You're Not Wrong, You're Just Unsatisfiable - Investigating ASP Code Generation with LLMs via Fine-tuning and Feedback Loops

Finn Alberts (852685751)[a], Arjan de Weerd (852646163)[a]

[a]*Open Universiteit, Heerlen, The Netherlands*

Abstract

Large Language Models (LLMs) show promise for generating Answer Set Programming (ASP) code from natural language, but reliably producing programs that are both syntactically and semantically correct remains challenging. Prior work has shown that iterative syntax feedback can improve compilability, while semantic correctness often lags behind. In this paper, we investigate whether combining supervised fine-tuning with a dual-stage repair architecture—consisting of syntactic validation and an LLM-based semantic feedback loop—can improve ASP code generation. We fine-tune a Qwen2.5-7B-Instruct model using a custom dataset build using publicly available Clingo code, augmented with synthetic repair tasks to explicitly train debugging behavior. We evaluate our approach on four established scheduling domains using multiple experimental setups with and without fine-tuning and semantic feedback. Our results show that fine-tuning leads to near-perfect syntactic correctness and a substantial improvement in semantic accuracy. However, the introduction of a semantic feedback loop does not yield additional gains and in some cases degrades performance due to unreliable semantic validation. These findings suggest that semantic validation itself should be treated as a learning problem, motivating future work on fine-tuning dedicated semantic validator models.

Keywords: Answer Set Programming, Large Language Models, Code Generation, Fine-tuning, Iterative Feedback, Semantic Validation, Program Repair, Declarative Programming, Logic Programming, Clingo

1. Introduction

Since the launch of ChatGPT, the popularity of Large Language Models (LLMs) has grown massively, with these models becoming accessible to the general public due to the introduction of user-friendly, conversational systems, which significantly lowered the barrier to entry for interacting with advanced AI models. Naturally, this led to these models being applied for code development as well, with significant accelerations in the development process as a result (Mohamed et al., 2025). Meanwhile, research towards the potential of LLMs for declarative programming has been limited (Coppolillo et al., 2024).

One of these declarative programming languages is Answer Set Programming (ASP). ASP is a powerful language for knowledge representation and expresses problems as logical rules. An ASP solver can then derive so-called Stable Models which represent all possible solutions given the constraints set by those rules. It should be noted, however, that specifying those logical rules can be challenging, as it requires expertise on ASP, which limits its accessibility. The use of LLMs for the generation of ASP programs could help reduce this challenge. Furthermore, ASP code generation through LLMs opens the possibility of integrating ASP (and therefore knowledge representation and reasoning) into conversational agents such as ChatGPT, similarly to how tools like Python are currently integrated. However, generating fully syntactically and semantically correct ASP programs using LLMs remains an open problem, which we will discuss further in Section 2.

2. Background

2.1. Answer Set Programming

Answer set programming is a logic programming approach that falls under the declarative paradigm, which means the valid solution is specified and a solver like Clingo is used to find it. ASP programs consist of rules that specify conditions on values of variables (Lifschitz, 2019). These rules consist of a head and a body, split by the :-, and are of the form:

$$a_0 \text{ :- } a_1, \ldots, a_m, \text{ not } a_{m+1}, \ldots, \text{not } a_n.$$

The *not* denotes negation as a failure, and a_i are called atoms. Atoms have the form $p(t_1, \ldots, t_n)$ where terms (t_i) may be constants, variables, arithmetic or functional terms. A literal is either an atom or its negation. The head of a rule is derived if all literals are satisfied, meaning positive literals are true and negated ones are not proven true. Rules with an empty body are called facts while rules with an empty head are constraints.

To add expressive power to the language, several extensions are defined such as choice rules (for selection), aggregates (for numerical reasoning) and optimization statements (for ranking solutions). The optimization statement is especially relevant as it guides the solver to prefer certain answer sets over others, while the choice rule allows the solver to decide which atoms to include. In this rule, the head is an expression in braces, representing the possible atoms that may be chosen given lower (l) and upper (u) bounds:

$$l\{a_1, \ldots, a_m\}u \text{ :- } a_{m+1}, \ldots, a_n, \text{ not } a_{n+1}, \ldots, \text{not } a_o.$$

The *generate, define and test* is a common structure to define answer set programs. The *generate* part creates the search space of possible solutions. The *test* part filters out invalid solutions by applying constraints. The *define* part specifies definitions which make the test easier to express. Examples of ASP implementations and applications can be found on the official project of the Potsdam Answer Set solving community at `https://github.com/potassco`. For a more detailed introduction to ASP we refer to Lifschitz (2019).

2.2. LLMs approaches for ASP

The use of LLMs for ASP is not new and several approaches have already been studied. One of these approaches follows a Few-Shot, Chain of Thought (CoT) prompting approach, where natural language is first converted into Controlled Natural Language (CNL) (similar to Borroto et al. (2024)) which is then fed into an LLM pipeline, where the output is built in several chained steps (Heyninck et al., 2025; O'Brien, 2025; Rosenberg, 2025). Results show that the CoT approach outperforms zero-shot approaches using a single prompt, but are not yet fully correct in both syntax and semantics. Results from Ishay et al. (2023) show similar results.

2.3. Fine-tuning

Supervised Fine-Tuning (SFT) adapts pre-trained language models to specific downstream tasks (Ouyang et al., 2022). However, fully updating parameters is computationally expensive and risks catastrophic forgetting. To address this, Parameter-Efficient Fine-Tuning (PEFT) methods like Low-Rank Adaptation (LoRA) (Hu et al., 2022) freeze the pre-trained weights and inject trainable rank decomposition matrices into the transformer layers. This approach significantly reduces memory requirements while maintaining model quality, often combined with quantization (QLoRA) (Dettmers et al., 2023) for further efficiency.

In the context of ASP, fine-tuning open-weights models offers a cost-effective alternative to proprietary LLMs. Coppolillo et al. (2024) applied QLoRA to fine-tune a lightweight Gemma-2B model on synthetic atomic patterns (LLASP). Similarly, van der Westhuizen (2025) fine-tuned a Phi-2 model using AspPy2, a framework for generating procedural templates. While these studies demonstrate the feasibility of learning ASP syntax, they rely on template-heavy data which limits linguistic diversity. Furthermore, van der Westhuizen (2025) found that chained architectures (Natural Language → Controlled Natural Language → ASP) often underperformed due to information loss.

2.4. Iterative feedback

The use of iterative feedback loops to improve code generation is an active field of study, with recent work focusing on self-correcting LLMs (Chen et al., 2023; Ravi et al., 2025). Empirical studies for improving the syntax quality of code have been conducted by Quoc et al. (2024) and Ravi et al. (2025). Quoc et al. (2024) showed how using a syntax checker and code executor for detecting and correcting code can lead to higher code quality. Ravi et al. (2025) introduce a framework using multiple iterative feedback loops. In their work, the model will try and fix incorrect code based on the syntax error until the code compiles or a retry limit is reached.

