The Incremental Valiant recognizer uses the same table with the alternative index $tier$, row, X, where $tier = col + row$.

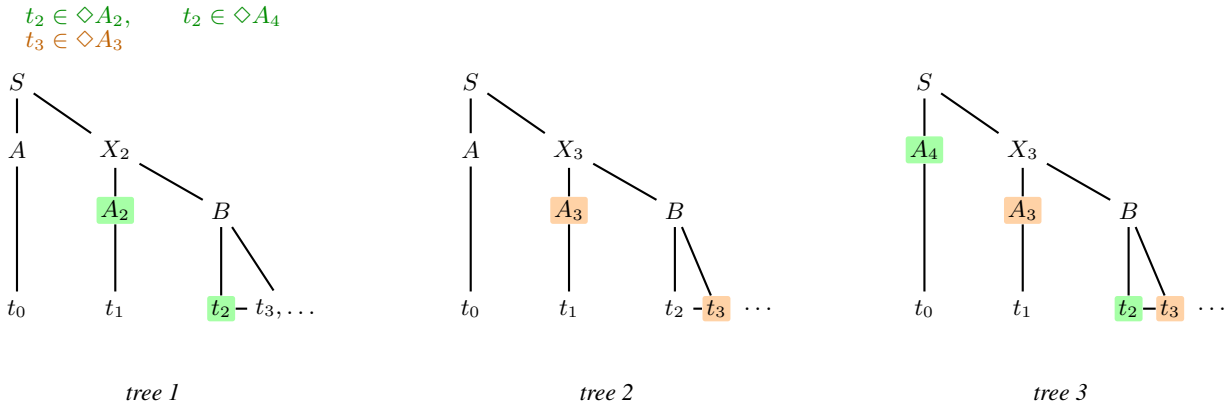

Figure 5: Three trees used in the counter example (highlighted nodes correspond to the committed constraints).

The CDoT parser inherits the boolean matrix $\boldsymbol{TIER}[row, X]$, and the left-foot boolean tensor $\mathbf{LF}[tier, row, X \to A\ B]$, with the last dimension indexed by binarized grammar rules.

Necessary ($\Box$) is enforced by denying the parent node from forming until the necessary condition is met. In the case that the underlying context-free grammar allows a finite number of consequent terminals, we assume the prescribed necessary requirement does not exceed this limit and the LLM completes the necessary requirement before the limit is exhausted. If this assumption is violated and the grammar does not provide an alternative path, generation will terminate.

Permitted ($\Diamond$) enforcement allows an antecedent non-terminal A instance to be used in anchoring grammar rule $X \to A\ B$ until a violating consequent terminal instance is committed.

C.1 Preprocessing

After a new terminal instance t_n is completed, we need to find left feet that 1) have all necessary conditions **fulfilled** after accepting t_n, 2) can give permission to **enable** t_n. This can be achieved by simple dynamic programming shown as algorithm 1. We omit proof for simplicity.

$$\begin{aligned} &\forall j \le \textsc{Fulfiller}[row, col] \Leftrightarrow \\ &\quad [t_j \dots t_n]_b \in \Box\big([t_{col} \dots t_{col+row}]_a\big) \\ &\textsc{Enabler}[row, col] \Leftrightarrow \\ &\quad t_n \in \Diamond([t_{col} \dots t_{col+row}]_a) \end{aligned} \tag{1}$$

C.2 Commit Wave

We update the parser for newly formed terminal instance t_n. We call the persistent state alive left foot, $\mathbf{LF}_n^{\mathbf{ALIVE}}$. This requires a bottom-up and a top-down pass that can be run in parallel. The pseudo code can be found as algorithm 2 and algorithm 6.

C.3 Probe Wave

Probe wave finds all valid next terminal types and confirms each doentic left foot belongs to a derivation that supports current prefix $t_0 \dots t_n$.

Additionally, to confirm each consequent terminal type have a unconstrained path, we run an additional bottom-up incremental Valiant step with set of grammar rules excluding corresponding deontic anchors. If a terminal type has a unconstrained path, the constrainer receive its context-free definition as specification.

Shown as algorithm 10 and algorithm 11

C.4 Post Processing

If a antecedent terminal is under a valid deontic left feet, then $\psi^{\Diamond}$ resulting from the corresponding deontic function should be added to enforcer during generation of the consequent terminal (algorithm 13). This takes a simple top-down dynamic programming with $O(|D| \cdot n^2)$ complexity.

