Leveraging Large Language Models to Generate Answer Set Programs

Adam Ishay\textsuperscript{1}, Zhun Yang\textsuperscript{1}, Joohyung Lee\textsuperscript{1,2}
\textsuperscript{1}Arizona State University
\textsuperscript{2}Samsung Research
\{aishay, zyang90, joolee\}@asu.edu

Abstract

Large language models (LLMs), such as GPT-3 and GPT-4, have demonstrated exceptional performance in various natural language processing tasks and have shown the ability to solve certain reasoning problems. However, their reasoning capabilities are limited and relatively shallow, despite the application of various prompting techniques. In contrast, formal logic is adept at handling complex reasoning, but translating natural language descriptions into formal logic is a challenging task that non-experts struggle with. This paper proposes a neuro-symbolic method that combines the strengths of large language models and answer set programming. Specifically, we employ an LLM to transform natural language descriptions of logic puzzles into answer set programs. We carefully design prompts for an LLM to convert natural language descriptions into answer set programs in a step-by-step manner. Surprisingly, with just a few in-context learning examples, LLMs can generate reasonably complex answer set programs. The majority of errors made are relatively simple and can be easily corrected by humans, thus enabling LLMs to effectively assist in the creation of answer set programs.

1 Introduction

Transformer-based large language models (LLMs) have recently shown remarkable success in many downstream tasks, demonstrating their general reasoning capability across diverse problems. However, while LLMs excel in generating System 1 thinking, they struggle with System 2 thinking, resulting in output that is often inconsistent and incoherent (Nye et al. 2021). This is because LLMs are basically trained to predict subsequent words in a sequence and do not appear to have a deep understanding of concepts such as cause and effect, logic, and probability, which are essential for reasoning.

To address the issue, Nye et al. (2021) propose a dual-system model that combines the strengths of LLMs and symbolic logic to achieve improved performance on reasoning tasks. They leverage an LLM to generate a System 1 proposal and employ symbolic computation to filter these proposals for consistency and soundness.

We are interested in situations where problems are described in natural language and solving them requires deep reasoning. A system needs to take into account linguistic variability and be able to perform symbolic reasoning. We take logic puzzles as the testbed as they are well-suited for this purpose.

We first note that GPT-3 (Brown et al. 2020) and GPT-4\textsuperscript{1} by themselves struggle with solving logic puzzles, despite various prompts we tried. On the other hand, we find that they can convert the natural language descriptions of the puzzles into declarative answer set programming languages (Lifschitz 2008; Brewka, Niemelä, and Truszczynski 2011) surprisingly well. Even the errors these LLMs make are mostly simple for humans to correct. We hope that our finding will ease the efforts of writing answer set programs and expand the application of answer set programming to a broader audience.

The remainder of this paper is organized as follows. Section 2 offers a brief overview of related work on automated solving of logic puzzles. Sections 3 and 4 delve into the proposed approach in detail. Section 5 presents experimental results and performance evaluations of the approach. Section 6 shows more examples demonstrating the generalizability of our method.

The code is available at https://github.com/azreasoners/gpt-asp-rules.

2 Preliminaries

2.1 Large Language Models (LLMs)

LLMs have significantly improved natural language processing, achieving strong performance on a variety of tasks using few-shot learning (Brown et al. 2020). However, LLMs remain weak at tasks that involve complex reasoning (Creswell, Shanahan, and Higgins 2022; Valmeekam et al. 2022), and scaling model size alone is not enough to achieve good performance (Rae et al. 2021). It has been shown that various prompting methods improve accuracy on reasoning tasks (Wei et al. 2022; Zhou et al. 2022; Creswell, Shanahan, and Higgins 2022). Nye et al. (2021) present a dual-system model which uses an LLM as a semantic parser and couples it with a custom symbolic module to achieve performance gains on reasoning tasks. This framework combines the strengths of LLMs for parsing complex natural language and symbolic logic for handling

\textsuperscript{1}Throughout the paper, we use GPT-3 to refer to the “text-davinci-003” model and GPT-4 to refer to the “gpt-4-0314” (released March, 2023) model in the OpenAI API.
complex reasoning. However, the authors had to use hand-engineered set of constraints for the latter part. To our knowledge, our work is the first to use LLMs to generate logic rules to solve complex reasoning tasks.

