# Neuro-Symbolic Reasoning with Large Language Models and Answer Set Programming: A Case Study on Logic Puzzles

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

**Abstract**

Large pre-trained language models such as GPT-3 and ChatGPT have demonstrated exceptional performance in various natural language processing (NLP) tasks and have shown the ability to solve certain reasoning problems. However, their reasoning capabilities are limited and shallow, despite the application of various prompting techniques. On the other hand, while formal logic can handle complex reasoning, 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 logic programming languages. Specifically, we employ GPT-3 to transform natural language descriptions of logic puzzles into answer set programs. We carefully design prompts for GPT-3 to convert natural language descriptions into answer set programs in a step by step manner. Surprisingly, with just a few in-context learning examples, GPT-3 can generate reasonably complex answer set programs. Most errors it makes are simple for humans to correct, enabling GPT-3 to assist humans in writing answer set programs productively.

## 1 Introduction

Transformer-based large language models (LLMs) have recently shown remarkable success on many downstream tasks, demonstrating their general reasoning capability on diverse problems. However, while LLMs are good at generating System 1-like sequences, they are not good at System 2-inspired logical thinking, and their output is often inconsistent and incoherent (Nye et al. 2021). This is because LLMs are 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 crucial 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 performance gains on reasoning tasks. They use an LLM to produce 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) alone [1] does not have the ability to solve logic puzzles, despite various prompts we tried. On the other hand, we find that it can convert the natural language descriptions of the puzzles into answer set programs (Lifschitz 2008; Brewka, Niemelä, and Truszczynski 2011) surprisingly well. This is in part thanks to the declarative semantics of answer set programs. Even the errors GPT-3 makes are mostly simple for humans to correct. We hope that our finding will enable expanding the application of answer set programming to non-experts.

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 a performance evaluation of the approach. Section 6 shows more examples demonstrating the generalizability of our method.

The code is available at https://github.com/azreasoners/gpt3-asp-rule.

## 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 couple 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 to handle complex reasoning. However, the authors had to use hand-engineered set of constraints for the latter part. To our

[1]Throughout the paper, we use the text-davinci-003 model.

knowledge, our work is the first to use LLMs to generate constraint rules to solve complex reasoning tasks.

## 2.2 Automated Logic Puzzle Solving

Works on solving logic puzzles typically involve a natural language to logic formalism mapping, which usually includes some problem simplification such as tailoring 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, 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) addresses the ”Jobs Puzzle” in a number of ways, focusing on encodings of the problem into logical formalisms. However, the logical representations of clues are not automatically generated. In a similar domain, Puzzler (Milicevic, Near, and Singh 2012) uses a general link parser to translate to the language Alloy for solving in a mostly automated process, although the types are assumed to be given. LogicSolver (Nordstrom 2017) follows a similar approach to Puzzler but replaces Alloy with a custom solver, and conducts a more thorough evaluation.

Several works utilize an English-to-ASP translation. (Schwitter 2013) presents a solution to the “Jobs Puzzle” in controlled natural language (Schwitter 2010), which, like (Shapiro 2011), involves manually translating the English statements, though they are more closely aligned with the original puzzle. Some of these works consist of trained models. (Baral and Dzifcak 2012) employs a $\lambda$-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) trains a maximum entropy-based model to extract relations for each clue, which are then converted into a common ASP rule format such that a stable model corresponds to the puzzle solution. LGPSolver (Jabrayilzade and Tekir 2020) uses DistilBERT, a transformer-based model, as a classifier that can distinguish between representative rule types. With the clue classification, the authors use a hand-crafted 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 typically follow 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