D Knight and Knave by Puzzle Size

E Knight and Knave Constrainer Grammar

Algorithm 1: Deontic Preprocessing

Input: Terminal sequence $t_0 \dots t_n$, deontic rule with antecedent type a and consequent type b, necessary function $\Box$, permitted function $\Diamond$

Output: $\textsc{Fulfilled}[row, col]$, $\textsc{Enabler}[row, col]$

```
// Base case: row 0
```

for $col \leftarrow 0$ **to** $n-1$ **do**

  if $\text{type}(t_{col}) \neq a$ **then**

    $\textsc{Fulfilled}[0, col] \leftarrow +\infty$;

    $\textsc{Enabler}[0, col] \leftarrow \mathit{false}$;

  else

    $J \leftarrow \{\, j \mid [t_j \dots t_n]_b \in \Box(t_{col}) \,\}$;

    $J \leftarrow J \cup \{-\infty\}$;

    $\textsc{Fulfilled}[0, col] \leftarrow max(J)$;

    $\textsc{Enabler}[0, col] \leftarrow t_n \in \Diamond t_{col}$;

```
// Inductive step:
```

for $row \leftarrow 1$ **to** $n-1$ **do**

  $L^{FF} \leftarrow \textsc{Fulfilled}[row{-}1, :{-}1]$;

  $R^{FF} \leftarrow \textsc{Fulfilled}[row{-}1, 1{:}]$;

  $\textsc{Fulfilled}[row] \leftarrow \min(L^{FF}, R^{FF})$;

  $L^{EB} \leftarrow \textsc{Enabler}[row{-}1, :{-}1]$;

  $R^{EB} \leftarrow \textsc{Enabler}[row{-}1, 1{:}]$;

  $\textsc{Enabler}[row] \leftarrow L^{EB} \vee R^{EB}$;

