Declarative Knowledge Distillation from Large Language Models for Visual Question Answering

Thomas Eiter[1] , **Jan Hadl**[1] , **Nelson Higuera**[1] , **Johannes Oetsch**[2]
[1]Institute for Logic and Computation, TU Wien, Favoritenstraße 9–11, 1040 Vienna, Austria
[2]Department of Computing, Jönköping University, Gjuterigatan 5, 551 11 Jönköping, Sweden
{thomas.eiter, jan.hadl, nelson.ruiz}@tuwien.ac.at, johannes.oetsch@ju.se

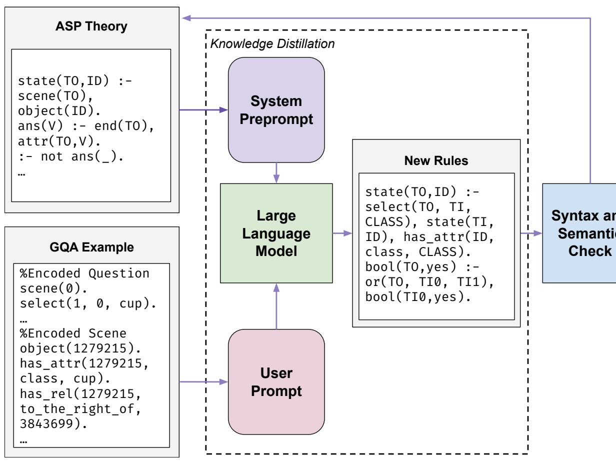
Abstract

Visual Question Answering (VQA) is the task of answering a question about an image and requires processing multimodal input and reasoning to obtain the answer. Modular solutions that use declarative representations within the reasoning component have a clear advantage over end-to-end trained systems regarding interpretability. The downside is that crafting the rules for such a component can be an additional burden on the developer. We address this challenge by presenting an approach for declarative knowledge distillation from Large Language Models (LLMs). Our method is to prompt an LLM to extend an initial theory on VQA reasoning, given as an answer-set program, to meet the requirements of the VQA task. Examples from the VQA dataset are used to guide the LLM, validate the results, and mend rules if they are not correct by using feedback from the ASP solver. We demonstrate that our approach works on two VQA datasets: CLEVR and GQA. Our results confirm that distilling knowledge from LLMs is in fact a promising direction besides data-driven rule learning approaches.[1]

1 Introduction

Visual question answering (VQA) (Antol et al. 2015; Goyal et al. 2017) is a challenging problem with valuable applications (Barra et al. 2021; Lin et al. 2023); it is the task of providing an accurate answer for a question about a visual scene. This requires not just a joint understanding of vision and text, but also the ability to follow complex chains of reasoning operations.

Neurosymbolic approaches to VQA (Mao et al. 2019; Yi et al. 2018; Amizadeh et al. 2020; Eiter et al. 2022b; Surís, Menon, and Vondrick 2023; Johnston, Nogueira, and Swingler 2023, etc.) use deep learning for perception, produce a symbolic representation of the input image and question, and then perform reasoning on this representation in a purely symbolic way. These approaches are interpretable, transparent, and can be extended easily due to their compositional structure. A promising direction in this regard is to use logic-based formalisms for the reasoning component. We are in particular interested in using Answer-Set Programming (ASP) (Brewka, Eiter, and Truszczynski 2011; Lifschitz 2019), a prominent knowledge representation framework, for the reasoning module of such systems. Besides concise representations, advantages are that non-determinism allows for multiple answers if desired, ambiguity in the perception often can be resolved in the reasoning module (Eiter et al. 2022b), and one can more easily add explanation capabilities (Eiter et al. 2023). The downside is that crafting the rules for such a component is not always easy and can be an additional burden on the developer.



```
state(TO,ID) :-
scene(TO),
object(ID).
ans(V) :- end(TO),
attr(TO,V).
:- not ans(_).
…
```

```
%Encoded Question
scene(0).
select(1, 0, cup).
…
%Encoded Scene
object(1279215).
has_attr(1279215,
class, cup).
has_rel(1279215,
to_the_right_of,
3843699).
…
```

```
state(TO,ID) :-
select(TO, TI,
CLASS), state(TI,
ID), has_attr(ID,
class, CLASS).
bool(TO,yes) :-
or(TO, TI0, TI1),
bool(TI0,yes).
```

Figure 1: An overview of our knowledge distillation method.

We address this challenge by presenting an approach for *declarative knowledge distillation* from Large Language Models (LLMs) (Vaswani et al. 2017; Zhao et al. 2023) (see Fig. 1). The assumption is that we want to develop a modular solution for a VQA task for which we have, as is commonly the case, a dataset of questions, scenes, and answers at our disposal. Further, the reasoning component involves already an initial ASP theory that covers some but not all aspects of the VQA task. Our method is to prompt an LLM to extend this initial theory to meet the remaining requirements. Examples from a VQA dataset are used to guide the LLM, validate the results, and mend the generated rules if they are not correct using feedback from the ASP solver. More specifically, if we encounter an example from the dataset for which the system does not yet give the correct answer, we prompt the LLM with a description of the missing behaviour, the scene,

[1]This work has been supported by the Bosch Center for AI. Code for reproducing our experiments is available as an online repository: https://github.com/pudumagico/KR2024.

question, and expected answer to suggest rules to extend the current program. We then run an ASP solver to test the new program. If the output of the ASP solver does not match the expected answer, we give the wrong answer and the expected answer to the LLM and prompt it to correct it accordingly. If the ASP solver gives a syntax error, we ask the LLM to fix it. If the test passes, we proceed with regression testing on previous examples to ensure that the system will still answer any question it could answer before the extension. We also experiment with presenting examples in batches to reduce the number of calls to the LLM.

Our approach bears similarities with methods from statistical-relational learning (SRL) (Raedt and Kersting 2017), as there the goal is also to find rules that generalise examples from a dataset. While SRL is data driven, our method uses examples only to guide a knowledge distillation process via conditioning an LLM; we go into more detail in the related work section.

We use two VQA datasets to evaluate our knowledge distillation method: CLEVR (Johnson et al. 2017) and GQA (Hudson and Manning 2019). CLEVR uses synthetic scenes and is designed to challenge VQA systems with compositional questions that require breaking down the overall task into a tree of primitive operations that is evaluated recursively. GQA uses real images that depict complex visual scenes and diverse questions with a large number of possible answers that involve reasoning on the image scene graph, i.e., the objects in the scene, their attributes, and relations. The VQA system that we use for CLEVR is NSVQASP (Eiter et al. 2023). For GQA, we introduce the system GS-VQA, which has been recently developed (Hadl 2023). Notably, GS-VQA is able to perform VQA in a *zero-shot manner*, i.e., without training or fine-tuning of components to the current dataset. This is a significant advantage over related systems that require a costly training process. Both NSVQASP and GS-VQA use ASP for reasoning.

Our experiments show that LLMs can understand and produce ASP rules to implement new reasoning operators, and thus can effectively assist in the process of generating the reasoning component. This methodology offers a promising avenue for extracting domain-specific knowledge from LLMs in the realm of VQA and offers a viable alternative to data-driven rule learning approaches.

The remainder of the paper is organised as follows. We discuss related work in Section 2 and present our VQA approaches in Section 3. Then, we introduce the knowledge distillation method in Section 4, evaluate it in Section 5, and conclude the article in Section 6.

2 Related Work

Answer-Set Programming and LLMs. Recently, LLMs have been explored in the context of ASP. Ishay, Yang, and Lee (2023) observed that LLM reasoning capabilities are shallow, but they can serve as a highly effective semantic parser to transform input into ASP representations. These are then used to solve logical puzzles. We also recently proposed to use LLMs to parse question into ASP facts in the context of VQA for images of graphs (Bauer et al. 2023). Rajasekharan et al. (2023) showed that the combination of LLMs with ASP in their STAR framework produces good results for natural language tasks that require qualitative reasoning, mathematical reasoning, and goal-directed conversation. Our work is different as we do not focus on semantic parsing but on the more challenging task of knowledge distillation, where we aim for a system that can produce sound logical rules capturing knowledge about a particular domain. Recent work by Zhu et al. (2023) explores the use of LLMs to learn rules from arithmetic and kinship relationships, yet the rules they learn do not contain variables and their semantics is informal.

Statistical-relational learning. Similar to our approach, methods from statistical-relational learning (SRL) (Raedt and Kersting 2017), in particular from inductive logic programming (ILP) (Muggleton 1991; Muggleton and Raedt 1994; Cropper et al. 2022), aim at producing rules from example data and a background theory. SRL has seen great advances in terms of scalability by, e.g., applying gradient-based boosting (Gutmann and Kersting 2006), and systems like ILASP (Law, Russo, and Broda 2020) and FastLAS (Law et al. 2020) provide means for inductively learning expressive ASP programs.

However, SRL takes a statistical and probabilistic learning perspective that is in essence data driven. Our method is orthogonal to that, as it does not aim at learning. On the surface, we also use a data set, but the role is very different as it guides a knowledge distillation process via conditioning an LLM. ILP uses a search-based approach where the solutions produced are correct and minimal under some criteria. A key aspect of many ILP systems are *mode declarations* that define the syntactic form of allowed rules to restrict the search space of possible programs. Mode declarations are in a formal language, and they tacitly assume an intuition about the form of the solution. For the distillation approach , we do not need that; we only elicit knowledge that is already present in the LLM, and the information in the prompt that instructs what rules we want is informal and in natural language. When prompting LLMs, rules are general by command—while optimality is not enforced, it may happen implicitly.

Modular neurosymbolic VQA. There are several VQA systems that feature a modular architecture which combines subsymbolic with symbolic components (Yi et al. 2018; Mao et al. 2019; Amizadeh et al. 2020; Eiter et al. 2022b; Surís, Menon, and Vondrick 2023; Johnston, Nogueira, and Swingler 2023). Specifically, Yi et al. (2018) used a pipeline to extract a *scene graph* (a list of all objects detected in the image with their attributes and positions) from the image. They then translated the provided question into a structured representation of the reasoning steps, called *functional program*, and executed this program on the structural scene representation to obtain an answer. The authors showed excellent results on the popular CLEVR dataset (Johnson et al. 2017). This approach has been advanced with logic-based reasoning processes, e.g., by Differentiable First-Order Logic (∇-FOL) (Amizadeh et al. 2020), or by ASP (Eiter et al. 2022a). These reasoning processes can consider not just the most probable scene-graph prediction, but rather the entire vector of probabilities as output by the object detection and attribute/relation classifier networks that form the visual perception component of the VQA pipeline. Foundational models such as Vision-Language Models (VLMs) like BLIP-2 (Li et al. 2023) and SimVLM (Wang et al. 2022) have become sufficiently strong through their pre-training regimes to generalise well to multiple different datasets. Approaches that use these VLMs as components are, e.g., ViperGPT (Surís, Menon, and Vondrick 2023), CodeVQA (Subramanian et al. 2023), and PnP-VQA (Tiong et al. 2022).

3 VQA Approaches and Datasets

In this section, we present the two modular neurosymbolic VQA approaches for the two datasets, GQA and CLEVR, that we are going to use for our evaluation. Both systems use ASP to derive answers from a symbolic scene representation. So we start by reviewing the basics of ASP next.

3.1 Answer-Set Programming

Answer-Set Programming (ASP) (Brewka, Eiter, and Truszczynski 2011; Lifschitz 2019) is a well-known approach to declarative problem solving, in which solutions to a problem are described by sets of logical rules. Efficient ASP solvers for evaluating the rules are readily available.[2]

For our concerns, an ASP program is a finite set P of rules r of the following form:

$$a :-\ b_1, \ldots,\ b_n,\ \mathit{not}\ c_1, \ldots,\ \mathit{not}\ c_n \quad m, n \geq 0$$

where a, all b_i, and all c_j are atoms in a first-order predicate language, and not stands for negation as failure (aka. weak negation). We allow that a may be missing (viewed as falsity); then r acts as a constraint. Intuitively, the rule means that whenever all b_i are true and none of the c_j can be shown to be true, then a must be true. Some rules appear in Fig 2b.

The semantics of a ground (variable-free) ASP program is given in terms of answer sets, which are Herbrand models that satisfy a stability condition (Gelfond and Lifschitz 1988). A Herbrand interpretation of P is a set I of ground atoms in the language induced by P (intuitively, the atoms that are true). Such an I is a model of P if for each rule r in P either (i) $a \in I$ or (ii) $\{b_1, \ldots, b_n\} \not\subseteq I$ or (iii) $I \cap \{c_1, \ldots, c_n\} \neq \emptyset$; that is, I satisfies r viewed as implication in classical logic.

An interpretation I is then an answer set of P, if I is a $\subseteq$-minimal model of the program $P^I = \{r \in P \mid I$ satisfies neither (ii) nor $(iii)\}$. Intuitively, I must result by applying the rules r whose bodies "fire" w.r.t. I starting from facts.

The semantics of programs with variables is defined in terms of their groundings (uniform replacement of variables in rules with all possible ground terms).

ASP features further constructs such as choice rules (which allows to select among alternatives under cardinality bounds) and weak constraints (i.e., soft constraints expressing costs for an objective function that is minimised); notably, the latter allow for modeling numeric uncertainty and to single out the most likely from answer sets of a program or a range of most likely answer sets. For more details on ASP, we refer to (Brewka, Eiter, and Truszczynski 2011; Calimeri et al. 2020).

[2]We use clingo (v. 5.6.2) from https://potassco.org/.

3.2 Zero-Shot VQA for the GQA Dataset

We next introduce our system GS-VQA for zero-shot VQA on the GQA dataset (Hudson and Manning 2019). It does not require any training but relies on foundation models for processing the visual scene and ASP for deducing answers.

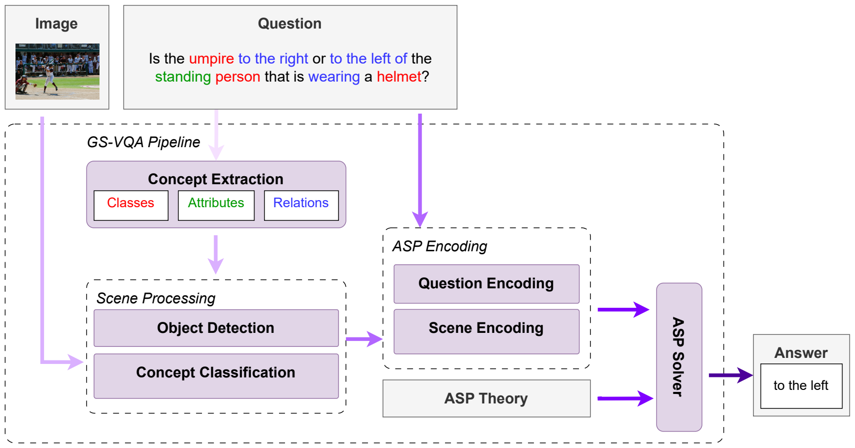
The GQA dataset. We use the state-of-the-art GQA dataset, which has been widely adopted in the recent literature (Amizadeh et al. 2020; Surís, Menon, and Vondrick 2023; Liang et al. 2020; Li et al. 2023). It contains over 22M open and binary questions that are complex in structure, involve a wide variety of reasoning skills, and have a large number (1 878) of possible answers. The questions cover more than 100 000 images from the Visual Genome dataset (Krishna et al. 2017) that present real-world scenes with a wide variety of object classes, attributes, and relations. GQA comes with two types of supplementary data that greatly aid in the development of our pipeline: First, each natural-language question from the test split comes with a functional representation of its required reasoning steps. Second, a Visual Genome scene graph is provided for every image in the test split of the dataset, which allows us to verify soundness of our ASP encoding under perfect visual information.

The GS-VQA pipeline. Our system, shown in Fig 2a, resembles other modular neurosymbolic models consisting of three distinguishable modules for language, vision, and reasoning. As mentioned, one of the advantages of using GQA is that we already have a functional representation that we can use for every question. Neurosymbolic systems have already shown that even classic neural networks such as the LSTM (Hochreiter and Schmidhuber 1997) are able to properly translate natural language into these representations. Moreover, LLMs can now be used for the task of semantic parsing, which may generalise much better than trained neural networks for specific datasets. We assume that the representation is given, and thus we only implement a script that translates it to our ASP representation. An example of this is shown in Fig. 2b under "Question Encoding". The predicates are linked by the numbers in the first argument, which represent steps to compose questions.

We next present our reasoning module as a fixed logical program at first, but we show in a later section how to use LLMs to extend an initial incomplete logical theory. Afterwards, we describe the vision module, where we identify the task of scene-graph generation and use our question-driven approach to solve it.

The ASP theory. The GS-VQA pipeline constructs symbolic encodings of both the input question and the input image, which we call *question encoding* and *scene encoding*, respectively. Similar to a related approach (Eiter et al. 2022a), we use ASP as the symbolic formalism for these encodings. This provides us not only with a mature ecosystem of tooling and solvers, but more importantly, allows us to capture the uncertainty in the class, attribute, and relation predictions of the scene-processing component.

The ASP theory consists of a set of rules that—in contrast to the question and scene encoding—do not change from question to question and encode the semantics of the



(a) An overview over the full GS-VQA pipeline.

```
%Scene Encoding
object(o1).
has_obj_weight(o1, 1971).
has_attr(o1, class, person).
has_attr(o1, name, baseball_player).
{has_attr(o1, pose, standing)}.
...

%Question Encoding
scene(0).
select(1, 0, helmet).
relate(2, 1, person, wearing, subject).
filter_any(3, 2, standing).
choose_rel(4, 3, umpire, to_the_left_of,
          to_the_right_of, subject).
end(4).

%ASP Theory
state(TO,ID) :- select(TO, TI, C),
               state(TI, ID),
               has_attr(ID, _, C).
bool(TO,no)  :- exist(TO, TI),
               not bool(TO,yes).
...
```

(b) Examples of excerpts from the question encoding, scene encoding, and the ASP theory.

Figure 2: GS-VQA takes the image and question as input and uses a question-driven approach to generate a partial scene graph. We generate an ASP representation of both partial scene graph and question. These are then solved along an ASP theory to derive the correct answer.

reasoning operations that can appear as part of the question encoding.

Revisiting the rules in Figure 2b, there T_i and T_o are variables representing input/output step references, ID represents an object id, C a class, A an attribute category, V an attribute value, and R a relation. The entire theory, described as part of the online repository, is solved alongside both scene and question encodings to produce an answer.

Question-driven scene-graph generation. The zero-shot nature of the pipeline presents a challenge for constructing a complete scene graph of visually complex scenes: due to the general-purpose nature, inference with the VLMs that GS-VQA uses for scene graph generation is far more resource intensive than adhoc trained models. As such, constructing a full scene graph in which likelihoods for all possible classes, attributes, and relations in the dataset are present for every object detected in the scene is untenable within a time bound that a human user might find acceptable.

To resolve this issue, the pipeline resorts to *"question-driven" partial scene-graph extraction*, where only information is extracted from the scene that is relevant for answering the question at hand. To this end, the *concept-extraction component* determines which object classes, attributes, and relations are relevant from the semantic representation of the input question. Conceptually, it takes a natural language question and outputs a tuple (C, A, R), where C is a set of classes, A is a set of attribute categories, and R is a set of relations. We use the functional representation that comes with every question in GQA to accomplish this. For the example question in Fig. 2a, the tuple is $(\{\texttt{helmet}, \texttt{umpire}, \texttt{person}\}, \{\texttt{pose}\}, \{\texttt{wearing}, \texttt{to\_the\_left\_of}, \texttt{to\_the\_right\_of}\})$.

