Fine-Tuning LLMs for Answer Set Programming

Erica Coppolillo[1,2*], Francesco Calimeri[1,3], Giuseppe Manco[2], Simona Perri[1], Francesco Ricca[1]

[1*]Department of Computer Science, University of Calabria, Rende, 87036, Cosenza, Italy.
[2]ICAR-CNR, Rende, 87036, Cosenza, Italy.
[3]DLVSystem Srl, Rende, 87036, Cosenza, Italy.

*Corresponding author(s). E-mail(s): erica.coppolillo@unical.it;
Contributing authors: francesco.calimeri@unical.it; giuseppe.manco@icar.cnr.it; simona.perri@unical.it; francesco.ricca@unical.it;

Abstract

Large Language Models (LLMs) have demonstrated impressive capabilities across a wide range of natural language processing tasks, including code generation. While substantial progress has been made in adapting LLMs to generate code for various imperative programming languages, their effectiveness in handling declarative paradigms, such as Answer Set Programming (ASP), remains largely underexplored. This paper takes a step toward bridging that gap by investigating the potential of LLMs for ASP code generation. We begin with a systematic evaluation of several foundational LLMs, moving towards state-of-the-art models. We show that, despite their extensive training, large parameter counts, and significant computational backing, older models exhibit poor performance in generating syntactically and semantically correct ASP programs, while most recent ones mainly achieve impressive results. However, to overcome the need for huge computational power, we introduce LLASP, a fine-tuned, lightweight model specifically trained to encode ASP programs. In this regard, we extensively explore the effectiveness of fine-tuning by curating several dedicated datasets suitable for ASP encoding with increasing levels of complexity. First, we show that LLASP is effective in encoding template-based core problems in ASP; second, that the training strategy can be pushed forward to disregard the need for templating and make the generation prompt-invariant; and lastly, we show that even complex problems can be effectively encoded, beyond core tasks. Experimental results also show that LLASP significantly outperforms both its non-fine-tuned counterparts and most general-purpose LLMs,

particularly in terms of semantic correctness, achieving a good trade-off between accuracy and resource-efficiency. Experimental code is publicly available at: https://anonymous.4open.science/r/LLASP-1324/.

Keywords: Answer Set Programming, Large Language Models, Evaluation, Fine-tuning

1 Introduction

Answer Set Programming (ASP) (Brewka et al. 2011; Lifschitz 2019) is a powerful paradigm for knowledge representation and reasoning, offering numerous advantages that contribute to its widespread use across different application areas. A key strength of ASP lies in its ability to represent complex knowledge in a concise and intuitive manner, allowing users to model sophisticated problems with relative simplicity (Baral 2010). ASP adopts a declarative approach to modeling and solving combinatorial, planning, and reasoning tasks (Gebser et al. 2013). Its non-monotonic semantics facilitate the representation of incomplete information and default reasoning, which are essential features for modeling real-world domains. Furthermore, ASP's expressiveness enables the seamless integration of diverse types of knowledge, such as rules, constraints, and preferences, thereby promoting flexible and effective problem-solving (Lifschitz 2019).

Traditionally, ASP has been viewed as a specialized tool intended for domain experts and knowledge engineers who possess a deep understanding of the problem domain. These users employ ASP to define constraints, rules, and preferences in a compact and expressive form. Nevertheless, despite its high-level abstraction, authoring ASP code remains a daunting task for non-expert users, due to both the unfamiliarity with declarative paradigms and the inherent complexity of application scenarios.

In this context, the development of tools to streamline and partially automate ASP program generation becomes essential. Such tools aim to reduce the gap between natural language specifications and executable ASP code, ultimately making the language more accessible (Erdem and Yeniterzi 2009; Fang and Tompits 2018; Schwitter 2018; Caruso et al. 2024).

Recent breakthroughs in Artificial Intelligence and Machine/Deep Learning have introduced Large Language Models (LLMs) as pivotal technologies across a variety of natural language processing tasks (Raiaan et al. 2024; Minaee et al. 2024). Among these, text-to-code generation has gained considerable traction. LLMs are trained on massive datasets comprising code repositories, technical discussions, documentation, and related web data, enabling them to understand the contextual subtleties of source code and generate contextually relevant outputs.

A natural evolution of this research line involves leveraging LLMs for the automatic generation of code in declarative programming languages. This direction addresses the longstanding challenge of translating high-level natural language specifications into ASP code, in line with the broader ambition of automating programming through natural language. Utilizing LLMs for ASP program synthesis provides several compelling advantages: richer semantic alignment through expressive descriptions of the

problem domain; increased accessibility due to context awareness and interoperability; and overall gains in efficiency and productivity. Despite these potential benefits, existing literature reveals a noticeable gap in this area, with only a few preliminary studies addressing the topic (Ishay et al. 2023; Borroto et al. 2024).

To address this shortcoming, we present a comprehensive investigation into the capabilities of current LLMs to generate ASP programs. We begin by identifying structural patterns in ASP programs corresponding to core problem-solving tasks and assess how well existing LLM architectures can capture and reproduce these patterns. Building on this characterization, we propose a novel methodology for enhancing ASP code generation: we construct a custom dataset focused on representative problem domains and fine-tune an LLM accordingly. Our experiments reveal that a fine-tuned lightweight base model can outperform much larger LLMs, both in syntactic correctness and semantic fidelity, demonstrating its practical potential.

Next, we push the boundaries of fine-tuning a small, lightweight LLM by showing that (i) template-based prompts can be overcome, in favor of a *prompt-invariant* generation; and (ii) that more complex problems beyond core programs can be effectively encoded.

This work is an extension of a study that appeared in a conference publication (Coppolillo et al. 2024). In the conference version of our research, we make the following contributions:

- To the best of our knowledge, this was the first extensive comparison of (at the time) state-of-the-art LLMs in the context of ASP code generation. Our findings suggest that, despite their potential, older LLMs still underperformed in terms of syntactic and semantic program correctness.
- We show that a tailored training strategy, even if applied on a lightweight LLM, can outperform state-of-the art large-size models. To address this, we curated a comprehensive dataset and used it to produce **LLASP**, the finetuned version of a Gemma 2B base-model specifically trained to capture ASP fundamental patterns.
- Via a comprehensive experimental evaluation, we prove that LLASP generate ASP programs that are highly valuable in terms of both syntactic and semantic accuracy, overcoming significantly larger and more powerful LLMs.

Compared to the conference version, we here extend the contributions by including the following improvements:

- We substantially broaden the evaluation of LLMs in generating ASP programs by embracing state-of-the-art models. Our findings highlight that more recent ones significantly improve their performance.
- We deeply explore the potential of fine-tuning compact LLMs, demonstrating their capacity to generalize beyond templated outputs and address more complex problems.
- Lastly, we discuss the limitations of our approach and outline promising directions for future work, informed by further exploratory experiments.

The remainder of the paper is organized as follows. Section 2 reviews related work on automated code generation and the challenges specific to ASP. Section 3 introduces

relevant background concepts. In Section 4, we present our methodology and design principles for evaluating and fine-tuning LLMs. Section 5 details the experimental setup while Section 6 provides our findings. Finally, Section 8 concludes the paper, discussing our limitations and suggesting directions for future research.

2 Related Work

The benefits of automating code synthesis have been widely acknowledged in the literature (Ernst and Bavota 2022; Kalliamvakou 2022; Peng et al. 2023; Dakhel et al. 2023), and most mainstream programming languages are now supported by automatic code generation tools (Chen et al. 2021). In this landscape, Large Language Models (LLMs) play a central role, and their performance in code generation tasks has been thoroughly benchmarked (Xu et al. 2022; Wang et al. 2023). Recently, Raihan et al. (2025) investigate LLM performance on introductory programming assignments, showing that while contemporary models can produce syntactically valid and often correct solutions, their performance varies considerably depending on task complexity and problem structure.

In this context, the effectiveness of fine-tuning LLMs to improve code generation, especially for imperative programming languages, has been well established (Ma et al. 2024). Related research has evolved along two complementary directions: model-level adaptation (e.g., supervised fine-tuning and alignment) and data-free inference-time strategies (e.g., advanced prompting or decoding mechanisms). Although our work aligns primarily with the former, it is important to contextualize this contribution within the broader landscape of model improvement methodologies.

An emerging line of research explores how advanced prompting can elicit domain-specialized reasoning without requiring any fine-tuning or additional data. Notably, (Wang et al. 2025a) introduced the Metamemory Workflow (M^2WF), which leverages metacognitive prompting to guide models through stages of recall, evaluation, and planning before code generation. This framework demonstrates that careful prompt orchestration can improve the capabilities of the base model and even approximate fine-tuned performance in code-related tasks by activating latent reasoning capabilities of pretrained models. In a related fashion, Li et al. (2025) studied the robustness of Transformer-based models through code transformation on two popular programming languages, Java and Python. Addressing a different perspective, Jiao et al. (2025) explore the problem of code vulnerability detection, introducing DeepVulHunter, a multi-round analysis framework designed to augment LLMs' ability to identify security flaws. Their results demonstrate that structured, iterative prompting can significantly enhance model robustness and accuracy.

A parallel but distinct research direction targets the decoding process itself rather than the model parameters. Advanced decoding algorithms modify token selection during inference to enhance fluency, coherence, and correctness, especially in code and symbolic reasoning tasks. For instance, (Wang et al. 2025b) propose Fuzzy-Assisted Contrastive Decoding, which integrates fuzzy neural networks with contrastive decoding to dynamically penalize incoherent or erroneous generations.

Beyond technical performance, Schneider (2025) focuses on the cognitive dimension of human-LLM collaboration, analyzing how interactions with LLMs shape users' mental models. The study reveals that users often develop inaccurate or overly confident beliefs about model agency, competence, and reliability. These mental model shifts can influence how individuals interpret LLM output, delegate tasks, or detect error factors.

In this broad landscape, the realm of declarative programming addresses an important research goal: the development of tools that facilitate and automate the synthesis of Answer Set Programming (ASP) code. The overarching objective is to narrow the gap between high-level natural language descriptions and their corresponding ASP implementations (Erdem and Yeniterzi 2009; Fang and Tompits 2018; Schwitter 2018; Caruso et al. 2024). Early work in this direction targeted the translation of logic puzzles described in simplified English into ASP, using techniques such as λ-calculus and probabilistic combinatory categorial grammars (Baral and Dzifcak 2012).