2.2 Automated Logic Puzzle Solving

Works focused on solving logic puzzles typically involve a mapping from natural language to logic formalism. This process often includes problem simplification techniques, such as tailoring the puzzle to a specific domain, restricting natural language input to a certain form, or assuming additional inputs like enumerated types. Lev et al. (2004) employ a specialized automated multi-stage parsing process to convert natural language text into an intermediate form called Semantic Logic, which is then converted into First Order Logic to finally evaluate on law school admissions tests (LSAT) and the Graduate Records Examination (GRE). Shapiro (2011) manually encodes the “Jobs Puzzle” in a few different logical formalisms and compare them. Puzzler (Milicevic, Near, and Singh 2012) uses a general link parser to translate puzzles into the Alloy language for solving, primarily through an automated process, albeit with assumed types. LogicSolver (Nordstrom 2017) follows a similar approach to Puzzler but replaces Alloy with a custom solver and conducts a more comprehensive evaluation.

Several works utilize translations into the language of answer set programming (ASP) (Lifschitz 2008; Brewka, Niemelä, and Truszczynski 2011). Schwitter (2013) addresses the “Jobs Puzzle” by representing the problem using controlled natural language (Schwitter 2010), which can be further turned into ASP. Baral and Dizifcak (2012) employ a λ-calculus-based approach and trains a model that converts a manually simplified version of natural language clues into ASP rules for solving Zebra puzzle-type logic puzzles. Mitra and Baral (2015) train a maximum entropy-based model to extract relations for each clue, which are then converted into a common ASP rule format, where a stable model corresponds to the puzzle solution. LGPSolver (Jabrayilzade and Baral 2015) exploits a maximum entropy-based model to classify natural language input to a certain form, or assuming additional inputs like enumerated types. With the clue classification, the authors use a handcrafted clue to Prolog translation (as opposed to ASP) and compute the solution. The works mentioned involve some combination of manual processing and/or brittle problem-specific translations. Our work distinguishes itself by being both fully automated and featuring a general pipeline, leveraging the extensive translation capacity available from LLMs.

2.3 Generate-Define-Test with ASP

ASP programs are typically written following the Generate-Define-Test structure, which generates potential solutions (Generate) and eliminates invalid ones based on certain constraints (Test). The Generate portion usually includes choice rules, while the Test portion consists of a set of constraints that prune out invalid solutions. An additional part of the program, the Define portion, includes necessary auxiliary predicates that are used in the Test portion.

3 Method

![Figure 1: Flow of Generating Answer Set Programs from Logic Puzzle in English](image)

In order to find a solution to a logic puzzle, we utilize GPT-3 to convert the puzzle into an answer set program so that the stable model (a.k.a. answer set) encodes the solution.² Although GPT-3 exhibits strong capabilities, we discovered that it cannot generate a correct answer set program without being guided by carefully engineered prompts. These prompts instructs GPT-3 to reliably extract constants and generate accurate predicates and rules. In this paper, we detail our prompt engineering efforts.

Figure 1 illustrates the structure of our pipeline, which utilizes GPT-3 step by step to generate an ASP program. Similar to how a human would approach the task, our pipeline first extracts the relevant objects and their categories. Then, it generates a predicate that describes the relations among the objects from different categories. Using the generated information, the pipeline further constructs an ASP program in the style of Generate-Define-Test.

Let $F_c$ and $F_p$ denote the Constant Extraction and Predicate Generation steps in Figure 1. Let $F_{r1}$ and $F_{r2}$ represent the two parts of the Rule Generation step, i.e., the Generate part and the Define & Test part, respectively. Our pipeline can be modeled by the following equations that map a puzzle story $q$ to an ASP program $\Pi = \Pi_{\text{generate}} \cup \Pi_{\text{define,test}}$.

$$c = F_c(q) \quad \Pi_{\text{generate}} = F_{r1}(c, p) \quad \Pi_{\text{define,test}} = F_{r2}(q, c, p).$$

Here, $c$ and $p$ denote extracted objects and generated predicates. Each step $F_c$ is realized by GPT-3 with 2-shot prompting, i.e., only 2 examples in each prompt.

3.1 Constant Extraction

The first step in the pipeline is to extract constants or entities from the given story along with their corresponding categories. To accomplish this, we invoke GPT-3 using Prompt C, which consists of three parts: instruction, examples, and a query.

**Prompt C:**

²Though this section mostly mentions GPT-3, GPT-4 can be used instead.
Given a problem, extract all different constants and their categories in the form "category: constant_1; constant_2; ...; constant_n". Here, the format of each constant is turned into either an integer or a string surrounded by double quotes, e.g., "some name".

