NL2LOGIC: AST-Guided Translation of Natural Language into First-Order Logic with Large Language Models

Rizky Ramadhana Putra[1], Raihan Sultan Pasha Basuki[2], Yutong Cheng[1], Peng Gao[1]

[1]Virginia Tech, Blacksburg, VA, USA
[2]Universitas Ary Ginanjar, Jakarta, Indonesia
{rizky, yutongcheng, penggao}@vt.edu
raihansultan.pashabasuki@students.uag.ac.id

Abstract

Automated reasoning is critical in domains such as law and governance, where verifying claims against facts in documents requires both accuracy and interpretability. Recent work has adopted a structured reasoning paradigm that parses first-order logic (FOL) rules from natural language and delegates inference to automated solvers. With the rise of large language models (LLMs), methods such as GCD and CODE4LOGIC leverage their reasoning and code generation capabilities to enhance logic parsing. However, these approaches suffer from (1) fragile syntax control, due to weak enforcement of global grammar consistency, and (2) low semantic faithfulness, as they lack fine-grained clause-level semantic understanding. To address these challenges, we propose NL2LOGIC, a FOL translation framework that uses an AST as an intermediate layer, combining a recursive LLM-based semantic parser with an AST-guided generator that deterministically produces solver-ready code. On the FOLIO, LogicNLI, and ProofWriter benchmarks, NL2LOGIC attains 99% syntactic accuracy and improves semantic correctness by 30% over state-of-the-art baselines. Moreover, integrating NL2LOGIC into Logic-LM yields near-perfect executability and improves downstream reasoning accuracy by 31% over Logic-LM's original few-shot unconstrained FOL translation module.

1 Introduction

Natural language documents, especially in domains such as law, policy, and governance, encode complex logical relations that must be interpreted precisely for downstream reasoning and compliance checking. However, natural language is inherently ambiguous and unstructured, making it difficult for machines, and even humans, to ensure logical consistency, detect contradictions, or verify claims across documents. This gap has motivated research on *automated reasoning over natural language*, where models assess whether a claim is entailed by supporting evidence. Early approaches rely on neural entailment frameworks (Evans et al., 2018; Bowman et al., 2015; Rocktaschel et al., 2016) that employ neural networks to classify entailment and contradiction. However, these models remain opaque black boxes, lacking interpretability and formal verifiability. To improve transparency and explainability, recent research introduces a structured reasoning paradigm that first *derives explicit logical representations from text* and then verifies claims through automated reasoning engines. This transition improves inference transparency and auditability by explicitly representing intermediate proof steps, rather than only final results.

In recent years, Large Language Models (LLMs) have shown notable progress in logical reasoning, particularly when guided by prompting strategies such as few-shot examples and Chain-of-Thought (CoT) prompting. Building on this progress, recent research has developed *LLM-powered approaches* for translating natural language (NL) into first-order logic (FOL), leveraging the models' strong natural language understanding and code generation capabilities to enable automated reasoning with explicit semantics such as FOL.

To improve NL-to-FOL translation performance, several approaches have developed specialized pipelines that enhance LLMs with additional structural guidance and symbolic control. A representative work, Grammar-Constrained Decoding (GCD) (Raspanti et al., 2025), enforces token-level syntactic correctness by encoding the target formal language as a context-free grammar and constraining the LLM's decoding process to adhere to this grammar. CODE4LOGIC (Liu, 2025) adopts a complementary approach. It leverages the code understanding capability of code generation models. Rather than generating FOL formulas directly, it uses few-shot prompting to produce Python code that encodes the grammar tree, which, when exe-

cuted, generates the corresponding FOL rule.

Despite their contributions, these methods face two key limitations:

- **Fragile syntax control.** GCD restricts generation at the token level through context-free grammars. While this enforces local syntactic correctness, grammatically valid outputs do not necessarily constitute valid logical statements, such as those requiring variable or signature consistency. CODE4LOGIC uses in-context learning (ICL) to generate a grammar tree and guides subsequent code generation with this structure. However, despite leveraging ICL, the free-form generation process remains prone to hallucination, and any errors in the tree can propagate into the following code generation stage, leading to invalid or inconsistent outputs.

- **Limited semantic faithfulness.** Existing approaches formulate NL–to–FOL translation as a single-step text-to-text task, mapping entire paragraphs to complete logical programs and forcing models to perform comprehension and translation simultaneously. Handling each clause iteratively is essential, as clauses often encode complex logical relations that can overwhelm the LLM if processed and translated all at once. Without this iterative decomposition, the LLM tends to rely on shallow token prediction rather than genuine logical understanding, making hallucinations more likely when sentence structures become complex.

To address these limitations, we propose NL2LOGIC, a framework that translates natural language sentences into *semantically faithful* and *syntactically correct* logical rules. Rather than adopting an imprecise, unconstrained, and single-step approach, the *semantic parser* decomposes each sentence into clauses and iteratively extracting first-order logic components (e.g., predicates, quantifiers, and logical connectives) to construct a First-Order Logic Abstract Syntax Tree (FOLAST). Parsing each clause iteratively enables the model to make controlled, grammar-guided decisions, ensuring clause-level accuracy that composes into a globally coherent logical representation. Then, the *AST-guided generator* ensures **syntactic correctness** by deterministically compiling the FOLAST through a two-pass algorithm: the first pass registers all constants, variables, and relation signatures, while the second pass assembles scoped expressions following solver-specific grammar. This design enforces strict syntactic validity while preserving semantic alignment, producing executable logical rules compatible with solvers such as Z3 (De Moura and Bjørner, 2008) and SMT-LIB (Barrett et al., 2010).

We evaluate NL2LOGIC on three widely used natural language inference (NLI) datasets, FOLIO (Han et al., 2024), ProofWriter (Tafjord et al., 2021), and LogicNLI (Tian et al., 2021), using the Z3 reasoning engine for consistent formal verification. We also integrate NL2LOGIC into Logic-LM (Pan et al., 2023), a representative neuro-symbolic framework, demonstrating its practical value as a modular component.

The evaluation addresses three research questions: (RQ1) whether the generated formulas are syntactically valid; (RQ2) whether they preserve the intended semantics for entailment prediction; (RQ3) whether the integration of NL2LOGIC improves existing neuro-symbolic systems. Across twelve models ranging from 0.5B to 27B parameters, NL2LOGIC achieves near-perfect syntax correctness, improves semantic accuracy by an average of 30% over the strongest baseline (GCD), and improves the existing neuro-symbolic framework by 31% on downstream reasoning task over Logic-LM's original unconstrained FOL translation module. These results demonstrate that our decoupled, AST-based design provides stronger syntactic control and more faithful semantic alignment, advancing automated reasoning over natural language through formal, interpretable symbolic representations. To facilitate future research, we release our implementation at: `https://github.com/peng-gao-lab/nl2logic`.

2 Related Work

Natural Language to Formal Logic Translation. Research on formal logic translation has evolved across several paradigms. Early rule-based methods (Bos and Markert, 2005; Zettlemoyer and Collins, 2012; Barker-Plummer et al., 2009; Abzianidze, 2017) offer precise control but lack robustness to linguistic variation. Neural models later generated logical forms directly from text (Lu et al., 2022; Cao et al., 2019), improving generalization but struggling with rare operators and complex syntax. More recently, neuro-symbolic systems commonly adopt unconstrained few-shot LLM translation integrated with logic solvers (Pan

et al., 2023; Olausson et al., 2023; Xu et al., 2024; Sanyal et al., 2022; Quan et al., 2024, 2025). Structured pipelines that add syntactic constraints (Raspanti et al., 2025; Liu, 2025) further improve performance, yet the resulting formulas often remain semantically unfaithful to the source text.