Subsequently, significant efforts have been devoted to the development of Controlled Natural Languages (CNLs) for ASP. CNLs are subsets of natural language with constrained grammar and vocabulary, designed to preserve interpretability while enabling formal translation. For example, BIOQUERYCNL (Erdem and Yeniterzi 2009) defines syntactic rules for constructing CNL statements and provides an algorithm to compile these into ASP. Fang et al. (Fang and Tompits 2018) introduced a CNL framework based on LANA annotations, implemented within the SeaLion IDE. Later, Schwitter proposed PENG$^{\text{ASP}}$, a CNL framework for both specifying and verbalizing ASP programs (Schwitter 2018). More recently, Dodaro et al. presented CNL2ASP (Caruso et al. 2024), a comprehensive, publicly available tool that translates CNL input into ASP code.

While CNLs offer a more accessible syntax compared to traditional ASP code, they still impose formal grammatical constraints, requiring users to conform to a structured input format. As a result, they do not fully eliminate the challenges non-expert users face when learning and applying ASP.

Beyond CNLs, several approaches have sought to leverage the capabilities of LLMs in conjunction with ASP. Nye et al. (Nye et al. 2021) proposed a dual-system architecture using GPT-3, which combines a neural module (System 1) with a logic-based reasoning component (System 2) to produce semantic parsers from natural language and integrate them with ASP reasoning modules. Similarly, Yang and Ishay (Yang et al. 2023) demonstrated how GPT-3 can act as a few-shot semantic parser for converting natural language into ASP logical forms, without requiring retraining for specific question-answering tasks.

In the context of logic puzzle solving, Ishay and Yahav (Ishay et al. 2023) employed prompt engineering to guide LLMs in producing ASP programs, using datasets such as the one introduced by Mitra and Baral (Mitra and Baral 2016). Moreover, the STAR framework (Rajasekharan et al. 2023) has illustrated how LLMs and ASP can be integrated for performing robust natural language understanding tasks.

Despite these advances, the literature still lacks a systematic and dedicated approach to using LLMs for ASP code synthesis.

A noteworthy exception is the recent work by Borroto et al. (2024), who introduced NL2ASP (Borroto et al. 2024), a framework for generating ASP code from natural language via a two-step architecture. Specifically, NL2ASP first applies neural machine translation (NMT) (Stahlberg 2020) to convert natural language into the CNL defined by Dodaro et al., and then uses the CNL2ASP tool to translate the intermediate statements into ASP code. Their system employs Transformer-based NMT models (T5-small and Bart-base) and achieves encouraging results.

Nevertheless, several key distinctions set our work apart from the NL2ASP approach. First, while NL2ASP focuses on graph-based problem domains, our framework targets general-purpose ASP code generation, leveraging fundamental structural patterns that are domain-independent. Second, our method eliminates the need for an intermediate representation (i.e., CNL), generating ASP code directly from natural language input. Third, although Borroto et al. noted the limited performance of LLMs on ASP code generation tasks, their study did not conduct an exhaustive empirical evaluation. In contrast, our paper offers a comprehensive analysis of LLM performance in this context, filling a critical gap in the current literature.

3 Preliminaries

3.0.1 Large Language Models.

A Large Language Model ($\mathcal{L}$) can be formally defined as a stochastic function f that maps an input token sequence $x = [x_1, x_2, ..., x_n]$ to an output sequence $y = [y_1, y_2, ..., y_m]$, where n and m denote the lengths of the input and output sequences, respectively. The model captures the conditional probability distribution $P_{\mathcal{L}}(y|x)$, reflecting complex dependencies and structures in natural language.

Given that $x \in V^*$ and $y \in W^*$, with V and W representing token vocabularies, both sequences must adhere to some underlying grammar. The function f samples y from the distribution $P_{\mathcal{L}}(\cdot|x)$, defining the model's generative process.

This probability distribution is computed via the Transformer Encoder-Decoder architecture (Vaswani et al. 2017). Initially, the input tokens are embedded into dense vectors that capture contextual similarity and encode positional information. These embeddings are then processed through stacked Transformer layers.

The Encoder is composed of multiple identical layers, each combining a self-attention mechanism to assign relevance weights among tokens, and a feedforward neural network that independently transforms each position. The Decoder layers follow a similar design, incorporating masked self-attention to preserve autoregressive generation, Encoder-Decoder attention to align with the input context, and position-wise feedforward networks.

The Decoder's final output is passed through a linear projection followed by a softmax function, producing a probability distribution over the vocabulary of possible next tokens. This allows the model to compute $P_{\mathcal{L}}(y_i|x, y_{:i-1})$, where $y_{:i-1}$ denotes the prefix of the output sequence up to token $i-1$. Token generation proceeds sequentially by sampling from this conditional distribution.

Several architectural variants have been introduced, including Causal-Decoders, Prefix-Decoders, Autoregressive models, and Mixture-based extensions (Naveed et al.

2025; Raiaan et al. 2024). However, since this paper does not focus on architectural innovation, the internal structure of the LLM is treated as a black box.

The training of $\mathcal{L}$ typically unfolds in two main phases. The first is pre-training, which is generally based on next-token prediction: given x and a prefix $y_{:i-1}$, the model learns to predict the next token y_i. The second phase is supervised fine-tuning (SFT), which adjusts the model to predict entire output sequences from inputs using a curated dataset $\mathcal{D} = \left\{(x^{(1)}, y^{(1)}), ..., (x^{(N)}, y^{(N)})\right\}$, where N is the number of training pairs.

This fine-tuning is driven by minimizing a loss function, typically cross-entropy, defined as:

$$\ell\left(y^{(i)}, P_{\mathcal{L}}\left(y^{(i)}|x^{(i)}\right)\right) = \sum_{j} y_j^{(i)} \log P_{\mathcal{L}}\left(y_j^{(i)}|x^{(i)}, y_{:j-1}^{(i)}\right).$$

This loss formulation is adopted throughout this paper, although other formulations are possible.

3.0.2 Answer Set Programming.

Answer Set Programming (ASP) is a robust declarative formalism for Knowledge Representation and Reasoning that has garnered substantial attention due to its high expressive power and the availability of mature, efficient solvers (Gebser et al. 2018). ASP programs are fully declarative, i.e., they are insensitive to the ordering of literals and rules, and allow for concise modeling of a wide range of complex problems.

As any other declarative paradigms, ASP fundamentally differs from standard imperative approaches (e.g., Python, C, Java) in both its conceptual and computational models. Specifically, in ASP, the programmer specifies *what* conditions or relationships should hold within a problem domain, rather than *how* to achieve them procedurally.

Here, we briefly recall the language's syntax and provide an intuitive account of its semantics, particularly focusing on the notion of *answer sets*, which plays a critical role in the validation phase described in Section 5.6. For an in-depth treatment and advanced features of ASP, the reader is referred to standard references (Brewka et al. 2011; Eiter et al. 2009; Calimeri et al. 2016).

A *term* may be a constant or a variable. An *atom* is an expression of the form $p(t_1, \ldots, t_n)$, where p is a predicate symbol of arity n and each t_i is a term. A program P is a finite collection of rules of the form:

$$\alpha_1 | \cdots | \alpha_k \ :- \ \beta_1, \ldots, \beta_n, \ not \ \beta_{n+1}, \ldots, \ not \ \beta_m,$$

where $k \geq 0$, $m \geq 0$, and all α_i, β_j are atoms. The head of the rule is a disjunction of atoms, while the body is a conjunction of literals (positive or negated via default negation).

A rule with a single atom in the head and an empty body represents a *fact*, i.e., an assertion known to be true. Conversely, a rule with an empty head defines a *(strong) constraint*, which enforces that the body must not be simultaneously satisfied. A strong

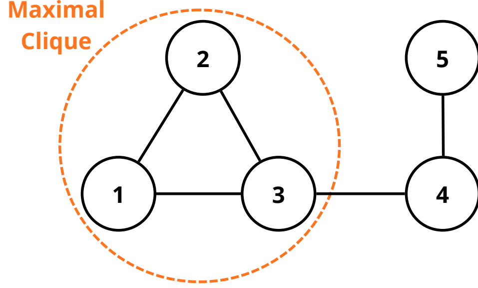


Fig. 1: Anecdotal example of a graph $G = (V, E)$ with a maximal clique.

constraint is modeled as:

$$:\text{-}\ l_1, ..., l_n.$$

ASP also supports *weak constraints*, which enable the modeling of preferences and soft conditions, useful for optimization tasks. A weak constraint has the form:

$$:\sim l_1, ..., l_n. \ \ [w@l, t_1, ..., t_m],$$

where w is a weight, l is a priority level, and $t_1, ..., t_m$ are optional terms. A violation of such a constraint incurs a cost w at level l.

A program, rule, or atom is said to be *ground* if it contains no variables. Models are defined over ground programs. Specifically, a *model* is a set of ground atoms that satisfies all rules in a program P.

ASP's semantics is defined in terms of *answer sets*, which are special models that represent alternative, self-consistent views of the world (Gelfond and Lifschitz 1991; Brewka et al. 2011). An ASP program may admit multiple answer sets, or none at all, each corresponding to a possible solution of the encoded problem.

To solve a computational problem, one encodes it as a set of ASP rules and facts that describe the domain and the specific instance. Answer sets of the resulting program then correspond to valid solutions. If no answer set exists, the problem instance has no solution.

When weak constraints are present, the optimal answer sets are those that minimize the accumulated cost at the highest priority level. Among those, preference is given to answer sets with minimal cost at the next lower level, and so on.

Anecdotal Example. To ensure comprehension, we provide an example of an ASP program encoding the Maximal Clique problem, a standard task in the ASP literature. Given an undirected graph $G = (V, E)$, the problem consists of finding the largest subsets of vertices in a graph where every pair of vertices is connected by an edge.

Consider the graph reported in Figure 1. In this simple example, G exhibits two cliques C_1, C_2, where $C_1 = \{v_1, v_2, v_3\}$, and $C_2 = \{v_4, v_5\}$, with C_1 being maximal. Translating G in ASP facts would result in:

```
vertex(1..5).
edge(1,2). edge(1,3).
```

```
edge(2,1). edge(2,3).
edge(3,1). edge(3,2). edge(3,4).
edge(4,3). edge(4,5).
edge(5,4).
```

A possible ASP encoding for the clique problem is the following.

```
in_clique(V) | out_clique(V) :- vertex(V).

:- in_clique(U), in_clique(V), U!=V, not edge(U, V).
:~ out_clique(V). [1@1, V]
```

The program collocates each vertex of the graph to be inside or outside the clique (first line). A strong constraint requires that an edge must link two distinct nodes within the clique (second line). Finally, a weak constraint ensures that only the *maximal* clique is produced.

The answer set of this program, instantiated (i.e., grounded) over G, consists of:

```
in_clique(1). in_clique(2). in_clique(3).
out_clique(4). out_clique(5).
```

where facts are excluded to enhance readability.