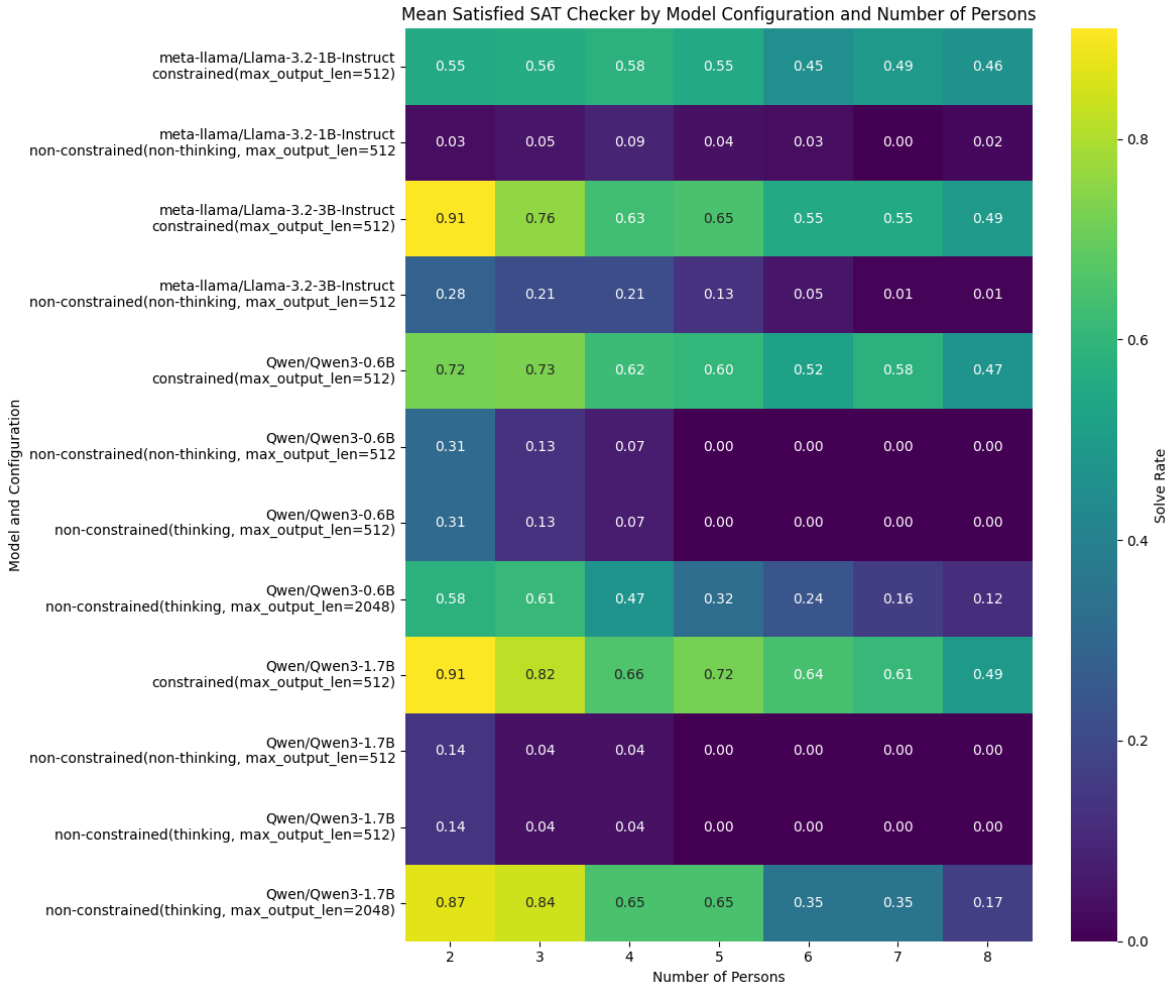

Figure 6: Knight and Knave test: Constrained models outperform unconstrained models across all puzzle sizes.

Algorithm 2: Commit↑: Computing $\mathsf{LF}_n^{\mathsf{ALIVE}}$ (Bottom-Up)

Input : ENABLER, FULFILLER, $\mathsf{LF}_{n-1}^{\mathsf{ALIVE}}$, FIRST, t_n
Output : $\mathsf{LF}_n^{\mathsf{ALIVE}}$

`// Initialization`
$\mathsf{LF}_n^{\mathsf{ALIVE}} \leftarrow$ **false** everywhere
$\mathsf{PARENT}^{\mathsf{FREE}}, \mathsf{PARENT}^{\mathsf{CONSTR}} \leftarrow$ **false** everywhere
$\mathsf{ANCESTOR}^{\mathsf{FREE}}, \mathsf{ANCESTOR}^{\mathsf{CONSTR}} \leftarrow$ **false** everywhere

`// Base Case: row = 0`
foreach *terminal rule* $(X \to t_n)$ **do**
  $\mathsf{PARENT}^{\mathsf{FREE}}[0, X] \leftarrow \mathbf{true}$

foreach $V \in \mathcal{N}$ **do**
  $\mathsf{ANCESTOR}^{\mathsf{FREE}}[0, V] \leftarrow \mathsf{PARENT}^{\mathsf{FREE}}[0, V]$

`AddAncestor(`$\mathit{ANCESTOR}^{\mathit{FREE}}[0]$, *FIRST*`)`

`// Inductive Step`
for $row_X \leftarrow 1$ **to** n **do**
  for $row_B \leftarrow 0$ **to** $row_X - 1$ **do**
    $tier_A \leftarrow n - row_B - 1$
    $row_A \leftarrow row_X - row_B - 1$
    foreach *rule* $(X \to A\ B)$ **do**
      if $\neg\, \mathit{LF}_{n-1}^{\mathit{ALIVE}}[tier_A, row_A, X \to A\ B]$ **then**
        continue
      `Merge`$^{\uparrow}_{\texttt{commit}}(row_X, row_B, tier_A, row_A, X \to A\ B)$
  `AddAncestor(`$\mathit{ANCESTOR}^{\mathit{FREE}}[row_X]$, *FIRST*`)`
  `AddAncestor(`$\mathit{ANCESTOR}^{\mathit{CONSTR}}[row_X]$, *FIRST*`)`

`// Finalization: Create new left feet at tier` n
for $row \leftarrow 0$ **to** n **do**
  foreach *rule* $(X \to A\ B)$ **do**
    $\mathsf{LF}_n^{\mathsf{ALIVE}}[n, row, X \to A\ B] \leftarrow \mathsf{PARENT}^{\mathsf{FREE}}[row, A] \vee \mathsf{PARENT}^{\mathsf{CONSTR}}[row, A]$

return $\mathsf{LF}_n^{\mathsf{ALIVE}}$

Algorithm 3: $\text{Merge}^{\uparrow}_{\textit{commit}}$ (Context-Free)