Using the (C, A, R) output tuple from the concept-extraction component, the *scene-processing component* has the task of extracting a question-driven partial scene graph. We represent the graph as a list $O = [o_1, \ldots, o_n]$ of objects $o_i = (id, s, B, c, A_o, R_o)$, each having a unique identifier id, a score s between 0 and 1 denoting the scene-processing confidence in the object detection, a bounding box B, a class c, and sets A_o and B_o of attribute and relation likelihoods, respectively. The class c is either in C or a sub-class of maximal specificity of a class c' in C. The latter ensures that an object cannot, e.g., be detected as just a "person", but must be detected as a maximally specific class like "baseball player". It also means that the scene graph contains only objects of classes that are deemed relevant to answering the question (hence the description as a "partial" scene graph). The set A_o contains, for each possible value v of each attribute category a in A, a likelihood between 0 and 1 that v applies to a for the object. Finally, R_o contains, for each relation $r \in R$ and each other detected object $o_j \neq o_i$, a likelihood between 0 and 1 that $r(o_i, o_j)$ applies. The scene graph $O = [o_1, o_2, \ldots, o_n]$ generated by when processing the scene is easily translated using a script into our ASP representation. Fig. 2b shows an excerpt of such representation under "Scene Encoding."

Evaluation of GS-VQA on GQA. We evaluate the GS-VQA pipeline on the balanced test-dev set of GQA, which contains 12 578 questions. The evaluation uses the larger ViT-L/14 variant of OWL-ViT (Minderer et al. 2022) for object detection and the smaller ViT-B/32 variant of CLIP (Radford et al. 2021) for concept classification.

In Table 1, we present a comparison between our system and other models on GQA. We classify them as *fine-tuned models*, which require training, and *zero-shot*, which do not. We further classify zero-shot models into four sub-categories:

- **End-to-End**: End-to-end systems are those that rely solely on neural networks for computing the answer.
- **Neurosymbolic**: Neurosymbolic systems like ours are those that combine both neural networks for parsing data and symbolic execution to calculate the answers.
- **Question-Symbolic**: Such methods extract a symbolic representation from only the input question, usually in the

	Model	Category	Accuracy
Fine-tuned	MAC	End-to-End	55.4%
	LXMERT	End-to-End	60.0%
	CRF	End-to-End	**72.1%**
Zero-shot	FewVLM	End-to-End	29.3%
	GS-VQA (ours)	Neurosymbolic	39.5%
	PnP-VQA	Semi-Symbolic	42.3%
	BLIP-2	End-to-End	44.7%
	ViperGPT	Question-Symbolic	48.1%
	CodeVQA	Question-Symbolic	**49.0%**

Table 1: Comparison of GS-VQA's accuracy on the test-dev set of GQA with other state-of-the-art approaches for VQA.

form of some programmatic specification of the reasoning steps needed to arrive at the question's answer.

- **Semi-Symbolic**: PnP-VQA (Tiong et al. 2022) extracts a symbolic representation of the image but does not perform its reasoning purely symbolically, hence we classify this method as semi-symbolic.

GS-VQA answers 39.5% of all questions of GQA's test-dev set correctly, with the current best zero-shot VQA model, CodeVQA (Subramanian et al. 2023), obtaining an accuracy of 49.0%, followed closely by ViperGPT (Surís, Menon, and Vondrick 2023). However, CodeVQA and ViperGPT translate input questions into Python code that may contain queries to another VQA model; thus the performance of the latter, which is PnP-VQA (Tiong et al. 2022) resp. BLIP-2 (Li et al. 2023), should be considered as their baseline.

While our model does not improve on the the state-of-the-art in terms of accuracy, it uses ASP for deducing answers which comes with a range of advantages with the most important ones being explainability and non-determinism to deal with ambiguous inputs (Eiter et al. 2022b). More specifically, ASP allows for transparent execution, verifiability, and transferability, as well as the possibility of reasoning under different modalities, e.g. abduction on instances for computing contrastive explanations (Eiter et al. 2023).

The components used for our system were run locally, in contrast to others such as ViperGPT, which allows for easy testing of our solution. Furthermore, as our model is modular, components can be easily replaced by better ones, and it can be expected that its performance improves when the underlying foundation models get better.

3.3 NSVQASP for the CLEVR Dataset

The second dataset we consider is CLEVR (Johnson et al. 2017), which uses synthetic scenes but challenges VQA systems with more complex compositional questions. While our approach for GQA is a novel contribution of this paper, we use an existing VQA system for CLEVR, namely NSVQASP (Eiter et al. 2022c), that we have extended by an explanation component in recent work (Eiter et al. 2023). We revisit its basic functionality in the remainder of this section.

The CLEVR dataset. CLEVR was designed to test VQA system with compositional questions that involve making several reasoning steps to derive the correct answer. The dataset contains synthetically generated images with different objects in it. These objects vary in their shape (cube, cylinder, sphere), colour (brown, blue, cyan, gray, green, purple, red, yellow), size (big, small), and material (metal, rubber).

CLEVR questions require, e.g., identifying objects, counting, filtering for attributes, comparing attributes, and spatial reasoning. They are formulated in natural language, but, as for GQA, a functional representation is provided as well that can be directly parsed into ASP facts. For illustration, the question "How many large things are either cyan metallic cylinders or yellow blocks?" then becomes

```
end(8). count(8,7). filter_large(7,6). union(6,3,5).
filter_cylinder(3,2). filter_cyan(2,1).
filter_metal(1,0).
filter_cube(5,4). filter_yellow(4,0). scene(0).
```

which encodes an execution tree of operations to derive the answer, where indices refer to output (first argument) and input (remaining arguments) of the respective operations.

The NSVQASP pipeline for CLEVR. The architecture of NSVQASP is similar to the one of GS-VQA. In fact, the latter system was inspired by the design of the former. The scenes in CLEVR are however less complex, and we use the popular object-detection framework YOLOv5[3] to identify all objects in the image, their attributes, and further bounding-box information.

As for GS-VQA, ASP is used for the reasoning module. More specifically, we use a uniform ASP encoding that describes how to derive the answer for a question and a scene, both given as translations into ASP facts, by step-wise evaluating the execution tree of operations from the question encoding. More information, code, and the full ASP encoding can be found online.[4]

4 Knowledge Distillation Method

Our VQA approaches use a hand-coded ASP theory that correctly computes the answer to any question in the dataset, given a correct representation of both the question and the image. The ASP encoding is constructed around a fixed dataset. If the dataset is extended, rules need to be modified or added to handle new examples. In general, these new rules must be crafted by a human; here, an LLM based system can come to aid and provide automated support. To this end, we aim for a reasoning module that manages the theory by being able to recognise which examples it can handle and which it cannot, and in the latter case, by adding rules in a way such that the example can be solved. We propose a method of *declarative knowledge distillation*, where the model we distill from is an LLM, and the knowledge that is distilled from it is represented in ASP rules.

4.1 Preprompt

We first present a preprompt that instructs the LLM to only return ASP rules that extend an initial theory Init. Theory

[3]https://ultralytics.com/yolov5.

[4]https://github.com/pudumagico/nsvqasp.

Init is a partial encoding for the task at hand that we want to extend. Our preprompt consists of several components:

1. **Introduction**: We present the setting of VQA and clarify that we have already parsed both scene and question into correct representations.
2. **Language Syntax**: We describe the syntax of the language we use to represent questions and scenes, in our case ASP.
3. **Scene and Question Explanation**: We explain the representation that is used for the scene graph and questions and give a examples.
4. **Answer Format**: We describe the format of the answers to the questions.
5. **Initial Theory**: We present the initial theory Init that must be updated as necessary.
6. **Task Explanation**: Finally, we explain the input the LLM will receive from now on and the expected response (see Listing 1).

The input after the preprompt are question/scene/answer tuples (Q, S, A) in the language of ASP. The task is the following: when our system recognises that the current theory cannot handle the instance presented, then the LLM is prompted to add rules such that the correct answer can be derived. The expected LLM output is a list of ASP rules.

```
Your task is to keep the ASP theory
updated with rules that allows us
to answer questions.
We provide an initial theory that
can handle some instances.
The prompt input will consist of one or
more questions in the ASP representation.

Strictly follow these guidelines:
1. Only output the new ASP Rules.
2. Do not add facts as rules.
3. New rules should be as general
as possible, i.e., have a low number of
constants and high number of variables.
4. Do not output any natural language.
```

Listing 1: Excerpt from the Task Explanation part of our preprompt.

4.2 Rule Distillation Algorithm

With the task explained to the LLM by the preprompt, we can start to present examples from the dataset. For each question/scene/answer tuple $p = (Q, S, A)$, we do the following steps (and repeat them for the same example at most r times if not successful):

1. **Prompting:** We prompt the LLM with p as additional input, and we get as a response R.
2. **Solving:** We concatenate R with the initial theory to get theory Res. Then, we run an ASP solver on theory Res alongside the instance pair (Q, S).
   (a) **Syntax Check:** If we get a syntax error, we pass the error message to the LLM and prompt it to revise R accordingly. We try this at most m times.
   (b) **Semantic Check:** We check whether the answer we get from the solver is correct, i.e., coincides with A. If not, we pass the actual answer and the expected answer to the LLM and task it to update R; we try this at most m times.
3. **Regression Testing:** To avoid that adding rules to the theory renders past examples incorrect, we test Res on all previously seen examples. Only if all tests pass we keep Res, otherwise we disregard the extension R.

The algorithm has two parameters, r is the number of retries per example, and m is the number of retries for mending incorrect rules (defaults are $r = m = 1$). Mending rules is potentially expensive as it requires more calls to the LLM; it can be turned off if preferred.

Example Sampling Strategies. VQA datasets contain millions of instances and going blindly through them can make the distillation process ineffective. Choosing a small but representative sample can yield better results faster. We propose two strategies to group instances together:

- ***Predicate Count***: We group all the instances by the number of predicate occurrences that appear in the ASP question representation. For this, we create a dictionary whose keys are the numbers of predicates and the contents are questions with such length.
- ***Predicate Relevance***: Here, we group examples based on the predicates that appear in the question representation. We first create a dictionary whose keys are all the predicates that appear in any question representation. Then we populate the dictionary with questions where the key predicate appears in the question representation.

We then sample examples from the group created for the chosen strategy.

Batch Optimization. We present prompt instances one by one which results in one LLM call for each example. This is not vert efficient and can misguide the LLM into implementing rules that only solve that particular example, yet our aim is to have general rules that can handle a considerable portion of the dataset. Considering that LLMs have increasingly bigger context sizes, we also investigate the option to present prompt instances in batches, where each batch contains up to b singular instances. We observe that the scene representation is usually much larger than the question representation and only contains a small number of predicates, and the variance comes more from the constants in them. Considering this, whenever we use our batch strategy, we present the prompt instance $p = \{Q_1, \dots, Q_b\}$, where we include only the question representation in the Q_i and drop the scene representations. Now the LLM must create rules general enough to pass the semantic check for all the examples in the batch. There is an expected trade off between batch size and accuracy, however: with large b, the semantic check and regression testing might be too strict for the LLM to produce rules that correctly cover all examples in one shot.

5 Knowledge Distillation Experiments

We conduct a series of experiments to evaluate our knowledge distillation method on GQA and CLEVR to answer the following research questions:

(R1) Given an ASP-based VQA system and a VQA task, can our approach extend the ASP reasoning component to deal with questions that require reasoning operations/steps that are not yet implemented?

(R2) What LLMs are suitable for our method?

(R3) Can our method cope with the challenge of removing, either randomly or in a more controlled way, increasingly large parts from an initial complete theory?

(R4) What are the effects of the more resource-friendly batch processing variant and mending switched off for our method?

The evaluation platform is a workstation with an Intel Core i7-12700K CPU, 32GB of RAM, and an NVIDIA GeForce RTX 3080 Ti GPU with 12GB of video memory. All experiments were run 5 times. We include average accuracy, standard deviation, as well as the minimum and maximum value obtained. For reproducibility, we logged all our parameters, random seeds, and input prompts.

LLM selection. Before going into the details of our knowledge distillation experiments, we describe how we selected LLMs for further consideration.

We ran the ASP programs of our VQA systems on examples from GQA and CLEVR. Then we selected ca. $45k$ examples where the answer was correctly calculated for GQA, and we then divided this set into a training and a test suite of ca. $35k$ and $10k$ instances, respectively. For CLEVR, we selected $50k$ examples and split them into $35k$ for training and $15k$ for testing.[5]

For the selected examples, we ran preliminary experiments on a large array of LLMs, both local and online API-based ones. Local models, like GPT4ALL (Anand et al. 2023) "wizardlm-13b", showed very poor performance when prompted to produce ASP rules. Some API models, like "mistral-medium", where too slow in coming up with a response for our purposes and were thus excluded.

We selected the three top performers, which were GPT-4 (OpenAI 2023), GPT-3.5, and Mistral (Jiang et al. 2023); more specifically, the models used are "gpt-4-1106-preview", "gpt-3.5-turbo-1106" and "mistral-small", respectively.

Experiment 1. Our first experiment is designed to address **(R1)** and **(R2)**. We start with the complete ASP encoding for each VQA approach presented in Section 3 and remove all rules that mention a particular predicate P that occurs in some question representation. We then prompt the LLM with examples that contain P in their question representation to repair the theory. This simulates a scenario where we the initial ASP theory is not yet capable to address all requirements of the VQA task and needs to be extended.

We use the predicate ordering strategy with a sample size of $k = 10$. As we only sample from questions where P is present, the number of examples used for each run is k.

The results for GQA and CLEVR are shown in Table 2. Column $\mathrm{Init} \setminus P$ shows the accuracy of the initial ASP encoding with a predicate P removed. The drop in accuracy varies depending on the number of questions affected and the role of the predicate that was removed (e.g., "select" results in a deeper drop). The other columns show the performance of the considered LLMs; here and in other tables, "—" means no improvement. By a large margin, GPT-4 is the most suitable LLM for this task for both datasets. With GPT-4, we could obtain rules that improve over Init for every predicate P. For the other models, there the quality of suggested repairs differs largely. GPT-3.5 is also able to mend Init but to a lesser degree than GPT-4, as it does not produce any new rules for some predicates. When it does, the accuracy is lower than the one of GPT-4. The Mistral model performs comparable to GPT-3.5 when it finds correct rules. This, however, happened less often and the gain in accuracy then is minimal.

Experiment 2. This experiment is designed to address **(R3)**. It is a challenge experiment where we took the complete ASP encodings and then removed a random sample of s percent of the rules from it. For GQA, the ASP theory consists of 60 rules; for CLEVR, the program comprises 72 rules. After rendering this theory incomplete, we prompted the LLM to restore it. We tested this setting for $s = \{10, 20, 50\}$ on the two best models from the first experiment, i.e., GPT-4 and GPT-3.5. We used the length ordering strategy with $k = 2$, which yields 20 examples per run, and $r = 1$ retries.

The results are given in Table 3. For GQA, GPT-4 is quite capable of producing ASP rules that improve the accuracy of the initial theory. On the other hand, GPT-3.5 starts to falter for GQA, as the gain in accuracy is dramatically reduced for $s = 20, 50$, and no gain is reported when $s = 10$. GPT-3.5 outperforms GPT-4 in few cases for CLEVR, which can be explained by the variance of the initial theory when randomly removing rules.

Experiment 3. This experiment tests our batch optimization approach and aims at answering **(R4)**. We consider batch sizes $b = 2, 5, 10$ and the length ordering strategy with $k = 11$, which yields 100 examples per run. We use $r = 3$ retries for examples and $m = 2$ retries for mending.

Like the previous experiment, this one is also based on the complete ASP theory, but we now selected by hand parts of it to generate three different initial theories. By their size, we call them Light, Medium and Heavy. Theory Light consists of five rules only that tell our system how to start and end the processing of a question, but nothing else. We added more rules to Light to generate Medium and even further rules for Heavy. For this experiment, we focused only on the most capable model, namely GPT-4.

Table 4 shows that GPT-4 still retains its capability to produce meaningful ASP rules even in this very challenging scenario. We can observe that small theories do not work well with bigger batches as they do not provide enough background for the LLM to produce suitable rules. When the size of the initial theory increases, the LLM can handle bigger batches and produces rules of higher quality.

Ablation Experiment. We finally study the effect of the mending step in our method when the LLM suggests rules that are either syntactically or semantically not correct. As mentioned in the previous section, these checks can be turned off, which results in fewer LLM calls.

[5]The expression "training set" refers here to the examples used when running the knowledge distillation method.

P	Init $\setminus P$	GPT-4	GPT-3.5	Mistral
query	48.84	97.67 ± 18.05 $(89.16, 98.92)$	70.02 ± 19.36 $(48.84, 85.53)$	—
exist	86.36	99.75 ± 00.50 $(98.86, 99.98)$	87.65 ± 02.41 $(86.36, 91.95)$	89.68 ± 04.20 $(86.36, 95.66)$
or	92.18	100.0 ± 00.00 $(100.0, 100.0)$	93.03 ± 01.90 $(92.18, 96.44)$	93.20 ± 01.66 $(92.18, 96.02)$
filter	81.60	98.21 ± 00.40 $(97.49, 98.40)$	83.15 ± 03.47 $(81.60, 89.37)$	81.70 ± 00.24 $(81.60, 82.14)$
choose_attr	92.12	95.98 ± 05.37 $(88.73, 99.83)$	93.73 ± 01.36 $(92.31, 95.83)$	92.12 ± 00.01 $(92.12, 92.15)$
verify_rel	93.72	98.60 ± 01.11 $(96.73, 99.43)$	—	—
select	9.53	99.94 ± 00.07 $(99.87, 100.0)$	27.42 ± 40.01 $(9.53, 99.01)$	—
negate	98.59	98.54 ± 00.20 $(98.59, 98.74)$	—	—
relate	56.89	69.38 ± 12.50 $(56.89, 85.25)$	—	—
two_different	98.94	100.0 ± 00.00 $(100.0, 100.0)$	99.39 ± 00.55 $(98.94, 100.0)$	—
two_same	98.83	99.99 ± 00.00 $(99.99, 100.0)$	99.05 ± 00.53 $(98.83, 100.0)$	—

(a) Results for GQA.

P	Init $\setminus P$	GPT-4	GPT-3.5	Mistral
exist	79.63	99.48 ± 00.70 $(98.72, 100.0)$	87.82 ± 11.11 $(98.72, 100.0)$	80.21 ± 06.36 $(72.16, 90.02)$
unique	29.19	97.67 ± 18.05 $(89.16, 98.92)$	—	—
count	98.01	99.60 ± 00.88 $(98.01, 100.0)$	98.01 ± 01.12 $(98.01, 98.40)$	—
equal_integer	96.61	99.92 ± 00.17 $(99.61, 100.0)$	97.80 ± 01.26 $(96.61, 99.60)$	—
and	93.67	100.0 ± 00.00 $(100.0, 100.0)$	97.46 ± 03.46 $(93.67, 100.0)$	—
relate_left	84.73	100.0 ± 00.00 $(100.0, 100.0)$	96.18 ± 07.63 $(84.73, 100.0)$	94.14 ± 08.02 $(84.73, 100.0)$
filter_large	68.54	100.0 ± 00.00 $(100.0, 100.0)$	87.41 ± 17.23 $(68.54, 100.0)$	81.12 ± 17.23 $(68.54, 100.0)$
query_shape	72.23	100.0 ± 00.00 $(100.0, 100.0)$	100.0 ± 00.00 $(100.0, 100.0)$	94.44 ± 12.43 $(72.23, 100.0)$
same_color	94.79	99.36 ± 00.87 $(98.41, 100.0)$	100.0 ± 00.00 $(100.0, 100.0)$	97.07 ± 02.70 $(94.79, 100.0)$

(b) Results for CLEVR.