Problem 1:
Consider N-Queens Puzzle on a chessboard of size 8x8. The goal is to assign 8 queens on the chessboard so that no two queens can share the same row, column, or diagonal.

Constants:
- index_of_row: 1; 2; 3; 4; 5; 6; 7; 8.
- index_of_column: 1; 2; 3; 4; 5; 6; 7; 8.

Problem 2:
"Against the Grain" offers hand-made wooden furniture at reasonable prices. Each item is made by an in-house employee. Using only the clues that follow, match each item to the employee who crafted it, and determine its price and the type of wood used to make it. Remember, as with all grid-based logic puzzles, no option in any category will ever be used more than once.

1. Bonita’s piece costs $325.
2. The item made of poplar costs more than Yvette’s piece.
3. Tabitha’s item costs 50 dollars less than Yvette’s piece.
4. The $275 item is either the piece made of ash or Yvette’s piece.

Constants:
- employee: "Bonita"; "Yvette"; "Tabitha".
- price: 225; 275; 325.
- wood_type: "ash"; "poplar"; "sandalwood".

Problem 3:

Constants:

The above two examples are chosen to cover two cases of object extraction. For the N-Queens problem, the constants 1, ..., 8 are not described in the Problem 1 statement (Line 4) but can be inferred. For the second puzzle, however, all constants in Lines 18-20 are mentioned in the example story provided in Lines 11-15.

The second puzzle is also intentionally selected to give an example for GPT-3 so that certain constants (e.g., $225) can be turned into valid integers (e.g., 225) so that arithmetic can be applied correctly later when generating rules later on, while others should be surrounded by double quotes. We experimented with various prompts to instruct GPT-3 to generate all non-numeric constants in lowercase and replace special characters with underscores. However, GPT-3 was unable to strictly adhere to these instructions and consequently made more errors.

3.2 Predicate Generation
The next step in the pipeline is to generate predicates p that describe the relations among the extracted constants. We use GPT-3 on the Prompt P below.

Prompt P:

Line 1 is a general instruction describing the task of predicate generation, and that the generated predicates should follow the form of "predicate(X1, X2, ..., Xn)" where each Xi is a distinct variable that represents a category of constants.
Again, the two examples follow. Lines 3–4 are a copy of the first example in Lines 3–8 of Prompt C (where we omit Lines 4–8 from Prompt C to reduce the space). Lines 6–9 continue the first example, where it now generates the predicates with variables as arguments following the instruction. It also contains two comments (starting with symbol %). The first comment in Line 7 recalls the categories of constants and assigns a different variable to each category. The second comment in Line 8 gives the English reading of the predicate and variables, and emphasizes the link between each variable and a category of constants. Similarly, Lines 11–17 present the second example.

Next, the story and constants are given for the third problem and GPT-3 is prompted to generate the predicate for that example, given the general instruction and the preceding two examples.

Given the extracted constants c and generated predicates p, the next step in the pipeline is to generate ASP rules II, consisting of the Generate part and the Define&Test part.

### 3.3 Rule Generation: Generate

The Generate part of an ASP program defines all possible mappings of constants from different categories. This is done by choice rules. In this step, an ASP program Π\text{generate} is obtained by calling GPT-3 with Prompt R1.