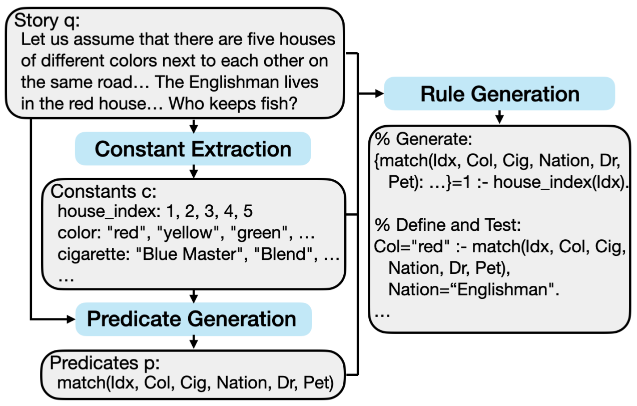
Story q:
Let us assume that there are five houses of different colors next to each other on the same road… The Englishman lives in the red house… Who keeps fish?
Constant Extraction
Constants c:
house_index: 1, 2, 3, 4, 5
color: "red", "yellow", "green", …
cigarette: "Blue Master", "Blend", …
…
Predicate Generation
Predicates p:
match(Idx, Col, Cig, Nation, Dr, Pet)
Rule Generation
% Generate:
{match(Idx, Col, Cig, Nation, Dr, Pet): …}=1 :- house_index(Idx).
% Define and Test:
Col="red" :- match(Idx, Col, Cig, Nation, Dr, Pet), Nation="Englishman".
…

Figure 1: Flow of Generating Answer Set Programs from Logic Puzzle in English

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 answer set encodes the solution. 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 $\mathcal{F}_c$ and $\mathcal{F}_p$ denote the *Constant Extraction* and *Predicate Generation* steps in Figure 1. Let $\mathcal{F}_{r1}$ and $\mathcal{F}_{r2}$ 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 $\Pi = \Pi_{generate} \cup \Pi_{define\_test}$.

$$c = \mathcal{F}_c(q) \qquad p = \mathcal{F}_p(q, c)$$
$$\Pi_{generate} = \mathcal{F}_{r1}(c, p) \qquad \Pi_{define\_test} = \mathcal{F}_{r2}(q, c, p)$$

Here, $c$ and $p$ denote extracted objects and generated predicates. Each step $\mathcal{F}_*$ 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.

In the absence of guidance on the format of objects, GPT-3 extracts 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. Instead of applying post-processing to rename them, we employ GPT-3 to generate a syntactically correct form in one step. This is achieved by invoking GPT-3 with the following prompt.

**Prompt C**:

```
1   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".
2
3   Problem 1:
4   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.
5
6   Constants:
7   index_of_row: 1; 2; 3; 4; 5; 6; 7; 8.
8   index_of_column: 1; 2; 3; 4; 5; 6; 7; 8.
9
10  Problem 2:
11  "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.
12  1. Bonita's piece costs $325.
13  2. The item made of poplar costs more than Yvette's
        piece.
14  3. Tabitha's item costs 50 dollars less than the
        piece made of sandalwood.
15  4. The $275 item is either the piece made of ash or
        Yvette's item.
16
17  Constants:
18  employee: "Bonita"; "Yvette"; "Tabitha".
19  price: 225; 275; 325.
20  wood_type: "ash"; "poplar"; "sandalwood".
21
22  Problem 3:
23  <story>
24
25  Constants:
```

Line 1 provides a general instruction for the task of extracting objects and directing GPT-3 to generate them in the form of "category: constant$_1$; ...; constant$_n$". Then, two examples follow: Lines 6-8 for Problem 1 specified in Lines 3-4, and Lines 17-20 for Problem 2 specified in Lines 10-15. By replacing Line 23 ($\langle$story$\rangle$) with a new example story and invoking GPT-3 with the above prompt, a new list of categories and constants for that story is generated, as with the previous two examples.