Table 2: Results for the knowledge distillation method when attempting to restore Init after all rules that mention a predicate P are removed.

$s(\%)$	GPT-4	GPT-3.5
10	91.32 ± 07.33 $(80.07, 97.10)$	—
20	76.89 ± 14.19 $(54.83, 88.69)$	31.98 ± 26.87 $(00.00, 69.31)$
50	25.60 ± 22.03 $(00.00, 51.56)$	06.29 ± 07.54 $(00.00, 18.30)$

(a) Results for GQA.

$s(\%)$	GPT-4	GPT-3.5
10	90.85 ± 09.33 $(74.94, 97.57)$	—
20	35.95 ± 12.90 $(24.18, 57.92)$	38.22 ± 14.16 $(19.37, 55.37)$
50	11.51 ± 12.32 $(00.00, 27.65)$	29.26 ± 27.94 $(00.00, 60.22)$

(b) Results for CLEVR.

Table 3: Knowledge distillation results when attempting to restore a complete ASP theory after a percentage s of rules is randomly removed.

b	Light	Medium	Heavy
1	$56, 26 \pm 10.23$ $(34.54, 61.28)$	81.45 ± 05.07 $(76.86, 87.91)$	83.85 ± 02.49 $(81.38, 87.77)$
2	32.71 ± 04.31 $(25.72, 43.15)$	79.83 ± 03.42 $(75.11, 83.03)$	74.32 ± 02.91 $(75.86, 80.54)$
5	16.62 ± 05.28 $(10.51, 17.59)$	69.68 ± 31.12 $(24.18, 82.19)$	84.25 ± 04.59 $(78.93, 89.48)$
10	—	15.38 ± 12.30 $(11.62, 31.75)$	84.75 ± 04.20 $(80.64, 90.85)$

(a) Results for GQA.

b	Light	Medium	Heavy
1	84.68 ± 26.42 $(38.23, 100.0)$	86.97 ± 04.35 $(83.89, 90.05)$	95.40 ± 03.83 $(91.25, 98.81)$
2	75.4 ± 33.78 $(27.84, 99.88)$	18.68 ± 04.50 $(15.67, 26.09)$	88.51 ± 04.46 $(83.37, 91.25)$
5	17.06 ± 29.55 $(00.00, 51.19)$	17.79 ± 03.00 $(15.67, 19.92)$	94.39 ± 03.71 $(91.33, 98.52)$
10	—	—	89.88 ± 09.04 $(77.68, 98.81)$

(b) Results for CLEVR.

Table 4: Results for the knowledge distillation method when using batch sizes b and the different initial theories Light, Medium and Heavy.

We only present the results for the previous three experiments on GQA in Tables 5–7; for CLEVR, they look similar. As one would expect, the results are worse most of the time, although not always, when mending is disabled. The rate of improvement with mending ranges from a couple of percentage points to up to ca. 20% (row for “select”).

P	Init \ P	GPT-4	GPT-3.5	Mistral
query	48.84	99.02 ± 00.04 $(99.02, 99.03)$	55.90 ± 15.80 $(48.84, 84.17)$	—
exist	86.36	99.09 ± 01.76 $(95.94, 99.97)$	87.83 ± 03.29 $(86.36, 93.73)$	86.60 ± 0.05 $(86.36, 87.57)$
or	92.18	100.0 ± 00.00 $(100.0, 100.0)$	93.03 ± 01.90 $(92.18, 96.44)$	94.41 ± 3.35 $(92.18, 99.73)$
filter	81.60	98.56 ± 00.58 $(98.47, 98.63)$	82.38 ± 0.078 $(81.92, 82.50)$	84.99 ± 7.59 $(81.60, 98.59)$
choose_attr	92.12	98.65 ± 02.65 $(93.91, 99.88)$	95.16 ± 04.26 $(92.03, 99.84)$	92.08 ± 0.06 $(91.98, 92.12)$
verify_rel	93.72	95.74 ± 03.49 $(93.72, 99.08)$	94.30 ± 01.17 $(93.72, 96.06)$	—
select	9.53	81.69 ± 36.81 $(9.53, 100.0)$	28.94 ± 39.82 $(9.53, 100.0)$	—
negate	98.59	98.72 ± 00.02 $(98.59, 99.12)$	—	—
relate	56.89	58.02 ± 02.19 $(57.54, 61.91)$	57.29 ± 00.09 $(56.89, 58.91)$	—
two_different	98.94	100.0 ± 00.00 $(100.0, 100.0)$	—	—
two_same	98.83	99.60 ± 00.02 $(99.50, 100.0)$	99.09 ± 00.05 $(98.83, 100.0)$	—

Table 5: Ablation study for GQA: Results when attempting to restore Init after removing rules for P without mending step.

$s(\%)$	GPT-4	GPT-3.5
10	89.25 ± 11.89 $(72.65, 99.15)$	—
20	69.44 ± 31.15 $(17.69, 96.31)$	3.68 ± 05.03 $(00.00, 09.26)$
50	30.36 ± 13.10 $(18.35, 51.14)$	06.03 ± 13.49 $(00.00, 30.18)$

Table 6: Ablation study for GQA: Results when attempting to restore the complete theory T after a percentage s of rules is randomly removed without mending step.

b	Light	Medium	Heavy
1	$51,43 \pm 08.56$ $(42.72, 59.85)$	80.41 ± 05.34 $(74.54, 85.01)$	76.70 ± 00.76 $(75.82, 77.19)$
2	27.06 ± 09.60 $(21.09, 38.14)$	77.70 ± 05.42 $(75.25, 83.92)$	77.60 ± 03.85 $(73.41, 81.00)$
5	15.23 ± 02.28 $(12.60, 16.49)$	60.16 ± 34.17 $(21.08, 84.45)$	86.28 ± 10.84 $(27.93, 94.71)$
10	—	19.55 ± 07.41 $(13.03, 27.62)$	73.93 ± 06.50 $(66.42, 77.69)$

Table 7: Ablation study for GQA: Results when using batch sizes b and different initial theories without mending step.

Discussion. We turn back to our research questions from the beginning of this section.

(R1) Experiment 1 shows that LLMs are capable of completing an ASP program that has all rules for a single operation removed. This reflects the case when a dataset is extended with new examples that require reasoning operations that are not yet encoded.

(R2) Regarding the suitability of different LLMs for declarative knowledge distillation, we conclude that only grand-scale LLMs, with GPT-4 currently the market leader, are able to tackle this problem effectively. Arguably, the LLMs that we used have more knowledge about mainstream programming languages such as Python than logical programming languages. It will be interesting so see whether small, self-hosted language models will eventually catch up in the future.

(R3) When challenged with restoring increasingly large parts of an ASP theory, the current approach reaches its limits. Only the most powerful model we used is still able to produce sound ASP rules.

(R4) Our experiments on batch processing and the ablation study helped to illuminate the trade off between performance and reducing the number LLM calls. In conclusion, better performance can be bought by using more prompts, which can be expensive if a subscription-based LLM is used.

6 Conclusion

We have presented an approach for declarative knowledge distillation using LLMs to find rules that extend the reasoning component of a VQA system to extend its capabilities. This process uses examples from a dataset as guidance and relies entirely on prompting without the need to train or fine-tune the used language models. The benefit of using logic-based methods in combination with foundation models is that we obtain systems that behave in an interpretable and verifiable way to ensure correct reasoning.

We have evaluated our approach on two VQA datasets: CLEVR (Johnson et al. 2017), consisting of synthetic scenes and compositional questions, and GQA (Hudson and Manning 2019), with real-world images and questions that require to process complex scene graphs. For CLEVR, we used a recently developed VQA system (Eiter et al. 2023) while for GQA, we have introduced a novel modular zero-shot neurosymbolic system that uses a question-driven approach to obtain partial scene graphs relevant to answering a question. Notably, by employing logical rules it facilitates developing advanced reasoning capabilities, including explainability.

Our knowledge distillation method shows promise for automatising the process ASP modelling for VQA, a complex scenario that requires understanding of intricate representations. This gives also inspiration for a future use case as a programming assistant, similar to GitHub's copilot (Dakhel et al. 2023), where the system helps the user by suggesting possible paths forward. Another direction for future work is to investigate ways to balance performance of the distillation approach and the number of LLM calls. Enhancing the LLM approach with concepts from ILP and a possible combination would also be interesting to explore.

References

Amizadeh, S.; Palangi, H.; Polozov, A.; Huang, Y.; and Koishida, K. 2020. Neuro-Symbolic Visual Reasoning: Disentangling "Visual" from "Reasoning". In *Proceedings of the 37th International Conference on Machine Learning (ICML)*, volume 119 of *Proceedings of Machine Learning Research*, 279–290. PMLR.

Anand, Y.; Nussbaum, Z.; Treat, A.; Miller, A.; Guo, R.; Schmidt, B.; Community, G.; Duderstadt, B.; and Mulyar, A. 2023. Gpt4all: An ecosystem of open source compressed language models. *CoRR* abs/2311.04931.

Antol, S.; Agrawal, A.; Lu, J.; Mitchell, M.; Batra, D.; Zitnick, C. L.; and Parikh, D. 2015. VQA: Visual Question Answering. In *2015 IEEE International Conference on Computer Vision(ICCV)*, 2425–2433. IEEE Computer Society.

Barra, S.; Bisogni, C.; Marsico, M. D.; and Ricciardi, S. 2021. Visual question answering: Which investigated applications? *Pattern Recognit. Lett.* 151:325–331.

Bauer, J. J.; Eiter, T.; Ruiz, N. H.; and Oetsch, J. 2023. Neuro-Symbolic Visual Graph Question Answering with LLMs for Language Parsing. In *Workshop on Trends and Applications of Answer Set Programming (TAASP 2023)*.

Brewka, G.; Eiter, T.; and Truszczynski, M. 2011. Answer set programming at a glance. *Commun. ACM* 54(12):92–103.

Calimeri, F.; Faber, W.; Gebser, M.; Ianni, G.; Kaminski, R.; Krennwallner, T.; Leone, N.; Maratea, M.; Ricca, F.; and Schaub, T. 2020. ASP-core-2 input language format. *Theory and Practice of Logic Programming* 20(2):294–309.

Cropper, A.; Dumancic, S.; Evans, R.; and Muggleton, S. H. 2022. Inductive logic programming at 30. *Mach. Learn.* 111(1):147–172.

Dakhel, A. M.; Majdinasab, V.; Nikanjam, A.; Khomh, F.; Desmarais, M. C.; and Jiang, Z. M. J. 2023. Github copilot AI pair programmer: Asset or liability? *J. Syst. Softw.* 203:111734.

Eiter, T.; Higuera, N.; Oetsch, J.; and Pritz, M. 2022a. A Neuro-Symbolic ASP Pipeline for Visual Question Answering. *Theory and Practice of Logic Programming* 22(5):739–754.

Eiter, T.; Higuera, N.; Oetsch, J.; and Pritz, M. 2022b. A neuro-symbolic ASP pipeline for visual question answering. *Theory Pract. Log. Program.* 22(5):739–754.

Eiter, T.; Higuera, N.; Oetsch, J.; and Pritz, M. 2022c. A neuro-symbolic ASP pipeline for visual question answering. *Theory and Practice of Logic Programming* 22(5):739–754.

Eiter, T.; Geibinger, T.; Higuera, N.; and Oetsch, J. 2023. A logic-based approach to contrastive explainability for neurosymbolic visual question answering. In *Proceedings of the 32nd International Joint Conference on Artificial Intelligence (IJCAI 2023)*, 3668–3676. ijcai.org.

Gelfond, M., and Lifschitz, V. 1988. The stable model semantics for logic programming. In *Proceedings of the 5th International Conference and Symposium on Logic Programming*, 1070–1080. MIT Press.

Goyal, Y.; Khot, T.; Summers-Stay, D.; Batra, D.; and Parikh, D. 2017. Making the V in VQA Matter: Elevating the Role of Image Understanding in Visual Question Answering. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 6325–6334. IEEE Computer Society.

Gutmann, B., and Kersting, K. 2006. TildeCRF: Conditional random fields for logical sequences. In *Proceedings of the 17th European Conference on Machine Learning (ECML 2006)*, volume 4212 of *Lecture Notes in Computer Science*, 174–185. Springer.

Hadl, J. 2023. GS-VQA: Zero-shot neural-symbolic visual question answering with vision-language models. Master's thesis, Technische Universität Wien, Wien.

Hochreiter, S., and Schmidhuber, J. 1997. Long short-term memory. *Neural Comput.* 9(8):1735–1780.

Hudson, D. A., and Manning, C. D. 2019. GQA: A New Dataset for Real-World Visual Reasoning and Compositional Question Answering. In *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 6700–6709. Computer Vision Foundation / IEEE.

Ishay, A.; Yang, Z.; and Lee, J. 2023. Leveraging large language models to generate answer set programs. In *Proceedings of the 20th International Conference on Principles of Knowledge Representation and Reasoning, KR 2023, Rhodes, Greece, September 2-8, 2023*, 374–383.

Jiang, A. Q.; Sablayrolles, A.; Mensch, A.; Bamford, C.; Chaplot, D. S.; de las Casas, D.; Bressand, F.; Lengyel, G.; Lample, G.; Saulnier, L.; Lavaud, L. R.; Lachaux, M.-A.; Stock, P.; Scao, T. L.; Lavril, T.; Wang, T.; Lacroix, T.; and Sayed, W. E. 2023. Mistral 7b.

Johnson, J.; Hariharan, B.; van der Maaten, L.; Fei-Fei, L.; Zitnick, C. L.; and Girshick, R. B. 2017. CLEVR: A diagnostic dataset for compositional language and elementary visual reasoning. In *2017 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017*, 1988–1997. IEEE Computer Society.

Johnston, P.; Nogueira, K.; and Swingler, K. 2023. NS-IL: neuro-symbolic visual question answering using incrementally learnt, independent probabilistic models for small sample sizes. *IEEE Access* 11:141406–141420.

Krishna, R.; Zhu, Y.; Groth, O.; Johnson, J.; Hata, K.; Kravitz, J.; Chen, S.; Kalantidis, Y.; Li, L.; Shamma, D. A.; Bernstein, M. S.; and Fei-Fei, L. 2017. Visual Genome: Connecting Language and Vision Using Crowdsourced Dense Image Annotations. *International Journal of Computer Vision* 123(1):32–73.

Law, M.; Russo, A.; Bertino, E.; Broda, K.; and Lobo, J. 2020. Fastlas: Scalable inductive logic programming incorporating domain-specific optimisation criteria. In *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, The Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020, The Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, February 7-12, 2020*, 2877–2885. AAAI Press.

Law, M.; Russo, A.; and Broda, K. 2020. The ILASP system for inductive learning of answer set programs. *CoRR* abs/2005.00904.

Li, J.; Li, D.; Savarese, S.; and Hoi, S. 2023. BLIP-2: Bootstrapping Language-Image Pre-training with Frozen Image Encoders and Large Language Models.

Liang, W.; Niu, F.; Reganti, A.; Thattai, G.; and Tur, G. 2020. LRTA: A Transparent Neural-Symbolic Reasoning Framework with Modular Supervision for Visual Question Answering.

Lifschitz, V. 2019. *Answer Set Programming*. Springer.

Lin, Z.; Zhang, D.; Tao, Q.; Shi, D.; Haffari, G.; Wu, Q.; He, M.; and Ge, Z. 2023. Medical visual question answering: A survey. *Artif. Intell. Medicine* 143:102611.

Mao, J.; Gan, C.; Kohli, P.; Tenenbaum, J. B.; and Wu, J. 2019. The neuro-symbolic concept learner: Interpreting scenes, words, and sentences from natural supervision. In *Proceedings of the 7th International Conference on Learning Representations (ICLR 2019)*. OpenReview.net.

Minderer, M.; Gritsenko, A. A.; Stone, A.; Neumann, M.; Weissenborn, D.; Dosovitskiy, A.; Mahendran, A.; Arnab, A.; Dehghani, M.; Shen, Z.; Wang, X.; Zhai, X.; Kipf, T.; and Houlsby, N. 2022. Simple Open-Vocabulary Object Detection. In *Computer Vision - ECCV 2022 - 17th European Conference*, volume 13670 of *Lecture Notes in Computer Science*, 728–755. Springer.

Muggleton, S. H., and Raedt, L. D. 1994. Inductive logic programming: Theory and methods. *J. Log. Program.* 19/20:629–679.

Muggleton, S. H. 1991. Inductive logic programming. *New Gener. Comput.* 8(4):295–318.

OpenAI. 2023. GPT-4 technical report.

Radford, A.; Kim, J. W.; Hallacy, C.; Ramesh, A.; Goh, G.; Agarwal, S.; Sastry, G.; Askell, A.; Mishkin, P.; Clark, J.; Krueger, G.; and Sutskever, I. 2021. Learning Transferable Visual Models From Natural Language Supervision. In *Proceedings of the 38th International Conference on Machine Learning (ICML)*, volume 139 of *Proceedings of Machine Learning Research*, 8748–8763. PMLR.

Raedt, L. D., and Kersting, K. 2017. Statistical relational learning. In *Encyclopedia of Machine Learning and Data Mining*. Springer. 1177–1187.

Rajasekharan, A.; Zeng, Y.; Padalkar, P.; and Gupta, G. 2023. Reliable natural language understanding with large language models and answer set programming. In *Proceedings 39th International Conference on Logic Programming, ICLP 2023, Imperial College London, UK, 9th July 2023 - 15th July 2023*, volume 385 of *EPTCS*, 274–287.

Subramanian, S.; Narasimhan, M.; Khangaonkar, K.; Yang, K.; Nagrani, A.; Schmid, C.; Zeng, A.; Darrell, T.; and Klein, D. 2023. Modular Visual Question Answering via Code Generation. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, 747–761. Association for Computational Linguistics.

Surís, D.; Menon, S.; and Vondrick, C. 2023. ViperGPT: Visual Inference via Python Execution for Reasoning. *CoRR* abs/2303.08128.

Surís, D.; Menon, S.; and Vondrick, C. 2023. ViperGPT: Visual Inference via Python Execution for Reasoning.

Tiong, A. M. H.; Li, J.; Li, B.; Savarese, S.; and Hoi, S. C. H. 2022. Plug-and-Play VQA: Zero-shot VQA by Conjoining Large Pretrained Models with Zero Training. In *Findings of the Association for Computational Linguistics: EMNLP 2022*, 951–967. Association for Computational Linguistics.

Vaswani, A.; Shazeer, N.; Parmar, N.; Uszkoreit, J.; Jones, L.; Gomez, A. N.; Kaiser, L. u.; and Polosukhin, I. 2017. Attention is all you need. In *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc.

Wang, Z.; Yu, J.; Yu, A. W.; Dai, Z.; Tsvetkov, Y.; and Cao, Y. 2022. SimVLM: Simple Visual Language Model Pretraining with Weak Supervision.

Yang, Z.; Ishay, A.; and Lee, J. 2020. Neurasp: Embracing neural networks into answer set programming. In *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI 2020*, 1755–1762. ijcai.org.

Yi, K.; Wu, J.; Gan, C.; Torralba, A.; Kohli, P.; and Tenenbaum, J. 2018. Neural-Symbolic VQA: Disentangling Reasoning from Vision and Language Understanding. In *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018*, 1039–1050.