**Prompt R1:**

<table>
<thead>
<tr>
<th>1</th>
<th>Given some categorized constants in the form &quot;category: constant_1; constant_2; ...; constant_n&quot; and some predicates about the relation among different categories of constants, write ASP (Answer Set Programming) rules to generate the search space of possible relations.</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Constants:</td>
</tr>
<tr>
<td>3</td>
<td>employee: &quot;Bonita&quot;; &quot;Yvette&quot;; &quot;Tabitha&quot;.</td>
</tr>
<tr>
<td>4</td>
<td>price: 225; 275; 325.</td>
</tr>
<tr>
<td>5</td>
<td>wood_type: &quot;ash&quot;; &quot;poplar&quot;; &quot;sandalwood&quot;.</td>
</tr>
<tr>
<td>6</td>
<td>Predicates:</td>
</tr>
<tr>
<td>7</td>
<td>% The categories include employee, price, and wood_type. We use different variables E, P, and W to represent employee, price, and wood_type.</td>
</tr>
<tr>
<td>8</td>
<td>% We match an employee E with price P and wood_type W, where E belongs to employee, P belongs to price, and W belongs to wood_type.</td>
</tr>
<tr>
<td>9</td>
<td>match(E, P, W)</td>
</tr>
<tr>
<td>10</td>
<td>ASP Rules:</td>
</tr>
<tr>
<td>11</td>
<td>% Define the constants in each category.</td>
</tr>
<tr>
<td>12</td>
<td>employee(&quot;Bonita&quot;; &quot;Yvette&quot;; &quot;Tabitha&quot;).</td>
</tr>
<tr>
<td>13</td>
<td>price(225; 275; 325).</td>
</tr>
<tr>
<td>14</td>
<td>wood_type(&quot;ash&quot;; &quot;poplar&quot;; &quot;sandalwood&quot;).</td>
</tr>
<tr>
<td>15</td>
<td>% For each employee E, it matches with exactly 1 price P and 1 wood type W.</td>
</tr>
<tr>
<td>16</td>
<td>(match(E, P, W): price(P), wood_type(W)=1 :- employee(E).</td>
</tr>
</tbody>
</table>

In the above prompt, (constants) and (predicates) are to be replaced for a new example. GPT-3 generates facts and choice rules following the last line of the prompt.

The task in this step is to write facts and choice rules based on the generated constants and predicates. Since this step doesn’t require the details of the story, we omit the story from the prompt to avoid unnecessary noisy information being included in the prompt. Each example only consists of constants, predicates, and ASP rules to be generated, i.e., facts and choice rules.

Similar to the previous prompts, Line 1 is a general instruction, Lines 3–20 provide an example, and Lines 22–28 are for the queried example. The example ASP rules in Lines 14–20 contain comments (Lines 14 and 19), which will also be generated for the queried example and help to gather semantic information before generating a rule.

### 3.4 Rule Generation: Define and Test

The Define&Test part of an ASP program contains constraints that “weed out” the stable models that do not correspond to valid answers. This step takes as input the puzzle story q, constants c, and predicates p. Semantically, the ASP rules represent the content in story q while, syntactically, the ASP rules must be formed by the extracted constants c and generated predicates p. The ASP program Π\text{define, test} is obtained by calling GPT-3 with Prompt R2.

**Prompt R2:**

<table>
<thead>
<tr>
<th>1</th>
<th>Consider the constraint in the following form</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>(&lt;C1&gt;; \langle C2&gt;; \ldots; \langle Cm \rangle ::= \langle L1 \rangle, \langle L2 \rangle, \ldots, \langle Ln \rangle. )</td>
</tr>
<tr>
<td>3</td>
<td>which says that if the conjunction (&lt;L1 \rangle \text{ and } &lt;L2 \rangle \text{ and } \ldots \text{ and } &lt;Ln \rangle) is true, then the disjunction of comparisons (&quot;&lt;C1 \rangle \text{ or } &lt;C2 \rangle \text{ or } \ldots \text{ or } &lt;Cm \rangle) must be true.</td>
</tr>
<tr>
<td>4</td>
<td>One can also add a restriction that &quot;exactly k of (&lt;C1 \rangle, \langle C2 \rangle, \ldots, \langle Cm \rangle) is true&quot; by using the following form</td>
</tr>
<tr>
<td>5</td>
<td>(&lt;C1 \rangle; \langle C2 \rangle; \ldots; \langle Cm \rangle \rightarrow k \rightarrow \langle L1 \rangle, \langle L2 \rangle, \ldots, \langle Ln \rangle. )</td>
</tr>
<tr>
<td>6</td>
<td>Given a problem, extract all constraints from the clues in the problem using only the provided constants and predicates.</td>
</tr>
</tbody>
</table>

**Problem 1:**

"Against the Grain" offers hand-made wooden furniture at reasonable prices. Each item is made by an in-house employee. Using only the clues that follow, match each item to the employee who crafted it, and determine its price and the type of wood used to make it. Remember, as with all grid-based logic puzzles, no option in any category will ever be used more than once.
1. Bonita’s piece costs $325.
2. The item made of poplar costs more than Yvette’s piece.
3. Tabitha’s item costs 50 dollars less than the piece made of sandalwood.
4. The $275 item is either the piece made of ash or Yvette’s item.

Constants:
employee: "Bonita"; "Yvette"; "Tabitha".
price: 225; 275; 325.
wood_type: "ash"; "poplar"; "sandalwood".