For comparison, we further provide a valid implementation of the same problem in Python. Notably, the code of the `find_cliques` function is taken from the official `networkx` implementation[1], a standard library for graph analysis and manipulation.

```
def find_cliques(G, nodes=None):
    if len(G) == 0:
        return

    adj = {u: {v for v in G[u] if v != u} for u in G}

    # Initialize Q with the given nodes and subg, cand with their
        nbrs
    Q = nodes[:] if nodes is not None else []
    cand = set(G)
    for node in Q:
        if node not in cand:
            raise ValueError(f"The given `nodes` {nodes} do not form
                a clique")
        cand &= adj[node]
```

[1] https://networkx.org/documentation/stable/reference/algorithms/generated/networkx.algorithms.clique.find_cliques.html

```
    if not cand:
        yield Q[:]
        return

    subg = cand.copy()
    stack = []
    Q.append(None)

    u = max(subg, key=lambda u: len(cand & adj[u]))
    ext_u = cand - adj[u]

    try:
        while True:
            if ext_u:
                q = ext_u.pop()
                cand.remove(q)
                Q[-1] = q
                adj_q = adj[q]
                subg_q = subg & adj_q
                if not subg_q:
                    yield Q[:]
                else:
                    cand_q = cand & adj_q
                    if cand_q:
                        stack.append((subg, cand, ext_u))
                        Q.append(None)
                        subg = subg_q
                        cand = cand_q
                        u = max(subg, key=lambda u: len(cand &
                            adj[u]))
                        ext_u = cand - adj[u]
            else:
                Q.pop()
                subg, cand, ext_u = stack.pop()
    except IndexError:
        pass

maximal_cliques = list(find_cliques(G))

print("Maximal Cliques:")
for clique in maximal_cliques:
    print(clique)
```

Clearly, formulating the problem in ASP enhances the clarity and conciseness of the program. This illustrative example underscores the suitability of ASP for representing

complex tasks that would otherwise entail considerably higher modeling complexity in conventional imperative coding paradigms.

4 Methodology and Knowledge Design

In this work, we pursue a twofold objective. First, we focus on evaluating general-purpose pre-trained Large Language Models (LLMs) that have not been explicitly optimized for ASP code generation. Given such a model $\mathcal{L}$ and a natural language specification x of a computational problem, our aim is to assess the model's ability to sample $y \sim P_{\mathcal{L}}(\cdot|x)$ such that y constitutes a valid ASP encoding aligned with the semantics of x.

Our second objective involves investigating the impact of fine-tuning on generation quality, particularly in the context of small-scale LLMs. A comprehensive evaluation of these goals requires the consideration of several key dimensions:

- **Dataset diversity and complexity.** The evaluation must be grounded on a diverse dataset containing a broad spectrum of problem specifications (x), varying in domain, complexity, and length. This variety ensures that we can thoroughly test $\mathcal{L}$'s robustness and versatility in producing ASP encodings under different conditions.
- **ASP compliance and robustness.** For each x, we examine whether the sampled output $y \sim P_{\mathcal{L}}(\cdot|x)$ complies with ASP's syntactic rules and accurately captures the intended semantics. This involves evaluating syntactic correctness, semantic alignment with the prompt, and the model's ability to preserve problem constraints.
- **Impact of model size.** We analyze how model architecture and parameter size affect performance in ASP generation. This includes comparing lightweight versus large models and quantifying trade-offs in terms of encoding precision, resource efficiency, and generalization capacity.
- **Generalization capability.** We assess whether $\mathcal{L}$, pre-trained on general-purpose corpora, can generalize to novel problem domains not explicitly seen during training. This dimension helps evaluate the model's flexibility and potential for zero-shot ASP generation.

In this section, we concentrate on the first dimension and defer the others to subsequent sections.

To approach ASP generation, we adopt a strategy inspired by the typical learning process of humans acquiring proficiency in ASP. This process begins with mastering simple encoding patterns that capture atomic knowledge units, such as representing Cartesian products, joining predicate extensions, implementing guessing strategies, enforcing value constraints, or defining transitive closures.

These basic building blocks can then be composed to construct more elaborate ASP rules. Due to the declarative nature of ASP, where the order of rules and literals is inconsequential, assembling complex programs often reduces to combining simpler, modular subprograms that encode specific aspects of the target problem.

Following this principle, we define a template dataset $\mathcal{T} = (\mathcal{P}, \mathcal{A})$, where $\mathcal{P}$ contains natural language descriptions of elementary tasks, and $\mathcal{A}$ contains corresponding *gold* ASP encodings.

The dataset is further partitioned into subsets associated with specific task categories: $\mathcal{T} = \{\mathcal{T}_{\mathcal{C}^1}, \mathcal{T}_{\mathcal{C}^2}, \ldots, \mathcal{T}_{\mathcal{C}^k}\}$, where $\mathcal{C}^i$ denotes the i-th class of modeled tasks, and k is the total number of distinct task types.

For instance, for the task category $\mathcal{C}^1$ related to transitive closure computation, the set $\mathcal{T}_{\mathcal{C}^1}$ includes pairs $(\mathcal{P}_{\mathcal{C}^1}, \mathcal{A}_{\mathcal{C}^1})$, where each pair links a task description to a suitable ASP encoding. These templates and their ASP solutions contain placeholders for predicate names, constants, or labels, which can be instantiated to create concrete training samples.

In the following, we present a concise overview of the core task types we investigate. Each is introduced with an informal description, followed by the template used for generation and a concrete example of a prompt paired with its corresponding ASP solution.

Guessing Assignments. We begin with a canonical ASP task in which each element of a domain must be assigned a unique label selected from a fixed set. In ASP, this is typically represented using disjunctive rules to encode the guessing mechanism. An illustrative example of a prompt-encoding pair is presented below.

```
Template: Write an ASP program for the following problem. Assign
    exactly a label among a given set of labels to a set of elements.
    The set of elements is expressed by predicate [PREDICATE]. The
    labels are [LABEL]+.

Prompt: Write an ASP program for the following problem. Assign
    exactly a label among a given set of labels to a set of elements.
    The set of elements is expressed by predicate city. The labels
    are moscow,rome,dubai.

Encoding: assign(X,"moscow") | assign(X,"rome") | assign(X,"dubai")
    :- city(X).
```

Expressing Constraints. Another common task in ASP involves enforcing mandatory conditions that must not be violated. Such requirements are typically modeled using *(strong) constraints*, which ensure that certain combinations of literals are disallowed. In some cases, auxiliary rules may be employed to facilitate the expression of these constraints. The following example illustrates a prompt that requests the prevention of a specific assignment, along with a corresponding ASP encoding that satisfies the condition.

```
Template: Write an ASP program for the following problem. Prevent the
    predicate [PREDICATE] with value [VALUE] from having label
    [LABEL].

Prompt: Write an ASP program for the following problem. Prevent the
    predicate car with value 11 from having label "red".
```

```
Encoding:  :- assign(11,"red").
```

Generating Combinations. A frequent requirement in ASP involves generating all possible combinations of elements from two distinct sets, i.e., computing their Cartesian product. This can be effectively achieved using a straightforward rule that pairs each element of the first set with every element of the second. Below, we present an example prompt that specifies this task, along with a corresponding ASP encoding that accomplishes it.

```
Template: Write an ASP program for the following problem. Generate
    all the combinations of elements from two sets. The two sets are
    represented by predicates [PREDICATE_1] and [PREDICATE_2].

Prompt: Write an ASP program for the following problem. Generate all
    the combinations of elements from two sets. The two sets are
    represented by predicates city and airport.

Encoding: combination(X,Y) :- city(X), airport(Y).
```

Joins. Joining elements from two distinct sets based on matching conditions over shared features is a common task in ASP. Such joins are typically implemented through rules whose bodies contain literals sharing a common variable, effectively encoding the desired matching logic. An example prompt describing this task, along with a corresponding ASP encoding, is provided below.

```
Template: Write an ASP program for the following problem. Consider
    predicate [PREDICATE_1] having fields [LABEL]+ and the predicate
    [PREDICATE_2] having fields [LABEL]+. Define a predicate
    [PREDICATE_1]_[PREDICATE_2] that associates to each [PREDICATE_1]
    the [LABEL] of [PREDICATE_2].

Prompt: Write an ASP program for the following problem. Consider
    predicate "owner" having fields "ID","surname",
    "name","restaurantID", and the predicate "restaurant" having
    fields "ID","description". Define a predicate "owner_restaurant"
    that associates to each owner the description of restaurant.

Encoding: owner_restaurant(X,Z) :- owner(X,_,_,Y),
                                   restaurant(Y,Z).
```

Transitive Closure. Transitive closure is a foundational concept used to define relational structures in numerous domains, as it captures not only direct relationships

between elements but also indirect ones that emerge through chains of connections. In ASP, expressing transitive closure typically requires multiple rules. The following example illustrates a prompt that requests the computation of a transitive closure, along with a corresponding ASP encoding that includes two rules: one to define direct relationships, and another to capture indirect ones via recursion.

```
Template:  Write an ASP program for the following problem. Define
    predicate [PREDICATE_1] as the transitive closure of predicate
    [PREDICATE_2].

Prompt: Write an ASP program for the following problem. Define predi
    cate "arrivals" as the transitive closure of predicate "flight".

Encoding:
    arrivals(X,Y) :- flight(X,Y).
    arrivals(X,Y) :- flight(X,Z),arrivals(Z,Y).
```

Expressing Preferences. To employ ASP for solving optimization problems, it is essential to express preferences over the set of admissible solutions. This is typically achieved by incorporating *weak constraints* into the program, which allow for the specification of desirable (but not mandatory) conditions. The following example presents a prompt that requests the expression of a specific preference, along with a corresponding ASP encoding that implements it.

```
Template: Write an ASP program for the following problem. I would
    prefer that predicate [PREDICATE] with value 18 is not associated
    with [LABEL]. If this occurs, it costs [COST] at level [LEVEL].

Prompt: Write an ASP program for the following problem. I would
    prefer that predicate house with value 18 is not associated with
    "flat". If this occurs, it costs 2 at level 2.

Encoding: :~ assign(18,"flat"). [2@2]
```

Filtering. When designing ASP programs, it is often necessary to restrict the extension of certain predicates according to specific filtering criteria. Below, we outline several common forms of such filters.

