Can LLMs Solve ASP Problems? Insights from a Benchmarking Study

Lin Ren[1,2], **Guohui Xiao**[1,2*], **Guilin Qi**[1,2], **Yishuai Geng**[1,2], **Haohan Xue**[1,2]

[1]School of Computer Science and Engineering, Southeast University, Nanjing, China
[2]Key Laboratory of New Generation Artificial Intelligence Technology and Its Interdisciplinary Applications (Southeast University), Ministry of Education, China
{renlin, guohui.xiao, gqi, ysgeng, thex1ay}@seu.edu.cn

Abstract

Answer Set Programming (ASP) is a powerful paradigm for non-monotonic reasoning. Recently, large language models (LLMs) have demonstrated promising capabilities in logical reasoning. Despite this potential, current evaluations of LLM capabilities in ASP are often limited. Existing works normally employ overly simplified ASP programs, do not support negation, disjunction, or multiple answer sets. Furthermore, there is a lack of benchmarks that introduce tasks specifically designed for ASP solving. To bridge this gap, we introduce ASPBench, a comprehensive ASP benchmark, including three ASP specific tasks: ASP entailment, answer set verification, and answer set computation. Our extensive evaluations on ASPBench reveal that while 14 state-of-the-art LLMs, including *deepseek-r1*, *o4-mini*, and *gemini-2.5-flash-thinking*, perform relatively well on the first two simpler tasks, they struggle with answer set computation, which is the core of ASP solving. These findings offer insights into the current limitations of LLMs in ASP solving. This highlights the need for new approaches that integrate symbolic reasoning capabilities more effectively. The code and dataset are available at https://github.com/HomuraT/ASPBench.

1 Introduction

Answer Set Programming (ASP) (Gelfond and Lifschitz 1991; Niemelä 1999; Dantsin et al. 2001) is a declarative programming paradigm designed for complex knowledge representation and problem-solving tasks. A key strength of ASP is its ability to model situations where conclusions must be revised as new information becomes available (Ginsberg 1980; Reiter 1988), enabling an adaptive and context-sensitive inference process. ASP provides a robust computational framework for various forms of logical reasoning, including default reasoning (Reiter 1980), and connects with abductive inference (Josephson and Josephson 1996), belief revision (Darwiche and Pearl 1997), stream reasoing (Beck, Dao-Tran, and Eiter 2018), and query answering (Wan et al. 2020).

Recently, large language models (LLMs) have demonstrated remarkable capabilities in diverse areas, such as information retrieval (Zhu et al. 2023), question answering (Yue et al. 2025), and code generation (Jiang et al. 2024). Furthermore, their potential in logical reasoning tasks is an active growing area of research (Liu et al. 2025). The question of whether LLMs possess logical reasoning ability, and if so, to what extent, has been widely explored (Huang and Chang 2023; Wang et al. 2024b), particularly in formalisms such as Description Logics (e.g., DL-Lite) (Wang et al. 2024a), Propositional Logic (Chan, Gaizauskas, and Zhao 2025; Chen et al. 2024a), and First-Order Logic (Chen et al. 2024a; Wang et al. 2024b). Beyond these monotonic formalisms, ASP offers a powerful framework for non-monotonic reasoning (NMR). Consequently, recent works increasingly leverage ASP to evaluate the NMR abilities of LLMs. Xiu, Xiao, and Liu (2022) created the dataset LogicNMR for evaluating default logic rules, which effectively only covers a simple ASP fragment. Parmar et al. (2024) introduced LogicBench for evaluating logical reasoning capabilities of LLMs, notably featuring patterns from non-monotonic logic (such as default reasoning). These works focus on evaluating ability of LLMs to perform symbolic reasoning. In contrast, Rudinger et al. (2020) and Allaway et al. (2023) explore reasoning with defeasible information from the implicit background knowledge of LLMs.

However, to the best of our knowledge, there are no benchmarks systematically evaluating the capabilities of LLMs for ASP. Previous studies have overlooked several key factors: *(1)* They often employ overly simplified ASP programs, for example, using predominantly unary predicates, focusing mainly on (stratified) default negation, and typi-

Dataset	Arity	Repr.	Ops.	MAS	ASE	ASV	ASC
δ-NLI	N/A	Tex	DN	N/A	×	×	×
ProofWriter	1	Tex	SN	N/A	✓	×	×
ruletaker	N/A	Tex	SN	N/A	✓	×	×
LogicNMR	1	Tex	SN, DN	Single	✓	×	×
generics-exemplars	N/A	Tex	N/A	N/A	×	×	×
LogicBench	1	Tex	SN, DN	Single	✓	×	×
ASPBench (Ours)	**Any**	**Sym & Tex**	**SN, DN, Disj, Cons**	**Multiple**	✓	✓	✓

Table 1: Comparison of ASPBench with other datasets for ASP reasoning. Columns show: Arity (predicate arity); Repr. (Symbolic/Textual representation); Ops. (Supported ASP Operations: SN - Strong Negation, DN - Default Negation, Disj - Disjunction, Cons - Constraints); MAS (Multiple Answer Sets support). Compared datasets are detailed in §2.

*Corresponding author

cally considering only a single answer set scenario. *(2)* The significant impact of predicate semantics (symbolic vs. descriptive names) on LLM reasoning in ASP—a key differentiator from traditional solvers—remains largely unexplored. *(3)* Previous benchmarks do not design evaluation tasks specifically for ASP.

To fill these gaps, we propose a novel benchmark for ASP, referred to as ASPBench. ASPBench features a novel framework for the automated generation of diverse ASP problems with varied rule styles and logical operations. The benchmark itself incorporates support for multiple answer sets, a broader range of ASP operators, and introduces three distinct ASP evaluation tasks: ASP entailment (ASE), answer set verification (ASV), and answer set computation (ASC). In addition to synthetic data, ASPBench includes real-world ASP programs collected from public sources. The differences between ASPBench and related datasets, in the context of evaluating ASP style reasoning, are shown in Table 1.

Our extensive experiments on ASPBench, evaluating 14 state-of-the-art LLMs, reveal several key insights: *(1)* **The characteristics of ASP programs significantly influence LLM performance**. LLMs struggle with complex tasks such as ASC on synthetic ASP programs. This challenge increases when dealing with real-world ASP programs, where performance significantly drops across all evaluated tasks. Furthermore, LLM reasoning is highly sensitive to program features, including their syntactic properties (e.g., whether they are positive, stratified, or head-cycle-free) and the number of answer sets. *(2)* **The inherent properties of LLMs and their inference strategies are vital for their ability to solve ASP**. Larger LLMs generally achieve higher performance while producing shorter outputs. Conversely, for LLMs with comparable parameters, the length of chain-of-thought tokens strongly correlates with ASP solving abilities, underscoring the value of increased test-time computation.

2 Related Work

This work is situated within the context of research on LLMs for logical reasoning, their application as logic solvers or code executors, and existing benchmarks for ASP solving.

2.1 Logical Reasoning with LLMs

Recently, LLMs have shown a powerful ability in various monotonic logical reasoning tasks, such as Multi-Step Reasoning (Saha et al. 2023; Fu et al. 2023) and Commonsense Reasoning (Tian, Zhang, and Peng 2023; Perak, Beliga, and Meštrović 2024). However, LLMs also exhibit notable limitations in reasoning tasks. Wang et al. (2024b) showed that LLM understanding of fundamental reasoning rules lags significantly behind human capability. Ishay, Yang, and Lee (2023) demonstrated the potential of LLMs in generating complex ASP programs through few-shot prompting, but most errors require manual correction. Srivatsa and Kochmar (2024) explored the challenges LLMs face in solving math word problems, while Li et al. (2024) demonstrated that LLMs perform considerably worse than neural program induction systems in reasoning tasks. Wang et al. (2024a) illustrated that LLMs struggle with understanding TBox NI transitivity rules. Parmar et al. (2024) showed that LLMs do not perform well in logic reasoning, even though they are in single inference rule scenarios.