The above two examples are chosen to cover two cases of object extraction. For the N-Queens problem, the constants $1, \ldots, 8$ 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**:

```
1   Given a problem and some categorized constants of the
         form "category: constant_1; constant_2; ...;
        constant_n", generate the minimum number of
        predicates to define the relations among the
        categories of constants. Each generated
        predicate is of the form "predicate(X1, X2, ...,
         Xn)" where X1, X2, ..., Xn are different
        variables and each variable X belongs to one of
        the categories. For each category, there must
        exist at least one variable of some predicate
        that belongs to this category.
2
3   Problem 1:
4   (Lines 4-8 from Prompt C: Omitted)
5
6   Predicates:
7   % The categories in Constants include index_of_row
        and index_of_column. We use different variables
        Ir and Ic to represent index_of_row and
        index_of_column.
8   % We assign a queen at row Ir and column Ic, where Ir
         belongs to index_of_row and Ic belongs to
        index_of_column.
9   assign(Ir, Ic)
10
11  Problem 2:
12  (Lines 11-20 from Prompt C: Omitted)
13
14  Predicates:
15  % The categories in Constants include employee, price
        , and wood_type. We use different variables E, P
        , and W to represent employee, price, and
        wood_type.
16  % 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.
17  match(E, P, W)
18
19  Problem 3:
20  <story>
21
22  Constants:
23  <constants>
```

```
24
25 Predicates:
```

Line 1 is a general instruction describing the task of predicate generation, and that the generated predicates should follow the form of "predicate($X_1$, ..., $X_n$)" where each $X_i$ 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 $\Pi$, 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 $\Pi_{generate}$ is obtained by calling GPT-3 with **Prompt R1**.

**Prompt R1**:

```
1  Given some categorized constants in the form "
       category: constant_1; constant_2; ...;
       constant_n" 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.
2
3  Constants:
4  employee: "Bonita"; "Yvette"; "Tabitha".
5  price: 225; 275; 325.
6  wood_type: "ash"; "poplar"; "sandalwood".
7
8  Predicates:
9  % The categories include employee, price, and
       wood_type. We use different variables E, P, and
       W to represent employee, price, and wood_type.
10 % 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.
11 match(E, P, W)
12
13 ASP Rules:
14 % Define the constants in each category.
15 employee("Bonita"; "Yvette"; "Tabitha").
16 price(225; 275; 325).
17 wood_type("ash"; "poplar"; "sandalwood").
18
19 % For each employee E, it matches with exactly 1
       price P and 1 wood type W.
20 {match(E, P, W): price(P), wood_type(W)}=1 :-
       employee(E).
21
22 Constants:
23 <constants>
24
25 Predicates:
26 <predicates>
27
28 ASP rules:
```

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 $\Pi_{define\_test}$ is obtained by calling GPT-3 with **Prompt R2**.