Filtering by values. This type of filtering involves selecting only those atoms in the extension of a predicate that match a given value. The following example presents a prompt that describes such a filtering condition, along with a corresponding ASP encoding that fulfills it.

```
Template: Write an ASP program for the following problem. Select all
    values associated to the predicate [PREDICATE] with label
    [LABEL].

Prompt: Write an ASP program for the following problem. Select all
    values associated to the predicate color with label "purple".

Encoding: select(X) :- color(X,"purple").
```

Filtering by negative. Another common filtering criterion involves excluding those atoms in a predicate's extension that satisfy a given condition. This may take the form of set difference operations between predicate extensions or the negation of compound conditions. The example below illustrates a prompt that requests filtering based on a negative condition applied to a selectively filtered portion of another predicate, along with the corresponding ASP encoding.

```
Template: Write an ASP program for the following problem. Select all
    values associated with predicate [PREDICATE_1] but not associated
    with predicate [PREDICATE_2] and label [LABEL].

Prompt: Write an ASP program for the following problem. Select all
    values associated with predicate vehicle but not associated with
    predicate moto and label "kawasaki".

Encoding: select(X) :- vehicle(X),
                       not moto(X,"kawasaki").
```

Filtering by numeric comparisons. Filtering parts of predicate extensions based on comparisons between terms is another common requirement in ASP. Such filters often involve numerical or relational conditions applied to specific attributes. The following example presents a prompt that requests filtering a predicate based on a numeric comparison, together with an appropriate ASP encoding that implements this logic.

```
Template: Write an ASP program for the following problem. Select all
    values associated with predicate [PREDICATE] with a value greater
    or equal than [VALUE].

Prompt: Write an ASP program for the following problem.
    Select all values associated with predicate
    size with a value greater or equal than 5.

Encoding: select(X) :- size(X,C), C>=5.
```

The presented collection, though limited in size, constitutes a rich and versatile foundation of essential ASP patterns. These building blocks can be combined and extended to construct more sophisticated programs capable of solving a wide range of problems.

For instance, many combinatorial problems involve selecting or associating elements from a given domain in accordance with specific constraints. Such problems can often be effectively addressed by combining ***guessing assignments*** with ***expressing constraints***, and potentially incorporating other task types introduced earlier. When optimization is also required, the problem can be further refined by ***expressing preferences*** through the use of weak constraints.

5 Experiments

In this section, we present the design of our experimental setup and provide an evaluation of the resulting outcomes. Building on the prompts introduced in the previous section, we aim to investigate the following research questions:

- **RQ1**: To what extent are state-of-the-art pre-trained LLMs capable of translating textual problem descriptions into accurate ASP programs?
- **RQ2**: Can LLMs be effectively fine-tuned to generate ASP rules that faithfully adhere to the given natural language specifications?
- **RQ3**: How suitable is the output produced by the fine-tuned LLM for practical use, particularly in terms of syntactic validity and semantic correctness?
- **RQ4**: Can the generation process be prompt-invariant?
- **RQ5**: Is it possible to extend the model capabilities to more complex problems?

In order to ensure reproducibility, experimental code is publicly available: https://anonymous.4open.science/r/LLASP-1324/.

5.1 Pre-trained LLMs

Our first objective is to evaluate the ability of pre-trained, foundational LLMs to generate valid ASP programs. To this end, we design a structured set of experiments aimed at benchmarking the performance of selected models in this specific task.

We focus on freely accessible LLMs that represent recent advancements in text generation, including both large-scale and lightweight architectures:

- **ChatGPT 3.5**[2]: Developed by OpenAI, this model is a fine-tuned version of GPT-3.5 optimized for conversational interactions and natural language understanding.
- **Copilot**[3]: A conversational AI assistant created by Microsoft, built on GPT-4 and integrated with multimodal capabilities such as DALL-E 3 for text-to-image generation.
- **Gemini** (Team 2024a)[4]: Introduced by Google, Gemini is a new class of multimodal LLMs with strong performance across vision, audio, and text tasks.

[2] https://chatgpt.com/
[3] https://learn.microsoft.com/en-us/copilot/overview
[4] https://gemini.google.com/app

- **Gemma** (Team 2024b)[5]: Released by the team behind Gemini, Gemma is a lightweight LLM family that achieves competitive results on tasks involving reasoning, comprehension, and safety.
- **LLaMa2** (Touvron et al. 2023)[6]: A dialogue-optimized LLM developed by Meta, LLaMa2 consistently outperforms many open-source alternatives across multiple evaluation benchmarks.
- **LLaMa3**[7]: The latest evolution of the LLaMa family, LLaMa3 demonstrates enhanced performance in instruction following, code generation, and reasoning tasks.
- **Mistral** (Jiang et al. 2023)[8]: Despite its smaller size (7B parameters in the initial release), Mistral has demonstrated remarkable efficiency and strong results across various benchmarks. Larger and more powerful versions have since been released.

Table 1 provides a comparative overview of the selected LLMs in terms of model size, architectural characteristics, and training data sources (Minaee et al. 2024). The models can be broadly categorized into two groups: large-scale models (with $\geq$ 70B parameters) and compact models (with $<$ 70B parameters). Among them, Gemma 2B stands out as the smallest model considered in our evaluation.

Table 1: Details on the evaluated pre-trained LLMs. Number of parameters is reported in billions(B)/trillions(T). E-D refers to the Encoder-Decoder architecture; D-Only is Decoder-Only; MoE is Mixture of Experts, and GQA is Grouped-Query Attention.

Model	No. Params.	Architecture	Training Data
ChatGPT 3.5	175B	E-D	Online sources
Copilot	1.5T	E-D	Github repositories
Gemini	1.6T	MoE	Docs, Books, Code
Gemma	2B-7B	D-Only	Docs, Maths, Code
LLaMa2	7B-13B	D-Only	Online sources
LLaMa3	8B-70B	D-Only + GQA	Online sources
Mistral	7B-141B	D-Only + GQA	Online sources

5.2 State-of-the-art LLMs

Next, we extend the analysis to state-of-the-art LLMs in generating ASP code starting from a textual description. Specifically, we revised the following models, selected from a wide range of heterogeneous LLM families:

[5] https://huggingface.co/blog/gemma
[6] https://www.llama2.ai/
[7] https://llama.meta.com/llama3/
[8] https://chat.mistral.ai/chat

- **ChatGPT-4o**[9]: OpenAI's flagship multimodal model capable of processing text, images, and audio inputs natively, offering faster and more cost-effective performance than GPT-4.
- **Claude 3.5 Sonnet**[10]: Anthropic's advanced language model excelling in natural language processing and content generation, with capabilities extending to visual content creation.
- **Command R+**[11]: Cohere's instruction-following conversational model optimized for complex Retrieval-Augmented Generation (RAG) workflows and multi-step tool use.
- **DeepSeek 2.5**[12]: An upgraded model combining DeepSeek-V2-Chat and DeepSeek-Coder-V2-Instruct, enhancing both general and coding capabilities.
- **DeepSeek Coder 2.5**[13]: A specialized version of DeepSeek 2.5 focused on code generation and understanding, integrating improvements from previous iterations.
- **Grok 2**[14]: xAI's frontier language model with advanced reasoning capabilities, excelling in chat, coding, and various academic benchmarks.
- **Grok 2 Mini**[15]: A streamlined variant of Grok 2, designed for speed without sacrificing quality, maintaining strong capabilities in reasoning and coding.
- **Jamba 1.5 Large**[16]: AI21 Labs' open model built on the SSM-Transformer architecture, demonstrating superior long-context handling and multilingual capabilities.
- **Jamba 1.5 Mini**[17]: A compact version of Jamba 1.5, optimized for efficiency while maintaining high performance in generative AI tasks.
- **LLaMa 3.1 Instruct**[18]: Meta's instruction-tuned model with 405 billion parameters and optimized for multilingual dialogue use cases, outperforming many open-source and closed chat models on common benchmarks.
- **LLaMa 3.2 Vision Instruct**[19]: An instruction-tuned model with 90 billion parameters optimized for visual recognition, image reasoning, captioning, and answering general questions about images.
- **Mistral Large 2**[20]: Mistral AI's dense, transformer-based LLM with 123 billion parameters, offering significant improvements in code generation, mathematics, and reasoning.
- **Mistral Nemo**[21]: Mistral AI's multilingual open-source model using a new tokenizer, Tekken, for efficient compression of natural language text and source code.

[9] https://openai.com/index/hello-gpt-4o/
[10] https://www.anthropic.com/news/claude-3-5-sonnet
[11] https://docs.cohere.com/v2/docs/command-r-plus
[12] https://api-docs.deepseek.com/news/news1210
[13] https://deepseekcoder.github.io/
[14] https://x.ai/news/grok-2
[15] https://x.ai/news/grok-2
[16] https://huggingface.co/ai21labs/AI21-Jamba-Large-1.5
[17] https://huggingface.co/ai21labs/AI21-Jamba-Mini-1.5
[18] https://huggingface.co/meta-llama/Llama-3.1-405B-Instruct
[19] https://huggingface.co/meta-llama/Llama-3.2-90B-Vision-Instruct
[20] https://ollama.com/library/mistral-large
[21] https://mistral.ai/news/mistral-nemo

- **Molmo**[22]: Allen Institute for AI's open vision-language model trained on a curated dataset of image-text pairs, achieving state-of-the-art performance among similar-sized multimodal models.
- **Nemotron 4**[23]: NVIDIA's large language model with 340 billion parameters, integrated into a synthetic data generation pipeline, assisting researchers and developers in building custom LLMs.
- **Pixtral Large**[24]: A Mistral model designed for multimodal tasks, including image generation and understanding, though specific details are limited.
- **Qwen 2.5 Instruct**[25]: An instruction-tuned model by Alibaba's Qwen team, optimized for various language tasks, including code generation and comprehension.

5.3 LLASP Fine-tuning

The subsequent stage of our methodology focuses on fine-tuning a lightweight language model and evaluating its performance on the problem categories introduced in Section 4.

To this end, we construct a dedicated training dataset by generating multiple instances derived from the templates associated with each task category. Concretely, for each problem class $\mathcal{T}_{\mathcal{C}^i}$, we create a variety of template instantiations to produce a collection of natural language prompts, each paired with a corresponding, semantically accurate ASP encoding.

Template instantiations are obtained using a predefined vocabulary of predicates, constants, and labels, as previously described in Section 4. Specifically, the vocabulary used to instantiate the LLASP dataset comprises 50 distinct words for the training set and 40 distinct words for the test set, with the two sets being obviously disjoint. The instantiated problems were generated by randomly sampling from the respective vocabularies with replacement. It is worth noting that, although the resulting instances may not always be semantically meaningful to humans (e.g., "Assign exactly one label from a given set of labels to a set of elements. The set of elements is expressed by the predicate city. The labels are red, tree, sky."), this lack of semantic coherence is irrelevant to the training objective, which is solely to learn the correct ASP encoding corresponding to each textual description.

The final dataset, denoted as $\mathcal{D}$, comprises approximately 4 million prompt–program pairs, with distribution across problem categories reflecting their relative complexity, as detailed in Table 2. The number of instances generated per task type is influenced by the syntactic complexity of the corresponding ASP encodings: tasks that permit greater structural variability, such as a higher number of predicates, atom arities, or ground instances, demand more extensive training data to ensure robust model learning.