Applying the ideas of Quoc et al. (2024) and Ravi et al. (2025), Alberts et al. (2025) expanded upon the work of Heyninck et al. (2025) by introducing a syntactic feedback loop. After the generation of a line of Clingo code, the code is validated by the Clingo API and possible error messages are fed back into the LLM to be repaired. Figures 1 and 2 provide an overview of the architecture used in their work. Although they did not fine-tune an LLM for self-debugging (as suggested by Jiang et al. (2025)), results from Alberts et al. (2025) show that including an iterative feedback loop results in an improvement in syntactic correctness, although results are still not yet perfect. Research from O'Brien (2025) shows similar results. Although achieving improved syntactic correctness, semantic correctness of the generated programs lagged behind showing room for improvement (Alberts et al., 2025).

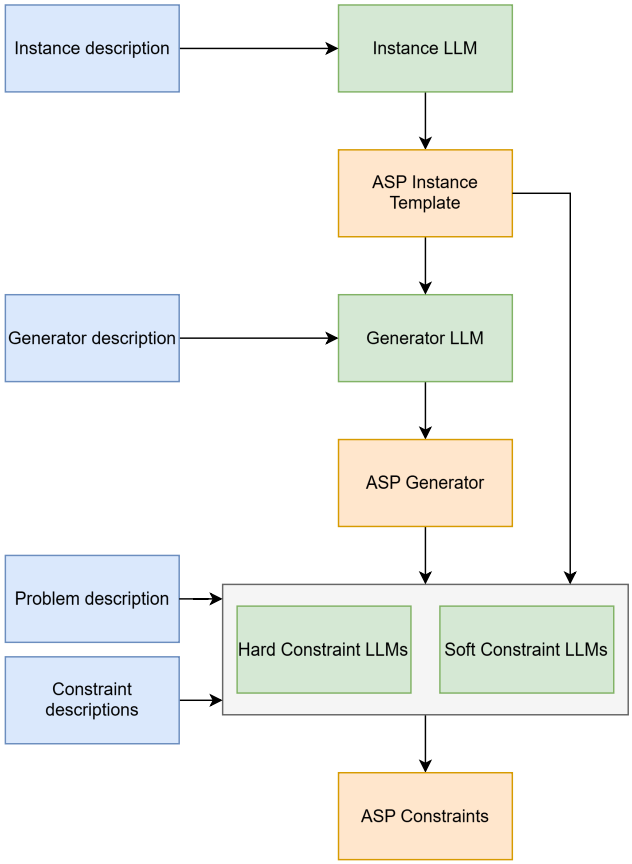

Figure 1: Chain of Thought as used by Alberts et al. (2025) and Heyninck et al. (2025). The ASP program is generated in three steps: generating instances, generating the generator, and generating the constraints. Each constraint is generated separately.

Research towards iterative feedback loops for semantic correctness is available as well. Although not specifically focused on semantic correctness, Madaan et al. (2023) proposes a framework where the model gives feedback on its own output. It is not specified what aspect the feedback should focus

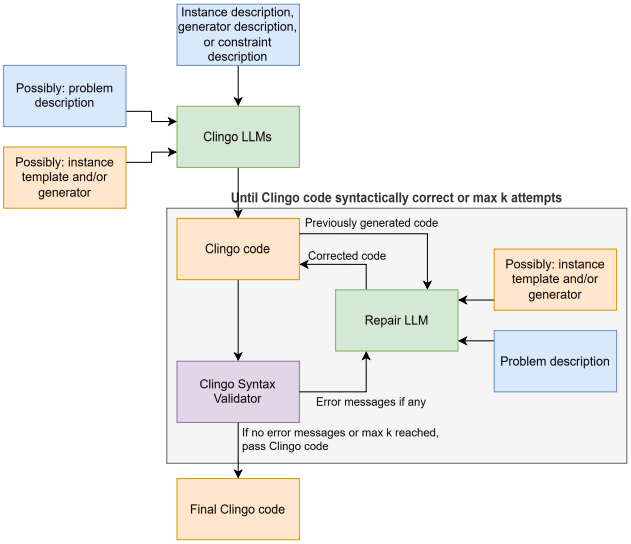

Figure 2: Integration of syntactic feedback loop by Alberts et al. (2025). This framework is applied for all LLMs from Figure 1.

on, and in the context of coding it could potentially focus on syntax or code efficiency, but also on semantics. This loop is repeated multiple times until maximum iterations are passed or the model judges that the output is satisfactory.

Chen et al. (2023) focus more on the coding domain and propose a framework where feedback is not only based on self-assessment, but on a combination of an explanation of the code (as generated by the model itself) and results from unit tests if available. Similar to Madaan et al. (2023) this process is repeated multiple times.

Furthermore, research from Jiang et al. (2025) suggests that we should train models to provide better feedback on code. Their results show an improvement in code quality using this approach.

3. Goal

In this paper, we will expand upon previous research, most notably the work from Alberts et al. (2025) and Heyninck et al. (2025), by expanding the existing feedback mechanism using the same concepts as Madaan et al. (2023) and Chen et al. (2023). We will introduce a semantic feedback loop alongside the syntactic feedback loop to investigate the impact on semantic correctness. Furthermore, we enhance the generation process by leveraging LoRA to fine-tune on a teacher-student distilled dataset based on ASP problems. Crucially, this training explicitly targets a "repair task" to internalize the debugging process found in Alberts et al. (2025) and neural program repair (Yasunaga and Liang, 2020), thereby improving the model's capability to write and fix syntactically correct ASP code. Our central research question for this paper is thus: *To what extent can a dual-stage repair architecture - combining syntax validation and semantic correction - and fine-tuning on Clingo examples improve the syntactic and semantic correctness of ASP programs generated by large language models?*

4. Data analysis

A major challenge in fine-tuning LLMs for ASP is the scarcity of high-quality, diverse datasets that align natural language with formal logic. To address this, we constructed a dataset using a Teacher-Student Distillation pipeline, designed to support modular code generation and iterative repair.

4.1. Distillation via Deconstruction

We curated a set of complex scheduling and combinatorial optimization problems from publicly available ASP problems written in Clingo (Potassco Group, 2026; Guerin, 2026; Banbara et al., 2019; Gebser et al., 2018; van der Westhuizen, 2025). Using the Gemini-3-flash-preview model as the "teacher", we decomposed these monolithic programs into modular components. The teacher was guided by a strict system prompt (see Appendix D) to:

1. **Deconstruct:** Isolate specific logic into modular categories: Instance Templates, Generators, Hard Constraints, and Soft Constraints.
2. **Refactor:** Modernize legacy ASP code into Clingo 5+ syntax (e.g., enforcing variable safety and correct aggregate syntax).
3. **Tone Variation:** Rewrite the natural language descriptions in five distinct personas (Academic, Casual Developer, Business, Succinct, Didactic) to prevent the student model from overfitting to specific keywords.