LLMs for Logical Reasoning. The development of logical reasoning capabilities in LLMs has seen significant progress through a range of approaches. One line of work breaks down complex reasoning into intermediate steps, often referred to as chain-of-thought prompting (Wei et al., 2022), while others show that simple step-by-step prompting can yield similar benefits without explicit examples (Kojima et al., 2022). However, these approaches lack the formal proof. To address this, some frameworks combine LLMs with automated reasoning tools to improve faithfulness (Creswell et al., 2022). Recent research has further explored integrating LLMs with symbolic solvers (Wang et al., 2023), treating LLMs as logical parsers rather than independent reasoners. In summary, prior work either lacks formal proof for logical reasoning or produces unfaithful logical forms due to overly simple, unconstrained translation methods.

3 Preliminary

First-Order Logic. First-Order Logic (FOL) is a formal system for expressing statements about entities, their properties, and relations (Enderton, 2001). Its syntax comprises *constants*, *variables*, *functions*, *relations*, *quantifiers* (e.g., $\forall$, $\exists$), and *logical connectives* (e.g., $\wedge$, $\vee$, $\rightarrow$, $\neg$). A well-formed FOL formula such as $\forall x.\, \mathit{Human}(x) \rightarrow \mathit{Mortal}(x)$ captures meaning in a precise, machine-verifiable form. Unlike natural language, FOL enforces strict grammar, enabling unambiguous interpretation and compatibility with automated reasoning systems.

Abstract Syntax Tree. An Abstract Syntax Tree (AST) is a tree-based data structure, commonly used in compilers and interpreters to represent the syntactic structure of a program (Alfred et al., 2007). Each node in an AST corresponds to a syntactic construct (e.g., operator, expression, declaration), enabling structural analysis and systematic code generation. We extend this concept to a first-order logic AST (*FOLAST*) defined by the standard FOL grammar in Fig. 2, serving as a backend-independent intermediate representation linking natural language to the generated code that will be executed by automated reasoning engines.

Reasoning Engines and Target Languages. Automated reasoning engines such as Z3 (De Moura and Bjørner, 2008), Prover9 (McCune, 2005–2010), and SMT-LIB (Barrett et al., 2010) solvers evaluate FOL statements for satisfiability, entailment, and consistency. Each engine requires strict, engine-specific syntax, e.g., Z3's Python API or SMT-LIB's symbolic format. However, these syntactic forms are rarely observed during LLM pretraining. Hence, directly generating solver code in an end-to-end manner leads to high syntax errors and inconsistent logical structures. Our *key insight* is to decouple logical parsing from code generation by introducing an intermediate AST representation, which captures the logical structure in an engine-agnostic form and can then be deterministically compiled into the target reasoning language. This separation enforces syntactic correctness, facilitates multi-engine generalization, and avoids LLM hallucinations tied to solver-specific syntax.

4 NL2LOGIC Design

NL2LOGIC adopts a parser-generator architecture designed to ensure syntactic correctness and semantic faithfulness in translating natural language into formal logic. As illustrated in Fig. 1, the pipeline consists of two stages: a *semantic parser*, which converts natural language text into a structured first-order logic abstract syntax tree (FOLAST), defined by the standard grammar in Fig. 2, and an *AST-guided generator*, which compiles the FOLAST into solver-executable code (e.g., Z3, SMT-LIB). We next describe each component.

4.1 Preprocessing

Documents are divided into well-defined sentences leveraging prior work on sentence boundary detection, as directly feeding long paragraphs into the parser risks hallucination and error propagation. Instead of rule-based splitting that relies only on punctuation and heuristics, we adopt SaT (Frohmann et al., 2024; Minixhofer et al., 2023), a learning-based model that predicts sentence boundaries using contextual and lexical cues. This ML-based approach distinguishes true sentence endings from punctuation used in abbreviations (e.g., *U.S.*, *Prof.*) or numeric expressions, thereby avoiding fragmentation errors. As a result, each logical sentence is reliably isolated, providing clean and accurate input to the semantic parser.

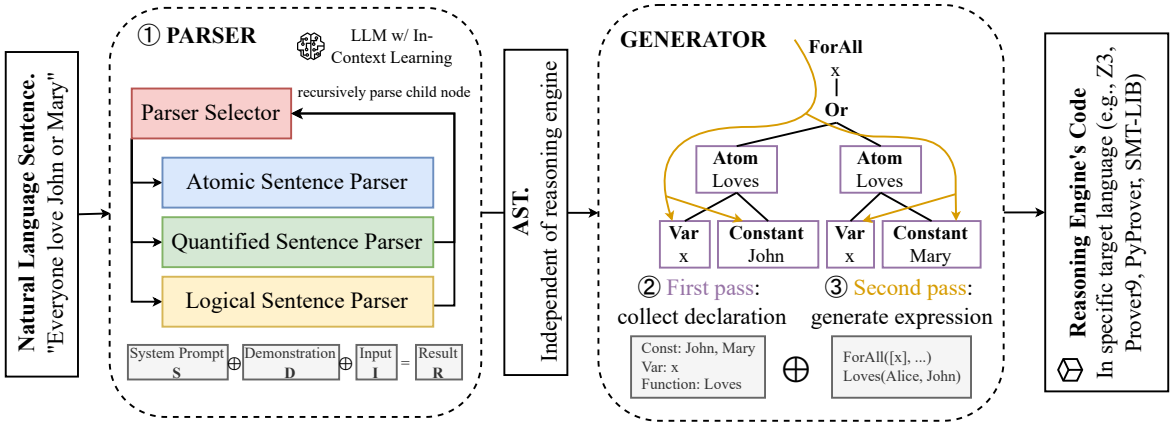


Figure 1: Overview of NL2LOGIC. The semantic parser (①) takes a natural language sentence and outputs a first-order logic abstract syntax tree (FOLAST) through a recursive, top-down approach. The AST is then compiled into the reasoning engine's target language through a two-pass algorithm. The first pass (②) collects constant, variable, and predicate signature declarations. The second pass (③) generates the concrete logical expression.

$$
\begin{array}{rcl}
\langle Term \rangle & ::= & \langle Variable \rangle \mid \langle Constant \rangle \\
\langle Atomic \rangle & ::= & \texttt{RelationName}(\langle Term \rangle) \\
& \mid & \texttt{RelationName}(\langle Term \rangle, \langle Term \rangle) \\
& \mid & \texttt{RelationName}(\langle Term \rangle, \langle Term \rangle, \langle Term \rangle) \\
\langle Quantified \rangle & ::= & \forall \langle Variable \rangle . \langle Formula \rangle \\
& \mid & \exists \langle Variable \rangle . \langle Formula \rangle \\
\langle Logical \rangle & ::= & \neg \langle Formula \rangle \\
& \mid & \langle Formula \rangle \wedge \langle Formula \rangle \\
& \mid & \langle Formula \rangle \vee \langle Formula \rangle \\
& \mid & \langle Formula \rangle \rightarrow \langle Formula \rangle \\
\langle Formula \rangle & ::= & \langle Atomic \rangle \\
& \mid & \langle Quantified \rangle \\
& \mid & \langle Logical \rangle
\end{array}
$$

Figure 2: Formal notation of abstract syntax tree (AST)

4.2 Semantic Parser

The semantic parser is the first and most critical stage of NL2LOGIC. It maps natural language sentences into a first-order logic abstract syntax tree (FOLAST; described in Section 3) that strictly enforces logical grammar. Without this stage, directly prompting an LLM to generate FOL symbols or solver-specific code (e.g., Z3, SMT-LIB) often yields syntax errors, hallucinations, or undeclared variables. This is demonstrated in our evaluations (Section 5) and caused by such formats are rarely observed during pre-training. By isolating parsing as a dedicated component, we guarantee that natural language is first converted into a syntactically valid and semantically transparent representation.