**Prompt R2**:

```
1  Consider the constraint in the following form
2  <C1>; <C2>; ...; <Cm> :- <L1>, <L2>, ..., <Ln>.
3  which says that if the conjunction "<L1> and <L2> and
        ... and <Ln>" is true, then the disjunction of
       comparisons "<C1> or <C2> or ... or <Cm>" must
       be true.
4
5  One can also add a restriction that "exactly k of <C1
       >, <C2>, ..., <Cm> is true" by using the
       following form
6  {<C1>; <C2>; ...; <Cm>}=k :- <L1>, <L2>, ..., <Ln>.
7
8  Given a problem, extract all constraints from the
       clues in the problem using only the provided
       constants and predicates.
9
10 Problem 1:
11 "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.
12  1. Bonita's piece costs $325.
13  2. The item made of poplar costs more than Yvette's
       piece.
14  3. Tabitha's item costs 50 dollars less than the
       piece made of sandalwood.
15  4. The $275 item is either the piece made of ash or
       Yvette's item.
16
17  Constants:
18  employee: "Bonita"; "Yvette"; "Tabitha".
19  price: 225; 275; 325.
20  wood_type: "ash"; "poplar"; "sandalwood".
21
22  Predicates:
23  % The categories include employee, price, and
       wood_type. We use different variables E, P, and
       W to represent employee, price, and wood_type.
24  % 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.
25  match(E, P, W)
26
27  Constraints:
28  % No option in any category will ever be used more
       than once.
29  {E1=E2; P1=P2; W1=W2}=0 :- match(E1,P1,W1), match(E2,
       P2,W2), (E1,P1,W1)!=(E2,P2,W2).
30
31  % 1. Bonita's piece costs $325.
32  P=325 :- match(E,P,W), E="Bonita".
33
34  % 2. The item made of poplar costs more than Yvette's
        piece.
35  P1>P2 :- match(E1,P1,W1), match(E2,P2,W2), W1="poplar
       ", E2="Yvette".
36
37  % 3. Tabitha's item costs 50 dollars less than the
       piece made of sandalwood.
38  P1=P2-50 :- match(E1,P1,W1), match(E2,P2,W2), E1="
       Tabitha", W2="sandalwood".
39
40  % 4. The $275 item is either the piece made of ash or
        Yvette's item.
41  {W="ash"; E="Yvette"}=1 :- match(E,P,W), P=275.
42
43  (Problem 2 omitted)
44
45  Problem 3:
46  <story>
47
48  Constants:
49  <constants>
50
51  Predicates:
52  <predicates>
53
54  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 $\Pi_{define\_test}$ generation and provides two rule forms for the target ASP rules. The first rule form

$$C_1; C_2; \dots; C_m \ \leftarrow \ L_1, L_2, \dots, L_n$$

says that "$C_1$ or ... or $C_m$ is true if $L_1$ and ... and $L_n$ are true." Here, each $L_i$ is a literal and each $C_i$ is a comparison in the input language of CLINGO, e.g., $A > B$, $A = B + 3$, etc. The second rule form

$$\{C_1; C_2; \dots; C_m\} = k \ \leftarrow \ L_1, L_2, \dots, L_n$$

additionally restricts that "exactly $k$ of $\{C_1, \dots, C_m\}$ 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. In this section, we explain two optional enhancements to the pipeline that strengthen its robustness when dealing with arbitrary sentences.

### 4.1 Constant Formatting

One common mistake of GPT-3 in rule generation is that it 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`).

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 $c$. The GPT-3 response in this step is the updated constants $c$, serving as an input to other steps in the pipeline.

```
Given categorized constants of the form "category:
    constant_1; constant_2; ...; constant_n", format
     the category and constants such that:
each category consists of only lowercase letters and
    underscores, and
each constant is either an integer or a string
    surrounded by double quotes, e.g., "United
    States".

There are two ways below to format constants and we
    must use the same way for all constants of the
    same category.
1. Turn all constants of the same category into
    integers with no space or special character.
2. Add double quotes around all constants of the same
     category.
Note that the 1st way has a higher priority, meaning
    that we must turn all constants of the same
    category into integers whenever possible. For
    example, twice or second can be turned into 2,
    September can be turned into 9, September 5th
    can be turned into 5 if all dates are in
    September, but 9:30am can only be turned into
    "9:30am" since no integer can represent 9:30am.
```

```

Original constants:
Employees: Bonita; Yvette; Tabitha.
Prices: $225; $275; $325.
Wood types: ash; poplar; sandalwood.

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

Original constants:
months: January; April; October; December.
times: 8:30AM; 10:30AM; 2:30PM; 3:30PM.
durations: 1 day; 3 days; 11 days; 12 days.

Formatted constants:
month: 1; 4; 10; 12.
time: "8:30AM"; "10:30PM"; "2:30PM"; "3:30PM".
duration: 1; 3; 11; 12.

Original constants:
⟨constants⟩

Formatted constants:
```

## 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

```
Of the person who won the prize in bioengineering and
     Sue Simpson, one won in 1976 and the other won
    in 1968.
```

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 person who won the prize in bioengineering and
    Sue Simpson are different.
The person who won the prize in bioengineering won in
     1976 or won in 1968.
Sue Simpson won in 1976 or won in 1968.
```