Input : $row_X, row_B, tier_A, row_A, (X \to A\ B)$ where $\text{type}(X \to A\ B) = \text{CF}$

if $\textit{PARENT}^{\textit{FREE}}[row_B, B]$ **then**
  $\textsf{PARENT}^{\textsf{FREE}}[row_X, X] \leftarrow \mathbf{true}$
  $\textsf{LF}_n^{\textsf{ALIVE}}[tier_A, row_A, X \to A\ B] \leftarrow \mathbf{true}$

if $\textit{PARENT}^{\textit{CONSTR}}[row_B, B]$ **then**
  $\textsf{PARENT}^{\textsf{CONSTR}}[row_X, X] \leftarrow \mathbf{true}$
  $\textsf{LF}_n^{\textsf{ALIVE}}[tier_A, row_A, X \to A\ B] \leftarrow \mathbf{true}$

if $\textit{ANCESTOR}^{\textit{FREE}}[row_B, B]$ **then**
  $\textsf{ANCESTOR}^{\textsf{FREE}}[row_X, X] \leftarrow \mathbf{true}$
  $\textsf{LF}_n^{\textsf{ALIVE}}[tier_A, row_A, X \to A\ B] \leftarrow \mathbf{true}$

if $\textit{ANCESTOR}^{\textit{CONSTR}}[row_B, B]$ **then**
  $\textsf{ANCESTOR}^{\textsf{CONSTR}}[row_X, X] \leftarrow \mathbf{true}$
  $\textsf{LF}_n^{\textsf{ALIVE}}[tier_A, row_A, X \to A\ B] \leftarrow \mathbf{true}$

Algorithm 4: $\text{Merge}^{\uparrow}_{\textit{commit}}$ (Strict Permitted)

Input : $row_X, row_B, tier_A, row_A, (X \to A\ B)$ where $\text{type}(X \to A\ B) = \textsc{Strict}$

$\textit{enabled} \leftarrow \textsf{ENABLER}[tier_A, row_A, X \to A\ B]$
$\textit{fulfilled} \leftarrow \textsf{FULFILLER}[tier_A, row_A, X \to A\ B] \geq (n - row_B)$

`// All paths must satisfy ◇ at this scope`

if $\textit{enabled} \wedge \left(\textit{ANCESTOR}^{\textit{FREE}}[row_B, B] \vee \textit{ANCESTOR}^{\textit{CONSTR}}[row_B, B]\right)$ **then**
  $\textsf{ANCESTOR}^{\textsf{CONSTR}}[row_X, X] \leftarrow \mathbf{true}$
  $\textsf{LF}_n^{\textsf{ALIVE}}[tier_A, row_A, X \to A\ B] \leftarrow \mathbf{true}$

if $\textit{enabled} \wedge \textit{fulfilled} \wedge \left(\textit{PARENT}^{\textit{FREE}}[row_B, B] \vee \textit{PARENT}^{\textit{CONSTR}}[row_B, B]\right)$ **then**
  $\textsf{PARENT}^{\textsf{CONSTR}}[row_X, X] \leftarrow \mathbf{true}$
  $\textsf{LF}_n^{\textsf{ALIVE}}[tier_A, row_A, X \to A\ B] \leftarrow \mathbf{true}$

Algorithm 5: Merge$^{\uparrow}_{commit}$ (Union Permitted)

Input : $row_X, row_B, tier_A, row_A, (X \to A\ B)$ where $\text{type}(X \to A\ B) = \textsc{Union}$

$enabled \leftarrow \mathsf{ENABLER}[tier_A, row_A, X \to A\ B]$
$fulfilled \leftarrow \mathsf{FULFILLER}[tier_A, row_A, X \to A\ B] \geq (n - row_B)$