During this distillation, we implemented a Teacher-Repair Loop. Every extracted snippet was verified against the Clingo compiler. If the teacher generated invalid code, the compiler error was fed back to the teacher to self-correct the snippet before it was added to the dataset.

4.2. Synthetic Corruption for Repair Training

To enable the solver-in-the-loop capability, we augmented the dataset with a Synthetic Repair Task applied with a probability of $p = 0.5$. This ratio was selected to balance theoretical multi-task learning objectives with empirical regularization. Theoretically, the balanced split prevents catastrophic forgetting of the primary generation task while internalizing debugging logic. Empirically, we observed that higher repair probabilities led to lexical overfitting, where the model memorized specific predicates from the training set and hallucinated them into unrelated test solutions. The chosen $p = 0.5$ threshold effectively mitigates this memorization risk while establishing robust repair capabilities.

We programmatically injected errors into valid ASP snippets to mimic common LLM failure modes. The corruption logic included:

- **Syntax & Termination:** Removing periods or injecting Markdown code fences.
- **Aggregate Malformation:** Replacing ASP aggregates with invalid syntax (e.g., using `{{...}}` or pipes `|`).

- **Unsafe Variables:** Removing body literals to create unsafe variable errors.
- **Logic Drift:** Injecting Python or SQL-style operators (e.g., `and`, `or`, `==`, `<>`, `if`) to train the model to correct procedural hallucinations.

Each corrupted snippet was paired with its corresponding Clingo error message. The resulting training tuple is formatted as:

$$(\texttt{System} + \texttt{Description} + \texttt{Corrupted\_Code} + \texttt{Error\_Message}) \rightarrow (\texttt{Valid\_Code})$$

This explicitly trains the model to interpret compiler feedback and perform localized debugging.

4.3. Context-Aware Training Construction

Once the benchmark problems were deconstructed into modular components (Instance, Generator, Constraints), we transformed them into a supervised fine-tuning (SFT) dataset using the ChatML format. Crucially, a cumulative context strategy is employed to capture the inherent inter-line dependencies of ASP programs, such that each subsequent line is generated given the previously produced code.

In our pipeline, downstream components (like constraints) rely on the predicates defined in upstream components (like the instance template). To capture this during training, we constructed the user prompts cumulatively:

- **Instance Task:** The model is given only the natural language description.
- **Generator Task:** The model is given the description *and* the canonical Instance Template.
- **Constraint Task:** The model is given the description, the Instance Template, *and* the Generator.

For the Repair Task, we adopted an inverted context strategy. To simulate a debugging workflow, the User Prompt focuses exclusively on the immediate syntax error and the corrupted code. However, to ensure the model retains semantic awareness (e.g., knowing which predicates are valid), we injected the original Problem Description directly into the System Prompt. This allows the model to "reference" the problem context without distracting from the immediate error-correction task.

The exact structure of the prompts and context provided for each training task is detailed in Appendix A. To ensure the model adhered to the specific requirements of each modular task (e.g., preventing optimization logic in the generator), we utilized specialized system prompts for each stage, which are detailed in Appendix B.

5. Methodology and implementation

5.1. Semantic feedback loop

The semantic feedback loop expands upon the existing framework from Alberts et al. (2025). A general overview can be seen in Figure 3. Once syntactically correct Clingo code has been generated, another LLM, the Semantic Validation LLM, validates if the generated Clingo code matches the given intent, and if this is not the case provides a reason explaining the difference. This reason is then fed back into the Clingo LLM for repair. If no syntactically correct code could be generated (even after repair attempts), no semantic repair attempts are made. Furthermore, as semantic repair can cause new syntax issues, it is possible that the syntactic feedback loop is repeated multiple times, with the total possible amount of syntax repair iterations increasing from k for the framework from Figure 2 to $k(n+1)$ for the new framework with k being the maximal amount of iterations for the syntactic feedback loop and n being the maximal amount of iterations for the semantic feedback loop.

It should be noted that the Semantic Validation LLM can be a different model than the LLMs used for generation and repair. This allows for a fine-tuned model to be used for generation, while also being able to use a different (fine-tuned) model for semantic validation. The system prompt used for the Semantic Validation LLM can be found in Appendix C.

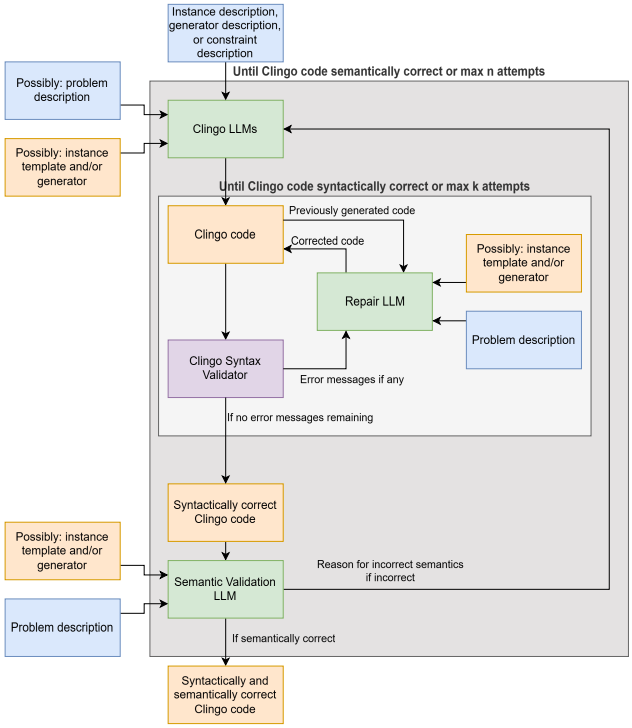

Figure 3: Integration of semantic feedback loop. The feedback loop expands upon the framework shown in Figure 2 by creating an additional loop which validates semantic correctness using an LLM.

5.2. Fine-tuning

We performed Supervised Fine-Tuning (SFT) using the Apple MLX framework, to efficiently train the `Qwen2.5-7B-Instruct` model on consumer hardware.

To determine the optimal capacity for the LoRA adapters, we conducted a comparative analysis between two configurations:

1. **Standard MLX Defaults:** The baseline configuration provided by the MLX framework ($r = 8$, dropout $= 0.0$,

$\alpha = 160.0$). This setting targets fewer parameters and is typically optimized for general text generation tasks.