The Sentence Paraphrasing step is done by calling GPT-3 with the following prompt, where ⟨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 “of A and B, one is C and the other is D.”

```
Copy a sequence of numbered sentences.

If a sentence is of the form "N. Of A and B, one is C
     and the other is D", replace it with 3
    sentences below.
N.1 A and B are different.
N.2 A is C or D.
N.3 B is C or D.

For every sentence, if it is not of the form "N. Of
     ... and ...", simply copy it without replacement
     . An easy way to determine if a sentence is not
     of the above form is to check if its first word
     is not of.

In the following example, one sentence is of the
     above form.
Given:
1. The squad from Grenada ended with 2 silver medals.
2. Of the team from Oman and the team that won 10
     silver medals, one finished with 2 gold medals
     and the other finished with 1 gold medal.
Copy:
1. The squad from Grenada ended with 2 silver medals.
2.1 The team from Oman and the team that won 10
     silver medals are different.
2.2 The team from Oman finished with 2 gold medals or
      finished with 1 gold medal.
2.3 The team that won 10 silver medals finished with
     2 gold medals or finished with 1 gold medal.

In the following example, no sentence is of the above
      form.
Given:
1. Tabitha's item costs 50 dollars less than the
     piece made of sandalwood.
2. The $275 item is either the piece made of ash or
     Yvette's item.
Copy:
1. Tabitha's item costs 50 dollars less than the
     piece made of sandalwood.
2. The $275 item is either the piece made of ash or
     Yvette's item.

Given:
⟨sentences⟩
Copy:
```

## 5 Experiments

We tested the above pipeline on the logic puzzles dataset from (Mitra and Baral 2015). Since the constants are provided in the dataset as necessary information to solve each puzzle, we apply Constant Formatting directly on the given constants to generate constants $c$.

The dataset consists of 50 training examples and 100 testing examples. When we design our prompts, we only refer to the training examples and evaluate our pipeline on both of them. Table 1 shows the performance of our approach to few-shot GPT-3 and the learning compared to a fully-supervised learning system LOGICIA (Mitra and Baral 2015). For the few-shot setting, we use the first two examples in the training set as the few-shot examples.[2] We

[2]There is some overlap between the training and test set, and two of the four correct cases for the baseline few-shot model on the test set are in the few-shot prompt. Hence the real performance is likely worse.

| Method | train set | test set |
|---|---|---|
| (Mitra and Baral 2015) | – | 71% |
| Few-shot GPT-3 | 2% | 4% |
| Zero-shot GPT-4 | – | 21% |
| Ours | 86% | 81% |

Table 1: Accuracy on 50 train and 100 test puzzles

| Step | Count |
|---|---|
| constant formatting | 1 |
| paraphrasing | 2 |
| constraint generation (syntax) | 11 |
| constraint generation (semantics) | 8 |

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

also tred a zero-shot setting with the very recently released GPT-4 [3], which outperforms GPT-3 in many benchmarks. Indeed, GPT-4 could solve 21% of the test puzzles correctly, which is remarkable compared with GPT-3's 4%. However, this is much lower than our method's 81%.

Besides that the direct execution of GPT-3 results in low performance, it is hard to understand why GPT-3 fails to solve puzzles as it is a blackbox; in other words, the result is hard to interpret.

On the other hand, in our method, although GPT-3 could still produce wrong ASP programs, the errors are much less and it is easy to identify the failure by examining the generated ASP programs. Table 2 summarizes the number of mistakes we encountered on 100 test data at each pipeline step. The majority (18) of mistakes happen during constraint generation, consisting of 10 syntax errors and 8 semantic errors. The syntax errors consist of small mistakes in terms, variables, or the location of atoms. The semantic errors are all about mathematical computation or comparison. Below we list a representative example for each type of errors from Table 2 and highlight the mistakes in red.

- **Failure to convert certain constants into integers during constant formatting.** The only mistake happens when, given constants "points: 181 points; 184 points; 187 points; 190 points", GPT-3 generated

```
points:  "181 points"; "184 points"; "187
    points"; "190 points".
```

while it should generate constants in integers below to be able to apply arithmetic operations.

```
points: 181; 184; 187; 190.
```

- **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.