`// Already permitted: propagate without re-checking`
if $\mathit{ANCESTOR}^{\mathit{CONSTR}}[row_B, B]$ **then**
  $\mathsf{ANCESTOR}^{\mathsf{CONSTR}}[row_X, X] \leftarrow \mathbf{true}$
  $\mathsf{LF}_n^{\mathsf{ALIVE}}[tier_A, row_A, X \to A\ B] \leftarrow \mathbf{true}$
if $\mathit{PARENT}^{\mathit{CONSTR}}[row_B, B]$ **then**
  $\mathsf{PARENT}^{\mathsf{CONSTR}}[row_X, X] \leftarrow \mathbf{true}$
  $\mathsf{LF}_n^{\mathsf{ALIVE}}[tier_A, row_A, X \to A\ B] \leftarrow \mathbf{true}$

`// Free` t_n `entering deontic scope: check` $\Diamond$
if $enabled \wedge \mathit{ANCESTOR}^{\mathit{FREE}}[row_B, B]$ **then**
  $\mathsf{ANCESTOR}^{\mathsf{CONSTR}}[row_X, X] \leftarrow \mathbf{true}$
  $\mathsf{LF}_n^{\mathsf{ALIVE}}[tier_A, row_A, X \to A\ B] \leftarrow \mathbf{true}$
if $enabled \wedge fulfilled \wedge \mathit{PARENT}^{\mathit{FREE}}[row_B, B]$ **then**
  $\mathsf{PARENT}^{\mathsf{CONSTR}}[row_X, X] \leftarrow \mathbf{true}$
  $\mathsf{LF}_n^{\mathsf{ALIVE}}[tier_A, row_A, X \to A\ B] \leftarrow \mathbf{true}$

Algorithm 6: Commit↓: Computing $\mathbf{LF}^{\text{ALIVE}}$ (Top-Down)

Input: $\textsc{Enabler}, \textsc{Fulfiller}, \mathbf{LF}^{\text{ALIVE}}_{n-1}, \textsc{First}, t_n$
Output: $\mathbf{LF}^{\text{ALIVE}}_n$
`// Initialization`
$\mathbf{LF}^{\text{ALIVE}}_n \leftarrow$ **false** everywhere;
$\textsc{Child}^{\textsc{Free}}, \textsc{Child}^{\textsc{Constr}} \leftarrow$ **false** everywhere;
$\textsc{Ancestor}^{\textsc{Free}}, \textsc{Ancestor}^{\textsc{Constr}} \leftarrow$ **false** everywhere;

`// Base case: Start symbol at row` n
foreach $V \in$ *Nonterminals* **do**
  $\textsc{Ancestor}^{\textsc{Free}}[n, V] \leftarrow (V = S)$;
$\textsc{AddDescendant}(\textsc{Ancestor}^{\textsc{Free}}[n], \textsc{First})$;

`// Inductive step: top-down`
for $row_X \leftarrow n$ **to** 1 **do**
  for $row_B \leftarrow 0$ **to** $row_X - 1$ **do**
    $tier_A \leftarrow n - row_B - 1$;
    $row_A \leftarrow row_X - row_B - 1$;
    foreach *rule* $(X \to A\ B)$ **do**
      if $\neg\, \mathbf{LF}^{\text{ALIVE}}_{n-1}[tier_A, row_A, X \to A\ B]$ **then continue**;
      $\textsc{Merge}\downarrow (row_X, row_B, tier_A, row_A, X \to A\ B)$;
  $\textsc{AddDescendant}(\textsc{Ancestor}^{\textsc{Free}}[row_X - 1], \textsc{First})$;
  $\textsc{AddDescendant}(\textsc{Ancestor}^{\textsc{Constr}}[row_X - 1], \textsc{First})$;
return $\mathbf{LF}^{\text{ALIVE}}_n$;

Algorithm 7: Merge↓ (Context-Free)