Predicates:
% The categories include employee, price, and wood_type. We use different variables E, P, and W to represent employee, price, and wood_type.
% We match an employee E with price P and wood type W, where E belongs to employee, P belongs to price, and W belongs to wood_type.
match(E, P, W)

Constraints:
% No option in any category will ever be used more than once.
(E1=E2; P1=P2; W1=W2)=0 :- match(E1, P1, W1), match(E2, P2, W2), (E1, P1, W1)!=(E2, P2, W2).
% 1. Bonita’s piece costs $325.
P=325 :- match(E, P, W), E="Bonita".
% 2. The item made of poplar costs more than Yvette’s piece.
P1>P2 :- match(E1, P1, W1), match(E2, P2, W2), W1="poplar", E2="Yvette".
% 3. Tabitha’s item costs 50 dollars less than the piece made of sandalwood.
P1=P2-50 :- match(E1, P1, W1), match(E2, P2, W2), E1="Tabitha", W2="sandalwood".
% 4. The $275 item is either the piece made of ash or Yvette’s item.
(W="ash", E="Yvette")=1 :- match(E, P, W), P=275.

(Problem 2 omitted)

Problem 3:

<story>

Constants:
<constants>

Predicates:
<predicates>

Constraints:

In the above prompt, <story> is a new puzzle, and <constants>, <predicates> are generated by GPT-3 for that story using Prompt C and Prompt P in Section 3.1 and 3.2.

Lines 1–8 are a general instruction describing the task of Problem text generation and provides two rule forms for the target ASP rules. The first rule form

\[ C_1; C_2; \ldots; C_n \leftarrow L_1, L_2, \ldots, L_n \]

says that “C₁ or ... or Cₙ is true if L₁ and ... and Lₙ are true.” Here, each Lᵢ is a literal and each Cᵢ is a comparison in the input language of CLINGO, e.g., \( A > B, A = B + 3 \), etc. The second rule form

\[ \{ C_1; C_2; \ldots; C_m \} \leftarrow L_1, L_2, \ldots, L_n \]

additionally restricts that “exactly k of \{C₁,...,Cₙ\} must be true.” In principle, the first rule form is enough to represent various constraints. However, since the second rule form is syntactically closer to certain complex sentences related to cardinality, e.g., “either ... or ...”, “neither ... nor ...”, or “no ... is ...”, etc, we found that GPT-3 works much better when we also include the second rule form.

4 Optional Enhancements to the Pipeline

Section 3 presented a general pipeline that automatically writes an ASP program for a puzzle in natural language using LLM. This section explains two optional enhancements that strengthen its robustness.

4.1 Constant Formatting

In the Constant Extraction step (Section 3.1), GPT-3 may extract the names of the objects as they appear in the puzzle story, such as $225, Sue Simpson, and 8:30 AM, which do not conform to the syntax of the input language of answer set solver CLINGO. Also, GPT-3 applies arithmetic computations (e.g., L₁=L₂+3) to constants surrounded by double quotes (e.g., L₂ is constant "9 inches") instead of constants that are integers (e.g., L₂ is constant 9).

A rule-based post-processing could be applied to turn them into the right syntax, but alternatively, we employ GPT-3 to generate syntactically correct forms. We found that this method requires significantly less efforts and is more general because GPT-3 applies the constant formatting correctly even for unforeseen formats using some “common sense,” which is lacking in the rule-based approach. We use the following prompt for this.

The Constant Formatting step is done by calling GPT-3 with the following prompt, where <constants> at the end of the prompt is replaced by the original (extracted) constants c obtained by the Constant Extraction step (Section 3.1). The GPT-3 response in this step is the updated constants c, serving as an input to other steps in the pipeline.
4.2 Sentence Paraphrasing

Sometimes sentences may need to be paraphrased before an LLM can correctly generate rules from them. The Sentence Paraphrasing step provides the opportunity to not only simplify or formalize the sentences from the original question but also add the hidden information assumed to underlie the question. For example, the following sentence

is one clue in the example question in Section 3. The correct translation requires an LLM to turn the above sentence into at least 3 ASP rules, which would be hard for the current LLMs (e.g., GPT-3). Instead, we can ask GPT-3 to first paraphrase such kind of sentence into simpler ones below.