Zhao, W. X.; Zhou, K.; Li, J.; Tang, T.; Wang, X.; Hou, Y.; Min, Y.; Zhang, B.; Zhang, J.; Dong, Z.; Du, Y.; Yang, C.; Chen, Y.; Chen, Z.; Jiang, J.; Ren, R.; Li, Y.; Tang, X.; Liu, Z.; Liu, P.; Nie, J.; and Wen, J. 2023. A survey of large language models. *CoRR* abs/2303.18223.

Zhu, Z.; Xue, Y.; Chen, X.; Zhou, D.; Tang, J.; Schuurmans, D.; and Dai, H. 2023. Large language models can learn rules. *CoRR* abs/2310.07064.

A Supplementary Material

We provide this appendix as supplementary material together with an online repository with code, data, and logs at https://github.com/pudumagico/KR2024.

- **Appendix**: Contains additional information regarding GS-VQA presented in Section 3 (for all details, see (Hadl 2023)), our knowledge distillation method shown in Section 4, and the experiments from Section 5, based on a selection to provide an informative overview (full information is in the repository).
- **Code and Data Repository**: Contains the code to run GS-VQA and our knowledge distillation method on the GQA dataset. The data refers to the preprompts and theories used.
- **Logs Repository**: We provide the logs of our experiments for knowledge distillation as part of the online repository. For each of the three experiments made in Section 5, we present logs for the five runs per configuration. Each run has a folder containing all the updates made to the initial theory, a folder containing the instances used and a file that registers the random seed used, and all the arguments passed to the main function and accuracy scores for the run.

B GS-VQA

We present our full encoding for GQA and give detailed information on how we construct the question encoding and scene encodings. Then we give details about the implementation of our scene graph generation model.

B.1 ASP Theory

In Listing 2, we present the complete ASP theory introduced in Section 3.

Listing 2: The complete ASP theory used in GS-VQA

```
% ========== Scene Graph Definitions ==========
#defined is_attr/1.
#defined is_attr_value/2.
#defined object/1.
#defined has_obj_weight/2.
#defined has_attr/3.
#defined has_rel/3.

% ========== Base Operations ==========
% ---------- scene ----------
#defined scene/1.

state(TO,ID) :- scene(TO), object(ID).

% ---------- end ----------
#defined end/1.

ans(V) :- end(TO), attr_value(TO,V).
ans(A) :- end(TO), attr(TO,A).
ans(R) :- end(TO), rel(TO,R).
ans(B) :- end(TO), bool(TO,B).

% ---------- ans ----------
% At least one answer must be derivable
:- not ans(_).
#show ans/1.


% ========== Intermediary Operations ==========
% ---------- select ----------
#defined select/3.

state(TO,ID) :- select(TO, TI, CLASS), state(TI, ID), has_attr(ID, class, CLASS).

% ---------- filter ----------
#defined filter/4.

state(TO,ID) :- filter(TO, TI, ATTR, VALUE), state(TI, ID), has_attr(ID, ATTR, VALUE).

#defined filter_any/3.

state(TO,ID) :- filter_any(TO, TI, VALUE), state(TI, ID), has_attr(ID, _, VALUE).

% ---------- relate ----------
#defined relate/5.

state(TO, ID') :- relate(TO, TI, CLASS, REL, subject), state(TI, ID), has_attr(ID', class, CLASS), has_rel(ID', REL, ID).
state(TO, ID') :- relate(TO, TI, CLASS, REL, object), state(TI, ID), has_attr(ID', class, CLASS), has_rel(ID, REL, ID').

% relate_any
#defined relate_any/4.

state(TO, ID') :- relate_any(TO, TI, REL, subject), state(TI, ID), has_rel(ID', REL, ID).
state(TO, ID') :- relate_any(TO, TI, REL, object), state(TI, ID), has_rel(ID, REL, ID').

% relate_attr
#defined relate_attr/4.

state(TO, ID') :- relate_attr(TO, TI, CLASS, ATTR), state(TI, ID), has_attr(ID, ATTR, VALUE), has_attr(ID', class, CLASS), has_attr(ID', ATTR, VALUE), ID!=ID'.

% ---------- compare ----------
#defined compare/5.

state(TO,ID) :- compare(TO, TI0, TI1, VALUE, true), state(TI0, ID), state(TI1, ID'), has_attr(ID, _,
    ↪ VALUE), not has_attr(ID', _, VALUE
    ↪ ).
state(TO,ID') :- compare(TO, TI0, TI1,
    ↪ VALUE, true), state(TI0, ID),
    ↪ state(TI1, ID'), not has_attr(ID,
    ↪ _, VALUE), has_attr(ID', _, VALUE)
    ↪ .

state(TO,ID') :- compare(TO, TI0, TI1,
    ↪ VALUE, false), state(TI0, ID),
    ↪ state(TI1, ID'), has_attr(ID, _,
    ↪ VALUE), not has_attr(ID', _, VALUE
    ↪ ).
state(TO,ID) :- compare(TO, TI0, TI1,
    ↪ VALUE, false), state(TI0, ID),
    ↪ state(TI1, ID'), not has_attr(ID,
    ↪ _, VALUE), has_attr(ID', _, VALUE)
    ↪ .


% ========== Terminal Operations
    ↪ ==========
% ---------- query ----------
#defined query/3.

{ has_attr(ID, ATTR, VALUE) :
    ↪ is_attr_value(ATTR, VALUE)} = 1 :-
    ↪  query(TO, TI, ATTR), state(TI, ID
    ↪ ), ATTR != name, ATTR != class,
    ↪ ATTR != hposition, ATTR !=
    ↪ vposition.
attr_value(TO,VALUE) :- query(TO, TI,
    ↪ ATTR), state(TI, ID), has_attr(ID,
    ↪  ATTR, VALUE).

% ---------- verify ----------
% verify_rel
#defined verify_rel/5.

bool(TO, yes) :- verify_rel(TO, TI,
    ↪ CLASS, REL, subject), state(TI, ID
    ↪ ), has_attr(ID', class, CLASS),
    ↪ has_rel(ID', REL, ID).
bool(TO,no) :- verify_rel(TO, TI, CLASS,
    ↪  REL, subject), not bool(TO,yes).

bool(TO, yes) :- verify_rel(TO, TI,
    ↪ CLASS, REL, object), state(TI, ID)
    ↪ , has_attr(ID', class, CLASS),
    ↪ has_rel(ID, REL, ID').
bool(TO,no) :- verify_rel(TO, TI, CLASS,
    ↪  REL, object), not bool(TO,yes).

% verify_attr
#defined verify_attr/4.

bool(TO, yes) :- verify_attr(TO, TI,
    ↪ ATTR, VALUE), state(TI, ID),
    ↪ has_attr(ID, ATTR, VALUE).
bool(TO,no) :- verify_attr(TO, TI, ATTR,
    ↪  VALUE), not bool(TO,yes).

% ---------- choose ----------
% choose_rel
#defined choose_rel/6.
{has_rel(ID', REL, ID): has_attr(ID',
    ↪ class, CLASS); has_rel(ID', REL',
    ↪ ID): has_attr(ID', class, CLASS)}
    ↪ = 1 :- choose_rel(TO, TI, CLASS,
    ↪ REL, REL', subject), state(TI, ID)
    ↪ .
rel(TO, REL) :- choose_rel(TO, TI, CLASS,
    ↪  REL, REL', subject), state(TI, ID
    ↪ ), has_attr(ID', class, CLASS),
    ↪ has_rel(ID', REL, ID).
rel(TO, REL') :- choose_rel(TO, TI,
    ↪ CLASS, REL, REL', subject), state(
    ↪ TI, ID), has_attr(ID', class,
    ↪ CLASS), has_rel(ID', REL', ID).

{has_rel(ID, REL, ID'): has_attr(ID',
    ↪ class, CLASS); has_rel(ID, REL',
    ↪ ID'): has_attr(ID', class, CLASS)}
    ↪  = 1 :- choose_rel(TO, TI, CLASS,
    ↪ REL, REL', object), state(TI, ID).
rel(TO, REL) :- choose_rel(TO, TI, CLASS,
    ↪  REL, REL', object), state(TI, ID)
    ↪ , has_attr(ID', class, CLASS),
    ↪ has_rel(ID, REL, ID').
rel(TO, REL') :- choose_rel(TO, TI,
    ↪ CLASS, REL, REL', object), state(
    ↪ TI, ID), has_attr(ID', class,
    ↪ CLASS), has_rel(ID, REL', ID').

% choose_attr
#defined choose_attr/5.
{has_attr(ID, ATTR, VALUE); has_attr(ID,
    ↪  ATTR, VALUE')} = 1 :- choose_attr
    ↪ (TO, TI, ATTR, VALUE, VALUE'),
    ↪ state(TI, ID).
attr_value(TO, VALUE) :- choose_attr(TO,
    ↪  TI, ATTR, VALUE, VALUE'), state(
    ↪ TI, ID), has_attr(ID, ATTR, VALUE)
    ↪ .
attr_value(TO, VALUE') :- choose_attr(TO,
    ↪  TI, ATTR, VALUE, VALUE'), state(
    ↪ TI, ID), has_attr(ID, ATTR, VALUE'
    ↪ ).

% ---------- exist ----------
#defined exist/2.

bool(TO,yes) :- exist(TO, TI), state(TI,
    ↪ ID).
bool(TO,no) :- exist(TO, TI), not bool(
    ↪ TO,yes).

% ---------- different, same ----------
% all_different
#defined all_different/3.

bool(TO,no) :- all_different(TO, TI,
    ↪ ATTR), state(TI, ID), state(TI, ID
    ↪ '), has_attr(ID, ATTR, VALUE),
    ↪ has_attr(ID', ATTR, VALUE), ID !=
    ↪ ID'.
bool(TO,yes) :- all_different(TO, TI,
    ↪ ATTR), not bool(TO,no).

% all_same
```

```
#defined all_same/3.

bool(TO,no) :- all_same(TO, TI, ATTR),
    ↪ state(TI, ID), state(TI, ID'),
    ↪ has_attr(ID, ATTR, VALUE), not
    ↪ has_attr(ID', ATTR, VALUE), ID !=
    ↪ ID'.
bool(TO,yes) :- all_same(TO, TI, ATTR),
    ↪ not bool(TO,no).

% two_different
#defined two_different/4.

bool(TO,no) :- two_different(TO, TI0,
    ↪ TI1, ATTR), state(TI0, ID), state(
    ↪ TI1, ID'), has_attr(ID, ATTR,
    ↪ VALUE), has_attr(ID', ATTR, VALUE)
    ↪ .
bool(TO,yes) :- two_different(TO, TI0,
    ↪ TI1, ATTR), not bool(TO,no).

% two_same
#defined two_same/4.

bool(TO,no) :- two_same(TO, TI0, TI1,
    ↪ ATTR), state(TI0, ID), state(TI1,
    ↪ ID'), has_attr(ID, ATTR, VALUE),
    ↪ not has_attr(ID', ATTR, VALUE).
bool(TO,no) :- two_same(TO, TI0, TI1,
    ↪ ATTR), state(TI0, ID), state(TI1,
    ↪ ID'), not has_attr(ID, ATTR, VALUE
    ↪ ), has_attr(ID', ATTR, VALUE).
bool(TO,yes) :- two_same(TO, TI0, TI1,
    ↪ ATTR), not bool(TO,no).

% ---------- common ----------
#defined common/3.

attr(TO, ATTR) :- common(TO, TI0, TI1),
    ↪ state(TI0, ID), state(TI1, ID'),
    ↪ has_attr(ID, ATTR, VALUE),
    ↪ has_attr(ID', ATTR, VALUE), ATTR
    ↪ != name, ATTR != class, ATTR !=
    ↪ hposition, ATTR != vposition.
{attr(TO, ATTR): is_attr(ATTR)} = 1 :-
    ↪ common(TO, TI0, TI1).


% ========== Utility Operations
    ↪ ==========
% ---------- boolean ----------
% and
#defined and/3.

bool(TO,yes) :- and(TO, TI0, TI1), bool(
    ↪ TI0,yes), bool(TI1,yes).
bool(TO,no) :- and(TO, TI0, TI1), not
    ↪ bool(TO,yes).

% or
#defined or/3.

bool(TO,yes) :- or(TO, TI0, TI1), bool(
    ↪ TI0,yes).
bool(TO,yes) :- or(TO, TI0, TI1), bool(
    ↪ TI1,yes).
bool(TO,no) :- or(TO, TI0, TI1), not
    ↪ bool(TO,yes).

% ---------- unique ----------
#defined unique/2.

{state(TO,ID): state(TI,ID)} = 1 :-
    ↪ unique(TO, TI).
:~ unique(TO, TI), state(TO,ID),
    ↪ has_obj_weight(ID, P). [P, (TO, ID
    ↪ )]

% ---------- negate ----------
#defined negate/3.
state(TO, ID) :- negate(TO, TI0, TI1),
    ↪ state(TI0, ID), not state(TI1, ID)
    ↪ .
```

Next, we give a description of each of the operations that appear in our encoding.

Base Operations

Scene: The $scene()$ operation simply returns all objects in the scene as encoded by the Scene Encoding. Its single ASP rule is defined as follows:

$$state(T_o, I) :- scene(T_o), object(I). \tag{1}$$

End: The $end()$ operation converts the different possible answer types coming from its input step into a common $ans()$ predicate. Its rules are simple in structure:

$$ans(V) :- end(T_o), attr\_value(T_o, V). \tag{2}$$
$$ans(A) :- end(T_o), attr(T_o, A). \tag{3}$$
$$ans(R) :- end(T_o), rel(T_o, R). \tag{4}$$
$$ans(B) :- end(T_o), bool(T_o, B). \tag{5}$$

We add an integrity constraint that forbids solutions in which no $ans()$ predicate can be derived:

$$:- not\ ans(\_). \tag{6}$$

Intermediary Operations

Select: The $select()$ operation restricts its input set of objects to those that are members of a certain class. Its ASP rule is as follows:

$$\begin{aligned} state(T_o, I) :- &select(T_o, T_i, C), \\ &state(T_i, I), has\_attr(I, \text{class}, C). \end{aligned} \tag{7}$$

Filter: Similarly to the $select()$ operation, the $filter()$ operation restricts its input objects to those that have a specific value for a certain attribute category. For cases in which it is not known which category the filtered value belongs to, the analogous $filter\_any()$ operation is used. We present only the rule for the former, since the latter is identical except for the restriction to an attribute category:

$$state(T_o, I) :- filter(T_o, T_i, A, V),\; state(T_i, I), has\_attr(I, A, V). \tag{8}$$

Relate: Relate operations return objects connected to some input object through either a relation or a common attribute value. The $relate()$ operation covers the case of connection through a relation, and additionally requires that the connected objects belong to a certain class. We only present the rule variant in which the input object is the object of the relation, and the connected object is the subject; the variant with swapped subject/object positions is analogous:

$$state(T_o, I_2) :- relate(T_o, T_i, C, R, \text{subject}), state(T_i, I_1),\; has\_attr(I_2, \text{class}, C), has\_rel(I_2, R, I_1). \tag{9}$$

The $relate\_any()$ operation does not restrict the class of the connected objects, and its rules are the same as those for $relate()$ except for the omission of the $has\_attr(I_2, \text{class}, C)$ restriction.

The case of connection through a common attribute value is covered by the $relate\_attr()$ operation, whose sole rule is presented below:

$$state(T_o, I_2) :- relate\_attr(T_o, T_i, C, A), state(T_i, I_1),\; has\_attr(I_1, A, V), has\_attr(I_2, \text{class}, C),\; has\_attr(I_2, A, V), I_1 \neq I_2. \tag{10}$$

Compare: The $compare()$ operation takes two steps, an attribute value, and a mode as input. It checks if one of the input objects from the two input steps (respectively assumed unique) has the attribute value, and the other one does not. If so, it returns the object having the attribute value if the mode is `true`, and the object not having the attribute value if the mode is `false`. We present the rules for the mode `true` below, the rules for the mode `false` are analogous:

$$state(T_o, I_1) :- compare(T_o, T_{i1}, T_{i2}, V, \text{true}),\; state(T_{i1}, I_1), state(T_{i2}, I_2),\; has\_attr(I_1, \_, V), not\, has\_attr(I_2, \_, V). \tag{11}$$

$$state(T_o, I_2) :- compare(T_o, T_{i1}, T_{i2}, V, \text{true}),\; state(T_{i1}, I_1), state(T_{i2}, I_2),\; not\, has\_attr(I_1, \_, V), has\_attr(I_2, \_, V). \tag{12}$$

Terminal Operations

Query: The $query()$ operation returns the value of a specific attribute category for its input object (it implicitly assumes only one input object is present). Its rule is accompanied by a cardinality rule (shown in simplified form in Equation 14) that ensures that the optimization must assign exactly one value for the attribute category to the object. As with other operation-specific cardinality rules that follow, we do not enforce this constraint in general, since it may be beneficial to consider multiple similar attribute values (e.g., `brown`, `beige`) to apply to an object, for example for filter operations. The added rules are shown below:

$$attr\_value(T_o, V) :- query(T_o, T_i, A),\; state(T_i, I), has\_attr(I, A, V). \tag{13}$$

$$\{has\_attr(I, A, V) : is\_attr\_value(A, V)\} = 1 :-\; query(T_o, T_i, A), state(T_i, I). \tag{14}$$

Verify: Verify operations return a boolean that indicates whether the input object (assumed unique) is connected through a specific relation to some object with a certain class ($verify\_rel()$), or has a specific attribute value ($verify\_attr()$). Of the rules for $verify\_rel()$, we present only the case in which the input object is in the object position of the relation; the case for the subject position is analogous:

$$bool(T_o, \text{yes}) :- verify\_rel(T_o, T_i, C, R, \text{subject}),\; state(T_i, I_1), has\_attr(I_2, \text{class}, C),\; has\_rel(I_2, R, I_1). \tag{15}$$

$$bool(T_o, \text{yes}) :- verify\_attr(T_o, T_i, A, V),\; state(T_i, I), has\_attr(I, A, V). \tag{16}$$

For all of these rules, a dual one exists that forces the operation output to `no` if the conditions for `yes` are not fulfilled. The one for $verify\_attr()$ is shown below:

$$bool(T_o, \text{no}) :- verify\_attr(T_o, T_i, A, V),\; not\, bool(T_o, \text{yes}). \tag{17}$$

Choose: Choose operations are similar to Verify operations, but rather than asking *if* a certain relation or attribute value is present for the input object (again assumed unique), they ask *which* of two options is present. Of the rules for $choose\_rel()$, we again show only the case in which the input object is in the object position. Also, only the rule for one of the two options is presented, since the other (R_2 or V_2) is analogous:

$$rel(T_o, R_1) :- choose\_rel(T_o, T_i, C, R_1, R_2, \text{sub}),\; state(T_i, I_1), has\_attr(I_2, \text{class}, C),\; has\_rel(I_2, R_1, I_1). \tag{18}$$

$$attr\_value(T_o, V_1) :- choose\_attr(T_o, T_i, A, V_1, V_2),\; state(T_i, I),\; has\_attr(I, A, V_1). \tag{19}$$

For both $choose\_rel()$ and $choose\_attr()$, we again use cardinality rules to enforce that only one of the two options can apply:

$$\begin{aligned}\{&has\_rel(I_2, R_1, I_1) : has\_attr(I_2, \text{class}, C);\\ &has\_rel(I_2, R_2, I_1) : has\_attr(I_2, \text{class}, C)\} = 1 \text{:-}\\ &choose\_rel(T_o, T_i, C, R_1, R_2, \text{subject}), state(T_i, I_1).\end{aligned} \tag{20}$$

$$\begin{aligned}\{has\_attr(I, A, V_1); has\_attr(I, A, V_2)\} = 1 \text{:-}&\\ choose\_attr(T_o, T_i, A, V_1, V_2),&\\ state(T_i, I).&\end{aligned} \tag{21}$$

Exist: The Exist operation returns whether or not an input object exists:

$$bool(T_o, \text{yes}) \text{:-}\ exist(T_o, T_i), state(T_i, I). \tag{22}$$

Like for the Verify operations, a dual rule is added that outputs `no` if the rule outputting `yes` does not apply.