Input: $row_X, row_B, tier_A, row_A, (X \to A\,B)$ where type$(X \to A\,B) = \text{CF}$

if $\textsc{Child}^{\textsc{Free}}[row_X, X]$ **then**
  $\textsc{Child}^{\textsc{Free}}[row_B, B] \leftarrow$ **true**;
  $\mathbf{LF}_n^{\textsc{Alive}}[tier_A, row_A, X \to A\,B] \leftarrow$ **true**;

if $\textsc{Child}^{\textsc{Constr}}[row_X, X]$ **then**
  $\textsc{Child}^{\textsc{Constr}}[row_B, B] \leftarrow$ **true**;
  $\mathbf{LF}_n^{\textsc{Alive}}[tier_A, row_A, X \to A\,B] \leftarrow$ **true**;

if $\textsc{Ancestor}^{\textsc{Free}}[row_X, X]$ **then**
  $\textsc{Ancestor}^{\textsc{Free}}[row_B, B] \leftarrow$ **true**;
  $\mathbf{LF}_n^{\textsc{Alive}}[tier_A, row_A, X \to A\,B] \leftarrow$ **true**;

if $\textsc{Ancestor}^{\textsc{Constr}}[row_X, X]$ **then**
  $\textsc{Ancestor}^{\textsc{Constr}}[row_B, B] \leftarrow$ **true**;
  $\mathbf{LF}_n^{\textsc{Alive}}[tier_A, row_A, X \to A\,B] \leftarrow$ **true**;

Algorithm 8: Merge↓ (Strict Permitted)

Input: $row_X, row_B, tier_A, row_A, (X \to A\,B)$ where type$(X \to A\,B) = \textsc{Strict}$

$enabled \leftarrow \textsc{Enabler}[tier_A, row_A, X \to A\,B]$;

$fulfilled \leftarrow \textsc{Fulfiller}[tier_A, row_A, X \to A\,B] \geq (n - row_B)$;

`// All paths must satisfy ◇ at this scope`

if $enabled \wedge \big(\textsc{Ancestor}^{\textsc{Free}}[row_X, X] \vee \textsc{Ancestor}^{\textsc{Constr}}[row_X, X]\big)$ **then**
  $\textsc{Ancestor}^{\textsc{Constr}}[row_B, B] \leftarrow$ **true**;
  $\mathbf{LF}_n^{\textsc{Alive}}[tier_A, row_A, X \to A\,B] \leftarrow$ **true**;

if $enabled \wedge fulfilled \wedge \big(\textsc{Child}^{\textsc{Free}}[row_X, X] \vee \textsc{Child}^{\textsc{Constr}}[row_X, X]\big)$ **then**
  $\textsc{Child}^{\textsc{Constr}}[row_B, B] \leftarrow$ **true**;
  $\mathbf{LF}_n^{\textsc{Alive}}[tier_A, row_A, X \to A\,B] \leftarrow$ **true**;

Algorithm 9: Merge↓ (Union Permitted)

Input: $row_X, row_B, tier_A, row_A, (X \to A\,B)$ where type$(X \to A\,B) =$ UNION
$enabled \leftarrow$ ENABLER$[tier_A, row_A, X \to A\,B]$;
$fulfilled \leftarrow$ FULFILLER$[tier_A, row_A, X \to A\,B] \geq (n - row_B)$;

`// Already constrained: propagate (permission granted below)`
if ANCESTOR$^{\text{CONSTR}}[row_X, X]$ **then**
  ANCESTOR$^{\text{CONSTR}}[row_B, B] \leftarrow$ **true**;
  $\mathbf{LF}_n^{\text{ALIVE}}[tier_A, row_A, X \to A\,B] \leftarrow$ **true**;
if CHILD$^{\text{CONSTR}}[row_X, X]$ **then**
  CHILD$^{\text{CONSTR}}[row_B, B] \leftarrow$ **true**;
  $\mathbf{LF}_n^{\text{ALIVE}}[tier_A, row_A, X \to A\,B] \leftarrow$ **true**;

`// Free` t_n `entering deontic scope: check` $\Diamond$`, if satisfied` $\to$ `constrained`
if $enabled \wedge$ ANCESTOR$^{\text{FREE}}[row_X, X]$ **then**
  ANCESTOR$^{\text{CONSTR}}[row_B, B] \leftarrow$ **true**;
  $\mathbf{LF}_n^{\text{ALIVE}}[tier_A, row_A, X \to A\,B] \leftarrow$ **true**;
if $enabled \wedge fulfilled \wedge$ CHILD$^{\text{FREE}}[row_X, X]$ **then**
  CHILD$^{\text{CONSTR}}[row_B, B] \leftarrow$ **true**;
  $\mathbf{LF}_n^{\text{ALIVE}}[tier_A, row_A, X \to A\,B] \leftarrow$ **true**;