The dataset is partitioned into training and validation subsets using an 80–20 split, while preserving the proportion of examples per task category.

For fine-tuning, we selected Gemma 2B as our model of choice. As the smallest model among those evaluated, it provides a conservative baseline and enables a clearer

[22] https://huggingface.co/allenai/Molmo-7B-D-0924
[23] https://huggingface.co/nvidia/Nemotron-4-340B-Instruct
[24] https://mistral.ai/news/pixtral-large
[25] https://huggingface.co/Qwen/Qwen2.5-72B-Instruct

Table 2: Number of tuples and relative proportions of each problem within the dataset $\mathcal{D}$.

Problem	**No. Tuples**	**Proportion (%)**
Assignment	1,000,000	27
Constraint	500,000	13.5
Combination	100,000	2.7
Join	900,000	24.3
Transitive closure	100,000	2.7
Preference	400,000	10.8
Value filtering	100,000	2.7
Negative filtering	100,000	2.7
Numeric filtering	500,000	13.5
Total	3,700,000	100

assessment of the impact of our fine-tuning strategy. Supervised Fine-Tuning (SFT) is carried out using the `SFTTrainer` module from the `trl` libra

Table 3: Number of tuples and relative proportions of each problem within the dataset $\mathcal{D}^{\text{Inv}}$.

Problem	**No. Tuples**	**Proportion (%)**
Assignment	128,000	13.9
Constraint	64,000	7
Combination	56,000	6.1
Join	288,000	31.3
Transitive closure	64,000	7
Preference	32,000	3.5
Value filtering	96,000	10.4
Negative filtering	32,000	3.5
Numeric filtering	160,000	17.4
Total	920,000	100

5.4 Prompt Invariance

We aim at further exploring the impact of fine-tuning by making the code generation *prompt-invariant*, i.e., not constrained to the templating formats described in Section 4. To achieve this result, starting from $\mathcal{D}$, we construct an augmented dataset $\mathcal{D}^{\text{Inv}}$ as follows. For each problem in $\mathcal{D}$, we generate 21 variations semantically equivalent to the original prompt, exploiting Copilot for the rephrasing task. Out of these 21 variations, 20 has been used for training and the last one for testing. The proportion of problems in $\mathcal{D}^{\text{Inv}}$ is provided in Table 3. We used this new dataset to further refine the fine-tuned version of Gemma 2B, previously trained on $\mathcal{D}$. We refer to this new

Table 4: Number of tuples and relative proportions of each problem within the dataset $\mathcal{D}^{\text{Complex}}$.

Problem	**No. Tuples**	**Proportion (%)**
Assignment + Constraint	1,080,000	31.3
Join + Filtering	1,440,000	41.7
Combination + Negative Filtering	936,000	27
Total	3,456,000	100

model as LLASP$^{\text{Inv}}$. For consistency, the test set $\mathcal{D}_{test}^{\text{Inv}}$ used for evaluation consists of 9,000 tuples, as $\mathcal{D}_{test}$.

5.5 Complex Problems

Finally, we want to assess if more complex problems can be encoded via proper fine-tuning, beyond core tasks. To achieve this goal, we construct a new dataset $\mathcal{D}^{\text{Complex}}$, embracing three representative problems: **assignment + constraint**, consisting of a guess followed by a strong constraint; **join + filtering**, which executes a join and then select a predicate by a specific value; and **combination + negative filtering**, generating a combination from two sets and further filtering excluding a given predicate. We construct $\mathcal{D}^{\text{Complex}}$ by (i) concatenating the rules of the individual core tasks which constitute each complex problem, and (ii) instantiating them with the protocol described in Section 4. The statistics about this new dataset are depicted in Table 4. Different from the prompt-variance task, we used $\mathcal{D}^{\text{Complex}}$ to train the original Gemma 2B model, since the new knowledge has to be injected from scratch. We denote this new model as LLASP$^{\text{Complex}}$. Finally, similar to $\mathcal{D}_{test}$, $\mathcal{D}_{test}^{\text{Complex}}$ consists of 9,000 tuples with random predicates and labels that have never been used in the training set.

5.6 Evaluation Protocol

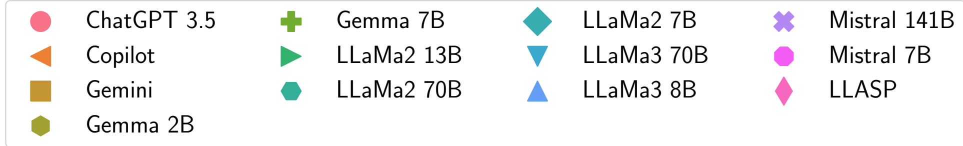
To evaluate the correctness of the generated ASP encodings, we utilize the *Clingo* ASP solver (Gebser et al. 2016) through its dedicated Python API[26]. Clingo is employed to compute the answer sets of a given ASP program P, thereby defining the function $s(P) = AS(P)$, where $AS(P)$ denotes the (possibly empty) set of answer sets associated with P.

Given a prompt x, let $y \sim P_{\mathcal{L}}(\cdot|x)$ be the ASP program generated by the model $\mathcal{L}$, and let y^* be the corresponding gold-standard encoding for x. We construct a set of facts F_{y^*} that encode the specific instance of the problem described by x. From these, we define two ASP programs: $P = y \cup F_{y^*}$ and $P^* = y^* \cup F_{y^*}$.

We then conduct two levels of evaluation:

- **Syntactic correctness:** We execute Clingo on P and check for parsing errors. If no such errors occur, we consider the generated program y to be syntactically valid.

[26] https://potassco.org/clingo/python-api/5.4/



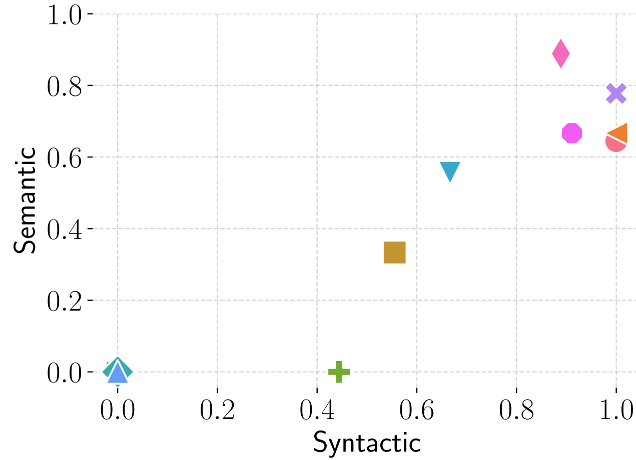


Fig. 2: Comparative analysis on both syntactic and semantic results. Some markers are hidden for overlapping in (0, 0).

- **Semantic correctness:** We compute the answer sets $s(P)$ and $s(P^*)$, and compare them. If $AS(P) = AS(P^*)$, we declare a semantic match, indicating that the generated program y is functionally equivalent to the gold standard y^* for the given problem instance.

6 Results

In the following section, we present the results of our experiments. Specifically, we first evaluate the capability of pioneering LLMs in encoding ASP programs compared to our framework LLASP. Next, as enhancements to the conference version of this work, we delve into the fine-tuning methodology, evaluating the performance of LLASP tailored for the prompt-invariance task and for more complex ASP encoding beyond core problems.

6.1 Pioneering LLMs vs LLASP

In the initial set of experiments, we perform a comparative evaluation between the pre-trained models described in Section 5.1 and LLASP, our fine-tuned instance of Gemma 2B. It is important to note that many of the pre-trained LLMs do not provide direct access via APIs. As a result, for each task category, we utilize the specific instantiations

Table 5: Comparative analysis in syntactic and semantic terms. Bold and underline entries denote the best and second-best scores, respectively.

Model	*Syntactic*	*Semantic*
ChatGPT 3.5	**1.**	0.64
Copilot	**1.**	0.67
Gemini	0.56	0.33
Gemma 2B	0.	0.
Gemma 7B	0.44	0.
LLaMa2 7B	0.	0.
LLaMa2 13B	0.	0.
LLaMa2 70B	0.	0.
LLaMa3 8B	0.	0.
LLaMa3 70B	0.67	0.56
Mistral 7B	0.91	0.67
Mistral 141B	**1.**	0.78
LLASP	0.89	**0.89**

introduced in Section 4. The corresponding prompts are submitted to each model, and the generated outputs are evaluated using the methodology described earlier.

To account for the stochastic nature of LLMs during sampling, each experiment is repeated five times, thereby ensuring a degree of statistical robustness in the results.

Table 5 presents the syntactic and semantic accuracy achieved by all evaluated pioneering models, while Figure 2 offers a visual summary of the outcomes. It is worth emphasizing that syntactic correctness does not guarantee semantic equivalence. For example, although Gemma 7B achieves a 44% syntactic accuracy rate, none of its outputs meet the semantic criteria.

Overall, no model achieves full correctness across all tasks. Lightweight models generally perform less effectively, with the exception of Mistral 7B, which demonstrates comparatively strong results.

Among the evaluated systems, ChatGPT, Copilot, and Mistral 141B attain perfect syntactic accuracy (100%). Interestingly, both Gemini and LLaMa3 70B exhibit lower performance, despite being of comparable scale. LLASP stands out as the model with the highest semantic accuracy across all evaluated LLMs.

A particularly notable observation is that, in the case of LLASP, syntactically valid outputs frequently correspond to semantically correct programs as well. Table 5 supports this finding, showing that this alignment between syntax and semantics is exclusive to LLASP. This is a remarkable outcome, especially when contrasted with the original (non-fine-tuned) Gemma 2B model, which produces entirely inconsistent results.

Table 6 further details the results for each task. Compared to large sized models, LLASP only fails on the join task, due to syntactic fails. By contrast, Mistral 141B exhibits semantic failures both on assignment and negative filtering.

Table 6: Generation accuracy in terms of syntactic (*Syn.*) and semantic (*Sem.*) perspective across all the considered pioneering LLMs. The provided scores represent the proportion of correct encodings over the total number of trials.