2.2 LLMs as Logic Solvers or Code Executors

Recently, code has been recognized as a powerful tool for LLMs to access and leverage external sources (Yang and others 2024). Meanwhile, there has been growing interest in exploring the role of LLMs as logic solvers and code executors. For example, Feng et al. (2023) utilized LLMs as Prolog logic solvers to address parsing errors in logic programs. Similarly, Chen et al. (2024b) explored how to guide LLMs in simulating logic solvers to execute Propositional Logic or Satisfiability Modulo Theories (SMT) programs, using natural language, Z3Py (Moura and Bjørner 2008), or SMT-LIB (Barrett, Stump, and Tinelli 2010). Additionally, Wang et al. (2024c) demonstrated that LLMs can serve as executors when generated Z3 programs fail during execution, and Lyu et al. (2024) explored the feasibility of using LLMs as Python code executors. Our work focuses on leveraging LLMs as ASP solvers.

2.3 Benchmarks for ASP Solving with LLMs

To evaluate the ASP solving ability of language models, several benchmarks have been proposed. δ-NLI (Rudinger et al. 2020) was introduced for non-monotonic inference by assessing belief changes with new information. LogicNMR (Xiu, Xiao, and Liu 2022) provides a dataset of textual ASP samples. generics-exemplars (Allaway et al. 2023) focused on reasoning about generics and their exceptions. For broader logical reasoning, ProofWriter (Tafjord, Dalvi, and Clark 2021) was developed for generating natural language proofs, and RuleTakers (Clark, Tafjord, and Richardson 2021) were created for emulating reasoning over textual rules. Leidinger, Rooij, and Shutova (2024) also explored belief stability in generics. Additionally, LogicBench (Parmar et al. 2024) provides benchmarks for logical reasoning, encompassing some non-monotonic scenarios (Santos et al. 2024). Borroto, Kareem, and Ricca (2024) introduced NL2ASP, a two-step architecture that demonstrates the potential for automated symbolic ASP program generation from natural language specifications. Our work focuses on a comprehensive evaluation of ASP solving ability of LLMs, using more complex synthetic ASP programs and real-world ASP programs.

3 Preliminary

3.1 Answer Set Programming

In this work, we employ the framework of Answer Set Programming (ASP) (Gelfond and Lifschitz 1988; Gelfond and Lifschitz 1991). An ASP program is a set of rules of the following form:

$$\omega_1(\mathbf{x}_1)|\ldots|\omega_k(\mathbf{x}_k) \leftarrow \alpha_1(\mathbf{x}_1),\ldots,\alpha_m(\mathbf{x}_m), \text{not } \alpha_{m+1}(\mathbf{x}_{m+1}),...,\text{not } \alpha_n(\mathbf{x}_n)$$

where each $\omega_i(\mathbf{x}_i)$ is an atom and each $\alpha_j(\mathbf{x}_j)$ is a literal of the form $\mathrm{p}(\mathbf{x}_j)$ (positive literal) or $-\mathrm{p}(\mathbf{x}_j)$ (negative literal),

and each $\mathbf{x}_i$ or $\mathbf{x}_j$ consists of variables and constants. In ASP, "not" and "−" are called the default negation (negation-as-failure) and the classical negation (strong negation), respectively. An ASP program or a rule is ground if there are no variables. A fact is a ground rule with $n = 0$. We often write an ASP problem as a pair (W, D) with W a set of facts, and D a set of rules.

The semantics of ASP are characterized by the notion of answer sets, also known as stable models (Gelfond and Lifschitz 1988; Wan et al. 2020). The semantics of an ASP program $P = (W, D)$ is defined via the Gelfond-Lifschitz transformation. Let S be a candidate set of ground atoms. The reduct P^S of the program P with respect to S is obtained from the set of all ground instances of rules in $D \cup W$ by:

(1) Deleting each rule r such that its body contains a default literal not α and $\alpha \in S$.

(2) Deleting all default literals not α from the bodies of the remaining rules.

The resulting program P^S is a positive (default-negation-free) disjunctive logic program. A set S is an answer set of P if and only if S is a minimal model of P^S. A model M of P^S is minimal if there is no model $M' \subset M$ of P^S. In this context, facts from W are treated as rules with empty bodies.

The ASP paradigm has been implemented in several ASP solvers, e.g., DLV (Alviano et al. 2017) and Clingo (Gebser et al. 2012).

3.2 Syntactic Classes

In this work, we are interested in the following syntactic classes of ASP programs based on their properties:

Positive Program: A program is *positive* if it contains *only* positive atoms (i.e., neither strong negation nor default negation occurs).

Stratified Program: A program is *stratified* when its dependency graph has no directed cycle that traverses a negative edge, i.e. default negation is never involved in recursion (Apt, Blair, and Walker 1988). Positive and strong literals may still form cycles. For examples, *P1* = {`-p :- -q.  -q :- -p.`} is stratified, because it has no recursion through default negation; *P2* = {`p :- not q.  q :- not p.`} is *unstratified* , because the cycle between `p` and `q` goes through default negation.

Head-Cycle-Free (HCF) Program: A program is *head-cycle-free* when its dependency graph is acyclic, i.e., it contains no directed cycle consisting solely of positive or negated atoms (Ben-Eliyahu and Dechter 1994). The presence of multiple atoms in a disjunctive head is allowed; what is prohibited is any positive recursion. For examples, *P3* = {`a | b :- d.  c :- a.  c :- b.  d.`} is HCF, because no positive cycle exists; *P4* = {`a | b :- c.  c :- a.  c :- b.`} is *not* HCF, because the positive dependency graph contains the cycle `a` → `c` → `a` (and similarly for `b`), i.e. a positive recursion.

3.3 Task Definitions

We define three distinct downstream tasks to evaluate the reasoning capabilities of models on ASP solving. These tasks cover different aspects of ASP reasoning:

(1) **ASP Entailment (ASE):** Given an ASP program P and a ground atom `a`. The task is to determine the truth state of `a` within the answer set S. The expected output is one of three possibilities: `true` (if `a` $\in S$), `false` (if `-a` $\in S$), or `unknown` (if neither `a` $\in S$ nor `-a` $\in S$).

(2) **Answer Set Verification (ASV):** Given an ASP program P (guaranteed to have one or more answer sets, potentially generated using disjunction in rule heads) and a candidate set of ground atoms C, the task is to determine whether C is an actual answer set of P.

(3) **Answer Set Computation (ASC):** Given an ASP program P, the task is to compute and return one of its correct answer sets.

4 ASPBench

We introduce ASPBench through a systematic three-stage generation pipeline (Figure 1), which enables precise control over sample complexity and diversity. *(1)* **ASP Graph Construction** (§ 4.1, 4.2): We construct an ASP Graph that represents the logical dependency structure. This graph-based representation allows us to control key properties such as reasoning depth and structural complexity. *(2)* **ASP Rule Generation** (§ 4.3): We transform the ASP Graph into concrete, syntactically valid ASP rules.s *(3)* **ASPBench Construction** (§ 4.4, 4.5): We construct the symbolic benchmark tailored to our three target tasks (i.e., ASP Entailment, ASP Verification, and ASP Computation) and generate the corresponding textual samples.

This section mainly introduces the architecture and key steps. The detailed procedures, hyperparameters, and other specific contents are provided in Appendix A.