2. **Logic-Optimized Settings:** A high-rank configuration adapted from van der Westhuizen (2025), designed specifically for ASP synthesis ($r = 32$, dropout $= 0.05$, $\alpha = 64$).

Our preliminary experiments demonstrated a significant performance divergence. While the standard configuration (MLX Default) was computationally lighter, it struggled to capture the rigid structural rules of ASP, frequently "hallucinating" invalid syntax. In contrast, the logic-optimized settings increased the trainable parameter count to approximately 46M (0.6%). This added capacity proved crucial for internalizing the logic of variable safety and constraints.

Based on these results, we adopted the logic-optimized configuration for all final experiments. Training was performed for a fixed duration of 900 iterations without early stopping, allowing the model to fully converge on the complex logic of the training set. We utilized an AdamW optimizer with a cosine decay scheduler, starting at a learning rate of $1e^{-5}$. The final hyperparameters are detailed in Table 1.

Parameter	Value
Base Model	Qwen2.5-7B-Instruct
Technique	QLoRA (4-bit)
LoRA Rank (r)	32
LoRA Alpha (α)	64
LoRA Dropout	0.05
Target Modules	All Linear Layers
Optimizer	AdamW
Learning Rate	$1e^{-5}$ (Cosine Decay)
Max Sequence Length	4096
Iterations	900 (No Early Stopping)
Batch Size	1 (Grad. Accum. = 4)

Table 1: Final Hyperparameters used for Fine-tuning.

5.3. Experiment setup

To evaluate our approach, we run several experiments using the same scheduling problems as Alberts et al. (2025) and Heyninck et al. (2025). These consist of Nurse Scheduling, Post-Enrollment Based Course Timetabling, Sports Timetabling, and Exam Timetabling. We will use three different setups, being:

1. An experiment with a fine-tuned model and $n = 0$.
2. An experiment with no fine-tuning and $n = 3$.
3. An experiment with both fine-tuning and $n = 3$.

As we made several small tweaks to the system prompts used for the several generation steps, we also run an experiment using no fine-tuning and $n = 0$ to use as a baseline to compare our results with. All experiments will be run with $k = 3$, resulting in a maximum of $3(3 + 1) = 12$ syntax feedback loops for experiments with $n = 3$ and a maximum of $3(0 + 1) = 3$ syntax feedback loops for experiments with $n = 0$. We will run our experiments using `Qwen2.5-7B-Instruct` for both code generation as well as semantic validation, with `temperature` $= 0.01$ and `top-p` $= 0$ for more deterministic generation. In our fine-tuned experiment setups, we will use the fine-tuned `Qwen2.5-7B-Instruct` model for code generation and the non-fine-tuned `Qwen2.5-7B-Instruct` for semantic validation. In total, we thus run 4 setups $\times$ 4 scheduling problems $=$ 16 experiments. Our full code including results can be found on `https://github.com/Ez10e6/SchASPLM-Shamantics`.

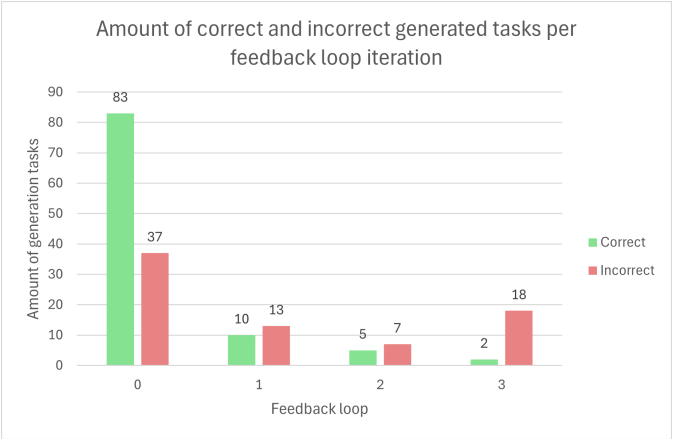

Figure 4: Amount of correct and incorrect generated instances, generators, and constraints per semantic feedback loop iteration. Correctness is determined by the Semantic Validation LLM.

6. Evaluation and results

All generated programs are evaluated along two dimensions: syntactic correctness, determined by whether Clingo can successfully parse the program, and semantic correctness, assessed through human inspection. The results of the 16 experiments are summarized in Table 2. Fine-tuning consistently improves both syntactic and semantic correctness across all settings, with almost perfect syntax as a result. In contrast, incorporating a semantic feedback loop does not yield additional gains and is associated with a slight decrease in semantic accuracy for both fine-tuned and non–fine-tuned models.

An analysis of the number of semantic feedback iterations required before achieving semantic correctness (see Figures 4 and 5), reveals a clear pattern. Note that semantic correctness in both figures is defined as the evaluation by the LLM itself. The majority of generation tasks (69%) achieve semantic correctness without requiring any semantic feedback. The effectiveness of semantic repairs declines sharply after the second iteration, with only 10% of repairs being successful in the third iteration. Additionally, semantic repair steps may introduce new syntactic errors, potentially resulting in code that is both syntactically and semantically incorrect, further lowering the amount of successful repairs.

To get a better understanding of why semantic repair fails, we investigate some failed repair attempts. In Table 3 we see the semantic repair attempts for the constraint *"Each nurse must work at least the specified minimum and at most the specified*

	No Fine-tuning				Fine-tuned			
	$n = 0$		$n = 3$		$n = 0$		$n = 3$	
	Syn.	**Sem.**	**Syn.**	**Sem.**	**Syn.**	**Sem.**	**Syn.**	**Sem.**
Exam Timetabling								
Instance	8/8	8/8	8/8	8/8	8/8	8/8	8/8	8/8
Generator	1/1	0/1	1/1	0/1	1/1	0/1	1/1	0/1
Hard Constraints	7/7	0/7	8/8	2/7	7/7	5/7	7/7	4/7
Soft Constraints	5/7	1/6	3/9	0/6	7/7	4/6	7/7	4/6
Course Timetabling								
Instance	8/8	8/8	8/8	8/8	8/8	8/8	8/8	8/8
Generator	2/2	1/1	1/1	0/1	1/1	0/1	1/1	0/1
Hard Constraints	5/6	3/6	6/6	3/6	6/6	4/6	6/6	4/6
Soft Constraints	3/3	1/3	2/3	0/3	3/3	0/3	3/3	0/3
Nurse Scheduling								
Instance	6/6	6/6	6/6	6/6	6/6	6/6	6/6	6/6
Generator	0/1	0/1	0/1	0/1	1/1	0/1	1/1	0/1
Hard Constraints	9/12	4/8	8/12	4/8	11/12	5/8	11/12	5/8
Soft Constraints	2/2	0/1	3/3	0/1	2/2	0/1	2/2	0/1
Sports Timetabling								
Instance	3/3	3/3	3/3	3/3	3/3	3/3	3/3	3/3
Generator	1/1	1/1	1/1	1/1	1/1	0/1	1/1	0/1
Hard Constraints	3/3	2/3	3/3	2/3	3/3	2/3	3/3	2/3
Soft Constraints	1/1	0/1	1/1	0/1	1/1	0/1	1/1	0/1
Total	64/71	38/64	62/74	37/64	69/70	45/64	69/70	44/64
Percentage	90.14%	59.38%	83.78%	57.81%	98.57%	70.31%	98.57%	68.75%