Different/Same: The Different/Same operations come in two distinct variations: The $all\_different()$ and $all\_same()$ operations take one step and an attribute category as input and check if across all objects from the input step, their sets of attribute values for the input category are pairwise disjoint (for $all\_different()$), or identical (for $all\_same()$):

$$\begin{aligned}bool(T_o, \text{no}) \text{:-}\ &all\_different(T_o, T_i, A), state(T_i, I_1),\\ &state(T_i, I_2), has\_attr(I_1, A, V),\\ &has\_attr(I_2, A, V), I_1 \neq I_2.\end{aligned} \tag{23}$$

$$\begin{aligned}bool(T_o, \text{no}) \text{:-}\ &all\_same(T_o, T_i, A), state(T_i, I_1),\\ &state(T_i, I_2), has\_attr(I_1, A, V),\\ &not\ has\_attr(I_2, A, V), I_1 \neq I_2.\end{aligned} \tag{24}$$

The other variant of the Different/Same operations, $two\_different()$ and $two\_same()$, have two input steps and check whether for the input objects from those steps (respectively assumed unique), their sets of attribute values for an attribute category are disjoint (for $two\_different()$), or identical (for $two\_same()$):

$$\begin{aligned}bool(T_o, \text{no}) \text{:-}\ &two\_different(T_o, T_{i1}, T_{i2}, A), state(T_{i1}, I_1),\\ &state(T_{i2}, I_2), has\_attr(I_1, A, V),\\ &has\_attr(I_2, A, V).\end{aligned} \tag{25}$$

$$\begin{aligned}bool(T_o, \text{no}) \text{:-}\ &two\_same(T_o, T_{i1}, T_{i2}, A), state(T_{i1}, I_1),\\ &state(T_{i2}, I_2), has\_attr(I_1, A, V),\\ &not\ has\_attr(I_2, A, V).\end{aligned} \tag{26}$$

$$\begin{aligned}bool(T_o, \text{no}) \text{:-}\ &two\_same(T_o, T_{i1}, T_{i2}, A), state(T_{i1}, I_1),\\ &state(T_{i2}, I_2), not\ has\_attr(I_1, A, V),\\ &has\_attr(I_2, A, V).\end{aligned} \tag{27}$$

Like for the other terminal operations returning booleans, a dual rule exists for each of the rules above.

Common: The Common operation has two input steps and returns an attribute category for which the input objects from those steps (respectively assumed unique) have a common value:

$$\begin{aligned}attr(T_o, A) \text{:-}\ &common(T_o, T_{i1}, T_{i2}), state(T_{i1}, I_1),\\ &state(T_{i2}, I_2), has\_attr(I_1, A, V),\\ &has\_attr(I_2, A, V).\end{aligned} \tag{28}$$

Utility Operations

Boolean: There are two Boolean operations that combine the boolean output(s) of terminal operation steps as expected according to boolean arithmetic, $and()$:

$$\begin{aligned}bool(T_o, \text{yes}) \text{:-}\ &and(T_o, T_{i1}, T_{i2}),\\ &bool(T_{i1}, \text{yes}), bool(T_{i2}, \text{yes}).\end{aligned} \tag{29}$$

$$bool(T_o, \text{no})\ \text{:-}\ and(T_o, T_{i1}, T_{i2}), not\ bool(T_o, \text{yes}). \tag{30}$$

And $or()$:

$$bool(T_o, \text{yes}) \text{:-}\ or(T_o, T_{i1}, T_{i2}), bool(T_{i1}, \text{yes}). \tag{31}$$

$$bool(T_o, \text{yes}) \text{:-}\ or(T_o, T_{i1}, T_{i2}), bool(T_{i2}, \text{yes}). \tag{32}$$

$$bool(T_o, \text{no})\ \text{:-}\ or(T_o, T_{i1}, T_{i2}), not\ bool(T_o, \text{yes}). \tag{33}$$

Unique: The Unique operation forces only one of its input objects objects to be returned, the one with the lowest weight:

$$\{state(T_o, I) : state(T_i, I)\} = 1 \text{:-}\ unique(T_o, T_i). \tag{34}$$

$$\begin{aligned}\text{:}\sim\ &unique(T_o, T_i), state(T_o, I),\\ &has\_obj\_weight(I, W).[W, (T_o, I)]\end{aligned} \tag{35}$$

Negate: The Negate operation has two input steps and returns all those input objects from the first step that are not present in the second one:

$$\begin{aligned}state(T_o, I) \text{:-}\ &negate(T_o, T_{i1}, T_{i2}),\\ &state(T_{i1}, I), not\ state(T_{i2}, I).\end{aligned} \tag{36}$$

Question Encoding Each operation in the functional representation of the question is—in general—translated into one ASP fact, although additional facts might be inserted to enforce certain properties that are implicit in the semantic representation of GQA (eg., uniqueness, inverted criteria).

We illustrate the encoding with the example question "Do the umpire and the person holding the green baseball bat have the same pose?", whose corresponding ASP Question Encoding is presented in Figure 3.

From the example it can be seen that the structure of the tree of reasoning operations in the semantic representation is encoded using step indices. Each predicate encoding an

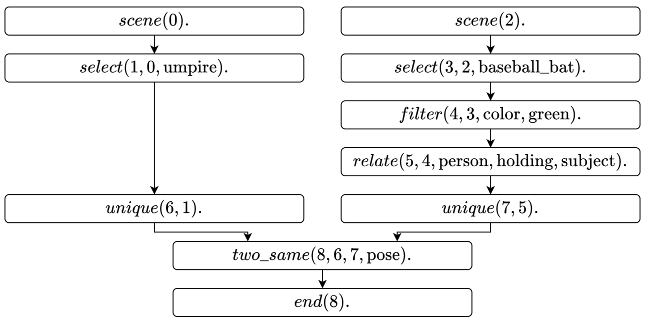


Figure 3: The Question Encoding for the question "Do the umpire and the person holding the green baseball bat have the same pose?"

operation takes as first argument the index of its output step, then one or more indices of its input step(s), followed by the remaining arguments specific to the operation. If an operation has no predecessor in the semantic representation, a $scene/1$ fact is inserted, and an $end/1$ fact is always added for the last step index. The mapping of all operations from the GQA semantic representation to their corresponding Question Encoding facts is shown in Table 8.

Scene Encoding As explained in in Section 3, the Scene Processing component outputs a partial scene graph represented as a list O of objects $o_i = (id_i, s_i, B_i, c_i, A_{o,i}, R_{o,i})$, each having a unique identifier $id_i \in \mathcal{I}$, a score $s_i \in [0, 1]$, a bounding box $B_i = (x, y, w, h) : \mathbb{R}^4_{\geq 0}$, a class $c_i \in \mathcal{C}$, attribute likelihoods $A_{o,i} : \mathcal{A} \to (\mathcal{V} \to [0, 1])$, and relation likelihoods $R_{o,i} : \mathcal{I} \to (\mathcal{R} \to [0, 1])$. The information for each object o_i is converted into a set of ASP facts and constraints as described in the following paragraphs.

Objects: To start, we add an $object(id_i)$ fact to the encoding to establish the existence of the object in the scene. To be able to select the object that the Object Detection component has the most confidence in when filtering a set of qualifying objects with the $unique()$ operation, we add a fact $has\_obj\_weight(id_i, w_o)$, where $w_o = \lfloor \min(-1000 \cdot \ln(s_i), 5000) \rfloor$. The formulation of w_o is taken from NeurASP (Yang, Ishay, and Lee 2020) and ensures that the "higher-is-better" score s_i is correctly adapted to a "lower-is-better" weight for the $unique()$ operation (see Equation 35). So, for our example object, the added facts would be:

$$object(\text{o1}). \tag{37}$$
$$has\_obj\_weight(\text{o1}, 1971). \tag{38}$$

Classes: For the object's class c_i and all of its parent classes (if any), we add a fact $has\_attr(id_i, \text{class}, c_i)$. This allows the object to be considered not just for operations filtering on its specific class, but also the parent classes it belongs to. Additionally, we add a fact $has\_attr(id_i, \text{name}, c_i)$. Questions in GQA that inquire the type of a certain object (e.g., "Who is wearing a shirt?") always have `query: name` as the terminal operation, and the expected answer is the object's most specific class. Having $has\_attr(id_i, \text{name}, c_i)$ in the encoding allows us to easily identify the most specific class of the object for these cases. For our example object, the added facts are thus:

$$has\_attr(\text{o1}, \text{class}, \text{alive}). \tag{39}$$
$$has\_attr(\text{o1}, \text{class}, \text{person}). \tag{40}$$
$$has\_attr(\text{o1}, \text{class}, \text{baseball\_player}). \tag{41}$$
$$has\_attr(\text{o1}, \text{name}, \text{baseball\_player}). \tag{42}$$

Attributes: For each attribute value v in category a in $A_{o,i}$, we add the choice rule $\{has\_attr(id_i, a, v)\}.$, which ensures that the ASP solver can consider both the case where the attribute value applies to the object, and the case where it doesn't. To include the likelihoods of those two cases into the consideration, we add the weak constraints $:\sim\ has\_attr(id_i, a, v).\,[w_{v^+}, (id_i, a, v)]$ and $:\sim$ $not\, has\_attr(id_i, a, v).\,[w_{v^-}, (id_i, a, v)]$, where $w_{v^+} = \lfloor \min(-1000 \cdot \ln(s_v), 5000) \rfloor$ and $w_{v^-} = \lfloor \min(-1000 \cdot \ln(1 - s_v), 5000) \rfloor$, s_v being the likelihood of v applying to the object according to $A_{o,i}$. For the example object and the `standing` attribute value of the `pose` category, we therefore add the following rules and constraints to the encoding:

$$\{has\_attr(\text{o1}, \text{pose}, \text{standing})\}. \tag{43}$$
$$:\sim\ has\_attr(\text{o1}, \text{pose}, \text{standing}). \tag{44}$$
$$[83, (\text{o1}, \text{pose}, \text{standing})]$$
$$:\sim\ not\, has\_attr(\text{o1}, \text{pose}, \text{standing}). \tag{45}$$
$$[2525, (\text{o1}, \text{pose}, \text{standing})]$$

Additionally, we add facts for the two attribute categories that are explicitly excluded from Scene Processing, `hposition` and `vposition`, whose value for each object is directly derived from the bounding box B_i. We divide the input image horizontally and vertically into thirds. For `hposition`, the object gets the value `left`, `middle`, or `right` if the center of its bounding box falls within the left, middle, or right horizontal third of the input image. Analogously for `vposition`, the object gets the value `top`, `middle`, or `bottom` if the center of its bounding box falls within the upper, middle, or lower vertical third of the input image. For our example object, given its bounding box center of $(205, 185)$ and the 500×334 dimensions of its corresponding input image, this yields the following facts:

$$has\_attr(\text{o1}, \text{hposition}, \text{middle}). \tag{46}$$
$$has\_attr(\text{o1}, \text{vposition}, \text{middle}). \tag{47}$$

Relations: For each other object $o_j = (id_j, s_j, B_j, c_j, A_{o,j}, R_{o,j}) \in O \setminus \{o_i\}$ and relation r in $R_{o,i}$, we proceed like we do for attribute values. We add the choice rule $\{has\_rel(id_i, r, id_j)\}.$, and the weak constraints $:\sim\ has\_rel(id_i, r, id_j).\,[w_{r^+}, (id_i, r, id_j)]$ and $:\sim\ not\, has\_rel(id_i, r, id_j).\,[w_{r^-}, (id_i, r, id_j)]$. Here, $w_{r^+} = \lfloor \min(-1000 \cdot \ln(s_r), 5000) \rfloor$ and

Category	**Operation**	**Resulting Fact(s)**
Select	`select: person`	$\mathit{select}(T_o, T_i, \text{person}).$
Filter	`filter color: red`	$\mathit{filter}(T_o, T_i, \text{color}, \text{red}).$
	`filter color: not(red)`	$\mathit{filter}(T_{o1}, T_i, \text{color}, \text{burnt}).$ $\mathit{negate}(T_{o2}, T_i, T_{o1}).$
	`filter: burnt`	$\mathit{filter\_any}(T_o, T_i, \text{burnt}).$
	`filter: not(burnt)`	$\mathit{filter\_any}(T_{o1}, T_i, \text{burnt}).$ $\mathit{negate}(T_{o2}, T_i, T_{o1}).$
Relate	`relate: person,holding,s`	$\mathit{relate}(T_o, T_i, \text{person}, \text{holding}, \text{subject}).$
	`relate: _,holding,o`	$\mathit{relate\_any}(T_o, T_i, \text{holding}, \text{object}).$
	`relate: person,same_pose,_`	$\mathit{relate\_attr}(T_o, T_i, \text{person}, \text{pose}).$
Query	`query: color`	$\mathit{unique}(T_{o1}, T_i).$ $\mathit{query}(T_{o2}, T_{o1}, \text{color}).$
Verify	`verify color: red`	$\mathit{unique}(T_{o1}, T_i).$ $\mathit{verify\_attr}(T_{o2}, T_{o1}, \text{color}, \text{red}).$
	`verify: burnt`	$\mathit{unique}(T_{o1}, T_i).$ $\mathit{verify\_attr}(T_{o2}, T_{o1}, \text{any}, \text{burnt}).$
	`verify rel: shorts,wearing,o`	$\mathit{unique}(T_{o1}, T_i).$ $\mathit{verify\_rel}(T_{o2}, T_{o1}, \text{shorts}, \text{wearing}, \text{object}).$
Choose	`choose color: red|blue`	$\mathit{unique}(T_{o1}, T_i).$ $\mathit{choose\_attr}(T_{o2}, T_{o1}, \text{color}, \text{red}, \text{blue}).$
	`choose: dried|wet`	$\mathit{unique}(T_{o1}, T_i).$ $\mathit{choose\_attr}(T_{o2}, T_{o1}, \text{any}, \text{dried}, \text{wet}).$
	`choose rel: bat,near|in,o`	$\mathit{unique}(T_{o1}, T_i).$ $\mathit{choose\_rel}(T_{o2}, T_{o1}, \text{bat}, \text{near}, \text{in}, \text{object}).$
	`choose less healthy` (analogous for `more`)	$\mathit{unique}(T_{o1}, T_{i1}).$ $\mathit{unique}(T_{o2}, T_{i2}).$ $\mathit{compare}(T_{o3}, T_{o1}, T_{o2}, \text{healthy}, \text{false}).$ $\mathit{query}(T_{o4}, T_{o3}, \text{name}).$
	`choose healthier`	$\mathit{unique}(T_{o1}, T_{i1}).$ $\mathit{unique}(T_{o2}, T_{i2}).$ $\mathit{compare}(T_{o3}, T_{o1}, T_{o2}, \text{healthy}, \text{true}).$ $\mathit{query}(T_{o4}, T_{o3}, \text{name}).$
Exist	`exist`	$\mathit{exist}(T_o, T_i).$
Different/ Same	`same color` (analogous for `different`)	$\mathit{unique}(T_{o1}, T_{i1}).$ $\mathit{unique}(T_{o2}, T_{i2}).$ $\mathit{two\_same}(T_{o3}, T_{o1}, T_{o2}, \text{color}).$
	`same: color` (analogous for `different`)	$\mathit{all\_same}(T_o, T_i, \text{color}).$
Common	`common`	$\mathit{unique}(T_{o1}, T_{i1}).$ $\mathit{unique}(T_{o2}, T_{i2}).$ $\mathit{common}(T_{o3}, T_{o1}, T_{o2}).$
And	`and`	$\mathit{and}(T_o, T_{i1}, T_{i2}).$
Or	`or`	$\mathit{or}(T_o, T_{i1}, T_{i2}).$

Table 8: The Question Encoding for each operation from GQA's semantic representation

$w_{r^-} = \lfloor \min(-1000 \cdot \ln(1 - s_r), 5000) \rfloor$, s_r being the likelihood of r applying with o_i as subject and o_j as object according to $R_{o,i}$. For the example object as subject, o_2 as object, and the `wearing` relation, we therefore extend our encoding as follows:

$$\{\mathit{has\_rel}(\text{o1}, \text{wearing}, \text{o2})\}. \tag{48}$$

$$:\sim \mathit{has\_rel}(\text{o1}, \text{wearing}, \text{o2}).\ [105, (\text{o1}, \text{wearing}, \text{o2})] \tag{49}$$

$$:\sim \mathit{not}\ \mathit{has\_rel}(\text{o1}, \text{wearing}, \text{o2}).\ [2302, (\text{o1}, \text{wearing}, \text{o2})] \tag{50}$$

B.2 Scene Graph Generation Implementation

We describe the full implementation details of the components of the GS-VQA pipeline that were introduced conceptually in Section 3.

Preprocessing We ignore a particular set of questions in GQA that are not answerable with the information contained in the ground-truth scene graph of the corresponding image. These questions refer to the image as a whole, rather than objects within it, and all start with a `select: scene` operation. Examples for this set of questions are: "Which place is it?", "Is it an outdoors scene?", and "What is the image showing?".

Furthermore, we singularize plural object classes since the semantic representation of GQA uses singular/plural inconsistently: for example, the question "Is there a pepper or a potato that is not red?" has the operation `select: potatoes` in its semantic representation.

Concept Extraction The Concept Extraction component's job is to produce a tuple (C, A, R) with the classes, attributes and relations relevant to answering the question. Table 9 contains all operations that can occur in the GQA semantic question representation and the classes, attributes, or relations extracted from them. However, we have to explain some particularities.

Let $\mathcal{C}$ be the set of all classes in the current dataset (GQA in our case). Accordingly, let $\mathcal{A}$ be the set of all attribute categories, $\mathcal{V}$ be the set of all attribute values, and $\mathcal{R}$ be the set of all relations in the dataset. Furthermore, let $cv : \mathcal{A} \to 2^{\mathcal{V}}$ be the mapping of attribute categories to the attribute values included in that category.

Regarding classes, certain questions reference objects whose class is not restricted by any operation of the question. For example, for the question "What is the man holding?", the object in question is only restricted by its `holding` relation to an object of class `man`, but could be of any class. In GQA's semantic question representation, these cases all contain a `relate` operation without class restriction (e.g., `relate: _,holding,o`). To signify to the Object Detection component that objects of all possible classes in the scene could be of relevance, a special `all` class is introduced and added to C in these cases. As such, strictly speaking $C \subseteq \mathcal{C} \cup \{\texttt{all}\}$, though we will include `all` in $\mathcal{C}$ for simplicity in the remainder of this thesis. Note that GQA does not have questions like "What is blue?" that do not contain a relation operation but still require all objects in the scene, though it would be easy to add support for them with a `select: _` operation for which the special `all` class is added to C as well.

Usually, operations in GQA's semantic question representation that work with attributes contain both a category and the relevant value(s) of that category (if any). However, GQA contains some attribute values that are not associated with any category (Table 9 contains some examples: `burnt`, `healthy`, `dried`). For these, operation variants without a category exist (e.g., `filter: burnt` instead of `filter color: red`). To handle these *standalone* values, the set A is actually split into two sets A_c, A_v, where attribute categories extracted from the semantic question representation are added to A_c, while relevant standalone values are added to A_v. The information passed to the Scene Processing component is therefore actually (C, A_c, A_v, R), with $A_c \subseteq \mathcal{A}$, $A_v \subseteq \mathcal{V}$, and $R \subseteq \mathcal{R}$. To declutter the notation in the succeeding sections, we consider standalone values to be part of the `any` attribute category, which is considered included in $\mathcal{A}$.