Model	**Assignment**		**Constraint**		**Combination**		**Join**		**Closure**		**Preference**		**Value Filter**		**Neg. Filter**		**Num. Filter**	
	Syn.	*Sem.*	*Syn.*	*Sem.*	*Syn.*	*Sem.*	*Syn.*	*Sem.*	*Syn.*	*Sem.*	*Syn.*	*Sem.*	*Syn.*	*Sem.*	*Syn.*	*Sem.*	*Syn.*	*Sem.*
ChatGPT 3.5	1.	0.8	1.	1.	1.	1.	1.	1.	1.	1.	1.	0.	1.	0.	1.	0.	1.	1.
Copilot	1.	0.	1.	1.	1.	1.	1.	1.	1.	1.	1.	0.	1.	1.	1.	0.	1.	1.
Gemini	0.	0.	1.	0.	0.	0.	1.	1.	1.	1.	0.	0.	0.6	0.	0.4	0.	1.	1.
Gemma 2B	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.
Gemma 7B	0.	0.	1.	0.	0.	0.	1.	0.	0.	0.	0.	0.	0.	0.	1.	0.	1.	0.
LLaMa2 7B	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.
LLaMa2 13B	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.
LLaMa2 70B	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.
LLaMa3 8B	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.
LLaMa3 70B	0.8	0.2	0.4	0.4	1.	1.	0.6	0.6	1.	1.	0.	0.	0.6	0.6	0.6	0.2	1.	1.
Mistral 7B	0.8	0.	1.	0.	1.	1.	1.	1.	1.	1.	0.4	0.	1.	1.	1.	1.	1.	1.
Mistral 141B	1.	0.	1.	1.	1.	1.	1.	1.	1.	1.	1.	1.	1.	1.	1.	0.	1.	1.
LLASP	1.	1.	1.	1.	1.	1.	0.	0.	1.	1.	1.	1.	1.	1.	1.	1.	1.	1.

Table 7: In-depth comparison using an extended dataset. Bold and underline entries denote the best and second-best scores, respectively.

Model	*Syntactic*	*Semantic*
ChatGPT 3.5	**1.**	0.67
Copilot	**1.**	0.57
Mistral 7B	**1.**	<u>0.85</u>
Mistral 141B	**1.**	0.69
LLASP	<u>0.93</u>	**0.93**

To further analyze the shortcomings of generated outputs, we offer anecdotal insights into the ASP encodings produced by the top-performing models identified in Figure 2: Mistral 141B, ChatGPT 3.5, Copilot, and Mistral 7B. Our focus here is on semantic correctness, which serves as the most meaningful indicator of quality in the context of ASP generation.

Surprisingly, all of the aforementioned models, except ChatGPT 3.5, consistently fail to generate semantically correct ASP encodings for the Guessing Assignment task, resulting in a semantic accuracy score of 0.0 (see Table 6). In contrast, ChatGPT 3.5 successfully produces a correct encoding in four out of five attempts, achieving a score of 0.8.

We highlight that, when evaluating LLASP, the generated program is compared against a single gold standard reference. During the fine-tuning phase, we deliberately restricted the set of valid encodings to a single canonical form (although multiple encodings could, in principle, be correct) in order to ensure training stability and achieve convergence. Since the model was trained exclusively to recognize this specific encoding, there is no risk of false negatives, that is, instances where a semantically equivalent program would be incorrectly classified as incorrect.

In contrast, for the evaluation of all other models (whose outputs could include multiple equivalent encodings), each of the five generated programs was manually validated by the authors, who possess expertise in ASP. Consequently, we can confidently regard the reported results as accurate.

Below, we present the encodings produced by each model in response to the Guessing Assignment prompt described in Section 4, alongside the corresponding output generated by LLASP.

Mistral 7B / 141B. While the generated programs are syntactically valid, they exhibit a semantic issue: they restrict the solution space by disallowing multiple assignments to the same label. This constraint is not specified in the prompt and leads to logical inconsistency when the number of labels is smaller than the number of elements to be labeled.

```
Encoding:
    1 { assigned(X, L) : label(L) } 1 :- city(X).
    :- assigned(X1, L), assigned(X2, L), X1 != X2.
```

ChatGPT 3.5. Similarly, the generated output constitutes a syntactically valid ASP encoding as verified by the Clingo solver; however, it fails to accurately capture the intended semantics of the textual prompt.

```
Encoding:
    1 { assign(X, L) : label(L) } 1 :- city(X).
    :- assign(X, L1), assign(Y, L1), X != Y, label(L1).
    :- city(X), not assign(X, _).
```

Copilot. In this case, the generated program conveys a semantics that deviates entirely from the intended problem description. Specifically, the provided fragment defines a join between the predicates `city` and `label` using the common term `City`, which does not align with the prompt's requirements.

```
Encoding:
    assign_label(City, Label) :-
        city(City), label(City, Label).
```

LLASP. Finally, we present an example of an incorrect ASP program generated by LLASP for the join task described in Section 4. The encoding contains a syntactic error related to the arity of the `owner` predicate, as well as an unintended inclusion of the `Z` predicate, both of which compromise the semantic correctness of the program.

```
Encoding: owner_restaurant(X,Z):-owner(X,Y),Z(Y).
```

As an additional evaluation, aimed at further validating the robustness of our findings, we compare LLASP with the previously considered models using an extended set of test instances. In this setting, prompts are varied with respect to the predicates and labels involved. The results of this comparison are summarized in Table 7.

Once again, LLASP demonstrates superior performance. Despite occasional syntactic issues, it consistently produces more reliable ASP encodings, outperforming the second-best model (Mistral 7B) by nearly ten percentage points in overall accuracy. Furthermore, we observe that whenever the generated encoding is syntactically correct, the semantics of the corresponding problem are also preserved.

Encouraged by these findings, we conducted a large-scale evaluation of LLASP. To this end, we constructed a test set $\mathcal{D}_{test}$ comprising 9,000 prompts, generated following the same methodology used for the training dataset $\mathcal{D}$, but instantiated with a distinct set of predicates and labels. The examples are uniformly distributed across the various task categories.

Table 8 reports LLASP's accuracy on $\mathcal{D}_{test}$. We make the following key observations: (i) in contrast to previous experiments, syntactic correctness does not always guarantee semantic correctness; (ii) nonetheless, the overall quality of the generated ASP programs remains high across all tasks, including the join task, which previously exhibited a failure case.

Given the scale and diversity of $\mathcal{D}_{test}$, this experiment provides strong evidence for the robustness and generalization capabilities of the proposed approach.

Table 8: Results in terms of syntactic and semantic generation accuracy of LLASP over $\mathcal{D}_{test}$. The overall performance of LLASP is highlighted in bold.

Problem	*Syntactic*	*Semantic*
Assignment	0.76	0.73
Constraint	1.	1.
Combination	1.	0.81
Join	0.95	0.91
Closure	1.	0.99
Preference	1.	0.88
Value Filtering	1.	0.89
Negative Filtering	1.	0.89
Numeric Filtering	1.	0.9
Total (avg)	**0.97**	**0.89**

6.2 State-of-the-art LLMs

As a natural extension of the conference paper, we considerably expand the evaluation of LLMs in ASP coding to state-of-the-art models. Table 9 presents the full set of generation scores, evaluating both syntactic and semantic accuracy across the set

Table 9: Generation accuracy in terms of syntactic (*Syn.*) and semantic (*Sem.*) perspective across all the considered state-of-the-art LLMs. The provided scores represent the proportion of correct encodings over the total number of trials.

Model	Assignment		Constraint		Combination		Join		Closure		Preference		Value Filter		Neg. Filter		Num. Filter	
	Syn.	*Sem.*	*Syn.*	*Sem.*	*Syn.*	*Sem.*	*Syn.*	*Sem.*	*Syn.*	*Sem.*	*Syn.*	*Sem.*	*Syn.*	*Sem.*	*Syn.*	*Sem.*	*Syn.*	*Sem.*
ChatGPT-4o	1.	1.	1.	1.	1.	1.	1.	1.	1.	1.	1.	1.	1.	1.	1.	1.	1.	1.
Claude 3.5 Sonnet	0.8	0.8	1.	1.	1.	1.	1.	1.	1.	1.	1.	1.	1.	1.	1.	1.	1.	1.
Command R+	0.	0.	1.	0.8	1.	1.	1.	1.	1.	1.	0.	0.	1.	1.	1.	1.	1.	1.
DeepSeek 2.5	1.	1.	0.8	0.4	1.	1.	1.	1.	1.	1.	0.8	0.	1.	1.	1.	1.	1.	1.
DeepSeek Coder 2.5	1.	0.	0.8	0.	1.	0.	1.	1.	1.	1.	1.	0.	1.	1.	1.	1.	1.	1.
Grok 2	1.	0.8	0.8	0.8	1.	0.4	1.	1.	1.	0.4	1.	0.8	1.	1.	0.6	0.2	1.	1.
Grok 2 Mini	1.	0.8	1.	0.	1.	1.	1.	1.	1.	1.	1.	0.4	1.	1.	0.6	0.	1.	1.
Jamba 1.5 Large	1.	1.	1.	1.	1.	1.	1.	1.	1.	1.	1.	0.	1.	0.8	1.	0.	1.	1.
Jamba 1.5 Mini	0.8	0.	1.	1.	0.2	0.	1.	1.	0.6	0.6	1.	0.	1.	0.	1.	1.	1.	1.
LLaMa 3.1 Instruct	1.	0.4	1.	0.6	1.	0.6	1.	0.6	0.8	0.4	0.6	0.2	1.	0.4	1.	0.6	1.	1.
LLaMa 3.2 Vision Instruct	1.	0.4	0.8	0.	0.8	0.4	0.6	0.4	0.6	0.4	1.	0.	1.	0.4	1.	0.4	1.	0.8
Mistral Large 2	1.	1.	1.	1.	1.	0.6	1.	1.	1.	1.	0.	0.	1.	1.	0.6	0.6	1.	1.
Mistral Nemo	0.	0.	1.	0.4	1.	1.	1.	1.	1.	1.	1.	0.	0.8	0.4	0.8	0.8	0.4	0.2
Molmo	0.	0.	0.	0.	0.8	0.8	1.	1.	1.	1.	1.	0.	1.	1.	1.	0.	1.	1.
Nemotron 4	1.	1.	0.8	0.	1.	0.4	1.	0.4	1.	1.	1.	0.	1.	0.8	1.	0.	1.	1.
Pixtral Large	1.	0.2	1.	0.	1.	0.8	1.	1.	1.	1.	0.	0.	1.	1.	1.	1.	1.	1.
Qwen 2.5 Instruct	1.	1.	1.	1.	1.	0.8	1.	1.	1.	1.	1.	0.	1.	1.	1.	1.	1.	1.
LLASP	1.	1.	1.	1.	1.	1.	0.	0.	1.	1.	1.	1.	1.	1.	1.	1.	1.	1.

Table 10: Performance analysis in syntactic and semantic terms on state-of-the-art LLMs. Bold and underline entries denote the best and second-best scores, respectively.