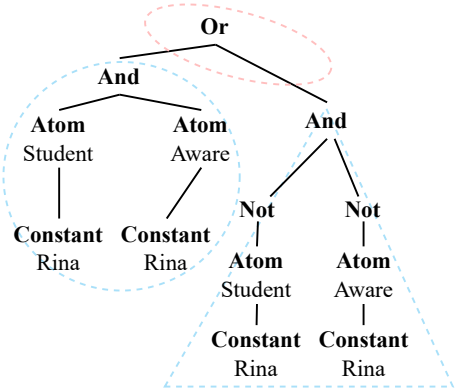
Conventional NLP parsers (e.g., dependency or constituency parsers) are inadequate for translating NL into formal logic. Rule-based parsers (Woods, 1970; Marcus, 1978) fail to capture the diversity and ambiguity of natural sentences, while conventional ML-based parsers (Kitaev and Klein, 2018; Kitaev et al., 2019) require large annotated corpora of NL and FOL pairs. In contrast, large language

Rina is either a student who is unaware that caffeine is a drug, or she is not a student and is aware that caffeine is a drug

(a) The parser decomposes a sentence containing a logical operator into its operator and operand(s).

- Rina is a student who is unaware that caffeine is a drug
- Rina is not a student and is aware that caffeine is a drug

(b) The extracted operands are rewritten as standalone sentences. Since they are not necessarily atomic, each operand is recursively fed back to the parser.



(c) The complete AST representation. The parser identifies only the outermost structure, while the operands are recursively fed back to the parser.

$$(\text{Student}(\text{Rina}) \wedge \text{Aware}(\text{Rina})) \vee (\neg\text{Student}(\text{Rina}) \wedge \neg\text{Aware}(\text{Rina}))$$

(d) The first-order logic rule representation

Figure 3: `LogicalSentenceParser` example.

models (LLMs) can perform in-context learning (Brown et al., 2020), allowing them to follow explicit grammar constraints and generate structured FOLAST outputs. Therefore, an *LLM-based parser* is both necessary and practical: it combines pretrained linguistic knowledge with these formal grammar constraints to produce accurate and generalizable logical representations.

Our parser operates recursively through specialized sub-modules, as shown in Fig. 1. The *Parser Selector* first classifies each sentence as `atomic`, `quantified`, or `logical`. An `atomic` sentence expresses a single relation, a `quantified` sentence introduces a quantifier (e.g., $\forall$, $\exists$), and a `logical` sentence contains logical connectives (e.g., $\wedge$, $\vee$, $\rightarrow$, $\neg$). Then, the corresponding sub-parser is invoked: the *Atomic Sentence Parser* extracts predicates and arguments; the *Quantified Sentence Parser* identifies quantifiers and variables, then recursively parses the quantified scope; and the *Logical Sentence Parser* detects operators, splits the input into operands, and recursively processes each operand. At each step, the parser focuses only on the outermost construct while delegating the remaining sub-sentences to recursive parsing calls. This *top-down recursion* incrementally builds the AST, mirroring the hierarchical composition of FOL expressions. This design limits error propagation, reduces the LLM's cognitive load, and maintains semantic faithfulness across sentences of varying complexity. The complete system prompts for each parser are provided in Appendix A.

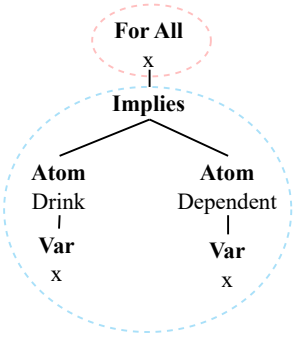
To illustrate the recursive mechanism, Fig. 3 and Fig. 4 present two representative cases, compound and quantified sentences. Fig. 3 shows how a disjunctive sentence is parsed by identifying the outermost operator (`Or`) and recursively decomposing its operands, each containing internal conjunctions and negations. At each step, the parser enforces local correctness by validating the syntactic and semantic consistency of each node before integrating it into the higher-level tree. For example, it ensures that every predicate has a valid argument (e.g., `Student(Rina)`) and that logical connectives such as `And` or `Not` combine clauses rather than partial phrases. Fig. 4 then illustrates a quantified structure, where the parser isolates the quantifier (`ForAll`), abstracts the variable (`x`), and recursively parses its scoped clause into atomic predicates such as `Drink(x)` and `Dependent(x)`. These two examples show how our parser consistently handles different logical forms by decomposing them into their outermost constructs and deferring sub-sentences to recursive parsing.

All people who regularly drink coffee are dependent on caffeine.

(a) The parser decomposes the sentence into its quantifier and scope.

x that regularly drink coffee are dependent on caffeine

(b) The scope is rewritten such that the quantified subject is replaced by a variable. Since the scope is not necessarily atomic, it is recursively fed back into the parser.



(c) The complete AST representation. The parser identifies only the outermost structure, while the scope is recursively fed back to the parser.

$$\forall x\,(\text{Drink}(x) \rightarrow \text{Dependent}(x))$$

(d) The first-order logic rule representation.

Figure 4: `QuantifiedSentenceParser` example.

4.3 AST-Guided Generator

The AST-guided generator converts the FOLAST produced by the parser into solver-ready code (e.g., Z3 or SMT-LIB). The generator deterministically maps each AST node to target-language expressions, mimicking how source-to-source compilers work (e.g., Cython (Behnel et al., 2010)). Its workflow follows a *two-pass algorithm*: the first pass traverses the AST to collect all variable, constant, and relation signature declarations, establishing a consistent global context; the second pass revisits each node to emit logical expressions that respect operator precedence, quantifier scope, and relation arity. The generator produces executable code in the reasoning engine's target language, preserving syntactic consistency and faithfully reflecting the logical structure defined by the FOLAST. We describe each pass below.

The first pass, *Declaration Collection*, constructs the symbol table for the target language (Algorithm 1). Each node is visited in preorder: declaration nodes (constants, variables, and relations)

Algorithm 1 First pass to collect declaration in an AST

Require: N ▷ *set of nodes in an AST*
Ensure: D ▷ *set of declaration*

```
D ← ∅
for n ∈ N do
    if n is a variable then
        D ← D ∪ {DECLAREVAR(n)}
    else if n is a constant then
        D ← D ∪ {DECLARECONST(n)}
    else if n is a relation then
        D ← D ∪ {DECLARERELATION(n)}
    else continue
```

record their signatures in a shared context, while expression nodes (e.g., `And`, `Or`, `Not`, `Implies`) are traversed only to process their children without generating expressions. Together, these recorded entries form a declaration environment, a mapping that registers all identifiers to their declared type or signature. This environment ensures that every symbol referenced in the subsequent expression generation stage (e.g., variable names or predicate signatures) has been properly introduced.

The second pass, *Expression Generation*, emits logical statements in the reasoning engine's target language (Algorithm 2). Each AST node is revisited to produce concrete code: constants and variables map to their declared identifiers, relational nodes expand into function calls with correct arity, and logical operators format operands according to the solver's grammar. Quantified nodes introduce scoped variables and generate expressions with explicit bindings, ensuring variable names remain unique and properly scoped. All generated expressions are then added to the solver's assertion set (e.g., `s.add(...)` in Z3). This pass converts the FOLAST into final executable reasoning rules, grounding natural language in formal logic.