Similar to the situation for classes, there is one operation that requires all attribute categories to be considered. The `common` operation, which appears in questions like "What do the plate and the silver watch have in common?", has as its answer the attribute category for which its two input objects have the same value. If one such operation occurs in the input question, we set $A_c = \mathcal{A}$.

For certain operations like `filter color: red`, we know that only a specific value (`red`) of the attribute category (`color`) is relevant to the question, but add the entire category to A_c regardless.

Scene Processing Having identified the concepts relevant to answering a question as a (C, A_c, A_v, R) tuple in the Concept Extraction component, we now have to actually construct the partial scene graph that contains this information from the input image. This partial scene graph is represented as a list O of objects $o_i = (id, s, B, c, A_o, R_o)$, each having a unique identifier $id \in \mathcal{I}$ for some set of alphanumeric identifiers $\mathcal{I}$, a detection confidence score $s \in [0, 1]$, a bounding box $B = (x, y, w, h) : \mathbb{R}^4_{\geq 0}$, a class $c \in \mathcal{C}$, attribute likelihoods $A_o : \mathcal{A} \to (\mathcal{V} \to [0, 1])$, and relation likelihoods $R_o : \mathcal{I} \to (\mathcal{R} \to [0, 1])$. Constructing the graph is done in two stages.

First, the Object Detection sub-component detects objects and their bounding boxes in the scene that conform to one of the classes in C. Second, the Concept Classification sub-component takes these objects as input and supplements the likelihoods for attributes and relations applying to them.

Object Detection For object detection, we use OWL-ViT (Minderer et al. 2022), which is capable of operating in an open-vocabulary manner, i.e., the classes for which it should detect objects can vary from invocation to invocation without any modifications to the model. Given a list of class labels to detect, OWL-ViT returns a list of object detections, each with the class label $c \in \mathcal{C}$ it was detected as, a detection confidence score $s \in [0, 1]$, and a bounding box $B = (x, y, w, h) : \mathbb{R}^4_{\geq 0}$, where (x, y) are the coordinates of the top-left corner of the bounding box (with the top-left corner of the image having coordinates $(0, 0)$), and w and h are the width and height of the bounding box.

We process the classes in C in a multi-staged manner, in the order of class specificity according to GQA's hierarchy. As an example to illustrate the explanations below, we assume that we have extracted the following set C of relevant classes from an input question: $C = \{\texttt{vehicle}, \texttt{van}, \texttt{driver}, \texttt{all}\}$.

1. As a first step, OWL-ViT is used to detect objects with classes of maximal specificity (i.e., classes without subclasses in the hierarchy), in this case `van` and `driver`. We directly use the class names as labels for OWL-ViT, and do not transform them to, e.g., "a photo of a $\{\texttt{class}\}$".

Category	Operation	Extracted Concepts
Select	`select: person`	`person` (C)
Filter	`filter color: red`	`color` (A_c)
	`filter color: not(red)`	`color` (A_c)
	`filter: burnt`	`burnt` (A_v)
	`filter: not(burnt)`	`burnt` (A_v)
Relate	`relate: person,holding,s`	`person` (C), `holding` (R)
	`relate: _,holding,o`	`all` (C), `holding` (R)
	`relate: person,same_pose,_`	`person` (C), `pose` (A_c)
Query	`query: color`	`color` (A_c)
	`query: name`	–
Verify	`verify color: red`	`color` (A_c)
	`verify: burnt`	`burnt` (A_v)
	`verify rel: shorts,wearing,o`	`shorts` (C), `wearing` (R)
Choose	`choose color: red|blue`	`color` (A_c)
	`choose: dried|wet`	`dried` (A_v), `wet` (A_v)
	`choose rel: bat,holding|wearing,o`	`bat` (C), `holding` (R), `wearing` (R)
	`choose less healthy` (analogous for `more`)	`healthy` (A_v)
	`choose healthier`	`healthy` (A_v)
Exist	`exist`	–
Different/ Same	`same color` (analogous for `different`)	`color` (A_c)
	`same: color` (analogous for `different`)	`color` (A_c)
Common	`common`	$\mathcal{A}$ (A_c)
And	`and`	–
Or	`or`	–

Table 9: The extracted concepts from each operation in GQA's semantic question representation. Classes are denoted with (C), attribute categories with (A_c), standalone values with (A_v), and relations with (R)

As a clean-up step, objects whose bounding boxes overlap by more than a threshold t_{bbox1} and have the same class c are combined, i.e., they are returned as one with the smallest bounding box enclosing both original boxes and the higher of the two object's detection confidence scores. The number of objects returned is limited in two ways: First, only objects for which OWL-ViT's detection confidence score exceeds a certain threshold t_s are returned. Second, if more than k_1 objects are detected for a class with a confidence score exceeding t_s, only the top k_1 objects (sorted by confidence score) are returned. The result is a list of object proposals $O_1 = [(s_1, B_1, c_1), (s_2, B_2, c_2), ...]$.

2. In the second step, for each class higher up in the class hierarchy, objects of all sub-classes of maximal specificity of that class are detected. Like in step 1, the threshold t_s applies, and the maximum number of objects returned for one higher-order class is limited by a separate threshold k_2. While OWL-ViT would also directly accept the higher-order class for detection, we require for each detected object a class of maximal specificity in case that it becomes the target of a `query: name` operation (for example, GQA expects questions like "What thing is the man holding?" to be answered with "burger" and not simply "food"). Another benefit to detecting the sub-classes of maximal specificity is that some higher-order classes are rather vague and therefore difficult for OWL-ViT to detect (for example, OWL-ViT struggles with detecting objects of class "watercraft", but is rather adept at detecting a "boat"). Note that this approach requires the class hierarchy for the current dataset, which we deem an acceptable compromise, since regular SGG or open-vocabulary object detection for all classes in the dataset would require at least the complete list of possible classes in any case.

For our example, OWL-ViT would be run with all sub-classes of maximal specificity of the higher-order class `vehicle`: `minivan`, `jeep`, `locomotive`, `bus`, `van`, etc. The list of returned object proposals O_2^* is merged with the list of objects O_1 detected in the first step to yield the step output list O_2 in the following way: if the bounding boxes of an object in O_1 and O_2^* overlap by more than a certain percentage threshold t_{bbox2}, and the classes of the two objects have a common ancestor/higher-order class, the object from O_1 is included in O_2, and the object from O_2^* is discarded. This ensures that objects are not considered multiple times, and that objects that might feasibly belong to multiple classes of maximal specificity (e.g., `van` and `minivan`) are considered under a class of maximal specificity implied by the question.

3. The third and final step runs only if the special `all` class is included in C. In this case, OWL-ViT is run once more, this time with all classes of maximum specificity in the hierarchy. Again the threshold t_s applies, and the length of the list O_3^* of object proposals returned by this run is limited by a threshold k_3. The list of new objects O_3^* is merged with the existing list O_2 in exactly the same manner as was done in step 2, yielding the final list of object proposals O_3.

Finally, each object proposal in O_3 is assigned a unique identifier $id \in \mathcal{I}$ to yield the list of partial scene graph objects $O = [(id_1, s_1, B_1, c_1), (id_2, s_2, B_2, c_2), ...]$.

Concept Classification Taking the list of partial scene graph objects O from the Object Detection sub-component, the Concept Classification sub-component uses CLIP (Radford et al. 2021) to supplement for each object the attribute and relation likelihoods A_o, R_o according to the question-relevant attribute categories A_c, attribute values A_v, and relations R from the Concept Extraction component.

Attribute Likelihoods To determine the attribute likelihoods $A_{o,i}$ for each object $o_i = (id_i, s_i, B_i, c_i) \in O$, a crop of the image area under its bounding box B_i is created. Since especially small objects often need context to be identified even by humans (e.g., an image crop of the bounding box of a wooden table leg would just show an indeterminate brown block), we add padding to all four sides of the bounding box before creating the image crop. The padding p_w added to the left and right sides of the bounding boxes is determined from the box width w_{bbox} and the image width w_{image} as follows, with the top/bottom padding determined analogously:

$$p_w = \left[1 - \tanh\left(2\frac{w_{\text{bbox}}}{w_{\text{image}}}\right)\right] * w_{\text{bbox}} \tag{51}$$

This formulation ensures that the bounding boxes of small objects (relative to the image size) are padded by a large amount relative to their size (up to $w_{\text{bbox}}/h_{\text{bbox}}$ on each side in the limit) to provide the required context around the objects, but the amount of padding decreases quickly as object size increases. To build the attribute likelihoods $A_{o,i}$, we need for each attribute value in A_v and each attribute value of each attribute category in A_c (so each attribute value $a \in A_v \cup \{a' \mid a_c \in A_c, a' \in cv(a_c)\}$) a number in $[0, 1]$ that represents the likelihood of that attribute value applying to object o_i. We can calculate the cosine similarity between CLIP's embedding of the padded object crop and the embedding of a textual prompt to get an indication of the semantic similarity between these two: prompts "fitting" the image crop will have higher cosine similarities than prompts that don't. However, this concept of semantic similarity requires a point of reference: what constitutes a "high" cosine similarity with a prompt embedding varies from image to image.

We use a modified version of their "target vs. neutral" approach to introduce this frame of reference: We introduce a neutral prompt, i.e., one that we are sure applies to the current object o_i. The neutral prompt follows the schema "a blurry photo of {a/an} $\{c_i\}$" (for example, if $c_i = \texttt{van}$, the neutral prompt would be "a blurry photo of a van"). We then build for each attribute value a a modified version more specific to that value, the target prompt, which follows the schema "a blurry photo of {a/an} $\{a\}$ $\{c_i\}$". For example, for $a = \texttt{red}$, the target prompt would be "a blurry photo of a red van".

If the additional information added to the target prompt actually applies to the object, CLIP should return a higher cosine similarity between its embedding and the object crop than between the neutral prompt embedding and the object crop. Conversely, if the additional information does not apply to the object, the cosine similarity should be lower than for the neutral prompt. We can therefore frame the computation of the likelihood of attribute value a applying to object o_i as a binary classification problem between "applies" and "does not apply", represented by the target and neutral prompts, respectively. To obtain values in $[0, 1]$, we simply compute the softmax over the cosine similarity values between the two prompts and the object crop. More formally, let $\mathbf{f}_{\text{CLIP}}(x)$ be the embedding vector of an image or a text prompt x as returned by CLIP's image or text encoder (which are already normalized). Furthermore, let p_t, p_n be the target and neutral prompts, and I the object crop. We compute the likelihood of attribute a applying as softmax $(\mathbf{f}_{\text{CLIP}}(I) \cdot \mathbf{f}_{\text{CLIP}}(p_t), \mathbf{f}_{\text{CLIP}}(I) \cdot \mathbf{f}_{\text{CLIP}}(p_n))_0$.

Relation Likelihoods For the relation likelihoods $R_{o,i}$ of each object o_i, we use the same "target vs. neutral" approach as for attributes. Between o_i and every other object $o_j = (id_j, s_j, B_j, c_j) \in O \setminus \{o_i\}$, we determine the smallest bounding box enclosing the two objects' original boxes, and use the crop of the image area under that bounding box. For each relation $r \in R$, the neutral prompt follows the schema "$\{c_i\}$ and $\{c_j\}$", while the target prompt adheres to the schema "$\{c_i\}$ $\{r\}$ $\{c_j\}$". For example, if $c_i = \texttt{woman}$, $c_j = \texttt{van}$, and $r = \texttt{driving}$, then the neutral prompt would be "woman and van", and the target prompt would be "woman driving van".

C Knowledge Distillation

We present our complete preprompt as presented in Section 4. It is composed of seven parts, one of them being an initial theory T_{init}. We present all the parts with exception of that one, in order they are: introduction, language syntax, scene encoding, question encoding, answer format and task. The theory is introduced after the answer format and before the task in the complete preprompt.

C.1 Introduction

Our introduction in Listing 3 sets the general setting of VQA, what is the input and their ASP translation, and informs the LLM of their task in a short manner before passing passing to the following sections.

```
1. Introduction
We are in the domain of Visual Question
    Answering.
The task consists of taking an image and
     question related to it as input and
     produce as output the correct
    answer.

We have already preprocessed both the
    image and question into correct
    Answer Set Programming
    representations.
Scene/Question pairs of ASP facts serve
    as the instance for an ASP program
    which we call the Theory.
This is a collection of rules that
    handles the input instance and
    calculates the correct answer.

Your task is to help us expand the
    Theory with new rules as new
    instances of questions appear.
We will first give you some definitions
    in the following sections and then
    present the concrete task.
```

Listing 3: Introduction section of our preprompt

C.2 Language Syntax

In Listing 4, we give specific syntax of the language of choice, in our case ASP. We do this in order to guide the LLM to produce syntactically correct rules. Supporting this we also give two special indications:

- We tell the LLM to avoid using semicolons in the body of rules. We do this to avoid falling in Prolog syntax, which is very similar to ASP. Although there are ASP rules with semicolons in the body that are correct, these are not necessary for our experiments.
- We give explanations on what an unsafe rule is in ASP. This happens when some variable is "unbound", and is a major source of errors when prompting the LLM.

```
2. Answer Set Programming Syntax

Answer Set Programming (ASP) is a form
    of declarative programming oriented
    towards difficult search problems.
Its syntax and usage can be summarized
    as follows:

Rules: The basic building block of an
    ASP program. A rule has a head and a
     body, and is written in the form:
    Head :- Body.
It means that if the body is true, then
    the head is also true.
Predicates in the Body are separated by
    commas, and not semicolons as in
    Prolog.
For example: flies(tweety) :- bird(
    tweety), not penguin(tweety).

Atoms: These are the basic propositions
    and can be any string of characters
    and numbers starting with a
    lowercase letter.

Literals: An atom or its negation. The
    negation used in ASP is negation as
    failure, denoted by not.
For example, not a means that a cannot
    be proven to be true.

Facts: These are rules without a body,
    stating something that is
    unconditionally true.
For example: bird(tweety).

Constraints: These are rules without a
    head, used to eliminate certain
    answers.
For example, :- not fly(tweety). states
    that any answer set where tweety
    does not fly is not acceptable.

Choice Rules: These allow for the
    generation of multiple answers,
    expressing that atoms can be freely
    chosen to be in the answer set.
For example, {fish(tweety);bird(tweety)}
     :- penguin(tweety). means that the
    penguin tweety may be a fish, a bird
     or none.

Comments: In ASP, comments start with a
    % and continue to the end of the
    line. They are ignored by the ASP
    solver.

In ASP, the use of ; to represent
    logical disjunction in the body of a
     rule is not allowed.
Instead, we need to express disjunction
    through separate rules.

In ASP, a variable in a rule is
    considered unsafe when a variable
    appears in the head of a rule or in
    a negative literal in the body, but
    not in a positive literal.
Examples of Unsafe Variables:

% Example 1: Unbound Y in negated
    predicate, S and T in regular
    predicates
p1(X) :- not q(X,Y), r(S), s(T).

% Example 2: Unbound Y in negated
    predicate, S in regular predicate, T
    in negated predicate
p2(X, B) :- not q(Y), r(S), not s(T).
```

Listing 4: Language Syntax section of our preprompt

C.3 Scene Encoding

The scene encoding explanation shown in Listing 5 and Listing 6, for GQA and CLEVR respectively, is a compressed version of what we have shown here in the appendix, where the predicates are explained and as example we show an except of an scene encoding.

```
3. Scene Representation

Consider the following ASP
    representation for the objects in an
     image.

Object Declaration: Each object(<id>)
    declares a unique object with a
    specific identifier.
For example, object(1279158) declares an
     object with the ID 1279158.
Attributes (has_attr): The has_attr(<
    object_id>, <attribute>, <value>)
    facts define various attributes of
    an object.
For example, has_attr(1279158, class,
    tree) means the object with ID
    1279158 is classified as a "tree".
    Attributes cover a range of
    properties like class, name, color,
    hposition (horizontal position),
    vposition (vertical position), and
    more.
Relations (has_rel): The has_rel(<
    object_id1>, <relation>, <object_id2
    >) facts describe the relationships
    between two objects.
For example, has_rel(1279158,
    to_the_right_of, 1279150) indicates
    that the object 1279158 is to the
    right of the object 1279150.
Each object in this set of facts seems
    to represent an element in a larger
    scene or model,
with attributes that describe its
    characteristics (like class, color,
    position) and relations that
    describe its spatial or logical
    connections to other elements in the
     scene.
For example, objects are classified into
     categories like tree, wheel, wing,
    etc., and their positions are
    described in relation to other
    objects (e.g., to_the_right_of,
    below, near).

An excerpt of an example of such an
    encoding is the following:

object(1120957).
has_attr(1120957, class, jet).
has_attr(1120957, vposition, middle).
object(1120970).
has_attr(1120970, class, window).
has_rel(1120970, to_the_right_of,
    1120977).
has_rel(1120970, to_the_right_of,
    1120980).
object(1120964).
has_attr(1120964, class, shirt).
has_attr(1120964, name, shirt).
has_attr(1120964, vposition, middle).
has_rel(1120964, to_the_left_of,
    1120963).
...
```

Listing 5: Scene Encoding section of our preprompt for GQA

```
3. Scene Representation

Consider the following ASP
    representation for the objects in an
     image.

It consists of possibly multiple
    predicates of the form 'obj(ID,X,Y,M,
    C,F,S)',
where 'ID' is a unique ID that defines
    the object, 'X, Y' are coordinates
    between 0 and 10,
'M' is a material, 'C' is a color, 'F'
    is a form and 'S' is a size.
An excerpt of an example of such an
    encoding is the following:

obj(0,324,201,rubber,purple,sphere,large
    ).
obj(1,282,166,rubber,purple,cylinder,
    small).
obj(2,216,94,metal,blue,sphere,large).
obj(3,127,115,metal,green,cube,large).
...
```

Listing 6: Scene Encoding section of our preprompt for CLEVR

C.4 Question Encoding

Similarly, Listing 7 and Listing 8, for GQA and CLEVR respectively, gives a description of all the predicates that can appear in a question encoding and then we give five examples of how these look like.

```
4. Question Representation

Consider the following ASP
    representation for natural language
    questions.
The representation shows the steps
    needed to solve the question.
The first number of each predicate
    indicates the output step.
The rest of the numbers indicate the
    input steps.
```

```
In this way chains of predicates are
    joint together to represent the
    reasoning steps needed to answer the
     question.

scene(S): Initializes all objects at
    step S.
select(S, I, O): Selects object O at
    step S, based on input I.
relate(S, I, O, R, Sub): Establishes
    relationship R between O and Sub at
    S.
unique(S, I): Asserts uniqueness of an
    object from I at S.
query(S, I, A): Queries attribute A of
    an object from I at S.
end(S): Concludes query at S.
filter(S, I, A, V): Filters objects by A
     and V at S.
relate_any(S, I, R, Sub): Establishes a
    general R at S.
filter_any(S, I, A): Filters objects by
    A presence at S.
negate(S, I, P): Negates condition from
    I at S.
exist(S, I): Checks existence of I at S.
verify_attr(S, I, A, V): Verifies A with
     V of I at S.
all_same(S, I, A): Checks if all from I
    share A.
choose_attr(S, I, A, O1, O2): Chooses
    between O1 and O2 for A at S.
choose_rel(S, I, O, R1, R2, Sub):
    Chooses between R1 and R2 at S.
common(S, I1, I2): Finds commonality
    between I1 and I2 at S.
two_different(S, I1, I2, A): Compares I1
     and I2 for different A at S.
two_same(S, I1, I2, A): Compares I1 and
    I2 for same A at S.
and(S, I1, I2): Logical AND of I1 and I2
     at S.
or(S, I1, I2): Logical OR of I1 and I2
    at S.
compare(S, I1, I2, C, B): Compares I1
    and I2 based on condition C and
    boolean B at S.

Examples of questions are the following,
     separated by rows of #:
scene(0).
select(1, 0, animal).
all_same(2, 1, class).
end(2).
#########
scene(0).
all_different(2, 1, class).
end(2).
#########
scene(0).
select(1, 0, pizza).
exist(2, 1).
end(2).
#########
scene(0).
select(1, 0, bed).
exist(2, 1).
end(2).
#########
scene(0).
select(1, 0, animal).
all_same(2, 1, class).
end(2).
#########
```