4.1 Definition of ASP Graph

Similar to the Rete algorithm (Forgy 1989), we use a graph structure to represent a logical program. An **ASP Graph** is a directed graph that serves as a structured and detailed representation of an ASP program.

The graph utilizes two types of nodes: **Rule Nodes** (N_R) represent the individual rules within the ASP program. Each rule node corresponds to a specific rule. **Predicate Nodes** (N_P) represent the unique predicates used in the program. Each predicate node corresponds to a distinct predicate.

Moreover, four types of edges define the logical operations between nodes: *(1)* **Default (P):** The predicate is not subject to any negation operators. *(2)* **Strong Negation (SN):** The predicate is subject to strong negation. *(3)* **Default Negation (DN):** The predicate is subject to default negation. *(4)* **Combined Negation (SN&DN):** The predicate is subject to both default and strong negation. The term "ASP Graph" in this work refers to this Rule-Predicate-Operation Graph.

4.2 ASP Graph Construction

The initial step in the framework is ASP Graph Construction, which aims to generate an ASP Graph according to hyperparameters (e.g., number of rules, predicates, arity, etc.). This process, illustrated in the top-left panel of Figure 1,

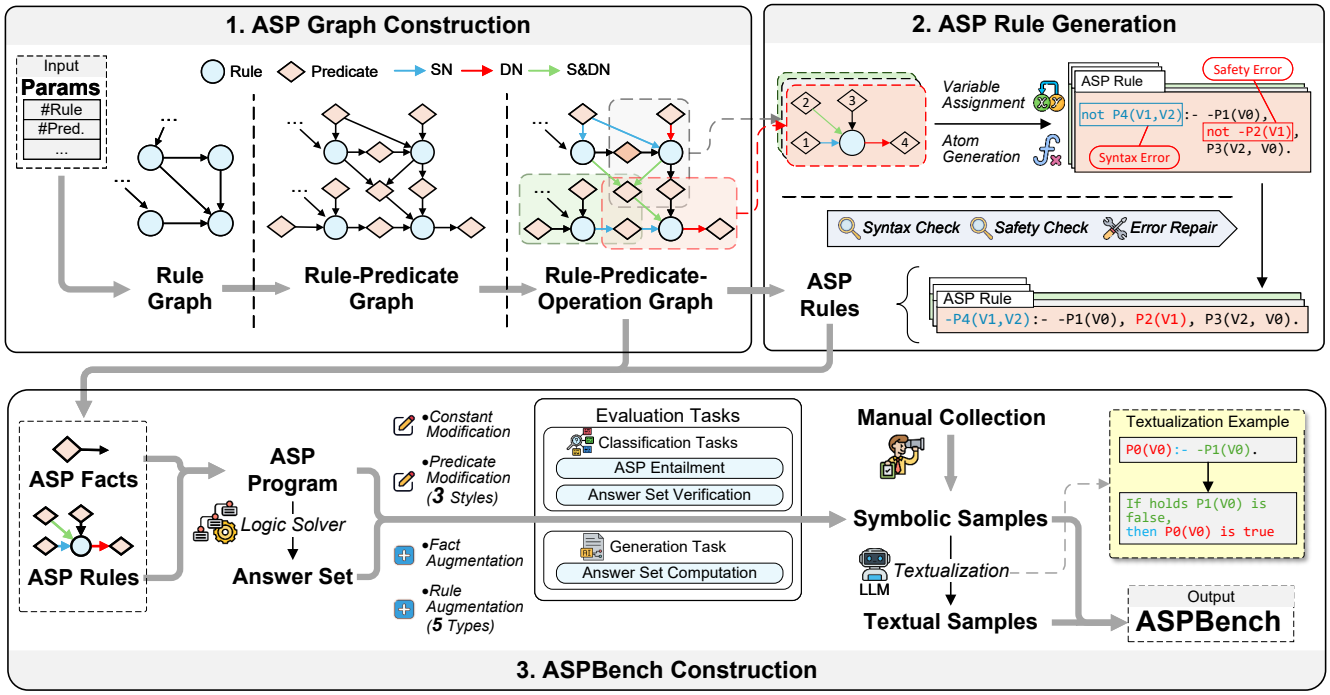

Figure 1: The generation framework for ASPBench

unfolds in three stages, progressively adding detail to the graph:

Rule Graph: This initial graph consists of a predefined number of rule nodes (N_R). Directed edges are then randomly generated between these nodes, up to a specified total number of edges. This random generation is guided by several constraints: *(a)* the resulting graph must be a directed acyclic graph with a single terminal node (no outgoing edges), and *(b)* every rule node must have at least one connecting edge. An edge $N_R^i \rightarrow N_R^j$ in this graph represents a dependency, where the head predicate of rule N_R^i appears in the body of rule N_R^j.

Rule-Predicate Graph: This stage refines the Rule Graph by introducing predicate nodes (N_P) and associated edges. First, specific predicates are introduced:

- For each edge $N_R^i \rightarrow N_R^j$ in the Rule Graph, a predicate N_P^k is inserted, forming $N_R^i \rightarrow N_P^k \rightarrow N_R^j$.
- For each source rule node N_R^{source} (no incoming $N_R \rightarrow N_R$ edges), an input predicate N_P^{in} is added with an edge $N_P^{in} \rightarrow N_R^{source}$.
- For the terminal rule node N_R^{term} (no outgoing $N_R \rightarrow N_R$ edges), an output predicate N_P^{out} is added with an edge $N_R^{term} \rightarrow N_P^{out}$.

Subsequently, the graph is extended by creating a specified number of new predicate nodes and edges.

Rule-Predicate-Operation Graph: The final stage refines the Rule-Predicate Graph by randomly assigning types to edges based on the probability distribution. The types of edges are P, SN, DN, or SN&DN. To ensure the derivability of the entire graph structure and facilitate quantitative analysis (e.g., controlling reasoning chain length), we set the incoming and outgoing edges to a unified type for each predicate node during generation.

Furthermore, to ensure structural diversity and avoid redundancy in the dataset, we keep only one sample from those with similar graph structures.

4.3 ASP Rule Generation

Following the construction of the Rule-Predicate-Operation Graph, the ASP Rule Generation step translates the graph into concrete ASP rules. As depicted in the top-right panel of Figure 1, this process involves several stages for each rule node in the graph:

(1) **Input Collection:** For a given rule node in the graph, all adjacent predicate nodes and the corresponding edge information (both type and direction) are gathered. This information determines which predicates constitute the head and body of the rule and the logical operations involved.

(2) **Initial Rule Formulation:** Based on the collected information, two sub-steps occur:

Variable Assignment: This step assigns appropriate variables to the atoms, determining the arity (n-ary) of each predicate. The initial arity (n-ary) for each predicate is randomly assigned within a predefined range. Once assigned, the arity of a predicate is maintained consistently whenever it appears in the program. If a predicate has been assigned variables previously, its arity is maintained. Additionally, to ensure the rationality of the generated rules, we ensure that the variables of one predicate satisfy the following two con-

ditions: *(a)* At least one of the variables must occur in the other predicates (either the body or the head). *(b)* All head predicate variables must be present in the body.

Atom Generation: This step converts predicate nodes into ASP atoms along with the assigned variables, using P-style identifiers (e.g., "`P1(V0)`").

These atoms are then assembled into an ASP rule based on the edge directions and types. If a rule node in the graph has multiple outgoing edges representing potential head predicates, the generation process can be adjusted according to the requirements of the task. By default, when generating programs intended for the ASE task, which requires a unique answer set, disjunctions are avoided and multiple rules with the same body are produced (e.g., "`a :- b.`", "`c :- b.`"). Conversely, for ASV and ASC tasks involving multiple answer sets, rules with disjunctions (e.g., "`a | c :- b.`") will be generated. Figure 1 shows an example of an ASP rule.