5 Evaluation

5.1 Evaluation Setup

Baseline. We compare NL2LOGIC against two representative baselines. The first baseline is Grammar-Constrained Decoding (GCD) (Raspanti et al., 2025), a state-of-the-art approach that enforces syntactic correctness by encoding the target formal language as a context-free grammar and constraining the LLM's decoding process accordingly. We run GCD using the authors' released

Algorithm 2 Second pass to generate expression in an AST

```
function GENERATEEXPRESSION(r)
    if r = atomic sentence then
        a ← ∅
        for x ∈ r.args do
            a ← a ∪ GENERATEEXPRESSION(x)
        name ← r.name
        return ATOMICSENTENCE(name, a)
    else if r = binary sentence then
        a ← r.left_operand
        a ← GENERATEEXPRESSION(a)
        b ← r.right_operand
        b ← GENERATEEXPRESSION(b)
        op ← r.operator
        return BINARYSENTENCE(a, op, b)
    else if r = negation sentence then
        s ← r.sentence
        n ← GENERATEEXPRESSION(s)
        return NEGATEDSENTENCE(n)
    else if r = quantified sentence then
        q ← r.quantifier
        s ← GENERATEEXPRESSION(r.scope)
        return QUANTIFIEDSENTENCE(q, s)
    else if r = variable or constant then
        return r.name
```

implementation and follow the original evaluation protocol.

The second baseline is Logic-LM (Pan et al., 2023), a representative neuro-symbolic framework that translates natural language into first-order logic using few-shot prompting, without explicit syntactic or semantic constraints on the generated formulas. To evaluate whether NL2LOGIC can strengthen existing neuro-symbolic systems, we replace Logic-LM's original NL-to-FOL translation component with NL2LOGIC while keeping all other components unchanged. For this comparison, we report both executable rate (i.e., whether the generated FOL formulas are syntactically valid and solver-executable) and downstream NLI accuracy.

For fairness, we evaluate using the same models (Gemma, Llama, Mistral, and Qwen) with sizes ranging from 0.5 to 27B parameters, as shown in Tables 1 and 3 to 5. For Logic-LM integration, we maintain the original pipeline but substitute only the FOL translation component.

Datasets. We use three natural language inference (NLI) datasets, LogicNLI (Tian et al., 2021),

ProofWriter (Tafjord et al., 2021), and FOLIO (Han et al., 2024). These datasets are widely used for NLI (Pan et al., 2023; Morishita et al., 2024) and first-order logic (FOL) translation tasks (Liu, 2025; Yang et al., 2024). In total, we use 3,000 premise-hypothesis pairs. Each premise consists of a set of sentences, with each sentence corresponding to one logical rule. The hypothesis is a single sentence expressible as one logical statement. The sentence structures are relatively simple, with minimal co-reference and inter-sentence dependencies. Each pair is labeled as `entailment`, `contradiction`, or `uncertain`. To obtain the entailment prediction $\hat{y}$, NL2LOGIC parses the premises p and hypotheses h to generate executable code for the automated reasoning engine SOLVER, as defined in Eq. (1).

$$\text{SOLVER}(p,h) = \begin{cases} \text{Ent.}, & p \models h \wedge p \not\models \neg h, \\ \text{Cont.}, & p \not\models h \wedge p \models \neg h, \\ \text{Unc.}, & \text{otherwise} \end{cases}$$
$$\hat{y} = \text{SOLVER}(p,h) \tag{1}$$

RQs and Metrics. Our evaluations aim to answer two research questions.

- **RQ1: Syntax correctness.** We assess whether NL2LOGIC generates rules that adhere to first-order logic syntax, as specified in Fig. 2. Syntax correctness is quantified using the correctness rate defined in Eq. (2), where N_{correct} is the number of sentences with correct syntax and N_{total} is the total number of sentences.

$$\text{Syntax Correctness Rate} = \frac{N_{\text{correct}}}{N_{\text{total}}} \tag{2}$$

- **RQ2: Semantic correctness.** Following standard practice in prior NL-to-FOL work (Raspanti et al., 2025; Liu, 2025), we assess semantic correctness through downstream natural language inference (NLI) accuracy. This indirect evaluation is necessary because FOL expressions admit many truth-equivalent variants (e.g., $\neg(P \wedge Q) \equiv \neg P \vee \neg Q$), making direct comparison against a single canonical form impractical. Each NLI problem consists of premises and a hypothesis, and correctness is measured as NL2LOGIC 's accuracy (Eq. (3)), defined as the proportion of predictions ($\hat{y}_i$) matching the gold labels (y).

$$\text{Accuracy} = \frac{\sum_{i=1}^{n} 1(\hat{y}_i = y_i)}{n} \tag{3}$$

- **RQ3: Integration with neuro-symbolic systems.** We evaluate whether NL2LOGIC 's AST-guided translation improves existing neuro-symbolic frameworks that rely on unconstrained zero-shot or few-shot prompting. Specifically, we integrate NL2LOGIC into the Logic-LM (Pan et al., 2023) pipeline by replacing its original FOL translation module. Performance is assessed using the same two key metrics, which are syntactic and semantic accuracy.

5.2 RQ1: Syntax Correctness

Result. Table 1 shows that NL2LOGIC **consistently outperforms** the grammar-constrained decoding (GCD) baseline (Raspanti et al., 2025) across all model families and scales, achieving **near-perfect syntactic correctness** (up to 0.99). The improvement is most pronounced in smaller models (e.g., Gemma-2-2B, Llama-3.2-1B), where syntax control is typically fragile. This demonstrates that NL2LOGIC 's *iterative parser* effectively mitigates syntax drift by incrementally constructing the abstract syntax tree (AST), thus reducing the model's cognitive burden. For larger models (e.g., Mistral-22B, Qwen-2.5-14B), NL2LOGIC saturates near 0.99 accuracy, showing that once local node validity is guaranteed, *global syntactic integrity emerges naturally*.

	FOLIO		LogicNLI	
Model	**GCD**	**NL2 LOGIC**	**GCD**	**NL2 LOGIC**
gemma-2-2b	0.56	**0.92**	0.25	**0.94**
gemma-2-9b	0.77	**0.98**	0.91	**0.99**
gemma-2-27b	0.81	**0.99**	0.93	**0.99**
llama-3.2-1b	0.27	**0.99**	0.08	**0.97**
llama-3.2-3b	0.29	**0.93**	0.28	**0.98**
llama-3.1-8b	0.13	**0.96**	0.68	**0.98**
ministral-8b	0.09	**0.99**	0.07	**0.98**
mistral-22b	0.29	**0.99**	0.46	**0.98**
qwen-2.5-0.5b	0.14	**0.66**	0.05	**0.80**
qwen-2.5-1.5b	0.36	**0.87**	0.09	**0.99**
qwen-2.5-3b	0.12	**0.81**	0.01	**0.99**
qwen-2.5-7b	0.01	**0.94**	0.01	**0.99**
qwen-2.5-14b	0.53	**0.92**	0.56	**0.99**

Table 1: Syntax correctness rate. NL2LOGIC consistently outperforms GCD and achieves near-perfect syntax correctness.

Error Analysis. However, errors may still occur when the LLM produces invalid nodes, rendering the accumulated AST incorrect. Two types of errors are commonly observed in our experiments.

```
{"quantifier": {
    "quantifier" : { ...
```

(a) In rare cases, the LLM produces incomplete JSON that exceeds the maximum token limit.

```
{"transitive_verb": "kind",
  "subject": "sophia",
  "object": "" }
```

(b) The `parser` applies an incorrect JSON schema for the sentence "Sophia is kind", causing the LLM to leave the object empty. As a result, the generated node is invalid.

Figure 5: Common errors that result in first-order logic syntax violations.