Model	*Syntactic*	*Semantic*
ChatGPT-4o	**1.**	**1.**
Claude 3.5 Sonnet	<u>0.98</u>	<u>0.98</u>
Command R+	0.78	0.76
DeepSeek 2.5	0.96	0.82
DeepSeek Coder 2.5	<u>0.98</u>	0.56
Grok 2	0.93	0.71
Grok 2 Mini	0.96	0.69
Jamba 1.5 Large	**1.**	0.76
Jamba 1.5 Mini	0.84	0.51
LLaMa 3.1 Instruct	0.93	0.53
LLaMa 3.2 Vision Instruct	0.87	0.36
Mistral Large 2	0.84	0.8
Mistral Nemo	0.78	0.53
Molmo	0.76	0.53
Nemotron 4	<u>0.98</u>	0.51
Pixtral Large	0.89	0.67
Qwen 2.5 Instruct	**1.**	0.87
LLASP	0.89	0.89

of state-of-the-art LLMs under consideration. For ease of interpretation, a condensed summary of these results, reporting average scores, is provided in Table 10. Several important observations emerge from this comparison. First, when juxtaposing the performance of these contemporary LLMs against previously analyzed models (refer to Tables 5 and 6), a clear trend of substantial improvement becomes evident. For

instance, while the top-performing legacy model, Mistral 141B, attained a perfect syntactic score but only 0.78 in semantic accuracy, a number of the newer models approach or even reach near-perfect performance on both metrics. Most notably, ChatGPT-4o achieves an outstanding average score of 1.0 for both syntactic correctness and semantic fidelity, marking a significant leap in the capabilities of general-purpose language models in structured generation tasks. This constitutes a pivotal finding, as it confirms that recent advancements in LLM development have placed increasing emphasis on enhancing programming proficiency, not only in imperative languages, but also in more specialized, declarative paradigms such as ASP. The high performance attained by these models highlights the growing maturity of LLMs in understanding and generating formally constrained code. However, it is essential to contextualize these results within the broader goals of our work. While state-of-the-art models set impressive benchmarks, they are often associated with prohibitive computational demands, rendering them impractical for many real-world applications with limited resources. As such, our focus diverges from competing with these large-scale models. Instead, we aim to investigate the potential of task-specific fine-tuning strategies that are feasible under resource constraints. Specifically, we explore how well smaller or more targeted models can perform when trained on carefully curated data, and whether such models can achieve competitive results in specialized tasks. The following sections delve into this line of inquiry, illustrating the efficacy of lightweight fine-tuning approaches in domains traditionally considered challenging for language models.

Table 11: Results in terms of syntactic and semantic generation accuracy of LLASP$^{\text{Inv}}$ over $\mathcal{D}^{\text{Inv}}_{test}$. The overall performance of LLASP is highlighted in bold.

Problem	*Syntactic*	*Semantic*
Assignment	1.	0.9
Constraint	1.	1.
Combination	0.98	0.77
Join	0.99	0.99
Closure	1.	0.9
Preference	0.93	0.93
Value Filtering	1.	0.83
Negative Filtering	1.	0.98
Numeric Filtering	1.	0.98
Total (avg)	**0.99**	**0.92**

6.3 Prompt Invariance

The performance of the model fine-tuned for the prompt-invariance task (LLASP$^{\text{Inv}}$) is presented in Table 11, both in terms of syntactic and semantic accuracy on $\mathcal{D}^{\text{Inv}}_{test}$.

These results demonstrate that applying a targeted fine-tuning procedure, specifically designed to enhance prompt-invariance, yields strong task-specific performance. Further, they show that the average performance of $\text{LLASP}^{\text{Inv}}$ on the $\mathcal{D}_{test}^{\text{Inv}}$ dataset exceeds the performance achieved by the original LLASP model on $\mathcal{D}_{test}$, demonstrating the effectiveness of the inversion-based approach (see Table 8). This performance gain underscores the effectiveness of task-oriented fine-tuning in adapting LLMs to specialized objectives. Notably, the improvement is particularly significant in the context of constrained domains such as declarative programming, where the model is required to generalize across a narrow but systematically structured set of prompts. These findings highlight the importance of aligning fine-tuning strategies with the structural and semantic demands of the target task to fully exploit the capabilities of pre-trained language models.

Table 12: Results in terms of syntactic and semantic generation accuracy of $\text{LLASP}^{\text{Complex}}$ over $\mathcal{D}_{test}^{\text{Complex}}$. The overall performance of LLASP is highlighted in bold.

Problem	*Syntactic*	*Semantic*
Assignment + Constraint	1.	1.
Join + Filtering	1.	1.
Combination + Negative Filtering	1.	0.97
Total (avg)	**1.**	**0.99**

6.4 Complex Problems

Finally, Table 12 reports the syntactic and semantic evaluation scores obtained on the $\mathcal{D}_{test}^{\text{Complex}}$ dataset by $\text{LLASP}^{\text{Complex}}$, a model trained from scratch to handle the generation of more complex problems extending beyond the core task set. In this more demanding setting, characterized by programs composed of multiple interrelated rules, the model demonstrates near-perfect performance across both syntactic and semantic dimensions. These results provide compelling evidence that task-specific training, especially from scratch, is crucial when the objective extends beyond fine-tuning pre-existing capabilities and instead requires the acquisition of fundamentally new knowledge or reasoning patterns. The effectiveness of $\text{LLASP}^{\text{Complex}}$ in this context underscores the necessity of tailoring the model development pipeline, including the training paradigm and data design, to the structural and cognitive complexity of the target domain. This also reinforces the broader insight that adapting the fine-tuning strategy to match the specificity and novelty of the task is key to achieving robust generalization and high-quality output in structured domains such as logic-based program synthesis.

7 Discussion

In the following, we address key aspects of our research, including the comparison between LLASP and the evaluated cutting-edge LLMs, as well as the constraints of the proposed method.

LLASP vs state-of-the-art. Our research indicates that LLASP offers a good balance between accuracy and efficiency. While some state-of-the-art models tested reach nearly flawless results (refer to Table 10), we observe that these models are either proprietary, available only via paid APIs (e.g., ChatGPT-4o, Claude Sonnet 3.5), or open-source but require significant computational resources for suitable adoption (e.g., Mistral Large 2). In contrast, our work demonstrates that competitive and reliable performance can be achieved through targeted fine-tuning at minimal cost, leveraging a free and lightweight model with only 2 billion parameters. This result emphasizes that strong task-specific capabilities do not necessarily require large-scale models or significant infrastructure investments. We argue that this direction is particularly valuable in scenarios where the end user:

- Seeks to apply LLMs to a specific task or a custom dataset.
- Lacks access to extensive computational resources or the financial means to rely on costly commercial APIs for frequent queries.
- Requires greater flexibility and customizability in adapting the model to specific application needs.

The recent literature has devoted increasing attention to such principles of *Green AI* (Ashraf et al. 2025b; Islam et al. 2025; Ashraf et al. 2025a), which advocate for AI systems that are both sustainable and energy-efficient, aiming to achieve high performance while minimizing environmental impact. Our methodology complies with this viewpoint, encouraging responsible and accessible AI development by optimizing the use of computational resources.

Limitations. Despite the provided insights, our study presents some limitations. First, we do not exhaustively investigate the impact of prompt engineering on model performance, since in the prompt-invariance task, we devise slightly different variations of the same instruction. Next, we limit our investigation to the ASP generic encodings discussed in Section 4, thus not considering domain-specific problems (e.g., graph-related tasks).

Further, in this extension, we show that a targeted fine-tuning is effective also in case of more complex problems beyond core tasks. We create such problems by concatenating simple rules, discarding meaningless and/or inconsistent constructs and choosing sub-rules which resemble ASP standard programs (such as "Guess + Constraint"). However, other choices can be addressed. For instance, future research could explore a specific domain of typical ASP problems, e.g., graphs domain, combinatorial problems, scheduling tasks.

Another assumption consists in encoding a single solution for each problem, thus limiting the freedom of the model in generating the programs. A possible solution would consist in introducing a *normalizer*, which is in charge to match the generated and the correct programs under a *semantic* perspective, with no structure constraints.

Equivalently, one could rely on a ad-hoc benchmarking framework, dedicated to declarative paradigms, and on custom metrics implemented for assessing the alignment of the two programs.

8 Conclusions and Future Work

In this work, we conducted a comprehensive investigation into the ability of Large Language Models (LLMs) to generate correct Answer Set Programming (ASP) encodings from natural language specifications. To the best of our knowledge, this represents the first in-depth comparative study of state-of-the-art LLMs in the context of ASP code generation.

Our results reveal that, despite their remarkable capabilities, original foundational LLMs fell short in this domain, exhibiting limited reliability in producing syntactically and semantically correct ASP programs. In this regard, we considerably expand the analysis of state-of-the-art models from a wide set of LLM families in performing ASP translation from textual description. In contrast to older models, we find that state-of-the-art LLMs show significant improvements, sometimes achieving a perfect score.

However, we demonstrate that task-specific fine-tuning of even small and lightweight LLMs can substantially improve performance and enable practical applicability, thus not necessitating huge computational power.

To this end, we introduce **LLASP**, a fine-tuned variant of the lightweight Gemma 2B base model. LLASP is trained on a purpose-built dataset designed to capture a broad range of fundamental ASP patterns, by relying on templating formats. Empirical results show that LLASP significantly outperforms larger and more powerful pre-trained models, particularly in terms of semantic accuracy, thus validating the effectiveness of our fine-tuning methodology. Additionally, we further explore the effectiveness of fine-tuning on Gemma 2B, showing results in two-fold directions: (i) the model can be trained to be *prompt-invariant*, thus discarding the need for templating; and (ii) longer and more *complex* problems can be effectively encoded, beyond core tasks.

We argue that our findings highlight promising opportunities for future research and suggest several directions for expanding and enhancing this line of investigation. First, the development of a novel evaluation metric is warranted, e.g., one that jointly considers syntactic correctness and the practical effectiveness of the generated ASP programs in solving the problems described by the prompts.

Moreover, we plan to explore new controlled training strategies that embed syntactic awareness of ASP directly into the learning process. This endeavor would necessitate a detailed examination of the model's internal architecture, along with the implementation of appropriate architectural adaptations to support such integration.

Finally, alternative approaches beyond fine-tuning merit investigation. In particular, techniques such as advanced prompt engineering or training a model from scratch tailored specifically for ASP generation represent compelling avenues for future research.

Acknowledgements

Acknowledgments

This work has been partially funded by: (1) MUR on D.M. 351/2022, PNRR Ricerca, CUP H23C22000440007; (2) PNRR project FAIR - Future AI Research (PE00000013), Spoke 9 - Green-aware AI, under the NRRP MUR program funded by the "NextGenerationEU"; (3) PNRR project Tech4You "Technologies for climate change adaptation and quality of life improvement", CUP H23C22000370006, under the NRRP MUR program funded by the "NextGenerationEU"; PNRR Project "SoBigData.it – Strengthening the Italian RI for Social Mining and Big Data Analytics", CUP B53C22001760006, under the NRRP MUR program funded by the "NextGenerationEU"; PNRR project SERICS - Security and Rights in the CyberSpace (PE00000014), CUP B29J24000680005, under the NRRP MUR program funded by the "NextGenerationEU"; (4) by MIMIT under project EI-TWIN n. F/310168/05/X56 CUP B29J24000680005, and project ASVIN n. F/360050/01-02/X75 CUP B29J2400020000. Francesco Calimeri and Francesco Ricca are members of the Gruppo Nazionale Calcolo Scientifico-Istituto Nazionale di Alta Matematica (GNCS-INdAM).