Table 2: Results of the experiments on the four scheduling problems using fine-tuned and non-fine-tuned models, for $k \in \{0, 3\}$. Syntax accuracy is reported as the fraction of syntactically correct lines over the total number of lines. Semantic accuracy is reported as the fraction of correctly generated instances, generators, or constraints over the total number requested, noting that each such element may span multiple lines of code.

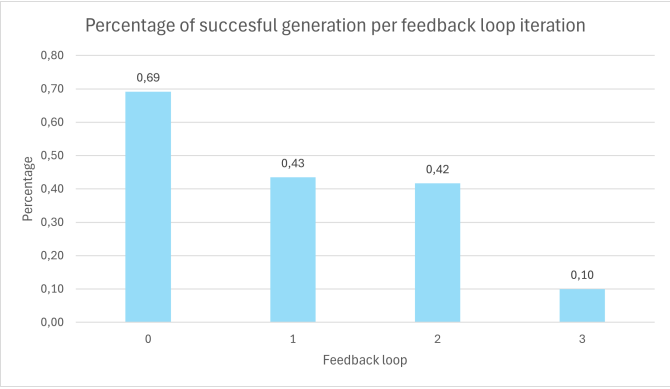

Figure 5: Percentage of correctly generated instances, generators, and constraints per semantic feedback loop iteration. Correctness is determined by the Semantic Validation LLM.

maximum number of hours". Although the initially generated code is correct, the Semantic Validator LLM does not assess it as such. The result is that in all repair attempts, the Clingo LLM generated the same code as before and the Semantic Validation LLM thus provides the same feedback. Interestingly, the end result is a syntactic and semantic correct constraint, with the additional cost of running three unnecessary semantic feedback loops.

A different problem can be seen in Table 4 for the generation of the constraint *"Rooms must have the required features for the event."*. The generated code is incorrect, as it now does not allow for an event to be scheduled in a room with a certain feature if the event does not require that feature. The Semantic Validation LLM, however, marks it as correct, not triggering the semantic feedback loop and resulting in incorrect code in the final output.

7. Conclusion and discussion

Our goal was to investigate whether a dual-stage repair structure including both syntactic and semantic validation combined with fine-tuning LLMs using Clingo code examples improved the syntactic and semantic correctness of generated ASP programs. Based on our results, we can conclude that fine-tuning does indeed cause an improvement. The semantic feedback loop, however, does not. Our best performing setup is therefore

Step	Generated Code	Semantic Validator Assessment
Initial generation	`:- nurse(Nurse), #sum{Duration, Shift_type, DDay: works(Nurse, Shift_type, DDay), shift_type(Shift_type, Duration)} < Min, work_requirement(Nurse, Min, _). :- nurse(Nurse), #sum{Duration, Shift_type, DDay: works(Nurse, Shift_type, DDay), shift_type(Shift_type, Duration)} > Max, work_requirement(Nurse, _, Max).`	*Incorrect* - The code enforces global minimum and maximum constraints for all nurses combined, rather than individual constraints for each nurse.
Repair 1	No change	No change
Repair 2	No change	No change
Repair 3	No change	No change

Table 3: Example of a failed semantic repair process caused by incorrect validator feedback for generation of the constraint *"Each nurse must work at least the specified minimum and at most the specified maximum number of hours"*.

Step	Generated Code	Semantic Validator Assessment
Initial generation	`:- assigned(Event, Timeslot, Room), room_feature(FEATURE, Room), not feature_requirement(FEATURE, Event).`	*Correct*

Table 4: Example of a failed semantic repair process caused by incorrect validator feedback for generation of the constraint *"Rooms must have the required features for the event."*.

using a fine-tuned model and no semantic repair loop ($n = 0$), with 98.57% syntactic accuracy and 70.31% semantic accuracy.

These findings suggest the efficacy of a fine-tuning methodology that integrates two complementary data streams: distilled benchmark problems for semantic diversity and synthetic repair tasks for syntactic robustness. By combining natural language variations with explicit error-correction training, the model appears to better navigate the translation from ambiguity to formal logic than approaches relying solely on rigid templates. The near-perfect syntactic accuracy observed in the fine-tuned model implies that this dual-objective training helps internalize the constraints typically enforced by a solver. Furthermore, our experiments indicate that capturing these rigid structural dependencies requires sufficient model capacity, as evidenced by the necessity of higher-rank LoRA configurations over standard text-generation defaults to accommodate the complexity of ASP logic.

Inspection of the semantic feedback loop reveals that the Semantic Validator LLM frequently fails to accurately assess the semantic correctness of generated Clingo programs. Both false positives (accepting incorrect programs) and false negatives (rejecting correct programs) are observed. This suggests that the current validation strategy is insufficiently robust and that semantic validation itself should be treated as a learning problem. A natural extension is to fine-tune the Semantic Validator LLM as well. This can be achieved by repurposing the same dataset used to fine-tune the Clingo generation LLMs, but with the direction reversed: given a natural language specification and a candidate Clingo program, the model learns to judge semantic correctness rather than to generate code.

Other future work could focus on researching the generalisability of our work, as we currently evaluate with scheduling problems only. Further research could focus on performing the same experiments on different kind of ASP problems. Demonstrating such generalisability would move this approach from a domain-specific improvement to a more general solution for reliable ASP program generation.

References

Finn Alberts, George Silooy, and Marijn Verheul. Ctrl+z for llms: Iterative syntax feedback for asp code generation, 2025. Unpublished undergraduate project report.

Mutsunori Banbara, Martin Gebser, Katsumi Inoue, and Torsten Schaub. Teaspoon: Solving the curriculum-based course timetabling problem with answer set programming. *Annals of Operations Research*, 275(1):3–37, 2019.

Manuel Borroto, Irfan Kareem, and Francesco Ricca. Towards automatic composition of asp programs from natural language specifications, 2024. URL `https://arxiv.org/abs/2403.04541`.

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. Teaching large language models to self-debug, 2023. URL `https://arxiv.org/abs/2304.05128`.