(3) **Error Checking and Repair:** The initial ASP rules may contain errors. Therefore, we carry out the following checks and repairs:

Syntax Check: Ensuring that the rule adheres to the syntax regulations of ASP (e.g., "`not`" cannot appear in the rule head).

Safety Check: Verifying that the rule is safe, meaning every variable appearing in the head or in a default negated literal also appears in a body literal that is not default negated (i.e., a positive or strongly negated literal).

Error Repair: The ASP rules are checked for syntax and safety errors using DLV2. If any errors are detected, an automated repair process will be carried out. This process attempts to make the rule valid by making minimal modifications, by applying a "negation flip" to predicate atoms. This operation simultaneously toggles both the strong negation status and the default negation status of an atom. For instance, applying this flip to "`not p`" changes it to "`-p`" ; similarly, applying it to "`not -p`" changes it to "`p`". The repair process iteratively modifies predicate negations, attempting progressively more changes until the rule passes the check. If a valid rule cannot be obtained after a set number of attempts, the sample generation for this specific rule will be discarded.

This step produces a set of syntactically correct and safe ASP rules, derived from the graph representation and ready for the ASPBench Construction phase.

4.4 ASPBench Construction

The final step, ASPBench Construction, synthesizes the symbolic and textual ASP samples, as illustrated in the bottom panel of Figure 1. This process involves four stages:

(1) **ASP Program Formulation:** Initial ASP facts are generated from input predicate nodes (those lacking incoming edges). The truth value ("`p.`" or "`-p.`") of a fact is determined by the unified type of its predicate node: types **P** and **SN** directly yield "`p.`" and "`-p.`" respectively. If the type includes default negation (**DN** or **SN&DN**), a "negation flip" operation is applied to determine the final truth value.

(2) **Diversification and Augmentation:** To enhance dataset diversity, several modifications are applied:

Fact and Rule Augmentation: To further enhance the diversity and complexity, we introduce 5 distinct types of additional rules to the ASP program. These rules are generated based on the target predicates (predicate nodes without outgoing edges) P_t, other predicates (predicate nodes with outgoing edges) P_o, or newly introduced predicates P_n. The structures of the additional rules are as follows: *(a)* The rule head contains predicates from P_t. *(b)* The rule body contains predicates from P_t. *(c)* Both the head and body consist solely of predicates from P_o. *(d)* Part of the predicates from P_n are in the head and/or body, others are from P_o. *(e)* Both the head and body consist solely of predicates from P_n. Furthermore, if P_n is introduced, additional facts regarding it will be added when generating rules.

Constant Modification: Constants in facts will be replaced with randomly generated names to vary the grounding (e.g., "`P1(V1)`" → "`P1("Tweety")`").

Predicate Styling: Predicate descriptions are modified using three distinct styles: *(a)* simple P-style identifiers (P1, P2,...); *(b)* related concepts drawn from ConceptNet triples. For example, replacing predicates "`a(V0)`" and "`b(V0)`" in "`a(V0) :- -b(V0).`" with "`flying`" and "`bird`" based on the ConceptNet relation "`bird, CapableOf, flying`", even if the resulting rule lacks real-world logical validity, such as "`flying(V0) :- -bird(V0).`"; or *(c)* random concepts are selected from the ConceptNet and used as predicate descriptions.

(3) **Symbolic Sample Generation:** The modified ASP program will be executed using DLV2 and then saved if answer sets are produced successfully. These answer sets will be parsed and, together with the program itself, will form the symbolic ASP sample.

(4) **Textualization:** The symbolic ASP sample (facts, rules, and potentially a query) is converted into a natural language description using template-based conversion rules, which is then proofread by an LLM[1] to fix grammatical and punctuation errors[2]. To mostly keep the logical structure during textualization, we use a simple but precise style. For instance, positive facts like "`p(X).`" are described as "p(X) is true", while negative facts like "`-p(X).`" are rendered as "p(X) is explicitly false". Rules such as "`h :- b1, not b2.`" are converted into conditional statements like "If b1 is true and there is no evidence that b2 is true, then h is true". Details about the textualization and the prompt are provided in the supplementary material (Appendix B, C).

This step produces a pair of corresponding symbolic and textual ASP samples, which constitute the samples of the ASPBench dataset.

4.5 Task Design

ASPBench is designed to evaluate LLMs on three core ASP reasoning tasks: ASP entailment, answer set verification, and answer set computation. This section provides details of the settings and generation process used by ASPBench to create samples for each task.

[1]We use gpt-4o-mini during textualization.

[2]We conducted a human check to ensure the quality of textual samples.

ASP Entailment For each ASE sample, we ensure that the ASP program P does not contain any disjunctions and has only one unique answer set S. The query atom a is positive and is generated from the terminal predicate node P_t. The label is the truth value of a with respect to S.

Answer Set Verification For each ASV sample, the program P is allowed to include disjunctions and is guaranteed to have at least one answer set. The label for the candidate set C is randomly assigned as either `true` or `false`. When the label is `true`, C is randomly selected from one of the answer sets of P. When the label is `false`, C is generated by taking a correct answer set of P and applying exactly one of the following modifications:

- *Flip Negation:* A fact is randomly selected from a correct answer set, and its negation status is flipped (e.g., "`p.`" becomes "`-p.`", or "`-p.`" becomes "`p.`").
- *Delete Fact:* A single fact is randomly removed from a correct answer set.
- *Add Modified Fact (Constants):* A fact is randomly selected from a correct answer set, its constants are altered, and this newly formed fact is added to the set.
- *Add Modified Fact (Predicate):* A fact is randomly selected from a correct answer set, its predicate name is changed, and this newly formed fact is added to the set.

Answer Set Computation For each ASC sample, the program P is allowed to include disjunctions and is guaranteed to have at least one answer set. Moreover, all the answer sets of P are saved. The output of the model for the ASC task is then evaluated for correctness by checking whether it matches one of these answer sets.

In summary, the ASE task in ASPBench is tailored for scenarios with unique answer sets by constraining rule generation. Conversely, the ASV and ASC tasks embrace the possibility of multiple answer sets, often involving head disjunction, thereby providing a comprehensive evaluation of reasoning over more complex ASP features.

5 Experiment

5.1 Dataset Statistics

ASPBench benchmark includes three datasets for evaluating the ASP solving capabilities of LLMs: ASP Entailment (ASE), Answer Set Verification (ASV), and Answer Set Computation (ASC). Each dataset consists of 1,000 samples, pairing symbolic ASP programs with textual descriptions. Table 2 details their statistics. Note that constraints are often ineffective in synthetic programs, so ASPBench only incorporates them in manually collected ASP programs.

Moreover, in addition to synthetic ASP programs, we collected 47 symbolically represented classic ASP programs from various categories, as shown in Table 3. To enhance diversity and prevent LLMs from providing correct answers by memorising programs, we extend each program by: *(1)* scaling up problem instances; *(2)* converting problems to P-style format.

Statistic	ASE	ASV	ASC
Program Size			
Avg. Rules	10.72	15.71	15.47
Avg. Facts	9.77	14.36	14.23
Pred. Arity	0-3	0-3	0-3
Max. Chain	1-7	-	-
Syntactic Classes			
Positive	10.1%	9.2%	10.3%
Stratified	34.5%	34.8%	35.7%
Head-Cycle-Free	42.9%	44.3%	44.9%
Answer Sets			
Avg. Count	1.00	3.33	3.44
Avg. Facts/Set	-	22.70	21.94
Label Dist.			
True	41.3%	50.7%	-
False	32.1%	49.3%	-
Unknown	26.6%	-	-
Pred. Style			
P-style	32.7%	32.7%	32.5%
Related	33.8%	32.5%	32.4%
Random	33.5%	34.8%	35.1%

Table 2: Statistics of the ASPBench dataset across different tasks (ASE, ASV, ASC).