The first type violates the given JSON output format entirely, resulting in missing nodes, as illustrated in Fig. 5a. Although rare, the LLM may hallucinate and produce incomplete or unparseable JSON. The second type conforms to the JSON schema but leaves required fields empty, resulting in invalid nodes, such as missing variable names in `Term` nodes or relation names in `Atom` nodes, as shown in Fig. 5b. Together, these cases account for less than 5% of all sentences. Error statistics are summarized in Table 2. We minimized these errors by employing strong output-control techniques, including the structured output feature in vLLM (Kwon et al., 2023) and few-shot prompting (Appendix A) to guide the LLM in producing structured JSON representations of AST nodes.

Model	Missing Nodes	Invalid Nodes	Total Sentences
gemma-2-2b	15	472	7100
gemma-2-9b	1	63	
gemma-2-27b	1	24	
llama-3.2-1b	127	5	
llama-3.2-3b	133	148	
llama-3.1-8b	107	97	
ministral-8b	100	9	
mistral-22b	100	13	
qwen-2.5-0.5b	13	1786	
qwen-2.5-1.5b	31	392	
qwen-2.5-3b	18	530	
qwen-2.5-7b	0	173	
qwen-2.5-14b	5	224	

Table 2: Error analysis on syntax correctness, accounting for less than 5% of all sentences.

5.3 RQ2: Semantic Correctness

Result. Table 3 shows that NL2LOGIC improves performance on natural language inference tasks by an average of 31% over the grammar-constrained decoding (GCD) baseline. This gain stems from NL2LOGIC 's parser, which incrementally constructs the abstract syntax tree (AST) instead of generating the entire logical rule in one step. The iterative design lowers the LLM's cognitive load when identifying entities, predicates, and logical connectives, which is particularly advantageous for smaller models. NL2LOGIC achieves relatively higher accuracy on LogicNLI than on FOLIO because LogicNLI contains simpler sentence structures with clearer connectives and clauses, making the top-down parsing approach more natural. In contrast, on FOLIO, some larger models with GCD surpass NL2LOGIC, suggesting that excessive structural decomposition may not always be beneficial, especially for sentences that are not naturally parse-able in a top-down manner.

	FOLIO		LogicNLI	
Model	**GCD**	**NL2 LOGIC**	**GCD**	**NL2 LOGIC**
gemma-2-2b	0.27	**0.35**	0.22	**0.41**
gemma-2-9b	**0.53**	0.38	0.23	**0.37**
gemma-2-27b	**0.61**	0.40	0.24	**0.37**
llama-3.2-1b	0.17	**0.38**	0.11	**0.33**
llama-3.2-3b	0.18	**0.36**	0.15	**0.37**
llama-3.1-8b	0.12	**0.37**	0.17	**0.37**
ministral-8b	0.16	**0.37**	0.04	**0.38**
mistral-22b	0.23	**0.38**	0.14	**0.34**
qwen-2.5-0.5b	0.17	**0.26**	0.12	**0.36**
qwen-2.5-1.5b	0.19	**0.34**	0.03	**0.41**
qwen-2.5-3b	0.08	**0.32**	0.01	**0.35**
qwen-2.5-7b	0.01	**0.37**	0.01	**0.34**
qwen-2.5-14b	0.34	**0.37**	0.27	**0.35**

Table 3: Semantic correctness measured by accuracy on natural language inference (NLI) tasks.

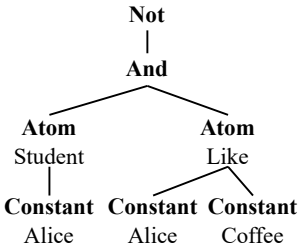
Observation and Error Analysis. There are two common semantic errors. First, words with implicit negation (e.g., unable, unaware, inconsistent) often confuse the parser. Despite careful prompt design, the parser sometimes misinterprets them and enters an infinite recursive parse. For instance, the sentence `'John is unable to walk'` is parsed back and forth to the sentence `'John is not unable to walk'`, as illustrated in Fig. 6a. Second, sentences with subtle operator ordering often lead to operator order errors. For example, the sentence Alice is not a student and does not like coffee is occasionally parsed with the `Not` operator preceding `And`, producing a semantically incorrect AST, as shown in Fig. 6b.

5.4 RQ3: Integration with Neuro-Symbolic Systems

To demonstrate NL2LOGIC's practical value as a modular component, we integrated it into Logic-

```
Parsing 'John is unable to walk'
Unary operator parser. Operator: 'Not'
|– Parsing 'John is not unable to walk'
   Unary operator parser. Operator: 'Not'
      ...
```

(a) Infinite recursion caused by parsing words with implicit negation.



(b) Incorrect operator ordering when parsing sentences with complex compound sentences. The sentence is "Alice is not a student and does not like coffee", where And operator should precede the Not operator.

Figure 6: Common semantic errors.

LM (Pan et al., 2023), a representative neuro-symbolic framework combining LLMs with symbolic solvers. Logic-LM originally uses few-shot prompting for FOL translation without explicit syntactic or semantic constraints. We replaced this module with NL2LOGIC while keeping all other pipeline components unchanged (Z3 solver, refinement mechanisms, answer selection), isolating the impact of our AST-guided translation on syntactic validity and downstream reasoning accuracy.

Table 4 presents executable rate comparison on ProofWriter (Tafjord et al., 2021) and FOLIO (Han et al., 2024). Logic-LM's original few-shot translation produces highly variable executable rates (0.01-0.94), with particularly poor performance on smaller models (e.g., 0.01 for gemma-2-2b, qwen-2.5-0.5b on ProofWriter). NL2LOGIC achieves near-perfect executable rate (0.99) across all 13 models and both datasets through iterative AST construction validating each logical construct incrementally.

This syntactic validity improvement translates into better semantic correctness. Table 5 shows accuracy on executable rules only, where predictions originate from logic solver execution rather than backup strategies (random guessing or CoT fallback). NL2LOGIC improves Logic-LM's accuracy by ∼31% on average. Gains are most pronounced on smaller models: on ProofWriter, gemma-2-2b improves from 0.01 to 0.58, llama-3.2-3b from 0.03 to 0.68, qwen-2.5-7b from 0.08 to 0.90, demonstrating that NL2LOGIC's structured decomposition reduces cognitive load by parsing natural language into FOL one clause at a time.

Model	ProofWriter		FOLIO	
	Logic LM	NL2 Logic	Logic LM	NL2 Logic
gemma-2-2b	0.01	**0.99**	0.07	**0.99**
gemma-2-9b	0.90	**0.99**	0.65	**0.99**
gemma-2-27b	0.94	**0.99**	0.60	**0.99**
llama-3.2-1b	0.72	**0.99**	0.83	**0.99**
llama-3.2-3b	0.25	**0.99**	0.38	**0.99**
llama-3.1-8b	0.90	**0.99**	0.58	**0.99**
ministral-8b	0.20	**0.99**	0.14	**0.99**
mistral-22b	0.85	**0.99**	0.65	**0.99**
qwen-2.5-0.5b	0.01	**0.99**	0.01	**0.99**
qwen-2.5-1.5b	0.01	**0.99**	0.05	**0.99**
qwen-2.5-3b	0.45	**0.99**	0.52	**0.99**
qwen-2.5-7b	0.15	**0.99**	0.55	**0.99**
qwen-2.5-14b	0.38	**0.99**	0.72	**0.99**

Table 4: Executable rate: Logic-LM original vs. with NL2LOGIC integration.