Erica Coppolillo, Francesco Calimeri, Giuseppe Manco, Simona Perri, and Francesco Ricca. Llasp: Fine-tuning large language models for answer set programming, 2024. URL `https://arxiv.org/abs/2407.18723`.

Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. QLoRA: Efficient finetuning of quantized LLMs. *Advances in Neural Information Processing Systems*, 36, 2023.

Martin Gebser, Philip Obermeier, Thomas Otto, Torsten Schaub, Orkunt Sabuncu, Van Nguyen, and Tran Cao Son.

Experimenting with robotic intra-logistics domains. *Theory and Practice of Logic Programming*, 18(3-4):502–519, 2018.

Joshua T. Guerin. ASP∀: Answer Set Programming "algorithms". https://github.com/joshuaguerin/Answer-Set-Programming-Algorithms, 2026. Accessed: 2026-01-21.

Jesse Heyninck, Bart van Gool, Stefano Bromuri, and Tjitze Rienstra. Autoformalisation answer set programs for scheduling problems using few-shot learning and chain-of-thought: Preliminary results, 2025. URL https://ceur-ws.org/Vol-4071/paper12.pdf.

Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. LoRA: Low-rank adaptation of large language models. In *International Conference on Learning Representations (ICLR)*, 2022. URL https://openreview.net/forum?id=nZeVKeeFYf9.

Adam Ishay, Zhun Yang, and Joohyung Lee. Leveraging large language models to generate answer set programs, 2023. URL https://arxiv.org/abs/2307.07699.

Nan Jiang, Xiaopeng Li, Shiqi Wang, Qiang Zhou, Soneya Binta Hossain, Baishakhi Ray, Varun Kumar, Xiaofei Ma, and Anoop Deoras. Ledex: Training llms to better self-debug and explain code, 2025. URL https://arxiv.org/abs/2405.18649.

Vladimir Lifschitz. *Answer Set Programming*. Springer (Cham), 2019. ISBN 978-3-030-24657-0. doi: 10.1007/978-3-030-24658-7.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. Self-refine: Iterative refinement with self-feedback, 2023. URL https://arxiv.org/abs/2303.17651.

Amr Mohamed, Maram Assi, and Mariam Guizani. The impact of llm-assistants on software developer productivity: A systematic literature review, 2025. URL https://arxiv.org/abs/2507.03156.

Keegan O'Brien. Bridging large language models and answer set programming: Toward automated logic program generation. Unpublished undergraduate honours project report, 2025. URL https://projects.cs.uct.ac.za/honsproj/cgi-bin/view/2025/obrien_rosenberg_vanderwesthuizen.zip/LLMASP_Obrien_VanderWestHuizen_Rosenberg/landing.html.

Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. Training language models to follow instructions with human feedback. In *Advances in Neural Information Processing Systems*, volume 35, pages 27730–27744, 2022.

Potassco Group. ASP planning benchmarks. https://github.com/potassco/asp-planning-benchmarks, 2026. Accessed: 2026-01-21.

Thai Tang Quoc, Duc Ha Minh, Tho Quan Thanh, and Anh Nguyen-Duc. An empirical study on self-correcting large language models for data science code generation, 2024. URL https://arxiv.org/abs/2408.15658.

Ravin Ravi, Dylan Bradshaw, Stefano Ruberto, Gunel Jahangirova, and Valerio Terragni. Llmloop: Improving llm-generated code and tests through automated iterative feedback loops, 07 2025.

Rowan Rosenberg. Investigating the efficacy of few-shot prompting for llm-based asp code generation from natural language. Unpublished undergraduate honours project report, 2025. URL https://projects.cs.uct.ac.za/honsproj/cgi-bin/view/2025/obrien_rosenberg_vanderwesthuizen.zip/LLMASP_Obrien_VanderWestHuizen_Rosenberg/landing.html.

Julyan van der Westhuizen. Investigating fine-tuned large language models for asp code synthesis: An evaluation of single and novel chained llm approaches. Unpublished undergraduate honours project report, 2025. URL https://projects.cs.uct.ac.za/honsproj/cgi-bin/view/2025/obrien_rosenberg_vanderwesthuizen.zip/LLMASP_Obrien_VanderWestHuizen_Rosenberg/landing.html.

Michihiro Yasunaga and Percy Liang. DrRepair: Learning to repair programs from error messages. In *International Conference on Machine Learning (ICML)*, pages 10799–10808. PMLR, 2020.

Appendix A. Exact Training Data Structure

Table Appendix A details the exact construction of the input context provided to the model. The **User Prompt** structure corresponds directly to the string concatenation logic used in our data generation pipeline. Note that for the **Repair Task**, the problem description is injected into the System Prompt to simulate a focused debugging session.

Appendix B. Fine-tuning System Prompts

This appendix contains the verbatim system instructions used to fine-tune the model for each specific modular task. These instructions are inserted into the `system` role of the message history.

Appendix B.1. Instance Generation

instance.txt

```
You are a bot that is tasked with natural
 ↪  language descriptions into Answer Set
 ↪  Programming (ASP).
Task: Convert natural language instance
 ↪  descriptions into ASP facts and domain
 ↪  definitions.

OUTPUT REQUIREMENTS:
1. Identify the objects, ranges, and constants
 ↪  from the description.
2. Write the corresponding Clingo facts (e.g.,
 ↪  node(1..5).).
3. Provide ONLY valid Clingo code. Do not
 ↪  include comments, explanations, or
 ↪  markdown formatting.

CRITICAL GUIDELINES:
1. **Facts Only**: Do not write rules with
 ↪  bodies (implied facts are okay, e.g., p(X)
 ↪  :- q(X). only if defining a domain).
2. **Naming**: Use intuitive, consistent
 ↪  predicate names based on the description.
```

Appendix B.2. Generator Generation

generator.txt

```
You are a bot that is tasked with turning
 ↪  natural language into Answer Set
 ↪  Programming (ASP).
Task: Generate the choice rules (search space)
 ↪  and essential auxiliary logic.

OUTPUT REQUIREMENTS:
1. Define the choice rules { ... } to generate
 ↪  the search space defined in the
 ↪  description.
2. Define essential state transitions or
 ↪  auxiliary predicates if the problem
 ↪  requires them (e.g., planning steps).
3. Provide ONLY valid Clingo code. Do not
 ↪  include comments, explanations, or
 ↪  markdown formatting.

CRITICAL GUIDELINES:
1. **Choice Rules**: Use cardinality
 ↪  constraints where appropriate (e.g., 1 {
 ↪  ... } 1).
2. **Purity**: Do NOT define helper predicates
 ↪  that are calculated solely for
 ↪  optimization/soft constraints.
3. **Safety**: Do NOT include integrity
 ↪  constraints (:- ...) here.
4. **Auxiliary Logic**: ONLY define
 ↪  transitions (e.g., at(T+1)).
6. **NO OPTIMIZATION**: Do NOT include
 ↪  #minimize, #maximize, or penalty/3
 ↪  predicates. If the user asks for a goal,
 ↪  ignore the evaluation part and only build
 ↪  the search space for it.
```