The Sentence Paraphrasing step is done by calling GPT-3 with the following prompt, where \((\text{sentences})\) at the end of the prompt is replaced by the numbered sentences in the queried puzzle story \(q\), and the GPT-3 response in text is used to replace the original sentences in \(q\). This prompt is dedicated to the logic puzzles from Puzzle Baron and only paraphrases one kind of sentence in the form \(\text{“of A and B, one is C and the other is D”}\).
To solve the puzzles, we apply Constant Formatting directly on the given constants in the dataset as necessary information to solve each puzzle. The dataset consists of 50 training examples and 100 testing examples. Table 1 shows the performance of our approach to zero-shot GPT-3/GPT-4, few-shot GPT-3/GPT-4, and a fully-supervised learning system LOGICIA (Mitra and Baral 2015). In the few-shot setting, we use the first two examples in the training set as the few-shot examples. GPT-3 with zero-shot and few-shot settings didn’t perform well, while zero-shot GPT-4 could solve 21% of the test puzzles correctly, which is significantly better than GPT-3’s performance. However, this is much lower than our method’s 81%. Interestingly, while the few-shot setting slightly improves over the zero-shot setting, this is quite different with GPT-4. This is likely because GPT-4 with the zero-shot setting was instructed to solve the puzzles in a step by step manner. However, for the few-shot setting, the examples only include the problem and solution, which may have discouraged GPT-4 from working through the puzzles in steps.

Besides the fact that the direct execution of the LLMs results in low performance, it is hard to understand why they fail to solve puzzles; in other words, the results are hard to interpret.

On the other hand, in our method, although the LLMs can still produce incorrect ASP programs, the errors are much less and it is easy to identify the failures by examining the generated ASP programs. Table 2 summarizes the number of mistakes we encountered on 100 test data at each pipeline step. Below we provide a representative example for each type of errors from Table 2 and highlight the mistakes in red. The examples are taken from GPT-3, although each type also occurs with GPT-4, except for the syntax error.

- **Failure to convert certain constants into integers during constant formatting.** The only mistake happens when, given constants “points: 181; 184; 187; 190”, GPT-3 generated while it should generate constants in integers below to be able to apply arithmetic operations.

- **Addition of wrong clues during paraphrasing.** One such error happens when paraphrasing the sentence “2. The book published in 2011 is either Orlando’s project or Jan’s project”. GPT-3 generated the following three sentences.

  - 187 points”; “190 points”.

The first sentence is simply a copy of the original sentence while the last two should not be there since they cannot be inferred from the original sentence.

- **Syntax error during constraint generation.** One typical syntax error is an incorrect variable name in a generated rule. For example, given the sentence “The study on the Liberty starts 2 months after Eunice’s project”, GPT-3 generated the following rule while the variable S in the second line should have been S1.

- **Semantic error during constraint generation.** One typical semantic error is caused by a wrong equation. For example, given the sentence “the $35,000 structure is 15 sq ft smaller than the $29,000 home”, GPT-3 generated while the equation in the head should have been S1=S2-15.

While our pipeline doesn’t achieve 100% accuracy on generated ASP programs, most failed puzzles only have one

### Table 1: Accuracy on 50 train and 100 test puzzles. GPT-3 refers to the model named “text-davinci-003” in the OpenAI API, while GPT-4 is the model named “gpt-4.”

<table>
<thead>
<tr>
<th>Method</th>
<th>train set</th>
<th>test set</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Mitra and Baral 2015)</td>
<td>–</td>
<td>71%</td>
</tr>
<tr>
<td>Zero-shot GPT-3</td>
<td>0%</td>
<td>2%</td>
</tr>
<tr>
<td>Few-shot GPT-3</td>
<td>4%</td>
<td>3%</td>
</tr>
<tr>
<td>Zero-shot GPT-4</td>
<td>12%</td>
<td>21%</td>
</tr>
<tr>
<td>Few-shot GPT-4</td>
<td>6%</td>
<td>7%</td>
</tr>
<tr>
<td>GPT-3 Generated ASP Rules</td>
<td>86%</td>
<td>81%</td>
</tr>
<tr>
<td>GPT-4 Generated ASP Rules</td>
<td>92%</td>
<td>92%</td>
</tr>
</tbody>
</table>

### Table 2: Mistakes on 100 test puzzles at different pipeline steps.

<table>
<thead>
<tr>
<th>Step</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPT-3</td>
<td></td>
</tr>
<tr>
<td>GPT-4</td>
<td></td>
</tr>
<tr>
<td>constant formatting</td>
<td>3</td>
</tr>
<tr>
<td>paraphrasing</td>
<td>2</td>
</tr>
<tr>
<td>constraint generation (syntax)</td>
<td>3</td>
</tr>
<tr>
<td>constraint generation (semantic)</td>
<td>13</td>
</tr>
</tbody>
</table>