Model	ProofWriter		FOLIO	
	Logic LM	NL2 Logic	Logic LM	NL2 Logic
gemma-2-2b	0.01	**0.58**	0.21	**0.36**
gemma-2-9b	0.45	**0.85**	0.26	**0.39**
gemma-2-27b	0.60	**0.90**	0.31	**0.41**
llama-3.2-1b	0.25	**0.47**	0.18	**0.38**
llama-3.2-3b	0.03	**0.68**	0.14	**0.36**
llama-3.1-8b	0.40	**0.91**	0.22	**0.38**
ministral-8b	0.18	**0.75**	0.04	**0.39**
mistral-22b	0.51	**0.78**	0.26	**0.41**
qwen-2.5-0.5b	0.01	**0.38**	0.01	**0.26**
qwen-2.5-1.5b	0.01	**0.48**	0.02	**0.34**
qwen-2.5-3b	0.17	**0.37**	0.14	**0.32**
qwen-2.5-7b	0.08	**0.90**	0.22	**0.37**
qwen-2.5-14b	0.32	**0.86**	0.31	**0.38**

Table 5: Accuracy on executable rules: Logic-LM original vs. with NL2LOGIC integration.

6 Conclusion

This paper presents NL2LOGIC, an AST-guided framework for translating natural language into first-order logic using large language models (LLMs). Unlike prior work that treats this task as free-form text generation with limited control, NL2LOGIC incrementally constructs the AST in a top-down manner. This approach provides stronger control over the LLM output, achieving near-perfect syntactic correctness, improving semantic accuracy by 30% on LogicNLI, FOLIO, and ProofWriter datasets, and improving existing neuro-symbolic system Logic-LM by 31% on both syntactic and downstream reasoning task accuracy.

Ethical Considerations

Datasets and Models. We rely solely on publicly available datasets and models, without involving human subjects or newly collected data. No personal, private, or sensitive information is used. Therefore, this work poses no risks related to privacy, consent, or data annotation ethics. All datasets and models are utilized in accordance with their respective licenses.

Potential Misleading Proof. We envision that rules generated by NL2LOGIC will be executed using automated reasoning engines such as Z3, PyProver, or SMT-LIB to verify a hypothesis against a set of premises. Although the goal is to obtain a formally provable answer, the proof must be interpreted carefully. Given a set of premises P and a hypothesis Q, $P \models Q$ does not necessarily imply that Q is proven. If $P \models \neg Q$ also holds, the result is *uncertain* instead of *entailment*. If the premises themselves are unsatisfiable (e.g., due to semantic errors during translation), P may entail any statement. Hence, one must examine all possible outcomes, which are $P \models Q$, $P \models \neg Q$, and even the satisfiability of P itself.

Use of AI Assistants. We acknowledge the use of AI assistants for grammar checking. The authors remain fully responsible for the scientific contributions, experimental results, and all claims presented in this paper.

Limitations

Dependencies on Multiple Sentences. NL2LOGIC converts text into first-order logic, one sentence at a time. It ensures consistent predicate arity, constant' names, and variable' names across sentences by prompting the LLM to use the base form of each word, such as the present tense for verbs and objects without modifiers. However, certain sentences require contextual information from adjacent sentences or the entire text to generate accurate logical representations. In such cases, NL2LOGIC does not yet handle co-reference or implicit relational links across sentences. For instance, given the premise "John is the father of Alice" and the hypothesis "John is the parent of Alice," separate translation would yield two distinct relations: `father` and `parent`. With full context, however, the predicate `father` could instead be expressed as a conjunction of `parent` and `male`.

Reliance on Structured Generations. NL2LOGIC relies on structured output capabilities available in both commercial APIs (e.g., OpenAI) and open-source LLM serving frameworks (e.g., vLLM). It prompts the LLM to generate JSON outputs following schemas specific to each parser type. However, not all LLM serving frameworks support this feature natively. For instance, `llama.cpp` requires integration with the `lm-format-enforcer` tool to enable structured output.

Acknowledgments

We would like to thank the anonymous reviewers for their constructive comments and suggestions. This work is supported in part by the National Science Foundation under grant CNS-2442171. Any opinions, findings, and conclusions made in this paper are those of the authors and do not necessarily reflect the views of the funding agencies.

References

Lasha Abzianidze. 2017. Langpro: Natural language theorem prover. *arXiv preprint arXiv:1708.09417*.

V Aho Alfred, S Lam Monica, and D Ullman Jeffrey. 2007. *Compilers principles, techniques & tools*. pearson Education.

Dave Barker-Plummer, Richard Cox, and Robert Dale. 2009. Dimensions of difficulty in translating natural language into first order logic. *International Working Group on Educational Data Mining*.

Clark Barrett, Aaron Stump, Cesare Tinelli, and 1 others. 2010. The smt-lib standard: Version 2.0. In *Proceedings of the 8th international workshop on satisfiability modulo theories (Edinburgh, UK)*, volume 13, page 14.

Stefan Behnel, Robert Bradshaw, Craig Citro, Lisandro Dalcin, Dag Sverre Seljebotn, and Kurt Smith. 2010. Cython: The best of both worlds. *Computing in Science & Engineering*, 13(2):31–39.

Johan Bos and Katja Markert. 2005. Recognising textual entailment with logical inference. In *Proceedings of Human Language Technology Conference and Conference on Empirical Methods in Natural Language Processing*, pages 628–635.

Samuel R. Bowman, Christopher Potts, and Christopher D. Manning. 2015. Recursive neural networks can learn logical semantics. In *Proceedings of the 3rd Workshop on Continuous Vector Space Models and their Compositionality*, pages 12–21, Beijing, China. Association for Computational Linguistics.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, and 12 others. 2020. Language models are few-shot learners. In *Advances in Neural Information Processing Systems*, volume 33, pages 1877–1901. Curran Associates, Inc.

Ruisheng Cao, Su Zhu, Chen Liu, Jieyu Li, and Kai Yu. 2019. Semantic parsing with dual learning. *arXiv preprint arXiv:1907.05343*.

Antonia Creswell, Murray Shanahan, and Irina Higgins. 2022. Selection-inference: Exploiting large language models for interpretable logical reasoning. *Preprint*, arXiv:2205.09712.

Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: an efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, page 337–340, Berlin, Heidelberg. Springer-Verlag.

Herbert B Enderton. 2001. *A mathematical introduction to logic*. Elsevier.

Richard Evans, David Saxton, David Amos, Pushmeet Kohli, and Edward Grefenstette. 2018. Can neural networks understand logical entailment? In *International Conference on Learning Representations*.

Markus Frohmann, Igor Sterner, Ivan Vulić, Benjamin Minixhofer, and Markus Schedl. 2024. Segment any text: A universal approach for robust, efficient and adaptable sentence segmentation. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 11908–11941, Miami, Florida, USA. Association for Computational Linguistics.

Simeng Han, Hailey Schoelkopf, Yilun Zhao, Zhenting Qi, Martin Riddell, Wenfei Zhou, James Coady, David Peng, Yujie Qiao, Luke Benson, and 1 others. 2024. Folio: Natural language reasoning with first-order logic. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 22017–22031.

Nikita Kitaev, Steven Cao, and Dan Klein. 2019. Multilingual constituency parsing with self-attention and pre-training. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 3499–3505, Florence, Italy. Association for Computational Linguistics.

Nikita Kitaev and Dan Klein. 2018. Constituency parsing with a self-attentive encoder. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 2676–2686.

Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. 2022. Large language models are zero-shot reasoners. *Advances in neural information processing systems*, 35:22199–22213.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*.

Junnan Liu. 2025. Few-shot natural language to first-order logic translation via code generation. In *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pages 10939–10960, Albuquerque, New Mexico. Association for Computational Linguistics.

Xuantao Lu, Jingping Liu, Zhouhong Gu, Hanwen Tong, Chenhao Xie, Junyang Huang, Yanghua Xiao, and Wenguang Wang. 2022. Parsing natural language into propositional and first-order logic with dual reinforcement learning. In *Proceedings of the 29th International Conference on Computational Linguistics*, pages 5419–5431.

Mitchell Philip Marcus. 1978. *A theory of syntactic recognition for natural language.* Ph.D. thesis, Massachusetts Institute of Technology.