Algorithm 10: Probe↑

Input: b – hypothetical terminal type at position n; $\mathbf{LF}^{\text{ALIVE}}$; FIRST
Output: $\mathbf{LF}^{\text{REACHABLE}\uparrow}$
$\text{ANCESTOR}^{\uparrow}[\cdot,\cdot] \leftarrow false$;
$\mathbf{LF}^{\text{REACHABLE}\uparrow}[\cdot,\cdot,\cdot] \leftarrow false$;
foreach *terminal rule* $(V \to b)$ **do**
  $\text{ANCESTOR}^{\uparrow}[0, V] \leftarrow true$;
ADDANCESTOR$(\text{ANCESTOR}^{\uparrow}[0], \text{FIRST})$;
for $row_X \leftarrow 1$ **to** n **do**
  for $row_B \leftarrow 0$ **to** $row_X - 1$ **do**
    $tier_A \leftarrow n - row_B - 1$;
    $row_A \leftarrow row_X - row_B - 1$;
    foreach *rule* $(X \to A\,B)$ **do**
      if $\mathbf{LF}^{\text{ALIVE}}[tier_A, row_A, X \to A\,B] \wedge \text{ANCESTOR}^{\uparrow}[row_B, B]$ **then**
        $\text{ANCESTOR}^{\uparrow}[row_X, X] \leftarrow true$;
        $\mathbf{LF}^{\text{REACHABLE}\uparrow}[tier_A, row_A, X \to A\,B] \leftarrow true$;
  ADDANCESTOR$(\text{ANCESTOR}^{\uparrow}[row_X], \text{FIRST})$;
return $\mathbf{LF}^{\text{REACHABLE}\uparrow}$

Algorithm 11: Probe↓

Input: $\mathbf{LF}^{\text{ALIVE}}$; FIRST
Output: $\mathbf{LF}^{\text{REACHABLE}\downarrow}$
$\text{ANCESTOR}^{\downarrow}[\cdot,\cdot] \leftarrow \mathit{false}$;
$\mathbf{LF}^{\text{REACHABLE}\downarrow}[\cdot,\cdot,\cdot] \leftarrow \mathit{false}$;
$\text{ANCESTOR}_{\text{uncon}}[\cdot,\cdot,\cdot] \leftarrow \mathit{false}$; `// indexed by` $[b, row, V]$
$\text{ANCESTOR}^{\downarrow}[n, \mathit{START}] \leftarrow \mathit{true}$;
AddDescendant($\text{ANCESTOR}^{\downarrow}[n]$, FIRST);
foreach *deontic consequent terminal* b **do**
  $\text{ANCESTOR}_{\text{uncon}}[b, n, \mathit{START}] \leftarrow \mathit{true}$;
  AddDescendant($\text{ANCESTOR}_{\text{uncon}}[b, n]$, FIRST);
for $row_B \leftarrow n-1$ **to** 0 **do**
  for $row_X \leftarrow row_B + 1$ **to** n **do**
    $tier_A \leftarrow n - row_B - 1$;
    $row_A \leftarrow row_X - row_B - 1$;
    foreach *rule* $(X \rightarrow A\ B)$ **do**
      if $\mathbf{LF}^{\text{ALIVE}}[tier_A, row_A, X \rightarrow A\ B] \wedge \text{ANCESTOR}^{\downarrow}[row_X, X]$ **then**
        $\text{ANCESTOR}^{\downarrow}[row_B, B] \leftarrow \mathit{true}$;
        $\mathbf{LF}^{\text{REACHABLE}\downarrow}[tier_A, row_A, X \rightarrow A\ B] \leftarrow \mathit{true}$;
        foreach *deontic consequent terminal* b **do**
          if $(X \rightarrow A\ B)$ *is not anchor of any deontic rule with consequent* b **then**
            if $\text{ANCESTOR}_{\text{uncon}}[b, row_X, X]$ **then**
              $\text{ANCESTOR}_{\text{uncon}}[b, row_B, B] \leftarrow \mathit{true}$;
  AddDescendant($\text{ANCESTOR}^{\downarrow}[row_B]$, FIRST);
  foreach *deontic consequent terminal* b **do**
    AddDescendant($\text{ANCESTOR}_{\text{uncon}}[b, row_B]$, FIRST);
return $\mathbf{LF}^{\text{REACHABLE}\downarrow}$