[3]GPT-4 was announced on the day of submission deadline. We couldn't help but try it! https://openai.com/product/gpt-4.

```
% 2.1 The book published in 2011 is either Orlando
    's project or Jan's project.
% 2.2 Orlando's project is the book
    published in 2011 or the 200 page
    project.
% 2.3 Jan's project is the book published
    in 2011 or the 200 page project.
```

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 generates the following rule

```
M=M1+2 :- match(S,M,Sh), match(S1,M1,Sh1),
    S="Eunice", Sh="Liberty".
```

while the variable `S` in red should be `S1` as shown below.

```
M=M1+2 :- match(S,M,Sh), match(S1,M1,Sh1),
    S1="Eunice", Sh="Liberty".
```

- **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

```
S1=S2+15 :- match(C1,S1,P1), match(C2,S2,P2), P1
    =35000, P2=29000.
```

while the equation should be `S1=S2-15` as shown below.

```
S1=S2-15 :- match(C1,S1,P1), match(C2,S2,P2), P1
    =35000, P2=29000.
```

While our pipeline doesn't achieve 100% accuracy on generated ASP programs, most failed puzzles only have one mistake and that mistake is easy to correct. This means that our pipeline could serve as a good suggestion tool to prepare draft ASP programs for users. For example, compared to designing all the ASP programs for 50+100 puzzles manually, it would save a significant amount of time to only check the correctness of the automatically generated rules for the programs that don't have a single stable model.

## 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, one general adjustment is to turn the last sentence (which is domain-specific for logic puzzles dataset) in Line 11 of **Prompt R1** into a numbered clue

```
0. No option in any category will ever be used more
     than once.
```

before Line 12 of **Prompt R1** so that this sentence is not assumed to always hold and GPT-3 doesn't always generate a constraint like Lines 28–29 of **Prompt R1** for any puzzle.

In the following part of this section, we show how our pipeline (with the general adjustment) can be further applied to generate ASP programs for Sudoku, its variants, and the Jobs Puzzle (Schwitter 2013).

## 6.1 Sudoku and its Variants

If we describe Sudoku problem with the following story $q$

```
The Sudoku problem is a classic computer science
     problem that involves filling a 9x9 grid with
     digits from 1 to 9.
1. The same number cannot appear twice in the same
     row.
2. The same number cannot appear twice in the same
     column.
3. The same number cannot appear twice in the 3x3 box
     . (You can use division / to check if 2 cells
     are in the same box.)
```

our pipeline generates the following ASP program $\Pi$.

```
1  % Define the constants in each category.
2  index_of_row(1; 2; 3; 4; 5; 6; 7; 8; 9).
3  index_of_column(1; 2; 3; 4; 5; 6; 7; 8; 9).
4  number(1; 2; 3; 4; 5; 6; 7; 8; 9).
5  % For each row Ir and column Ic, it assigns exactly 1
      number N.
6  {assign(Ir, Ic, N): number(N)}=1 :- index_of_row(Ir),
      index_of_column(Ic).
7
8  % 1. The same number cannot appear twice in the same
     row.
9  {N1=N2}=0 :- assign(Ir,Ic1,N1), assign(Ir,Ic2,N2), (
     Ic1,N1)!=(Ic2,N2).
10
11 % 2. The same number cannot appear twice in the same
     column.
12 {N1=N2}=0 :- assign(Ir1,Ic,N1), assign(Ir2,Ic,N2), (
     Ir1,N1)!=(Ir2,N2).
13
14 % 3. The same number cannot appear twice in the 3x3
     box.
15 {N1=N2}=0 :- assign(Ir1,Ic1,N1), assign(Ir2,Ic2,N2),
      (Ir1/3,Ic1/3)=(Ir2/3,Ic2/3), (Ir1,Ic1,N1)
     !=(Ir2,Ic2,N2).
```

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

```
((Ir1-1)/3, (Ic1-1)/3) = ((Ir2-1)/3, (Ic2-1)/3)
```

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 $\Pi$ becomes correct as Lines 2–3 of $\Pi$ are changed to the following facts.

```
index_of_row(0; 1; 2; 3; 4; 5; 6; 7; 8).
index_of_column(0; 1; 2; 3; 4; 5; 6; 7; 8).
```

## 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.

```
1. There are four people: Roberta, Thelma, Steve, and
     Pete.
2. Among them, they hold eight different jobs.
3. 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.
Question: Who holds which jobs?
```

This puzzle was considered a challenge for logical expressibility and automated reasoning (Shapiro 2011). It is also hard to solve for human experts because the problem description in natural language is not complete and contains ambiguous information.