W. McCune. 2005–2010. Prover9 and mace4. `http://www.cs.unm.edu/~mccune/prover9/`.

Benjamin Minixhofer, Jonas Pfeiffer, and Ivan Vulić. 2023. Where's the point? self-supervised multilingual punctuation-agnostic sentence segmentation. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 7215–7235, Toronto, Canada. Association for Computational Linguistics.

Terufumi Morishita, Gaku Morio, Atsuki Yamaguchi, and Yasuhiro Sogawa. 2024. Enhancing reasoning capabilities of llms via principled synthetic logic corpus. *Advances in Neural Information Processing Systems*, 37:73572–73604.

Theo Olausson, Alex Gu, Ben Lipkin, Cedegao Zhang, Armando Solar-Lezama, Joshua Tenenbaum, and Roger Levy. 2023. LINC: A neurosymbolic approach for logical reasoning by combining language models with first-order logic provers. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 5153–5176, Singapore. Association for Computational Linguistics.

Liangming Pan, Alon Albalak, Xinyi Wang, and William Wang. 2023. Logic-LM: Empowering large language models with symbolic solvers for faithful logical reasoning. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pages

3806–3824, Singapore. Association for Computational Linguistics.

Xin Quan, Marco Valentino, Louise A. Dennis, and Andre Freitas. 2024. Verification and refinement of natural language explanations through LLM-symbolic theorem proving. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 2933–2958, Miami, Florida, USA. Association for Computational Linguistics.

Xin Quan, Marco Valentino, Louise A. Dennis, and Andre Freitas. 2025. Faithful and robust LLM-driven theorem proving for NLI explanations. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 17734–17755, Vienna, Austria. Association for Computational Linguistics.

Federico Raspanti, Tanir Ozcelebi, and Mike Holenderski. 2025. Grammar-constrained decoding makes large language models better logical parsers. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 6: Industry Track)*, pages 485–499.

Tim Rocktaschel, Edward Grefenstette, Karl Moritz Hermann, Tomas Kocisky, and Phil Blunsom. 2016. Reasoning about entailment with neural attention. In *International Conference on Learning Representations (ICLR)*.

Soumya Sanyal, Harman Singh, and Xiang Ren. 2022. FaiRR: Faithful and robust deductive reasoning over natural language. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1075–1093, Dublin, Ireland. Association for Computational Linguistics.

Oyvind Tafjord, Bhavana Dalvi, and Peter Clark. 2021. ProofWriter: Generating implications, proofs, and abductive statements over natural language. In *Findings of the Association for Computational Linguistics: ACL-IJCNLP 2021*, pages 3621–3634, Online. Association for Computational Linguistics.

Jidong Tian, Yitian Li, Wenqing Chen, Liqiang Xiao, Hao He, and Yaohui Jin. 2021. Diagnosing the first-order logical reasoning ability through logicnli. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 3738–3747.

Bailin Wang, Zi Wang, Xuezhi Wang, Yuan Cao, Rif A Saurous, and Yoon Kim. 2023. Grammar prompting for domain-specific language generation with large language models. *Advances in Neural Information Processing Systems*, 36:65030–65055.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, and 1 others. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837.

William A Woods. 1970. Transition network grammars for natural language analysis. *Communications of the ACM*, 13(10):591–606.

Jundong Xu, Hao Fei, Liangming Pan, Qian Liu, Mong-Li Lee, and Wynne Hsu. 2024. Faithful logical reasoning via symbolic chain-of-thought. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 13326–13365, Bangkok, Thailand. Association for Computational Linguistics.

Yuan Yang, Siheng Xiong, Ali Payani, Ehsan Shareghi, and Faramarz Fekri. 2024. Harnessing the power of large language models for natural language to first-order logic translation. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 6942–6959, Bangkok, Thailand. Association for Computational Linguistics.

Luke S Zettlemoyer and Michael Collins. 2012. Learning to map sentences to logical form: Structured classification with probabilistic categorial grammars. *arXiv preprint arXiv:1207.1420*.

A System Prompts

The parser, described in Section 4 and illustrated in Fig. 1, iteratively constructs the abstract syntax tree (AST) by recursively analyzing each sentence component through specialized submodules: `ParserSelector`, `AtomicSentenceParser`, `QuantifiedSentenceParser`, and `LogicalSentenceParser`. The `ParserSelector` determines the appropriate parser for a sentence, while the remaining three parsers perform parsing and recursively invoke the `ParserSelector` for any child nodes. The system prompt for `ParserSelector` is shown in Fig. 7. While the system prompt for the `QuantifiedSentenceParser` is shown in Fig. 8. The `LogicalSentenceParser` involves multiple system prompts, as shown in Figs. 9 and 10. `AtomicSentenceParser` also involves multiple system prompts, as shown in Figs. 11 to 15.

ParserSelector System Prompt

```
You are an expert on classifying the sentence
by its overall structure:
A = An atomic logical statement. (no
quantifiers, no logical connectives, no
negation)
B = A quantified logical statement when the
sentence talks about a general rule that
covers many entities.
If the sentence only mentions specific proper
names, or an instances of variable (x,y,z), or
if quantifiers appear only inside part of the
sentence, then it should be classified as A,
C, or D instead.
C = A compound logical sentence, where each
part is connected with logical connectives
such as 'and', 'or', 'if...then', or 'only
if'.
D = A statement that contains literal negation
of another sentence ('not', 'no', 'dont',
'doesnt'). Only look for a literal negation.

Example 1:
Sentence: 'Alice sings.'
Answer: { "answer" : "A"}
...
Now, it is your turn

Input: {input sentence}
Answer:
```

Figure 7: The system prompt for selecting the appropriate parser. The first step in parsing a sentence is to classify whether it is an atomic, quantified, compound, or negation logical sentence.

QuantifiedSentenceParser System Prompt

```
You identified the sentence as a quantified
logical statement.


Task:
1. Select the correct quantifier:
- ForAll (e.g., all, every, each, no one) ->

the logical statement applies to ALL entities
- ThereExists (e.g., some, there is, at least
one, a) -> the logical statement applies to

SOME entities
2. Identify the variable (and all reference)
being quantified (the noun phrase that follows

the quantifier, e.g., "student", "person",
"dog") and replace it with a letter like x, y,
or z.

3. Rewrite the sentence WITHOUT the quantifier,
keeping the variable in place so the sentence
is still natural and understandable. Preserve

the exact wording and capitalization of all
subject and object names. If there is multiple
quantifier, just remove the outermost.

4. If the sentence is ambiguous, you should
rephrase it so that the next parser will
understand whether it is an atomic logical

sentence, or logical sentence with connectives,
or a quantified logical sentence.


Examples:

Input: "All students study hard."

Output: { "quantifier" : "ForAll", "variable"
: "x", "sentence_without_quantifier" : "x
study hard."}


...


Now, it is your turn

Input: {input sentence}

Output:
```

Figure 8: The system prompt for parsing a quantified logical sentence. It instructs the LLM to extract the quantifier, variable, and the scoped logical sentence.