Algorithm 12: AddAncestor / AddDescendant

Procedure AddAncestor($cell$, FIRST)
  foreach $V \in \mathit{Nonterminals}$ **do**
    if $\exists W : cell[W] \wedge \text{FIRST}[V, W]$ **then**
      $cell[V] \leftarrow \mathit{true}$;

Procedure AddDescendant($cell$, FIRST)
  foreach $W \in \mathit{Nonterminals}$ **do**
    if $\exists V : cell[V] \wedge \text{FIRST}[V, W]$ **then**
      $cell[W] \leftarrow \mathit{true}$;

Algorithm 13: Collect Permitted Constraints

Input : $\mathrm{LF}^{\mathrm{RRA}}[\mathrm{tier}, \mathrm{row}, \mathrm{rule}]$: `bool`
Input : Terminal sequence $t_0 \dots t_{n-1}$
Input : Deontic rules D
Output : $\psi_b^{\diamond}$: permitted specification sent to enforcer

$\mathrm{LF}^{\mathrm{RRA}}[\mathrm{tier}, \mathrm{row}]$	$\mathrm{Reachable}^{\uparrow} \wedge \mathrm{Reachable}^{\downarrow} \wedge \mathrm{Alive}$ for each left foot
$\mathrm{memo}[\mathrm{row}, \mathrm{col}]$	true if and only if underneath a RRA left foot
$\psi_b^{\diamond}$	permitted specification for next terminal of type b

for each *deontic rule with anchor* $a \gg A > X < B \ll b$ *and permitted deontic function* $\diamond$ **do**

`/* Initialize Memo with RRA */`
$\mathrm{memo}[\mathrm{row}, \mathrm{col}] \leftarrow \mathrm{LF}^{\mathrm{RRA}}[\mathrm{row} + \mathrm{col}, \mathrm{row}, X \rightarrow A\ B]$

`/* Propagate Reachability */`
for $\mathrm{row} \leftarrow (n-1)$ **to** 0 **do**
  $\mathrm{memo}[\mathrm{row}, :]$ `|=` $\mathrm{memo}[\mathrm{row}+1, :]$ `// vertical` $(\downarrow)$
  $\mathrm{memo}[\mathrm{row}, 1:]$ `|=` $\mathrm{memo}[\mathrm{row}+1, :\text{-}1]$ `// diagonal` $(\searrow)$

`/* Collect Active Antecedents */`
for $i \leftarrow 0$ **to** $n-1$ **do**
  if $\mathrm{type}(t_i) = a$ ***and*** $\mathrm{memo}[0, i]$ **then**
    $\psi_b^{\diamond} \leftarrow \psi_b^{\diamond} \cup \diamond(t_i)$

`/*` $\psi_b^{\diamond}$ `is sent to enforcer */`

```
start                       → bad_guess* good_guess
good_guess                  → sufficient_assignment_seq semi_3 GOOD_GUESS_TAIL
sufficient_assignment_seq   → assignment_seq
bad_guess                   → contradictory_assignment_seq semi_3 CONTRADICTORY_GUESS_TAIL
                            | repetition_assignment_seq semi_3 REPETITION_GUESS_TAIL
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
                            | GUESS_HEAD assignment
                            | unit_prop
assignment                  → ASSIGNMENT
ASSIGNMENT                  → (NAME "->" KK)
                            | AUX_VAR
repeating_assignment        → REPEATING_ASSIGNMENT
REPEATING_ASSIGNMENT^2      → ASSIGNMENT
semi_3                      → SEMI_3 "\n"
semi_2                      → SEMI_2 "\n"
semi_1                      → SEMI_1 "\n"
SEMI_3^3                    → ";"
SEMI_2^2                    → ";"
SEMI_1^1                    → ";"
GUESS_HEAD                  → "Here is a different and improved guess: {{\n"
GOOD_GUESS_TAIL             → "}} «DONE»"
CONTRADICTORY_GUESS_TAIL    → "}} contradiction\n\n"
REPETITION_GUESS_TAIL       → "}} Bad guess\n\n"
NAME                        → /(Jacob|Noah|Michael|Liam|Ella|. . . |Ava)/
KK                          → "knight" | "knave"
```

Figure 7: Context free grammar for LLM generations in the Knight and Knave problem.