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. 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,
      which means the chef is a female and the
     telephone operator is a male.
7. Roberta is not a boxer.
8. Pete has no education past the ninth grade. (Three
      jobs teacher, nurse, and police officer require
      higher education.)
9. Roberta, the chef, and the police officer went
     golfing together, meaning they are different
     people.
10.1 The same job cannot be assigned to different
     people.
10.2 Roberta is a female.
10.3 Thelma is a female.
10.4 Steve is a male.
10.5 Pete is a male.
10.6 The actor is a male.
```

```
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 {...}=k in a rule.

```
Given a problem as the background information, turn
    the provided categorized constants and
    predicates into rules.

Tasks:
1. Write a rule of the form "category(constant_1;
    constant_2; ...; constant_n)" to represent each
    row of categorized constants in the form "
    category: constant_1; constant_2; ...;
    constant_n".
2. Write a rule of the form "{predicate(X_1, X_2,
    ..., X_n): category_2(X_2), ..., category_n(X_n)
    }=k :- category_1(X_1)" for each predicate of
    the form "predicate(X_1, X_2, ..., X_n)".
```

Finally, GPT-3 generates the following ASP program:

```
% Task 1. Define the constants in each category.
person("Roberta"; "Thelma"; "Steve"; "Pete").
job("chef"; "guard"; "nurse"; "telephone operator"; "
    police officer"; "teacher"; "actor"; "boxer").
gender("male"; "female").
% Task 2. For each person P, there is exactly 2 jobs
    J and gender G.
{assign(P, J, G): job(J), gender(G)}=2 :- person(P).
% Done tasks 1 and 2.

% 5. The job of nurse is held by a male.
G="male" :- assign(P,J,G), J="nurse".

% 6. The husband of the chef is the telephone
    operator, which means the chef is a female and
    the telephone operator is a male.
G1="female" :- assign(P1,J1,G1), assign(P2,J2,G2), J1
    ="chef", J2="telephone operator".
G2="male" :- assign(P1,J1,G1), assign(P2,J2,G2), J1="
    chef", J2="telephone operator".

% 7. Roberta is not a boxer.
J!="boxer" :- assign(P,J,G), P="Roberta".

% 8. Pete has no education past the ninth grade. (
    Three jobs teacher, nurse, and police officer
    require higher education.)
{J="teacher"; J="nurse"; J="police officer"}=0 :-
    assign(P,J,G), P="Pete".

% 9. Roberta, the chef, and the police officer went
    golfing together, meaning they are different
    people.
P1!=P2 :- assign(P1,J1,G1), assign(P2,J2,G2), assign(
    P3,J3,G3), P1="Roberta", J2="chef", J3="police
    officer".
P2!=P3 :- assign(P1,J1,G1), assign(P2,J2,G2), assign(
    P3,J3,G3), P1="Roberta", J2="chef", J3="police
    officer".
P1!=P3 :- assign(P1,J1,G1), assign(P2,J2,G2), assign(
    P3,J3,G3), P1="Roberta", J2="chef", J3="police
    officer".

% 10.1 The same job cannot be assigned to different
    people.
{P1=P2}=0 :- assign(P1,J,G1),
    assign(P2,J,G2), (P1,G1)!=(P2,G2).

% 10.2 Roberta is a female.
G="female" :- assign(P,J,G), P="Roberta".

% 10.3 Thelma is a female.
G="female" :- assign(P,J,G), P="Thelma".

% 10.4 Steve is a male.
G="male" :- assign(P,J,G), P="Steve".

% 10.5 Pete is a male.
G="male" :- assign(P,J,G), P="Pete".

% 10.6 The actor is a male.
G="male" :- assign(P,J,G), J="actor".
```