5.2 Evaluation Setup

Models To evaluate the reasoning capability of LLMs using the ASPBench dataset, we conducted experiments on 14 LLMs. These models are categorized as follows: *(1) General LLMs*: *qwen2.5-7b*, *qwen2.5-14b* (Yang et al. 2025b), *glm-4-flash* (Team GLM 2024), *gpt4o-mini* (OpenAI 2024a), *gpt-4o* (OpenAI 2024b), *claude-3-haiku* (Anthropic 2024), *deepseek-v3* (Liu et al. 2024), and *gemini-2.5-flash-nothinking* (Doshi 2025). *(2) Reasoning-Optimized LLMs*: *qwen3-8b*, *qwen3-14b* (Yang et al. 2025a), *o3-mini* (OpenAI 2025b), *o4-mini* (OpenAI 2025a), *deepseek-r1* (Guo et al. 2025), and *gemini-2.5-flash-thinking* (Doshi 2025). For each task, we use the same prompt across all LLMs. The detailed prompts used in experiments are shown in Appendix C.

Metrics We evaluate performance using task-specific metrics: for ASP entailment and answer set verification, we use the macro-F1 score, which treats all classes equally to mitigate biases from imbalanced label distributions; for answer set computation, we employ Exact Match (EM), defining a sample as correct if any predicted answer set exactly matches any of the ground truth answer sets.

Implementation details Inspired by Zheng et al. (2023) and Tam et al. (2024), we do not forcefully restrict the output format of LLMs during reasoning to minimize potential interference with their actual reasoning capabilities. Instead, we employ *gpt-4o-mini* to convert the raw outputs into structured JSON format for automated evaluation. For the Answer Set Computation task, we conduct an alignment process before evaluation. This process maps predicted facts to ground truth answer sets, handling model outputs that may not strictly follow standard answer set representations. Moreover, we use the latest version of DLV, *DLV2*[3], to validate the correctness of the symbolic samples in ASPBench. Details about the prompts for each task are provided in the supplementary material (Appendix C).

[3]https://dlv.demacs.unical.it/

Category	Number	Characteristics and Representative Examples
Basic Logic Reasoning (BLR)	8	Classic logic puzzles requiring deductive reasoning and constraint satisfaction. *Examples:* Zebra puzzle, Who killed Agatha, Safe cracking puzzle, etc.
Combinatorial Search & Optimization (CSO)	15	Tasks involving state-space exploration, pattern matching, and combinatorial optimization. *Examples:* N-Queens, Sudoku, Magic square, Minesweeper, etc.
Constraint Satisfaction & Scheduling (CSS)	8	Resource allocation and temporal scheduling problems under constraints. *Examples:* Map coloring, Job scheduling, Traffic light coordination, etc.
Mathematical & Set Problems (MSP)	16	Problems involving numerical computations and set-based reasoning of varying complexity. *Examples:* Set covering, Euler Problem, Subset sum, Prime numbers, etc.

Table 3: Overview of manually collected ASP problems grouped by problem-solving paradigm and reasoning characteristics.

Model	ASP Entailment (F1)					Answer Set Verification (F1)						Answer Set Computation (EM)					
	Sym	Tex	P-style	RanW	RelW	Sym	Tex	P-style	RanW	RelW	RealP	Sym	Tex	P-style	RanW	RelW	RealP
General LLMs:																	
gpt-4o-mini	0.347	0.386	0.373	0.372	0.348	0.410	0.473	0.456	0.434	0.432	0.623	0.021	0.028	0.025	0.027	0.022	0.007
glm-4-flash	0.322	0.340	0.318	0.360	0.312	0.505	0.509	0.499	0.512	0.505	0.461	0.009	0.019	0.011	0.021	0.009	0.021
gpt-4o	0.563	0.651	0.577	0.631	0.606	0.519	0.549	0.550	0.515	0.533	0.518	0.083	0.083	0.069	0.104	0.074	0.028
claude-3.5-haiku	0.380	0.468	0.384	0.431	0.453	0.499	0.536	0.494	0.518	0.537	0.449	0.090	0.145	0.102	0.145	0.102	0.014
qwen2.5-7b	0.328	0.283	0.268	0.337	0.309	0.534	0.518	0.496	0.549	0.522	0.540	0.014	0.016	0.015	0.026	0.005	0.014
qwen2.5-14b	0.471	0.526	0.434	0.522	0.535	0.552	0.548	0.542	0.515	0.592	0.389	0.063	0.082	0.080	0.081	0.057	0.014
deepseek-v3	0.674	0.708	0.621	0.733	0.716	0.547	0.566	0.549	0.561	0.553	0.547	0.256	0.158	0.222	0.218	0.181	0.064
gemini-2.5-flash-nothinking	0.890	0.832	0.849	0.861	0.871	0.783	0.786	0.792	0.791	0.768	0.621	0.065	0.109	0.071	0.091	0.099	0.074
Reasoning-Optimized LLMs:																	
qwen3-8b	0.682	0.911	0.762	0.811	0.813	0.659	0.666	0.669	0.629	0.690	0.596	0.167	0.308	0.240	0.244	0.228	0.057
qwen3-14b	0.850	0.958	0.886	0.904	0.920	0.692	0.714	0.707	0.698	0.702	0.666	0.290	0.517	0.406	0.413	0.397	0.092
deepseek-r1	0.976	0.977	0.978	0.967	0.984	0.873	0.809	0.838	0.836	0.849	0.674	0.817	0.297	0.551	0.581	0.537	0.149
gemini-2.5-flash-thinking	0.967	0.870	0.909	0.923	0.922	0.794	0.775	0.775	0.778	0.796	0.521	0.244	0.234	0.206	0.258	0.257	0.171
o3-mini	0.982	0.984	0.986	0.980	0.984	0.859	0.818	0.840	0.831	0.843	0.759	0.600	0.531	0.531	0.590	0.574	0.264
o4-mini	0.972	0.965	0.964	0.968	0.973	0.800	0.823	0.809	0.812	0.812	0.702	0.645	0.604	0.591	0.656	0.624	0.239
Avg.	0.672	0.704	0.665	0.700	0.696	0.645	0.649	0.644	0.641	0.652	0.576	0.240	0.224	0.223	0.247	0.226	0.086

Table 4: Overall performance of different LLMs on various reasoning tasks using the ASPBench datasets and real-world ASP benchmarks. The table details F1 scores and EM across different input styles. Abbreviations: Sym (Symbolic representation), Tex (Textual representation), P-style (Program-style representation), RanW (Random concepts from ConceptNet), RelW (Related concepts from ConceptNet triples), RealP (Real-world ASP Programs).

5.3 Main Results

We report the main findings through the following four questions:

(1) How LLMs perform on ASP solving in general? Our analysis of Table 4 demonstrates that current LLMs have significant limitations when it comes to solving ASP. Their performance is generally poor and depends heavily on task structure and input modality.

From the results, we can observe that a **steep performance cliff exists with task complexity**. While LLMs achieve moderate F1 scores in ASE (68.8% for the Sym/Tex average) and ASV (64.7% for the Sym/Tex average), further analysis reveals significant differences in performance. Specifically, ASV is a binary classification task and therefore has a higher random baseline (e.g., 0.5 F1 score) than the three-class ASE. Furthermore, the peak performance achieved in ASE (e.g., 98.2% by o3-mini on Sym input) significantly outperforms that of ASV (e.g., 87.2% by deepseek-r1 on Sym input). These factors suggest that the performance gap between LLMs on these tasks is larger than the average scores suggest, and ASV is therefore a more challenging task. Moreover, their performance *drops sharply* in the more complex and practically vital **ASC** task. The average EM score for ASC is 23.2% in the synthetic dataset and drops further to 8.6% in real-world ASP programs. This highlights the significant challenges involved in generating complete and precise multi-step logical reasoning.