Footnote: 3For GPT-3/GPT-4, to avoid randomness, we use a temperature of 0 (deterministic) and a top P value of 1 (default setting).
6 More Examples

Previous approaches that automate logic puzzle solving either only predict constants and relations (Mitra and Baral 2015) or treat rule generation as a classification problem on a small set of rule templates (Jabrayilzade and Tekir 2020). In comparison, our method is generative, where rules are generated in an open-ended manner under the guidance of a few examples.

While it's hard to apply the previous methods to other domains without substantial changes, applying our pipeline to new domains requires only minor adjustments on the prompts. To apply our pipeline to other domains, we make a slight adjustment by turning the last sentence in Line 11 of Prompt R2 into a numbered clue “0. No option in any category will ever be used more than once.”, since it was specific to grid logic puzzles.

In the following part of this section, we show how our pipeline can be further applied to generate ASP programs for Sudoku and the Jobs Puzzle.

6.1 Sudoku

If we describe Sudoku problem with the following story

our pipeline generates the following ASP program II.

This ASP program II is almost correct except that the red part in Line 16 of II should be

since the row and column indices start from 1. This formula seems too difficult for GPT-3 to notice and generate unless some examples are provided. On the other hand, if we slightly adjust Lines 7–8 of Prompt C (Section 3.1) to make the indices start from 0, then the generated ASP program II becomes correct as Lines 2–3 of II are changed to the following facts.

GPT-4 also fails to generate the last rule correctly, although it makes a different mistake.

6.2 Jobs Puzzle

The Jobs Puzzle studied in (Schwitter 2013) asks one to assign 8 different jobs to 4 people while satisfying the given constraints. The full puzzle q is shown below.

This puzzle was considered a challenge for logical expressibility and automated reasoning (Shapiro 2011).

To apply our method to the Jobs Puzzle, some paraphrasing was needed before the Define&Test part of rule generation. We manually paraphrased the above puzzle to the following

There are four people: Roberta, Thelma, Steve, and Pete. Among them, they hold eight different jobs. Each holds exactly two jobs.
4. The jobs are: chef, guard, nurse, telephone operator, police officer (gender not implied), teacher, actor, and boxer.
5. The job of nurse is held by a male.
6. The husband of the chef is the telephone operator.
7. Roberta is not a boxer.
8. Pete has no education past the ninth grade.
9. Roberta, the chef, and the police officer went golfing together.
10. Question: Who holds which jobs?
by turning clues 1–4 as background story, clarifying clues 6, 8, and 9, and adding a few hidden clues numbered 10.X at the end.

As for the prompts, we only need to update Line 1 of Prompt R1 to the following to allow for \( \ldots = k \) in a rule.

Finally, GPT-3 generates the following ASP program:

Finally, GPT-3 generates the following ASP program:

which is almost correct with a single mistake in translating clue 10.1. If we just replace this constraint (in red) with

the corrected ASP program has exactly one stable model, which is the correct solution to the Jobs Puzzle.

Similarly, GPT-4 also failed to generate a completely correct ASP program. It also couldn’t generate a correct rule for constraint 10.1, and furthermore failed to produce the gender category in constant extraction step Prompt C), missing “gender: "male"; "female".”

7 Conclusion

LLMs are a relatively recent technology that have shown to be disruptive. Despite their wide range of applications, their responses are not always reliable and cannot be trusted.

Automatic rule generation is a difficult problem. However, by using LLMs as a front-end to answer set programming, we can utilize their linguistic abilities to translate natural language descriptions into the declarative language of answer set programs. Unlike previous methods that use algorithmic or machine learning techniques, we find that a pre-trained large language model with a good prompt can generate reasonably accurate answer set programs. We present a pipeline with general steps that systematically build an ASP program in a natural way. This method not only leads to higher accuracy but also makes the results interpretable.

We expect this type of work to expand the application of KR methods that may appear unfamiliar to non-experts. We also anticipate that this pipeline will serve as a suggestion tool to help users prepare valid constants, useful predicates, or draft ASP programs.
Acknowledgements
We are grateful to the anonymous referees for their useful comments. This work was partially supported by the National Science Foundation under Grant IIS-2006747.

References