Listing 7: Question Encoding section of our preprompt

```
4. Question Representation

Consider the following ASP
    representation for natural language
    questions.
The representation shows the steps
    needed to solve the question.
The first number of each predicate
    indicates the output step.
The rest of the numbers indicate the
    input steps.
In this way chains of predicates are
    joint together to represent the
    reasoning steps needed to answer the
     question.

scene(X):
   Indicates the start of a new scene or
        a situation.
   X: Identifier for the scene.

filter_property(X)
   Applies a filter based on a property
       (e.g., color, size, material) to
       objects in the scene.
   X: Identifier for the filter
       operation, or a specific property
        value.

unique(X)
   Specifies that the previous filter
       operation results in a unique
       object.
   X: Identifier linked to the outcome
       of the filter.

relate_direction(X):
   Defines a spatial relationship
       between objects.
   X: Specifies the identifier for the
       relationship.

and(X, Y)
   Logical AND operation, often used to
       combine conditions.
   X, Y: Identifiers of the conditions
       or scenes to be combined.

and(X, Y)
   Logical OR operation, often used to
       combine conditions.
   X, Y: Identifiers of the conditions
```

```
       or scenes to be combined.

query_attribute(X)
   Requests information about a
       particular attribute (e.g., color,
        material, shape) of the object.
   X: Identifier for the query operation.


exist(X)
   Asserts the existence of objects
       meeting previous criteria.
   X: Optionally identifies the
       particular existence check.

end(X)
   Marks the end of the query or scene
       description.
   X: Identifier that concludes the
       scene or query.

equal_[attribute](X, Y)
   Compares two attributes to assert
       equality.
   X, Y: Identifiers of the attributes
       or scenes being compared.

same_attribute
   Asserts that two or more objects
       share the same attribute (e.g.,
       size, material).
   X: Identifier for the comparison
       operation.

count(X)
   Count the objects that meet the
       criteria set before this
       predicate.
   X: Identifier for the count operation.


Examples of questions are the following,
     separated by rows of #:
scene(0).
filter_small(1).
filter_cyan(2).
filter_rubber(3).
unique(4).
relate_right(5).
scene(6).
filter_gray(7).
filter_metal(8).
unique(9).
relate_behind(10).
and(11,6).
unique(12).
query_shape(13).
end(14).
##############
scene(0).
filter_green(1).
filter_metal(2).
filter_cylinder(3).
unique(4).
relate_front(5).
filter_cylinder(6).
unique(7).
relate_left(8).
filter_small(9).
filter_rubber(10).
filter_sphere(11).
unique(12).
relate_front(13).
filter_metal(14).
filter_cube(15).
unique(16).
query_color(17).
end(18).
##############
scene(0).
filter_large(1).
filter_green(2).
filter_metal(3).
unique(4).
same_shape(5).
filter_small(6).
filter_yellow(7).
exist(8).
end(9).
##############
scene(0).
filter_blue(1).
unique(2).
relate_right(3).
filter_rubber(4).
filter_cylinder(5).
count(6).
scene(7).
filter_large(8).
filter_rubber(9).
filter_cube(10).
unique(11).
relate_behind(12).
filter_brown(13).
filter_rubber(14).
filter_cylinder(15).
count(16).
equal_integer(17,7).
end(18).
##############
```

Listing 8: Question Encoding section of our preprompt

C.5 Answer Format

The expected format in which the answer must be presented is explained in Listing 9. We give some examples of how these answer may look as well.

```
5. Answer Representation.

The answer computed by the theory will
    always be in a predicate of the form
     'ans(X)', where 'X' is the expected
     answer.
'X' can be a boolean, yes or no, it can
    be an attribute, it can be a
    relation or the value of some
    attribute.
```

```

Examples of these are:
ans(yes)
ans(red)
ans(man)
ans(left)
ans(pose)
```

Listing 9: Answer Format section of our preprompt

C.6 Task

Finally, Listing 10 explains with detail to the LLM what is expected from it. We describe the input prompt and present several rules that the LLM should adhere strictly.

```
7. Your Task.

Your task is to keep the ASP theory
    updated with rules that allows us to
     handle questions.
We provide an initial theory that can
    handle some instances.
The prompt input will consist of one or
    more questions in the ASP
    representation.
If there is a single question, it will
    come with its associated ASP scene
    encoding.
If there are multiple questions, they
    are separeted by # symbols.

Your task is to take questions and add
    rules to the theory such that it is
    able to handle them.

Strictly follow these guidelines:
1. Only output the new ASP Rules.
2. Do not add output ASP facts.
3. New rules should be as general as
    possible, i.e., have a low number of
     constants and high number of
    variables.
4. New rules should be as general as
    possible.
5. Do not use Code Block Format when
    returning the new rules, just plain
    text.
6. Do not add any unsafe rules.
7. You may add multiple rules if you are
     not sure, but these should not
    contradict themselves.
8. Do not add comments nor output
    explanations.
9. Do not output any natural language.
```

Listing 10: Task section of our preprompt

D Experiments

We present a table for the decomposed score for the GS-VQA evaluation, the light. medium and heavy theories for our knowledge distillation experiments and responses in the form of rules obtained from the LLMs.

D.1 GS-VQA Decomposed Results

In Table 10, we present the accuracy of the GS-VQA pipeline on fragments of the GQA test-dev that either belong to a specific question type (binary, i.e., answered with “yes”/“no”, or open), or include a specific reasoning operation. We see that the accuracy on binary questions is considerably higher than that on open questions. This result is both consistent with the literature (Amizadeh et al. 2020) and expected, since the space of possible answers for binary questions is significantly smaller. From the accuracy results of all questions containing a certain operation, we can similarly derive the expected result that the operations with the highest number of possible outputs ($common()$: all possible attribute categories; $relate\_any()$: all possible objects in the scene; $query\_attr()$: all possible values for an attribute category) generally lead to the lowest accuracy values.

By Question Type	
All	39.50%
Binary	55.13%
Open	30.64%
By Operation	
$select()$	39.50%
$filter()$	41.56%
$filter\_any()$	40.64%
$relate()$	36.88%
$relate\_any()$	8.59%
$relate\_attr()$	24.00%
$compare()$	39.13%
$query\_attr()$	23.43%
$query\_name()$	33.75%
$verify\_attr()$	56.81%
$verify\_rel()$	53.84%
$choose\_attr()$	62.49%
$choose\_rel()$	49.76%
$exist()$	55.52%
$two\_same()$	50.00%
$two\_different()$	41.30%
$all\_same()$	76.19%
$all\_different()$	60.00%
$common()$	6.25%
$and()$	53.62%
$or()$	52.80%

Table 10: The accuracy of the GS-VQA pipeline on the GQA test-dev set, shown split by question type and operation occurrence.

D.2 Theories

We now present the theories used in our batch experiments in Section 5. These are in order of the number of rules contained, the Light, Medium and Heavy theories.

Light Theory Listing 11 shows the Lght theory for GQA, the smallest handcrafted theory we use. It contains only 10 rules. We included the predicates that handle how to start and end the process ($scene, end, ans$), as well as those that handle $select, filter$ as an example for the LLM to expand on.

Listing 11: The light ASP theory for GQA

```
is_attr_value(ID, X) :- query(TO, TI, ATTR), state(TI, ID), has_attr(ID, ATTR, X).
is_attr(X) :- query(TO, TI, ATTR), state (TI, ID), has_attr(ID, ATTR, X).

state(TO,ID) :- scene(TO), object(ID).

state(TO,ID) :- select(TO, TI, CLASS), state(TI, ID), has_attr(ID, class, CLASS).
state(TO,ID) :- filter(TO, TI, ATTR, VALUE), state(TI, ID), has_attr(ID, ATTR, VALUE).
{state(TO,ID): state(TI,ID)} = 1 :- unique(TO, TI).

ans(V) :- end(TO), attr_value(TO,V).
ans(V) :- end(TO), attr(TO,V).
ans(V) :- end(TO), rel(TO,V).
ans(V) :- end(TO), bool(TO,V).
```

Listing 12 shows the Lght theory for CLEVR. It contains 20 rules. We included the predicates that handle how to start and end the process ($scene, end, ans$), as well as auxiliary predicates like $has_{s}ize, has_{c}olor$ that decompose the object predicate.

Listing 12: The light ASP theory for CLEVR

```
% State rules
state(0,ID) :- object(ID).
state(T+1,ID) :- scene(T), object(ID).

scene(X) :- scene(X,Y).

object(ID) :- obj(ID,_,_,_,_,_,_).
position(ID,X,Y) :- obj(ID,X,Y,_,_,_,_).
has_size(ID,SIZE) :- obj(ID,_,_,_,_,_, SIZE).
has_color(ID,COLOR) :- obj(ID,_,_,_, COLOR,_,_).
has_material(ID,MATERIAL):- obj(ID,_,_, MATERIAL,_,_,_).
has_shape(ID,SHAPE) :- obj(ID,_,_,_,_, SHAPE,_).

left_of(ID,ID') :- position(ID,X,Y), position(ID',X',Y'), state(T',ID'), ID!=ID', X<X'.
right_of(ID,ID') :- position(ID,X,Y), position(ID',X',Y'), state(T',ID'), ID!=ID', X>=X'.
in_front_of(ID,ID') :- position(ID,X,Y), position(ID',X',Y'), state(T',ID'), ID!=ID', Y>Y'.
behind_of(ID,ID') :- position(ID,X,Y), position(ID',X',Y'), state(T',ID'), ID!=ID', Y<=Y'.

% Derive answer (T must equal the last point in time)
ans(V) :- end(T), size(T,V).
ans(V) :- end(T), color(T,V).
ans(V) :- end(T), material(T,V).
ans(V) :- end(T), shape(T,V).
ans(V) :- end(T), bool(T,V).
ans(V) :- end(T), int(T,V).

:- not ans(_).

#show ans/1.

% Added rules to handle new instances
```

Medium Theory Listing 13 shows the subsequent Medium theory for GQA, that in addition to all the content of the Light theory, also contains rules that manage any intermediary operations, as shown in the comments of Listing 2. These include rules to handle $relate, relate\_any, relate\_attr$, etc.

Listing 13: The medium ASP theory for GQA

```
is_attr_value(ID, X) :- query(TO, TI, ATTR), state(TI, ID), has_attr(ID, ATTR, X).
is_attr(X) :- query(TO, TI, ATTR), state (TI, ID), has_attr(ID, ATTR, X).

state(TO,ID) :- scene(TO), object(ID).

state(TO,ID) :- select(TO, TI, CLASS), state(TI, ID), has_attr(ID, class, CLASS).
state(TO,ID) :- filter(TO, TI, ATTR, VALUE), state(TI, ID), has_attr(ID, ATTR, VALUE).
{state(TO,ID): state(TI,ID)} = 1 :- unique(TO, TI).

state(TO,ID) :- filter_any(TO, TI, VALUE), state(TI, ID), has_attr(ID, ATTR, VALUE).
state(TO, ID') :- relate(TO, TI, CLASS, REL, subject), state(TI, ID), has_attr(ID', class, CLASS), has_rel(ID', REL, ID).
state(TO, ID') :- relate(TO, TI, CLASS, REL, object), state(TI, ID), has_attr(ID', class, CLASS), has_rel(ID, REL, ID').
state(TO, ID') :- relate_any(TO, TI, REL, subject), state(TI, ID), has_rel( ID', REL, ID).
state(TO, ID') :- relate_any(TO, TI, REL, object), state(TI, ID), has_rel( ID, REL, ID').
state(TO, ID') :- relate_attr(TO, TI, CLASS, ATTR), state(TI, ID), has_attr(ID, ATTR, VALUE),
    ↪ has_attr(ID', class, CLASS),
    ↪ has_attr(ID', ATTR, VALUE'), VALUE
    ↪ ==VALUE', ID!=ID'.

ans(V)  :- end(TO), attr_value(TO,V).
ans(V)  :- end(TO), attr(TO,V).
ans(V)  :- end(TO), rel(TO,V).
ans(V)  :- end(TO), bool(TO,V).
```

Listing 14 shows the Medium theory for CLEVR, that in addition to all the content of the Light theory, also contains exemplary rules that manage some other intermediate predicates. These include rules to handle $filter\_large, query\_size, equal\_integer$, etc.

Listing 14: The medium ASP theory for CLEVR

```

% Spatial relation rules
state(T+1,ID) :- relate_left(T), state(T,
    ↪ ID'), left_of(ID,ID').
% Filtering rules
state(T+1,ID) :- filter_large(T), state(
    ↪ T,ID), has_size(ID,large).


% Query functions
size(T+1,SIZE) :- query_size(T), state(T,
    ↪ ID), has_size(ID,SIZE).


% Same-attribute relations
state(T+1,ID') :- same_size(T), state(T,
    ↪ ID), has_size(ID,SIZE), has_size(
    ↪ ID',SIZE), ID!=ID'.


% % Integer comparison
bool(T+1,yes) :- equal_integer(T,T'),
    ↪ int(T,V), int(T',V'), V=V'.
bool(T+1,no) :- equal_integer(T,T'), not
    ↪  bool(T+1,yes).


% Attribute comparison
bool(T+1,yes) :- equal_size(T,T'), size(
    ↪ T,V), size(T',V'), V=V'.
bool(T+1,no) :- equal_size(T,T'), not
    ↪ bool(T+1,yes).



% State rules
state(0,ID) :- object(ID).
state(T+1,ID) :- scene(T), object(ID).

scene(X) :- scene(X,Y).

object(ID) :- obj(ID,_,_,_,_,_,_).
position(ID,X,Y) :- obj(ID,X,Y,_,_,_,_).
has_size(ID,SIZE) :- obj(ID,_,_,_,_,_,
    ↪ SIZE).
has_color(ID,COLOR) :- obj(ID,_,_,_,
    ↪ COLOR,_,_).
has_material(ID,MATERIAL):- obj(ID,_,_,
    ↪ MATERIAL,_,_,_).
has_shape(ID,SHAPE) :- obj(ID,_,_,_,_,
    ↪ SHAPE,_).

left_of(ID,ID') :- position(ID,X,Y),
    ↪ position(ID',X',Y'), state(T',ID')
    ↪ , ID!=ID', X<X'.
right_of(ID,ID') :- position(ID,X,Y),
    ↪ position(ID',X',Y'), state(T',ID')
    ↪ , ID!=ID', X>=X'.
in_front_of(ID,ID') :- position(ID,X,Y),
    ↪  position(ID',X',Y'), state(T',ID
    ↪ '), ID!=ID', Y>Y'.
behind_of(ID,ID') :- position(ID,X,Y),
    ↪ position(ID',X',Y'), state(T',ID')
    ↪ , ID!=ID', Y<=Y'.

% Derive answer (T must equal the last
    ↪ point in time)
ans(V) :- end(T), size(T,V).
ans(V) :- end(T), color(T,V).
ans(V) :- end(T), material(T,V).
ans(V) :- end(T), shape(T,V).
ans(V) :- end(T), bool(T,V).
ans(V) :- end(T), int(T,V).

:- not ans(_).

#show ans/1.

% Added rules to handle new instances
```

Heavy Theory Listing 15 shows the Heavy theory for GQA, which contains all rules from the Medium theory and additionally, the rules to handle any utility function, such as *or*, *and*, *negate* and *unique*.

Listing 15: The heavy ASP theory for GQA

```
is_attr_value(ID, X) :- query(TO, TI,
    ↪ ATTR), state(TI, ID), has_attr(ID,
    ↪  ATTR, X).
is_attr(X) :- query(TO, TI, ATTR), state
    ↪ (TI, ID), has_attr(ID, ATTR, X).

state(TO,ID) :- scene(TO), object(ID).

state(TO,ID) :- select(TO, TI, CLASS),
    ↪ state(TI, ID), has_attr(ID, class,
    ↪  CLASS).
state(TO,ID) :- filter(TO, TI, ATTR,
    ↪ VALUE), state(TI, ID), has_attr(ID
    ↪ , ATTR, VALUE).
{state(TO,ID): state(TI,ID)} = 1 :-
    ↪ unique(TO, TI).

state(TO,ID) :- filter_any(TO, TI, VALUE
    ↪ ), state(TI, ID), has_attr(ID,
    ↪ ATTR, VALUE).
state(TO, ID') :- relate(TO, TI, CLASS,
    ↪ REL, subject), state(TI, ID),
    ↪ has_attr(ID', class, CLASS),
    ↪ has_rel(ID', REL, ID).
```

```
state(TO, ID') :- relate(TO, TI, CLASS, REL, object), state(TI, ID), has_attr(ID', class, CLASS), has_rel(ID, REL, ID').
state(TO, ID') :- relate_any(TO, TI, REL, subject), state(TI, ID), has_rel(ID', REL, ID).
state(TO, ID') :- relate_any(TO, TI, REL, object), state(TI, ID), has_rel(ID, REL, ID').
state(TO, ID') :- relate_attr(TO, TI, CLASS, ATTR), state(TI, ID), has_attr(ID, ATTR, VALUE), has_attr(ID', class, CLASS), has_attr(ID', ATTR, VALUE'), VALUE==VALUE', ID!=ID'.

bool(TO,yes) :- and(TO, TI0, TI1), bool(TI0,yes), bool(TI1,yes).
bool(TO,no) :- and(TO, TI0, TI1), not bool(TO,yes).
bool(TO,yes) :- or(TO, TI0, TI1), bool(TI0,yes).
bool(TO,yes) :- or(TO, TI0, TI1), bool(TI1,yes).
bool(TO,no) :- or(TO, TI0, TI1), not bool(TO,yes).
{state(TO,ID): state(TI,ID)} = 1 :- unique(TO, TI).
:~ unique(TO, TI), state(TO,ID), has_obj_weight(ID, P). [P, (TO, ID)]
state(TO, ID) :- negate(TO, TI0, TI1), state(TI1, ID), not state(TI0, ID).

ans(V) :- end(TO), attr_value(TO,V).
ans(V) :- end(TO), attr(TO,V).
ans(V) :- end(TO), rel(TO,V).
ans(V) :- end(TO), bool(TO,V).
```

Last, Listing 16 shows the Heavy theory for CLEVR, which contains all rules from the Medium theory and expands upon them, adding more rules similar to the ones included in Medium. For example, now the theory contains an array of rules to handle filtering questions such as *filter_small*, *filter_red*, *filter_brown*, etc.