For different types of LLMs, **Reasoning-Optimized LLMs achieve significant improvements**. Compared to *deepseek-v3* which achieves an EM of 20.7% in ASC, *deepseek-r1* achieves an EM of 55.7%, a substantial improvement. Similar improvements are observed in real-world ASP programs, where *qwen3-14b* shows a notable 9.2% EM compared to just 1.4% for *qwen2.5-14b*. However, even these improved models still present a gap for reliable practical application. This highlights that, despite beneficial optimization strategies, the capability of current LLMs to robustly solve these critical and complex reasoning tasks **requires further significant improvement**.

(2) How symbolic and textual representations influence the performance of LLMs in ASP solving? As shown in Table 4 (Avg. row), our results indicate that the relative performance of LLMs on symbolic versus textual representations is **highly task-dependent**.

LLMs generally perform better with textual input than symbolic input for classification tasks such as ASE (Tex: 70.4% vs Sym: 67.2% F1) and ASV (Tex: 64.9% vs Sym: 64.5% F1). This suggests that their reasoning is better aligned with the linguistic patterns of these contexts. Conversely, for the **ASC task**, symbolic inputs show a slightly higher average EM (Sym: 24.0% vs Tex: 22.4%). This difference may partly arise from the **challenge of aligning the natural language descriptions of textual outputs with structured ground truths**, which is an issue that is less prevalent for symbolic formats.

(3) How do the naming styles of predicates influence the ASP solving ability of LLMs? The choice of predicate

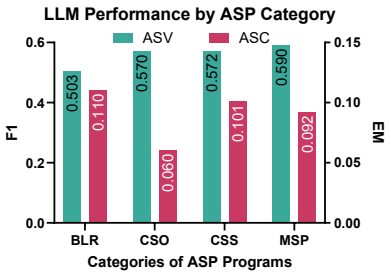

Figure 2: LLM performance on four real-world ASP problem categories.

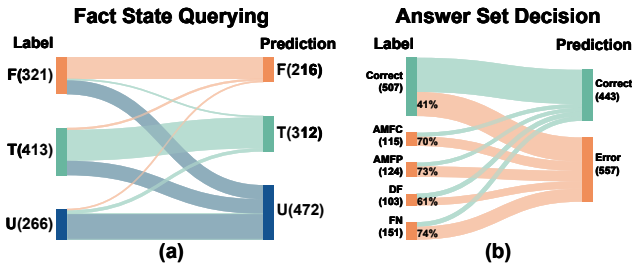

Figure 3: Sankey diagrams of average results on ASE and ASV. Truth-value abbreviations: T (true), F (false), U (unknown). Perturbation categories: AMFC (Add Modified Fact – Constants), AMFP (Add Modified Fact – Predicate), DF (Delete Fact), and FN (Flip Negation).

representation style **markedly influences LLM reasoning**.

Across all tasks, average results in Table 4 indicate that **LLMs reason more effectively with lexicalized predicates (RanW, RelW) than with simple P-style identifiers**. Specifically, P-style representations are consistently outperformed by at least one of the lexicalized predicate styles in each task category. For instance, in ASP entailment, the F1 score of P-style (66.5%) is surpassed by RanW (70.0%) and RelW (69.6%). This implies that predicates with some semantic grounding, whether they are randomly selected concepts from ConceptNet (as in RanW) or related concepts from ConceptNet triples (as in RelW), provide **more accessible anchors for reasoning** compared to simple identifiers (e.g., "P1", "P2" for P-style). This highlights a **sensitivity to how predicates are represented**, beyond just the choice between symbolic and textual inputs.

(4) How do LLMs perform in solving real-world ASP problems, compared to synthetic samples? Real-world ASP programs have similar or even fewer rules to synthetic samples, but they have more complex logical structures, such as constraints and iterative rules.

Compared with synthetic samples, the performance of LLMs on real-world ASP programs reveals a **notable performance gap**, with F1 scores dropping from 64.7% to 57.6% in ASV and EM scores plummeting from 23.2% to 8.6% in ASC (see the "RealP" columns in Table 4). For different categories, as shown in Figure 2, the two worst performing categories are Basic Logic Reasoning (BLR) and Combinatorial Search & Optimization (CSO), with the lowest F1 and EM scores, respectively. This denotes that current LLMs are almost unable to solve complex ASP problems.

Overall, these findings suggest that, despite their potential for solving basic ASP problems, **current LLMs lack the robust logical reasoning capabilities required for complex practical applications.**

5.4 Fine-grained Analysis

To investigate the limitations of LLMs in ASP solving, we perform fine-grained analysis on ASPBench. Case studies for each task are shown in the supplementary material (Appendix F).

ASP Entailment For ASP entailment (Figure 3 (a)), we observe that direct flips between the `true` and `false` states are rare; most errors arise from cases whose label is `true` or `false` but are predicted as `unknown`. Moreover, the proportion of `unknown` climbs from 26.6% in the ground truth to 47.2% in predictions. These observations reveal the following insights: *(1)* **Risk-averse bias.** LLMs would rather output `unknown` than risk making a polarity mistake, prioritising a lower risk of blatant contradiction over recall. *(2)* **Stricter ternary evaluation.** With a third truth value, errors hidden in binary metrics become visible. The increase in `unknown` predictions highlights the added difficulty of three-valued semantics.

Answer Set Verification For answer set verification (Figure 3 (b) and Figure 4 (a)), we observe the following insights:

Completeness blind spot: Ground-truth answer sets (*Correct*) and subsets formed by deleting a single fact (DF) have similar rates of being misjudged (41% vs. 39%), notably higher than for other situations. This indicates LLMs struggle to judge overall consistency, though they perform relatively well on local consistency.

Low sensitivity of solution space complexity: The performance of the LLM when evaluating candidate answer sets is not significantly affected by the complexity of the program, as measured by the total number of answer sets. For instance, accuracy only slightly decreases from 66% for programs with a single answer set to 61% for those with six. This general insensitivity indicates that the size of the solution space has a limited impact on this evaluation task. However, the observed minor decline suggests that very high program complexity could still pose an indirect challenge to these localized consistency checks.

Answer Set Computation For answer set computation (Figure 4 (b)), we observe that the performance of LLMs is sensitive to the number of answer sets in the program. Compared to ASV, LLMs are more sensitive to the number of answer sets in ASC. The following insights are observed:

Collapse in Model Computation: ASC requires the construction of a complete and coherent answer set based on the ASP program. This is a far more complex process than ASV, which, related to model checking, primarily involves verifying whether a given candidate is a valid answer set. This difference in complexity is clearly reflected in the performance of LLMs, with their ability to perform ASC tasks

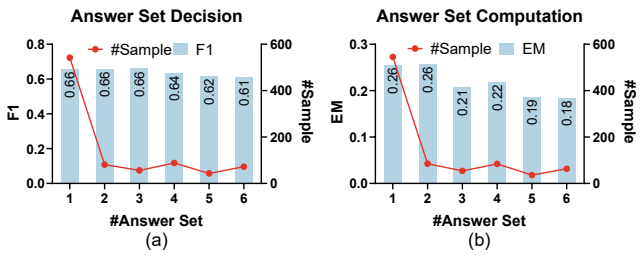

Figure 4: Effect of the number of answer sets on performance of LLMs.