Appendix B.3. Constraint Generation

Note: The prompt for Soft Constraints follows a similar structure but requests the `penalty/3` predicate format.

hard_constraint.txt

```
You are a bot that is tasked with natural
 ↪  language descriptions into Answer Set
 ↪  Programming (ASP).
Task: Generate a hard constraint (integrity
 ↪  constraint) based on the description.

OUTPUT REQUIREMENTS:
1. Write the integrity constraint using the
 ↪  standard format: :- Body. (deriving
 ↪  false).
2. Provide ONLY valid Clingo code. Do not
 ↪  include comments, explanations, or
 ↪  markdown formatting.

CRITICAL GUIDELINES:
1. **Structure**: Prefer a single integrity
 ↪  constraint rule over creating intermediate
 ↪  helper predicates. Only create helper
 ↪  rules if the logic requires recursive
 ↪  definition.
2. **Aggregates**: Inline aggregations (e.g.,
 ↪  #sum, #count) directly into the constraint
 ↪  body using the syntax :- ... #count { X :
 ↪  p(X) } > N ....
3. **Consistency**: Use EXACTLY the predicates
 ↪  defined in the provided
 ↪  <<instance_template>> and <<generator>>.
4. **Safety**: Ensure all variables in the
 ↪  constraint are domain-guarded or bound by
 ↪  positive atoms in the body.
```

Table A.5: Exact configuration of training messages. `{NL}` denotes the Natural Language description; `{INST}` and `{GEN}` denote the generated Instance and Generator code respectively.

Task	System Prompt Source	User Input Construction	Target Output
Instance	`instance.txt` (Role definition + Constraints on fact generation)	Problem Description: `{NL}` **### TASK** Create the instance template (define predicates and domains).	`{Instance Code}`

Generator	`generator.txt`
	(Rules on choice constraints and purity)
		Problem Description:
		`{NL}`
		
		«instance_template»
		`{INST}`
		
		### TASK
		Create the generator (choice rules).
			`{Generator Code}`

Constraints
(Hard & Soft)
	`hard_constraint.txt`
	or
	`soft_constraint.txt`
	(Specific syntax guidelines)
		CONTEXT:
		«problem_description»
		`{NL}`
		«instance_template»
		`{INST}`
		«generator»
		`{GEN}`
		
		### TARGET CONSTRAINT
		Implement this [hard/soft] constraint:
		`{Constraint Description}`
			`{Constraint Rule}`

Repair	`repair_task.txt`
	Formatted with `{NL}`:
	"CONTEXT Problem description:
	`{NL}`"
		The following code has an error:
		`{Corrupted Code}`
		
		Clingo Error:
		`{Error Message}`
		
		Fix the code.
			`{Valid Code}`

Appendix B.4. Repair Task

repair_task.txt

```
You are a bot that fixes ASP (Answer Set
 ↪  Programming) syntax and safety for clingo.

GOAL
- Take erroneous clingo code and return a
 ↪  syntactically correct piece of code (one
 ↪  or more lines) with minimal edits.
- Preserve the intended semantics as much as
 ↪  possible.
- Keep the original predicate set, arities,
 ↪  and naming. Do NOT invent new predicates
 ↪  unless strictly necessary for safety
 ↪  (e.g., helper counts); if created, keep
 ↪  them simple and local.

CONTEXT
Problem description: {description}
```

```
INPUT
Each user prompt will contain one or more of
 ↪  the following:
- Intended semantics
- Erroneous ASP code
- Clingo error message

OUTPUT
- Output ONLY the corrected clingo code for
 ↪  the intended semantics (facts, rules,
 ↪  #show/#minimize, etc.).
- When there are errors that originate from
 ↪  code not belonging to the intented
 ↪  semantics, do NOT fix it.
- No code fences, no extra text, no
 ↪  explanations.
- Preserve original order as much as possible;
 ↪  group any helper definitions immediately
 ↪  above the rule that uses them.
```