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

```
P1=P2 :- assign(P1,J,G1), assign(P2,J,G2).
```

the corrected ASP program would have exactly one stable model below, representing the correct solution to the Jobs Puzzle.

```
assign("Thelma","chef","female")
assign("Thelma","boxer","female")
assign("Steve","nurse","male")
assign("Steve","police officer","male")
assign("Pete","telephone operator","male")
assign("Pete","actor","male")
assign("Roberta","guard","female")
assign("Roberta","teacher","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.

## References

Baral, C., and 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*, 573–577.

Brewka, G.; Niemelä, I.; and Truszczynski, M. 2011. Answer set programming at a glance. *Communications of the ACM* 54(12):92–103.

Brown, T.; Mann, B.; Ryder, N.; Subbiah, M.; Kaplan, J. D.; Dhariwal, P.; Neelakantan, A.; Shyam, P.; Sastry, G.; Askell, A.; et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems* 33:1877–1901.

Creswell, A.; Shanahan, M.; and Higgins, I. 2022. Selection-inference: Exploiting large language models for interpretable logical reasoning. *arXiv preprint arXiv:2205.09712*.

Jabrayilzade, E., and Tekir, S. 2020. Lgpsolver-solving logic grid puzzles automatically. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, 1118–1123.

Lev, I.; MacCartney, B.; Manning, C. D.; and Levy, R. 2004. Solving logic puzzles: From robust processing to precise semantics. In *Proceedings of the 2nd Workshop on Text Meaning and Interpretation*, 9–16.

Lifschitz, V. 2008. What is answer set programming? In *Proceedings of the AAAI Conference on Artificial Intelligence*, 1594–1597. MIT Press.

Milicevic, A.; Near, J. P.; and Singh, R. 2012. Puzzler: An automated logic puzzle solver. Technical report, Massachusetts Institute of Technology (MIT).

Mitra, A., and Baral, C. 2015. Learning to automatically solve logic grid puzzles. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, 1023–1033.

Nordstrom, R. 2017. Logicsolver-solving logic grid puzzles with part-of-speech tagging and first-order logic. Technical report, University of Colorado, Colorado Springs.

Nye, M.; Tessler, M.; Tenenbaum, J.; and Lake, B. M. 2021. Improving coherence and consistency in neural sequence models with dual-system, neuro-symbolic reasoning. *Advances in Neural Information Processing Systems* 34:25192–25204.

Rae, J. W.; Borgeaud, S.; Cai, T.; Millican, K.; Hoffmann, J.; Song, F.; Aslanides, J.; Henderson, S.; Ring, R.; Young, S.; et al. 2021. Scaling language models: Methods, analysis & insights from training gopher. *arXiv preprint arXiv:2112.11446*.

Schwitter, R. 2010. Controlled natural languages for knowledge representation. In *Coling 2010: Posters*, 1113–1121.

Schwitter, R. 2013. The jobs puzzle: Taking on the challenge via controlled natural language processing. *Theory and Practice of Logic Programming* 13(4-5):487–501.

Shapiro, S. C. 2011. The jobs puzzle: A challenge for logical expressibility and automated reasoning. In *AAAI spring symposium: logical formalizations of commonsense reasoning*.

Valmeekam, K.; Olmo, A.; Sreedharan, S.; and Kambhampati, S. 2022. Large language models still can't plan (a benchmark for LLMs on planning and reasoning about change). In *NeurIPS 2022 Foundation Models for Decision Making Workshop*.

Wei, J.; Wang, X.; Schuurmans, D.; Bosma, M.; brian ichter; Xia, F.; Chi, E. H.; Le, Q. V.; and Zhou, D. 2022. Chain of thought prompting elicits reasoning in large language models. In Oh, A. H.; Agarwal, A.; Belgrave, D.; and Cho, K., eds., *Advances in Neural Information Processing Systems*.

Zhou, D.; Schärli, N.; Hou, L.; Wei, J.; Scales, N.; Wang, X.; Schuurmans, D.; Bousquet, O.; Le, Q.; and Chi, E. 2022. Least-to-most prompting enables complex reasoning in large language models. *arXiv preprint arXiv:2205.10625*.