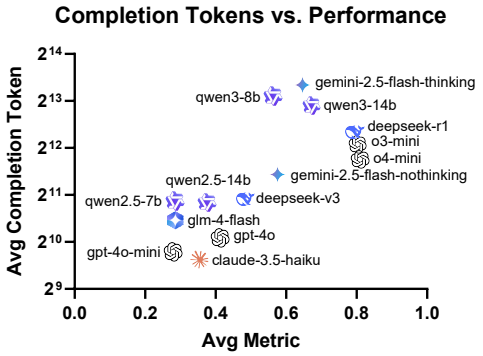

Figure 5: Average completion tokens vs. mean performance of LLMs. Mean performance combines metrics from ASP entailment (F1), answer set verification (F1), and answer set computation (EM).

decreasing sharply when ASP programs define three or more answer sets (e.g., EM from 26% for two answer sets to 21% for three).

5.5 Test-time and Model Scaling

Test-time scaling (Muennighoff et al. 2025) and model scaling (Kaplan et al. 2020) are two approaches to improve the performance of LLMs. To analyze the effect of these two approaches in our tasks, we report the average completion tokens and mean performance of LLMs, as shown in Figure 5.

The results visualize the link between average completion tokens (a proxy for thinking depth) and overall score: *(1)* **Longer reasoning chains tend to result in higher performance.** Reasoning-oriented variants such as *deepseek-r1*, *gemini-2.5-flash-thinking*, and *qwen3-14b* write longer chains than equally-sized base models and, in return, achieve noticeably higher performance—evidence that letting a model "think longer" at test time pays off. *(2)* **Larger models are more token-efficient.** Increasing parameters within the same family (e.g., *qwen3-14b* vs. *qwen3-8b*, *gpt-4o* vs. *gpt-4o-mini*) lifts performance while keeping chain length almost unchanged, showing that larger capacity delivers more signal per token.

Overall, this suggests that when GPU memory is limited, extending the reasoning chain is a cost-effective boost; with sufficient resources, scaling model parameters yield more reliable and shorter answers.

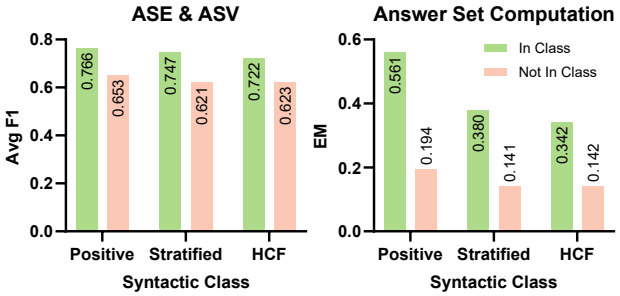

Figure 6: The fine-grained statistic of performance of LLMs on ASPBench with different syntactic classes.

5.6 Performance on Different Syntactic Classes

The syntactic structure of logic programs profoundly influences LLM reasoning performance (Figure 6). When programs adhere to specific syntactic constraints—Positive, Stratified, or HCF—LLMs demonstrate a striking improvement. This is most evident in ASC, where, for example, compared to non-Positive programs, Positive programs achieve a surge in EM scores from 19.4% to 56.1%—*nearly a threefold increase*. In simpler tasks like ASE and ASV, F1 scores are also boosted by over 10% when the positive constraint is met.

Furthermore, when cycles that violate Stratification or HCF constraints are present in programs, LLM performance also dramatically degrades. Particularly in ASC, under such conditions, EM scores fall by over 20% compared to programs that adhere to the respective Stratified or HCF constraints. Even for ASE and ASV, the presence of such unconstrained cycles causes scores to plateau around 0.60. This is significantly below the performance observed for programs that are Positive, or adhere to Stratification or HCF constraints.

This sharp divide highlights a fundamental deficit. While LLMs perform well with simple positive programs, their performance drops sharply when Stratification or HCF constraints are violated. In such cases, their average scores fall below those for general non-positive programs. **This exposes a critical lack of robust iterative and fixed-point reasoning in LLMs.**

6 Conclusion

In this work, we introduce ASPBench, a benchmark designed to evaluate ASP solving ability of LLMs. ASPBench includes diverse descriptions, predicates, and a rich set of logical operations. We define three key tasks: ASP entailment, answer set verification, and answer set computation, to rigorously assess LLM performance. Our experiments reveal significant limitations in the current ability of LLMs to handle ASP solving tasks. Here are a few potential future research directions that could mitigate the aforementioned limitations: *(1)* Develop hybrid architectures that integrate symbolic logic representation with neural networks to leverage the strengths of both approaches; *(2)* Propose new innovative methods specifically tailored to enhance ASP solving capability in LLMs.

Acknowledgments

This work is partially supported by National Nature Science Foundation of China under No. 62476058, and the Fundamental Research Funds for the Central Universities (2242025K30024). We thank the Big Data Computing Center of Southeast University for providing the facility support for the numerical calculations in this paper.

References

Allaway, E.; Hwang, J. D.; Bhagavatula, C.; et al. 2023. Penguins don't fly: Reasoning about generics through instantiations and exceptions. In *Proc. of EACL'2023*, 2610–2627.

Alviano, M.; Calimeri, F.; Dodaro, C.; et al. 2017. The ASP system DLV2. In *Proc. of LPNMR'2017*, 215–221.

Anthropic. 2024. Claude 3 Haiku: our fastest model yet. *Anthropic blog*.

Apt, K. R.; Blair, H. A.; and Walker, A. 1988. Towards a theory of declarative knowledge. In *Foundations of deductive databases and logic programming*. Elsevier. 89–148.

Barrett, C.; Stump, A.; and Tinelli, C. 2010. The smt-lib standard: Version 2.0. In *Proceedings of the 8th international workshop on satisfiability modulo theories (Edinburgh, UK)*, volume 13, 14.

Beck, H.; Dao-Tran, M.; and Eiter, T. 2018. LARS: A logic-based framework for analytic reasoning over streams. *Artif. Intell.* 261:16–70.

Ben-Eliyahu, R., and Dechter, R. 1994. Propositional semantics for disjunctive logic programs. *Annals of Mathematics and Artificial intelligence* 12:53–87.

Borroto, M.; Kareem, I.; and Ricca, F. 2024. Towards automatic composition of ASP programs from natural language specifications. *arXiv preprint arXiv:2403.04541*.

Chan, J.; Gaizauskas, R. J.; and Zhao, Z. 2025. RULEBREAKERS: Challenging LLMs at the crossroads between formal logic and human-like reasoning. In *Proc. of ICML'2025*.

Chen, M.; Li, G.; Wu, L. I.; et al. 2024a. Can language models pretend solvers? logic code simulation with LLMs. In *Proc. of SETTA'2024*, 102–121.

Chen, M.; Li, G.; Wu, L. I.; et al. 2024b. Can language models pretend solvers? logic code simulation with LLMs. *arXiv preprint arXiv:2403.16097*.

Clark, P.; Tafjord, O.; and Richardson, K. 2021. Transformers as soft reasoners over language. In *Proc. of the IJCAI'2021*, 3882–3890.

Dantsin, E.; Eiter, T.; Gottlob, G.; and Voronkov, A. 2001. Complexity and expressive power of logic programming. *ACM Comput. Surv.* 33(3):374–425.

Darwiche, A., and Pearl, J. 1997. On the logic of iterated belief revision. *Artif. Intell.* 89(1-2):1–29.

Doshi, T. 2025. Start building with Gemini 2.5 Flash. Online announcement.

Feng, J.; Xu, R.; Hao, J.; et al. 2023. Language models can be logical solvers. *arXiv preprint arXiv:2311.06158*.