References

Ashraf H, Danish SM, Leivadeas A, et al (2025a) Energy-aware code generation with llms: Benchmarking small vs. large language models for sustainable ai programming. arXiv:2508.08332

Ashraf H, Danish SM, Rahman S, et al (2025b) Toward green code: Prompting small language models for energy-efficient code generation. arXiv:2509.09947

Baral C (2010) Knowledge Representation, Reasoning and Declarative Problem Solving, 1st edn. Cambridge University Press, USA, https://doi.org/10.1017/CBO9780511543357

Baral C, Dzifcak J (2012) Solving puzzles described in english by automated translation to answer set programming and learning how to do that translation. In: Proceedings of the Thirteenth International Conference on Principles of Knowledge Representation and Reasoning. AAAI Press, United States, KR'12, p 573–577, https://doi.org/10.5555/3031843.3031913

Borroto M, Kareem I, Ricca F (2024) Towards automatic composition of ASP programs from natural language specifications. In: IJCAI, vol abs/2403.04541. ijcai.org, Cham, p to appear, https://doi.org/10.24963/ijcai.2024/685

Brewka G, Eiter T, Truszczyński M (2011) Answer set programming at a glance. Commun ACM 54(12):92–103. https://doi.org/10.1145/2043174.2043195

Calimeri F, Gebser M, Maratea M, et al (2016) Design and results of the fifth answer set programming competition. Artif Intell 231:151–181. https://doi.org/10.1016/J.ARTINT.2015.09.008

Caruso S, Dodaro C, Maratea M, et al (2024) CNL2ASP: converting controlled natural language sentences into ASP. Theory Pract Log Program 24(2):196–226. https://doi.org/10.1017/S1471068423000388

Chen M, et al (2021) Evaluating large language models trained on code. arXiv:2107.03374

Coppolillo E, Calimeri F, Manco G, et al (2024) Llasp: fine-tuning large language models for answer set programming. In: Proceedings of the 21st International Conference on Principles of Knowledge Representation and Reasoning, KR '24, https://doi.org/10.24963/kr.2024/78

Dakhel AM, Majdinasab V, Nikanjam A, et al (2023) Github copilot AI pair programmer: Asset or liability? J Syst Softw 203:111734. https://doi.org/10.1016/j.jss.2023.111734

Eiter T, Ianni G, Krennwallner T (2009) Answer set programming: A primer. In: Tessaris S, Franconi E, Eiter T, et al (eds) Reasoning Web. Semantic Technologies for Information Systems, 5th International Summer School 2009, Brixen-Bressanone, Italy, August 30 - September 4, 2009, Tutorial Lectures, Lecture Notes in Computer Science, vol 5689. Springer, Cham, pp 40–110, https://doi.org/10.1007/978-3-642-03754-2_2

Erdem E, Yeniterzi R (2009) Transforming controlled natural language biomedical queries into answer set programs. In: Cohen KB, Demner-Fushman D, Ananiadou S, et al (eds) Proceedings of the BioNLP 2009 Workshop. Association for Computational Linguistics, Boulder, Colorado, pp 117–124, URL https://aclanthology.org/W09-1315/

Ernst NA, Bavota G (2022) Ai-driven development is here: Should you worry? IEEE Softw 39(2):106–110. https://doi.org/10.1109/MS.2021.3133805

Fang M, Tompits H (2018) An approach for representing answer sets in natural language. In: Seipel D, Hanus M, Abreu S (eds) Declarative Programming and Knowledge Management. Springer International Publishing, Cham, pp 115–131, https://doi.org/10.1007/978-3-030-00801-7_8

Gebser M, Kaminski R, Kaufmann B, et al (2013) Answer Set Solving in Practice, 1st edn. Synthesis Lectures on Artificial Intelligence and Machine Learning, Springer, Cham, https://doi.org/10.1007/978-3-031-01561-8

Gebser M, Kaminski R, Kaufmann B, et al (2016) Theory solving made easy with clingo 5. In: ICLP (Technical Communications), OASIcs, vol 52. Schloss Dagstuhl

- Leibniz-Zentrum für Informatik, Austria, pp 2:1–2:15, https://doi.org/10.4230/OASIcs.ICLP.2016.2

Gebser M, Leone N, Maratea M, et al (2018) Evaluation techniques and systems for answer set programming: a survey. In: Lang J (ed) Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI 2018, July 13-19, 2018, Stockholm, Sweden. ijcai.org, Cham, pp 5450–5456, https://doi.org/10.24963/IJCAI.2018/769

Gelfond M, Lifschitz V (1991) Classical negation in logic programs and disjunctive databases. New Gener Comput 9(3/4):365–386. https://doi.org/10.1007/BF03037169

Ishay A, Yang Z, Lee J (2023) Leveraging large language models to generate answer set programs. In: Proceedings of the 20th International Conference on Principles of Knowledge Representation and Reasoning, KR '23, https://doi.org/10.24963/kr.2023/37

Islam MA, Jonnala DV, Rekhi R, et al (2025) Evaluating the energy-efficiency of the code generated by llms. arXiv:2505.20324

Jiang AQ, Sablayrolles A, Mensch A, et al (2023) Mistral 7b. arXiv:2310.06825

Jiao Y, Han J, Huang C (2025) Deepvulhunter: enhancing the code vulnerability detection capability of llms through multi-round analysis. Journal of Intelligent Information Systems 63(6):2237–2264. https://doi.org/10.1007/s10844-025-00982-0

Kalliamvakou E (2022) Research: quantifying github copilot's impact on developer productivity and happiness. URL https://github.blog/news-insights/research/research-quantifying-github-copilots-impact-on-developer-productivity-and-happiness/

Li Y, Qi S, Gao C, et al (2025) Understanding the robustness of transformer-based code intelligence via code transformation: Challenges and opportunities. IEEE Transactions on Software Engineering 51(2):521–547. https://doi.org/10.1109/TSE.2024.3524461

Lifschitz V (2019) Answer Set Programming, 1st edn. Springer Cham, Cham, https://doi.org/10.1007/978-3-030-24658-7

Ma Z, Guo H, Chen J, et al (2024) Llamoco: Instruction tuning of large language models for optimization code generation. arXiv:2403.01131

Minaee S, Mikolov T, Nikzad N, et al (2024) Large language models: A survey. arXiv:2402.06196

Mitra A, Baral C (2016) Addressing a question answering challenge by combining statistical methods with inductive rule learning and reasoning. In: AAAI. AAAI

Press, USA, pp 2779–2785, https://doi.org/10.1609/aaai.v30i1.10354

Naveed H, Khan AU, Qiu S, et al (2025) A comprehensive overview of large language models. ACM Trans Intell Syst Technol 16(5). https://doi.org/10.1145/3744746

Nye MI, Tessler MH, Tenenbaum JB, et al (2021) Improving coherence and consistency in neural sequence models with dual-system, neuro-symbolic reasoning. In: NeurIPS, pp 25192–25204, https://doi.org/10.5555/3540261.3542190

Peng S, Kalliamvakou E, Cihon P, et al (2023) The impact of AI on developer productivity: Evidence from github copilot. arXiv:2302.06590

Raiaan MAK, Mukta MSH, Fatema K, et al (2024) A review on large language models: Architectures, applications, taxonomies, open issues and challenges. IEEE Access 12:26839–26874. https://doi.org/10.1109/ACCESS.2024.3365742

Raihan N, Goswami D, Puspo SSC, et al (2025) On the performance of large language models on introductory programming assignments. Journal of Intelligent Information Systems https://doi.org/10.1007/s10844-025-00968-y

Rajasekharan A, Zeng Y, Padalkar P, et al (2023) Reliable natural language understanding with large language models and answer set programming. Electronic Proceedings in Theoretical Computer Science 385. https://doi.org/10.4204/eptcs.385.27

Schneider J (2025) Mental model shifts in human-llm interactions. Journal of Intelligent Information Systems 63(5):1737–1752. https://doi.org/10.1007/s10844-025-00960-6

Schwitter R (2018) Specifying and verbalising answer set programs in controlled natural language. Theory and Practice of Logic Programming 18:691 – 705. https://doi.org/10.1017/S1471068418000327

Stahlberg F (2020) Neural machine translation: A review. J Artif Intell Res 69:343–418. https://doi.org/10.1613/jair.1.12007

Team G (2024a) Gemini: A family of highly capable multimodal models. arXiv:2312.11805

Team G (2024b) Gemma: Open models based on gemini research and technology. arXiv:2403.08295

Touvron H, et al (2023) Llama 2: Open foundation and fine-tuned chat models. arXiv:2307.09288

Vaswani A, Shazeer N, Parmar N, et al (2017) Attention is all you need. In: Proceedings of the 31st International Conference on Neural Information Processing Systems. Curran Associates Inc., Red Hook, NY, USA, NIPS'17, p 6000–6010, https://doi.

org/10.5555/3295222.3295349

Wang S, Ding L, Zhan Y, et al (2025a) Leveraging metamemory mechanisms for enhanced data-free code generation in llms. arXiv:2501.07892

Wang S, Ding L, Zhan Y, et al (2025b) Fuzzy-assisted contrastive decoding improving code generation of large language models. IEEE Transactions on Fuzzy Systems 33(8):2689–2703. https://doi.org/10.1109/TFUZZ.2025.3575060

Wang Y, Le H, Gotmare A, et al (2023) CodeT5+: Open code large language models for code understanding and generation. In: Bouamor H, Pino J, Bali K (eds) Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing. Association for Computational Linguistics, Singapore, pp 1069–1088, https://doi.org/10.18653/v1/2023.emnlp-main.68

Xu FF, Alon U, Neubig G, et al (2022) A systematic evaluation of large language models of code. In: Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming. Association for Computing Machinery, New York, NY, USA, MAPS 2022, p 1–10, https://doi.org/10.1145/3520312.3534862

Yang Z, Ishay A, Lee J (2023) Coupling large language models with logic programming for robust and general reasoning from text. In: Rogers A, Boyd-Graber J, Okazaki N (eds) Findings of the Association for Computational Linguistics: ACL 2023. Association for Computational Linguistics, Toronto, Canada, pp 5186–5219, https://doi.org/10.18653/v1/2023.findings-acl.321