LogicalSentenceParser System Prompt

```
You parse a sentence into its OUTERMOST (top-level) logical operator and its two operands. Always choose the operator that

governs the entire sentence (outermost scope). Do not parse nested or inner operators here. Each operand, left and right, are a standalone

and complete sentence, not just a phrase, meaning it has at least subject and verb/to be. Output JSON matching:

operator: one of ["Not","And","Or","If", "OnlyIf","IfAndOnlyIf"]
left_operand: rewrite the left part as a clean,

standalone clause. Preserve the exact wording and capitalization of all subject and object names. But, resolve co-reference such as she,

he, it, etc.
right_operand: rewrite the right part as a clean, standalone clause. Preserve the exact

wording and capitalization of all subject and object names. But, resolve co-reference such as she, he, it, etc.


Decide by the main structure of the whole sentence:

- And: two clauses joined by and.
- Or: two clauses joined by or / either ... or.
- If: conditional "if ... then ..."

(antecedent = left, consequent = right).
- OnlyIf: "P only if Q" (left = P, right = Q; Q is required for P).

- IfAndOnlyIf: "iff / if and only if / exactly when / just in case".


Examples:

Input: "Alice sings and dances." Output:

{"operator": "And", "left_operand": "Alice sings", "right_operand": "Alice dances"}


...
Now, it is your turn


Input: {input sentence}
Output:
```

Figure 9: The system prompt for parsing a sentence involving a binary logical operator. It instructs the LLM to extract the operator and its two operands.

LogicalSentenceParser System Prompt

```
You parse a sentence whose top-level operator is unary negation.


Output JSON matching:
operator: always "Not"
operand : rewrite the sentence without the

outermost negation

Guidelines:

- Always set operator = "Not".
- For the operand, remove only the **outermost** negation.

- If there are multiple negations, strip just the outermost one and keep the inner ones.
- Rewrite the operand as a natural, grammatical

sentence.
- Do not add explanations or extra words.
- Return JSON only.


Examples:


Input: "It is not raining."
Output: {"operator": "Not", "operand": "It is raining"}


Input: "No student is absent."
Output: {"operator": "Not", "operand": "a

student is absent"}

Input: "It is not true that John is

not guilty." Output: {"operator": "Not", "operand": "John is not guilty"}


Input: "Nobody loves me." Output: {"operator": "Not", "operand": "Somebody loves me"}


Now, it is your turn

Input: {input sentence}

Answer:
```

Figure 10: The system prompt for parsing a sentence involving a negation operator. It instructs the LLM to extract the sentence excluding the negation.

AtomicSentenceParser System Prompt

```
You classify an ATOMIC natural-language predicate into one of:
A = Adjective/property

B = Intransitive verb (takes no object)
C = Transitive verb (takes exactly one object)
D = Ditransitive verb (takes two objects)

Examples:


Input: "Alice is tall."
Output: {"answer":"A"}


Input: "Alice is a student."
Output: {"answer":"A"}


Input: "Alice runs."
Output: {"answer":"B"}


Input: "The baby sleeps."
Output: {"answer":"B"}



...
Now, it is your turn

Input: {input sentence}
Output:
```

Figure 11: The system prompt for parsing an atomic logical sentence. It first instructs the LLM to determine whether the sentence involves an adjective, intransitive verb, transitive verb, or ditransitive verb predicate.

AtomicSentenceParser System Prompt

```
You extract the object and its adjective property from a simple atomic sentence.
Output JSON matching:

adjective: the describing word or phrase. use the base form with no modifier
obj: the entity being described (keep wording

and capitalization verbatim).  use the base form with no modifier


Rules:
-  Handle only atomic adjective/property sentences (e.g., "Alice is tall", "The dog is

happy").
- Subjects must be individual names or noun phrases without quantifiers ("all", "every",

"some", "no").
- Adjective may be single word or short phrase (e.g., "absent", "very tall").

- Do not paraphrase or change casing; copy terms exactly.
- Ignore tense/negation; just extract subject

and adjective.

Examples:


Input: "Alice is tall."
Output: {"adjective": "tall", "obj": "Alice"}


Input: "student is awesome."
Output:   {"adjective":   "awesome",   "obj":

"student"}

Input: "Bob is very tired."

Output:  {"adjective":  "very tired",  "obj":
"Bob"}



Now, it is your turn


Input: {input sentence}
Answer:
```

Figure 12: The system prompt for parsing an atomic logical sentence involving a subject and an adjective. It instructs the LLM to extract the subject and the adjective predicate.

AtomicSentenceParser System Prompt

```
You extract the subject and the main intransitive verb from a simple atomic sentence.

Output JSON matching:
verb: the main intransitive verb (copy exactly

as in the input). use the verb base form
subject: the entity performing the action (keep wording and capitalization verbatim)

Rules:
- Handle only atomic intransitive sentences (a

subject + intransitive verb, with no object, no quantifier, no negation).
- Subject is a proper name or noun phrase

(e.g., "Alice", "The student").
- Verb must appear exactly as written in the sentence (respect tense/aspect: "runs", "is

running", "slept").
- Do not paraphrase or alter capitalization.
- Sentences with objects, quantifiers, or

negation are out of scope.

Examples:

Input: "Alice runs."
Output: {"verb": "run", "subject": "Alice"}

Input: "The student sleeps."
Output: {"verb": "sleep", "subject": "The

student"}

Input: "Bob is running."

Output: {"verb": "run", "subject": "Bob"}

Input: "Alice swam."

Output: {"verb": "swim", "subject": "Alice"}

Now, it is your turn

Input: {input sentence}
Answer:
```

Figure 13: The system prompt for parsing an atomic logical sentence involving a simple subject and intransitive verb structure. It instructs the LLM to extract the subject and the verb predicate.

AtomicSentenceParser System Prompt

```
You extract the subject, the main transitive verb, and its single object from a simple atomic sentence.

Output JSON matching:
subject: the entity performing the action

(copy wording and capitalization verbatim)
verb: the main transitive verb. use the verb base form

obj: the object of the verb (copy wording and capitalization verbatim). use the base form in infinitive form


Examples:

Input: "Alice loves Bob."
Output: {"subject": "Alice", "verb": "love",

"obj": "Bob"}

Input: "The student reads a book."

Output: {"subject": "The student", "verb": "read", "obj": "a book"}


Input: "Bob is watching TV."
Output: {"subject": "Bob", "verb": "watch", "obj": "TV"}

Input: "Mary wrote a letter."
Output: {"subject": "Mary", "verb": "write",

"obj": "a letter"}

Input: "John loves swimming."

Output: {"subject": "John", "verb": "love", "obj": "to swim"}


Input: "Doe likes to read a book"
Output: {"subject": "Doe", "verb": "like", "obj": "to read a book"}


Now, it is your turn


Input: {input sentence}
Answer:
```

Figure 14: The system prompt for parsing an atomic logical sentence involving a subject and a transitive verb. It instructs the LLM to extract the subject, the verb predicate, and the object.

AtomicSentenceParser System Prompt

```
You extract the subject, the main ditransitive
verb, its indirect object, and its direct
object from a simple atomic sentence.

Output JSON matching:
subject: the entity performing the action

(copy wording and capitalization verbatim)
verb: the main ditransitive verb. use the base
verb form

indirect_obj: the recipient/beneficiary of
the action (copy wording and capitalization
verbatim). use the infinitive form if needed

direct_obj: the thing being
given/sent/shown/etc. (copy wording and
capitalization verbatim). use the infinitive

form if needed

Examples:

Input: "John gave Mary a book."
Output: {"subject": "John", "verb": "give",

"indirect_obj": "Mary", "direct_obj": "a
book"}


Input: "Alice sent Bob a letter."
Output: {"subject": "Alice", "verb": "send",
"indirect_obj": "Bob", "direct_obj": "a

letter"}
Input: "The teacher showed the students
a picture." Output: {"subject": "The

teacher", "verb": "show", "indirect_obj":
"the students", "direct_obj": "a picture"}


Input: "John gave a book to Mary."
Output: {"subject": "John", "verb": "give",
"indirect_obj": "Mary", "direct_obj": "a

book"}

...

Now, it is your turn

Input: {input sentence}

Answer:
```

Figure 15: The system prompt for parsing an atomic logical sentence involving a subject and a ditransitive verb. It instructs the LLM to extract the subject, the verb predicate, the direct object, and the indirect object.