Forgy, C. L. 1989. Rete: A fast algorithm for the many pattern/many object pattern match problem. In *Readings in artificial intelligence and databases*. Elsevier. 547–559.

Fu, Y.; Peng, H.; Ou, L.; et al. 2023. Specializing smaller language models towards multi-step reasoning. In *Proc. of ICML'2023*, 10421–10430.

Gebser, M.; Kaminski, R.; Kaufmann, B.; et al. 2012. *Answer Set Solving in Practice*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan and Claypool Publishers.

Gelfond, M., and Lifschitz, V. 1988. The stable model semantics for logic programming. In *Proc. of ICLP/SLP'1988*, 1070–1080.

Gelfond, M., and Lifschitz, V. 1991. Classical negation in logic programs and disjunctive databases. *NGC* 9:365–385.

Ginsberg, M. L. 1980. Readings in nonmonotonic reasoning.

Guo, D.; Yang, D.; Zhang, H.; et al. 2025. Deepseek-r1: Incentivizing reasoning capability in LLMs via reinforcement learning. *arXiv preprint arXiv:2501.12948*.

Huang, J., and Chang, K. C.-C. 2023. Towards reasoning in large language models: A survey. In *Proc. of Findings of ACL'2023*, 1049–1065.

Ishay, A.; Yang, Z.; and Lee, J. 2023. Leveraging large language models to generate answer set programs. *arXiv preprint arXiv:2307.07699*.

Jiang, J.; Wang, F.; Shen, J.; et al. 2024. A survey on large language models for code generation. *arXiv preprint arXiv:2406.00515*.

Josephson, J. R., and Josephson, S. G. 1996. *Abductive inference: Computation, philosophy, technology*. Cambridge University Press.

Kaplan, J.; McCandlish, S.; Henighan, T.; et al. 2020. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361*.

Leidinger, A.; Rooij, R. V.; and Shutova, E. 2024. Are LLMs classical or nonmonotonic reasoners? lessons from generics. In *Proc. of ACL'2024*, 558–573.

Li, Z.; Cao, Y.; Xu, X.; Jiang, J.; et al. 2024. LLMs for relational reasoning: How far are we? In *Proc. of LLM4CODE@ICSE'2024*, 119–126.

Liu, A.; Feng, B.; Xue, B.; et al. 2024. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437*.

Liu, H.; Fu, Z.; Ding, M.; et al. 2025. Logical reasoning in large language models: A survey. *arXiv preprint arXiv:2502.09100*.

Lyu, C.; Yan, L.; Xing, R.; et al. 2024. Large language models as code executors: An exploratory study. *arXiv preprint arXiv:2410.06667*.

Moura, L. D., and Bjørner, N. 2008. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, 337–340. Springer.

Muennighoff, N.; Yang, Z.; Shi, W.; et al. 2025. s1: Simple test-time scaling. *arXiv preprint arXiv:2501.19393*.

Niemelä, I. 1999. Logic programs with stable model semantics as a constraint programming paradigm. *Annals of mathematics and Artificial Intelligence* 25:241–273.

OpenAI. 2024a. GPT-4o mini: advancing cost-efficient intelligence. *OpenAI blog*.

OpenAI. 2024b. Hello GPT-4o. *OpenAI blog*.

OpenAI. 2025a. Openai o3 and o4-mini system card.

OpenAI. 2025b. Openai o3-mini technical report.

Parmar, M.; Patel, N.; Varshney, N.; et al. 2024. LogicBench: Towards systematic evaluation of logical reasoning ability of large language models. In *Proc. of ACL'2024*, 13679–13707.

Perak, B.; Beliga, S.; and Meštrović, A. 2024. Into LLM using RAG incorporating dialect understanding and prompt engineering techniques for causal commonsense reasoning. In *Eleventh Workshop on NLP for Similar Languages, Varieties, Proc. of the and Dialects (VarDial'2024)*, 220–229.

Reiter, R. 1980. A logic for default reasoning. *Artif. Intell.* 13(1-2):81–132.

Reiter, R. 1988. Nonmonotonic reasoning. In *Exploring artificial intelligence*. Elsevier. 439–481.

Rudinger, R.; Shwartz, V.; Hwang, J. D.; et al. 2020. Thinking like a skeptic: Defeasible inference in natural language. In *Proc. of Findings of ACL'2020*, 4661–4675.

Saha, S.; Yu, X. V.; Bansal, M.; et al. 2023. MURMUR: Modular multi-step reasoning for semi-structured data-to-text generation. In *Proc. of Findings of ACL'2023*, 11069–11090.

Santos, H.; Shen, K.; Mulvehill, A. M.; et al. 2024. A theoretically grounded question answering data set for evaluating machine common sense. *Data Intelligence* 6(1):1–28.

Srivatsa, K. A., and Kochmar, E. 2024. What makes math word problems challenging for LLMs? In *Proc. of Findings of NAACL'2024*, 1138–1148.

Tafjord, O.; Dalvi, B.; and Clark, P. 2021. ProofWriter: Generating implications, proofs, and abductive statements over natural language. In *Proc. Findings of ACL'2021*, volume ACL/IJCNLP 2021 of *Findings of ACL*, 3621–3634. Association for Computational Linguistics.

Tam, Z. R.; Wu, C.-K.; Tsai, Y.-L.; et al. 2024. Let me speak freely? a study on the impact of format restrictions on performance of large language models. *arXiv preprint arXiv:2408.02442*.

Team GLM. 2024. ChatGLM: A family of large language models from GLM-130B to GLM-4 all tools.

Tian, Y.; Zhang, F.; and Peng, N. 2023. Harnessing black-box control to boost commonsense in LM's generation. In *Proc. of EMNLP'2023*, 5417–5432.

Wan, H.; Xiao, G.; Wang, C.; et al. 2020. Query answering with guarded existential rules under stable model semantics. In *Proc. of AAAI'2020*, 1001–1008.

Wang, K.; Qi, G.; Li, J.; et al. 2024a. Can large language models understand DL-Lite ontologies? an empirical study. In *Proc. of Findings of EMNLP'2024*, 2503–2519.

Wang, S.; Wei, Z.; Choi, Y.; et al. 2024b. Can LLMs reason with rules? logic scaffolding for stress-testing and improving LLMs. In *Proc. of ACL'2024*, 7523–7543.

Wang, W.; Liu, K.; Chen, A. R.; et al. 2024c. Python symbolic execution with LLM-powered code generation. *arXiv preprint arXiv:2409.09271*.

Xiu, Y.; Xiao, Z.; and Liu, Y. 2022. LogicNMR: Probing the non-monotonic reasoning ability of pre-trained language models. In *Proc. of Findings of EMNLP'2022*, 3616–3626.

Yang, K., et al. 2024. If LLM is the wizard, then code is the wand: A survey on how code empowers large language models to serve as intelligent agents. *arXiv preprint arXiv:2401.00812*.

Yang, A.; Li, A.; Yang, B.; et al. 2025a. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*.

Yang, A.; Yu, B.; Li, C.; Liu, D.; et al. 2025b. Qwen2.5-1M technical report. *arXiv preprint arXiv:2501.15383*.

Yue, M.; Zhang, Y.; Liu, J.; et al. 2025. A survey of large language model agents for question answering. *arXiv preprint arXiv:2503.19213*.

Zheng, L. M.; Chiang, W.-L.; Sheng, Y.; et al. 2023. Judging llm-as-a-judge with mt-bench and chatbot arena. *Proc. of NeurIPS'2023* 46595–46623.

Zhu, Y.; Yuan, H.; Wang, S.; et al. 2023. Large language models for information retrieval: A survey. *arXiv preprint arXiv:2308.07107*.