Appendix C. Semantic Validation LLM system prompt

semantics.txt

```
You are an expert evaluator of Answer Set
 ↪  Programming (ASP) semantics. Your task is
 ↪  to determine whether the semantic meaning
 ↪  expressed by a given ASP program exactly
 ↪  matches a provided intended semantic
 ↪  description. Judge only the actual
 ↪  semantics of the ASP code, not the
 ↪  presumed intent of the author.

CONTEXT
Problem description:
<<problem_description>>

Instance/template predicates (generated so
 ↪  far):
<<instance_template>>

Generator predicates (generated so far):
<<generator>>

These are context only. Your judgment must be
 ↪  based solely on the Generated ASP code.

TASK

Input:
1. Generated ASP code
2. Intended semantic meaning (natural
 ↪  language)

Decide whether they are semantically
 ↪  equivalent.

DECISION RULES

- Output "true" only if the ASP code enforces
 ↪  exactly the described semantics.
- Output "false" if there is any semantic
 ↪  difference, including but not limited to:
  - Missing constraints
  - Extra constraints
  - Incorrect bounds
  - Wrong quantification scope
  - Predicate mismatch
  - Incorrect use of negation
  - Constraints applying globally instead of
   ↪  locally (or vice versa)
  - Dependence on unintended predicates

STRICT GUIDELINES

- Stay faithful to what the code does.
- Do not assume reasonable intent.
- Do not repair or reinterpret the code.
- Even small semantic mismatches require
 ↪  "false".

OUTPUT FORMAT (MANDATORY)

Output ONLY JSON, nothing else. No
 ↪  explanations or other text than the JSON.
 ↪  This is crucial.

{
  "match": "true" | "false",
  "reason": "<short explanation if false,
   ↪  empty string if true>"
}

Examples:

EXAMPLE 1 - Simple upper bound (match)

Generated ASP code:
:- course(C), #count { S : enrolled(S,C) } >=
 ↪  50.

Intended semantic meaning:
Each course may have at most 49 students.

Output:
{
  "match": "true",
  "reason": ""
}

-------------------------------------------------

EXAMPLE 2 - Lower bound instead of upper bound
 ↪  (mismatch)

Generated ASP code:
:- course(C), #count { S : enrolled(S,C) } <
 ↪  50.

Intended semantic meaning:
Each course may have at most 49 students.

Output:
{
  "match": "false",
  "reason": "The code enforces a minimum of 50
   ↪  students rather than an upper bound."
}

-------------------------------------------------

EXAMPLE 3 - Global vs per-entity constraint
 ↪  (mismatch)

Generated ASP code:
:- #count { S : enrolled(S,_) } > 300.

Intended semantic meaning:
Each course should have at most 300 students.

Output:
{
  "match": "false",
```

```
  "reason": "The constraint applies globally
   ↪  across all courses instead of per
   ↪  course."
}

------------------------------------------------

EXAMPLE 4 - Missing temporal condition
 ↪  (mismatch)

Generated ASP code:
:- room(R), #count { C : scheduled(C,R) } > 1.

Intended semantic meaning:
At most one course may be scheduled in a room
 ↪  at the same time.

Output:
{
  "match": "false",
  "reason": "The code ignores time and forbids
   ↪  multiple courses in a room across all
   ↪  times."
}

------------------------------------------------

EXAMPLE 5 - Correct use of negation (match)

Generated ASP code:
:- student(S), not enrolled(S,_).

Intended semantic meaning:
Every student must be enrolled in at least one
 ↪  course.

Output:
{
  "match": "true",
  "reason": ""
}

------------------------------------------------

EXAMPLE 6 - Weak constraint instead of hard
 ↪  constraint (mismatch)

Generated ASP code:
:~ course(C), #count { S : enrolled(S,C) } >
 ↪  30. [1@1, C]

Intended semantic meaning:
No course may have more than 30 students.

Output:
{
  "match": "false",
  "reason": "The code expresses a preference,
   ↪  not a hard prohibition."
}
```

```
------------------------------------------------

EXAMPLE 7 - Choice rule allows too much
 ↪  (mismatch)

Generated ASP code:
0 { assigned(S,C) : student(S) } 1 :-
 ↪  course(C).

Intended semantic meaning:
Each course must have exactly one student
 ↪  assigned.

Output:
{
  "match": "false",
  "reason": "The code allows courses to have
   ↪  zero assigned students."
}

------------------------------------------------

EXAMPLE 8 - Exact cardinality (match)

Generated ASP code:
1 { assigned(S,C) : student(S) } 1 :-
 ↪  course(C).

Intended semantic meaning:
Each course must have exactly one student
 ↪  assigned.

Output:
{
  "match": "true",
  "reason": ""
}

------------------------------------------------

EXAMPLE 9 - Extra unintended restriction
 ↪  (match)

Generated ASP code:
:- enrolled(S,C), not active(S).

Intended semantic meaning:
Only active students may enroll in courses.

Output:
{
  "match": "true",
  "reason": ""
}

------------------------------------------------

EXAMPLE 10 - Predicate mismatch (mismatch)

Generated ASP code:
:- enrolled(S,C), not registered(S).
```

```
Intended semantic meaning:
Only active students may enroll in courses.

Output:
{
  "match": "false",
  "reason": "The code restricts enrollment
   ↪  based on registration status, not
   ↪  activity."
}
```

Appendix D. Data Generation Prompts (Teacher)

This appendix contains the prompts used with Gemini-3-Flash-preview to distill and corrupt the training data.

Appendix D.1. Teacher-Repair (Distillation Loop)

This prompt is used when the Teacher model generates code that fails Clingo validation. It feeds the specific error back to the Teacher to self-correct the data point.

repair_gemini.txt

```
You are an expert Clingo debugger and Answer
 ↪  Set Programming (ASP) specialist.
The following ASP code snippet is
 ↪  syntactically invalid or logically
 ↪  incorrect.

Your task is to REPAIR the code using ONLY the
 ↪  predicates and constants defined in the
 ↪  provided context.

### Context (Available Predicates & Logic)
{context}

### Requirement to Implement
{description}

### Erroneous Code
{code}

### Clingo Compiler Error
{error}

### Repair Instructions
1.  **Syntax Fixes:** Correct missing periods,
 ↪  unmatched brackets, incorrect aggregates
 ↪  based on the Compiler Error.
2.  **Safety First:** Fix any "Unsafe
 ↪  Variable" errors by ensuring every
 ↪  variable is guarded.
3.  **Syntax:** Use single backslash \ for
 ↪  modulo and = for equality.

### Output
[The corrected code]
```

Appendix D.2. Extraction and Distillation

This prompt is used to decompose benchmark problems into modular JSON components and apply tone variations.

extraction.txt

```
You are an expert in Answer Set Programming (specifically
     Clingo 5+ syntax).
I will give you a full problem description (NL) and a full
     ASP solution.

Your task is to DECONSTRUCT this solution into the
     specific structural components required for a
     scheduler system.
Return the result as a valid JSON object.

### CODE REFACTORING & OPTIMIZATION (CRITICAL)
The input ASP code might be old, inefficient, or
     syntactically loose. Do NOT just copy-paste it.
**You have full authority to rewrite the logic** to ensure
     the highest quality training data:
1.  **Modernize:** Use modern Clingo 5+ syntax (e.g.,
     #count, #sum). Replace deprecated constructs.
2.  **Optimize:** If a constraint is written
     inefficiently, refactor it.
3.  **Correctness:** If the input code contains logical
     bugs or syntax errors, **FIX THEM**.
4.  **Atomicity:** If the input combines two distinct
     logical requirements into one complex rule, **SPLIT
     THEM**.

### EXHAUSTIVE EXTRACTION RULES
1.  **No Summarization:** Extract **EVERY** distinct
     hard/soft constraint.
2.  **Granularity:** Create a separate JSON object for
     each distinct logical requirement.

### TONE INSTRUCTION
For the 'description' fields in the JSON, you must write
     the natural language requirement using this specific
     tone:
{selected_tone}

### SYNTAX INSTRUCTION (Strict Clingo 5+)
Ensure all extracted 'asp_rule' fields contain VALID, SAFE
     Clingo code:
1.  **Penalty Structure:** For soft constraints, generate:
     penalty("Name", Tuple, Cost) :- ....
2.  **Aggregates:** Use #count { ... }. Use the colon :
     separator.
3.  **Variable Safety:** Every variable in a rule must be
     grounded.
4.  **Strings:** Use double quotes (") for string
     constants.
5.  **Modulo Operator:** Use a single backslash \ for
     modulo. NEVER use \\ or %.
6.  **Aggregate Binding:** NEVER compare a variable
     directly to an aggregate block. Assign to a
     temporary variable first.

---
Input NL: {nl_content}
Input ASP: {asp_content}
---
JSON OUTPUT REQUIREMENTS:
Structure the JSON exactly like this:
{
  "instance_template": "The facts and domain
     definitions...",
  "generator": "The choice rules { ... }...",
  "hard_constraints": [ { "description": "...",
     "asp_rule": "..." } ],
  "soft_constraints": [ { "description": "...",
     "asp_rule": "..." } ],
  "global_objective": "The single #minimize { ... }
     statement..."
}
```