Listing 16: The heavy ASP theory for CLEVR

```
% Uniqueness rule/constraint
state(T+1,ID) :- unique(T), state(T,ID).
:- unique(T), state(T,ID), state(T,ID'), ID!=ID'.

% Spatial relation rules
state(T+1,ID) :- relate_left(T), state(T,ID'), left_of(ID,ID').
state(T+1,ID) :- relate_right(T), state(T,ID'), right_of(ID,ID').

% Count rule
int(T+1,V) :- count(T), #count{ ID : state(T,ID) } = V.


% Exist rule
bool(T+1,yes) :- exist(T), state(T,ID).
bool(T+1,no) :- exist(T), not bool(T+1,yes).

% Filtering rules
state(T+1,ID) :- filter_large(T), state(T,ID), has_size(ID,large).
state(T+1,ID) :- filter_small(T), state(T,ID), has_size(ID,small).
state(T+1,ID) :- filter_gray(T), state(T,ID), has_color(ID,gray).
state(T+1,ID) :- filter_red(T), state(T,ID), has_color(ID,red).
state(T+1,ID) :- filter_blue(T), state(T,ID), has_color(ID,blue).
state(T+1,ID) :- filter_green(T), state(T,ID), has_color(ID,green).
state(T+1,ID) :- filter_brown(T), state(T,ID), has_color(ID,brown).

% Query functions
size(T+1,SIZE) :- query_size(T), state(T,ID), has_size(ID,SIZE).
color(T+1,COLOR) :- query_color(T), state(T,ID), has_color(ID,COLOR).

% Logical operators
state(T+1,ID) :- and(T,T'), state(T,ID), state(T',ID).

state(T+1,ID) :- or(T,T'), state(T,ID).
state(T+1,ID') :- or(T,T'), state(T',ID').

bool(T+1, yes) :- boolean_negation(T), bool(T, no).
bool(T+1, no) :- boolean_negation(T), not bool(T+1, yes).

% Same-attribute relations
state(T+1,ID') :- same_size(T), state(T,ID), has_size(ID,SIZE), has_size(ID',SIZE), ID!=ID'.
state(T+1,ID') :- same_color(T), state(T,ID), has_color(ID,COLOR), has_color(ID',COLOR), ID!=ID'.

% % Integer comparison
bool(T+1,yes) :- equal_integer(T,T'), int(T,V), int(T',V'), V=V'.
bool(T+1,no) :- equal_integer(T,T'), not bool(T+1,yes).

bool(T+1,yes) :- less_than(T,T'), int(T,V), int(T',V'), V<V'.
bool(T+1,no) :- less_than(T,T'), not bool(T+1,yes).


% Attribute comparison
bool(T+1,yes) :- equal_size(T,T'), size(T,V), size(T',V'), V=V'.
bool(T+1,no) :- equal_size(T,T'), not bool(T+1,yes).

bool(T+1,yes) :- equal_color(T,T'),
    ↪ color(T,V), color(T',V'), V=V'.
bool(T+1,no) :- equal_color(T,T'), not
    ↪ bool(T+1,yes).


% State rules
state(0,ID) :- object(ID).
state(T+1,ID) :- scene(T), object(ID).

scene(X) :- scene(X,Y).

object(ID) :- obj(ID,_,_,_,_,_,_).
position(ID,X,Y) :- obj(ID,X,Y,_,_,_,_).
has_size(ID,SIZE) :- obj(ID,_,_,_,_,_,
    ↪ SIZE).
has_color(ID,COLOR) :- obj(ID,_,_,_,
    ↪ COLOR,_,_).
has_material(ID,MATERIAL):- obj(ID,_,_,
    ↪ MATERIAL,_,_,_).
has_shape(ID,SHAPE) :- obj(ID,_,_,_,_,
    ↪ SHAPE,_).

left_of(ID,ID') :- position(ID,X,Y),
    ↪ position(ID',X',Y'), state(T',ID')
    ↪ , ID!=ID', X<X'.
right_of(ID,ID') :- position(ID,X,Y),
    ↪ position(ID',X',Y'), state(T',ID')
    ↪ , ID!=ID', X>=X'.
in_front_of(ID,ID') :- position(ID,X,Y),
    ↪  position(ID',X',Y'), state(T',ID
    ↪ '), ID!=ID', Y>Y'.
behind_of(ID,ID') :- position(ID,X,Y),
    ↪ position(ID',X',Y'), state(T',ID')
    ↪ , ID!=ID', Y<=Y'.

% Derive answer (T must equal the last
    ↪ point in time)
ans(V) :- end(T), size(T,V).
ans(V) :- end(T), color(T,V).
ans(V) :- end(T), material(T,V).
ans(V) :- end(T), shape(T,V).
ans(V) :- end(T), bool(T,V).
ans(V) :- end(T), int(T,V).

:- not ans(_).

#show ans/1.

% Added rules to handle new instances
```

D.3 LLM Communication Examples

We show now examples of the prompts we use with the LLM for GQA and CLEVR, alongside responses we got for the different experiments conducted.

Prompt We present examples of the prompts we give to the LLMs. The ones in Listing 17 and Listing 18 contains the prompt $p = (Q, S, A)$ for GQA and CLEVR respectively, and the following ones seen in Listing 19 and Listing 20, for GQA and CLEVR respectively, used for our batch experiments, contains the prompt $p = \{Q_1, \ldots, Q_b\}$, with b the batch size. We present an example where $b = 10$. Observe that for GQA, the scene representation is much bigger than the question representation, hence why it was dropped for the batch experiments, as the context size would not be enough for most models. We do the same for CLEVR to reduce the context used as well.

```
%Encoded Question
scene(0).
select(1, 0, furniture).
unique(2, 1).
query(3, 2, name).
end(3).
%Encoded Scene
object(4295542).
has_attr(4295542, class, sofa).
has_attr(4295542, name, sofa).
has_attr(4295542, class, furniture).
has_attr(4295542, color, beige).
has_attr(4295542, hposition, middle).
has_attr(4295542, vposition, middle).
has_rel(4295542, on, 4295539).
object(4295537).
has_attr(4295537, class, hair).
has_attr(4295537, name, hair).
has_attr(4295537, class, thing).
has_attr(4295537, color, gray).
has_attr(4295537, hposition, middle).
has_attr(4295537, vposition, top).
object(4295540).
has_attr(4295540, class, blanket).
has_attr(4295540, name, blanket).
has_attr(4295540, class, textile).
has_attr(4295540, any, plaid).
has_attr(4295540, hposition, right).
has_attr(4295540, vposition, middle).
has_rel(4295540, to_the_right_of,
    4295534).
has_rel(4295540, to_the_right_of,
    4071107).
has_rel(4295540, to_the_right_of,
    4071103).
has_rel(4295540, to_the_right_of,
    4295533).
has_rel(4295540, to_the_right_of,
    4295536).
has_rel(4295540, to_the_right_of,
    4295586).
has_rel(4295540, to_the_right_of,
    4295557).
object(4295552).
has_attr(4295552, class, pillow).
has_attr(4295552, name, pillow).
has_attr(4295552, class, object).
has_attr(4295552, color, tan).
has_attr(4295552, hposition, left).
has_attr(4295552, vposition, middle).
has_rel(4295552, to_the_left_of,
    4071103).
has_rel(4295552, to_the_left_of,
    4295589).
has_rel(4295552, behind, 4295535).
has_rel(4295552, to_the_left_of,
    4295535).
```

```
object(4071111).
has_attr(4071111, class, foot).
has_attr(4071111, name, foot).
has_attr(4071111, any, bare).
has_attr(4071111, hposition, right).
has_attr(4071111, vposition, bottom).
has_attr(4071111, vposition, middle).
has_rel(4071111, of, 4071103).
object(4295557).
has_attr(4295557, class, finger).
has_attr(4295557, name, finger).
has_attr(4295557, hposition, middle).
has_attr(4295557, vposition, bottom).
has_attr(4295557, vposition, middle).
has_rel(4295557, to_the_left_of,
    4295540).
object(4295586).
has_attr(4295586, class, sweater).
has_attr(4295586, name, sweater).
has_attr(4295586, class, clothing).
has_attr(4295586, color, red).
has_attr(4295586, hposition, left).
has_attr(4295586, vposition, middle).
has_rel(4295586, around, 4295589).
has_rel(4295586, to_the_left_of,
    4295540).
object(4071107).
has_attr(4071107, class, book).
has_attr(4071107, name, book).
has_attr(4071107, class, object).
has_attr(4071107, hposition, right).
has_attr(4071107, vposition, middle).
has_rel(4071107, to_the_left_of,
    4295540).
has_rel(4071107, to_the_right_of,
    4295536).
object(4295535).
has_attr(4295535, class, woman).
has_attr(4295535, name, woman).
has_attr(4295535, class, person).
has_attr(4295535, class, she).
has_attr(4295535, hposition, middle).
has_attr(4295535, vposition, middle).
has_rel(4295535, sitting_on, 4295542).
has_rel(4295535, lying_on, 4295542).
has_rel(4295535, on, 4295542).
has_rel(4295535, in_front_of, 4295552).
has_rel(4295535, to_the_right_of,
    4295552).
has_rel(4295535, to_the_right_of,
    4295538).
object(4295534).
has_attr(4295534, class, jeans).
has_attr(4295534, name, jeans).
has_attr(4295534, class, clothing).
has_attr(4295534, hposition, middle).
has_attr(4295534, vposition, bottom).
has_attr(4295534, vposition, middle).
has_rel(4295534, to_the_left_of,
    4295540).
has_rel(4295534, to_the_left_of,
    4295562).
has_rel(4295534, to_the_right_of,
    4295538).
object(4295533).
has_attr(4295533, class, dress).
has_attr(4295533, name, dress).
has_attr(4295533, class, clothing).
has_attr(4295533, color, yellow).
has_attr(4295533, hposition, middle).
has_attr(4295533, vposition, bottom).
has_attr(4295533, vposition, middle).
has_rel(4295533, to_the_left_of,
    4295540).
object(4071103).
has_attr(4071103, class, girl).
has_attr(4071103, name, girl).
has_attr(4071103, class, person).
has_attr(4071103, class, she).
has_attr(4071103, color, blond).
has_attr(4071103, hposition, middle).
has_attr(4071103, vposition, bottom).
has_attr(4071103, vposition, middle).
has_rel(4071103, to_the_left_of,
    4295540).
has_rel(4071103, to_the_right_of,
    4295552).
has_rel(4071103, to_the_left_of,
    4295562).
has_rel(4071103, to_the_right_of,
    4295538).
object(4295536).
has_attr(4295536, class, shirt).
has_attr(4295536, name, shirt).
has_attr(4295536, class, clothing).
has_attr(4295536, color, maroon).
has_attr(4295536, hposition, middle).
has_attr(4295536, vposition, middle).
has_rel(4295536, to_the_left_of,
    4071107).
has_rel(4295536, to_the_left_of,
    4295540).
object(4295562).
has_attr(4295562, class, foot).
has_attr(4295562, name, foot).
has_attr(4295562, any, bare).
has_attr(4295562, hposition, right).
has_attr(4295562, vposition, bottom).
has_attr(4295562, vposition, middle).
has_rel(4295562, to_the_right_of,
    4071103).
has_rel(4295562, to_the_right_of,
    4295534).
object(4295589).
has_attr(4295589, class, neck).
has_attr(4295589, name, neck).
has_attr(4295589, hposition, middle).
has_attr(4295589, vposition, middle).
has_rel(4295589, to_the_right_of,
    4295552).
object(4295539).
has_attr(4295539, class, floor).
has_attr(4295539, name, floor).
has_attr(4295539, class,
    urban_environment).
has_attr(4295539, class, place).
has_attr(4295539, any, hardwood).
has_attr(4295539, material, wood).
has_attr(4295539, hposition, middle).
has_attr(4295539, vposition, middle).
```

```
object(4295538).
has_attr(4295538, class, rug).
has_attr(4295538, name, rug).
has_attr(4295538, class, thing).
has_attr(4295538, color, blue).
has_attr(4295538, hposition, left).
has_attr(4295538, vposition, bottom).
has_attr(4295538, vposition, middle).
has_rel(4295538, on, 4295539).
has_rel(4295538, to_the_left_of,
    4295535).
has_rel(4295538, to_the_left_of,
    4071103).
has_rel(4295538, to_the_left_of,
    4295534).

%Expected Answer
ans(sofa).
```

Listing 17: Example of a prompt given to the LLM for GQA

```
%Encoded Question
scene(0).
filter_rubber(1).
filter_cylinder(2).
exist(3).
end(4).
%Encoded Scene
obj(0,348,171,rubber,blue,cube,large).
obj(1,257,210,rubber,yellow,cylinder,
    large).
obj(2,224,155,metal,gray,sphere,large).
%Expected Answer
ans(yes).
```

Listing 18: Example of a prompt given to the LLM for CLEVR

```
scene(0).
select(1, 0, baseball).
exist(2, 1).
end(2).
#########
scene(0).
select(1, 0, fence).
exist(2, 1).
end(2).
#########
scene(0).
select(1, 0, animal).
all_same(2, 1, class).
end(2).
#########
scene(0).
select(1, 0, couch).
exist(2, 1).
end(2).
#########
scene(0).
select(1, 0, woman).
exist(2, 1).
end(2).
#########
scene(0).
select(1, 0, animal).
all_different(2, 1, class).
end(2).
#########
scene(0).
select(1, 0, spider).
exist(2, 1).
end(2).
#########
scene(0).
select(1, 0, bicycle).
exist(2, 1).
end(2).
#########
scene(0).
select(1, 0, cell_phone).
exist(2, 1).
end(2).
#########
scene(0).
select(1, 0, bottle).
exist(2, 1).
end(2).
#########
```

Listing 19: Example of a batch prompt given to the LLM for GQA

```
scene(0).
filter_small(1).
filter_cyan(2).
filter_cylinder(3).
unique(4).
relate_left(5).
filter_small(6).
filter_metal(7).
filter_cylinder(8).
unique(9).
relate_behind(10).
filter_small(11).
filter_blue(12).
filter_metal(13).
filter_cube(14).
unique(15).
relate_front(16).
filter_small(17).
filter_cylinder(18).
exist(19).
end(20).
#########
scene(0).
filter_small(1).
filter_rubber(2).
filter_cylinder(3).
exist(4).
end(5).
#########
scene(0).
filter_large(1).
filter_metal(2).
filter_sphere(3).
unique(4).
same_color(5).
exist(6).
end(7).
```

```
#########
scene(0).
filter_gray(1).
filter_metal(2).
filter_sphere(3).
unique(4).
relate_left(5).
filter_metal(6).
filter_sphere(7).
unique(8).
relate_left(9).
filter_large(10).
unique(11).
relate_front(12).
filter_sphere(13).
exist(14).
end(15).
#########
scene(0).
filter_small(1).
filter_gray(2).
filter_sphere(3).
unique(4).
same_material(5).
filter_small(6).
exist(7).
end(8).
#########
scene(0).
filter_small(1).
filter_cyan(2).
unique(3).
relate_left(4).
filter_brown(5).
filter_metal(6).
filter_cylinder(7).
unique(8).
relate_front(9).
filter_large(10).
filter_metal(11).
unique(12).
relate_behind(13).
exist(14).
end(15).
#########
scene(0).
filter_brown(1).
filter_rubber(2).
unique(3).
same_shape(4).
exist(5).
end(6).
#########
scene(0).
filter_gray(1).
filter_metal(2).
filter_cylinder(3).
unique(4).
same_size(5).
exist(6).
end(7).
#########
scene(0).
filter_large(1).
filter_cyan(2).
unique(3).
relate_right(4).
filter_gray(5).
filter_cylinder(6).
exist(7).
end(8).
#########
scene(0).
filter_large(1).
filter_cyan(2).
filter_cylinder(3).
unique(4).
same_material(5).
exist(6).
end(7).
#########
```

Listing 20: Example of a batch prompt given to the LLM for CLEVR

Mending For the syntax and semantic mending, we provide the following two Listings that illustrate the prompts used for the case of the predicate *unique* of CLEVR using the Mistral model. Listing 21 shows the syntax mending prompt which contains the syntax error that the ASP solver gives and the rule to be fixed alongside instructions. Likewise, Listing 22 shows the prompt used for the semantic mending, which contains the incorrect answer obtained and the correct answer, plus the rules needed to be fixed. The prompt also contains the theory without the rule to be mended, which we omit for brevity.

Listing 21: Example of the syntax mending prompt

```
You must repair the syntax of the
    ↪ prompted Answer Set Programming
    ↪ rule(s).
The ASP Solver returned the following
    ↪ error: <block>:84:1-49: error:
    ↪ unsafe variables in:
  bool((T+1),no):-query_size(T);not
    ↪ unique(T,ID).
<block>:84:45-47: note: 'ID' is unsafe.
You must only output the fixed ASP rule(
    ↪ s) and any other rules included in
    ↪  the prompt that have correct
    ↪ syntax.
Do not output any natural language.
The output must be in plain text only!
Do not output the response as a code
    ↪ block!
The rule to be fixed is:
bool((T+1),no):-query_size(T);not unique
    ↪ (T,ID).
```

Listing 22: Example of the semantic mending prompt

```
The following rules were added to the
    ↪ theory to calculate the answer:
state(T+1,ID) :- same_color(T), state(T,
    ↪ ID), has_color(ID,COLOR),
    ↪ has_color(ID',COLOR), ID!=ID'.
They do not calculate the correct answer,
    ↪  which is: red,
but instead result in the following
    ↪ incorrect answer: [] (can me empty
    ↪ ).
You must only output the fixed ASP rule(
    ↪ s).
The rule(s) must be semantically
    ↪ different from the ones prompted
    ↪ and they should now calculate the
    ↪ correct answer.
Do not output any natural language.
The output must be in plain text only!
Do not output the response as a code
    ↪ block!
```

Responses First we present responses we received from the LLM models in the predicate removal experiment. We select responses from a run of each model for the experiments involving the predicates *or* shown in Listing 23, *filter* shown in Listing 24 and *exist* shown in Listing 25 for GQA and responses for the predicates *filter_large* shown in Listing 26, *same_color* shown in Listing 27 and *exist* shown in Listing 28 for CLEVR. Each of the listings contain all the responses from the LLMs, separated via comments. At a glance, is possible to see that GPT4 provides the most succinct yet general responses, which in some cases are exactly what we needed (see for example Listing 23). GPT3.5 responses are in general short, yet they start to be less general and contain constants in them, as shown in Listing 24. Mistral is the one that shows the worst behaviour, by providing rules that are long, winded and usually contain constants in them.

Listing 23: Or Predicate Responses

```
% GPT4 Response
bool(TO, yes) :- or(TO, TI0, TI1), bool(
    ↪ TI0, yes).
bool(TO, yes) :- or(TO, TI0, TI1), bool(
    ↪ TI1, yes).
bool(TO, no) :- or(TO, TI0, TI1), bool(
    ↪ TI0, no), bool(TI1, no).

% GPT3.5 Response
attr_value(TO, VALUE) :- filter(TO, TI,
    ↪ ATTR, VALUE), state(TI, ID),
    ↪ has_attr(ID, ATTR, VALUE).
bool(TO, yes) :- or(TO, TI0, TI1), bool(
    ↪ TI0, yes), bool(TI1, yes).
bool(TO, no) :- or(TO, TI0, TI1), bool(
    ↪ TI0, no), bool(TI1, no).

% Mistral Response
exist_fence(TO) :- scene(TO), select(T1,
    ↪  TO, fence), state(T1, ID),
    ↪ has_rel(ID, _, _).
ans(yes) :- end(TO), exist_fence(TO).
ans(no) :- end(TO), not exist_fence(TO).
has_rel(ID, REL, ID') :- scene(TO),
    ↪ select(T1, TO, CLASS), state(T1,
    ↪ ID),
has_attr(ID, class, CLASS), has_attr(ID',
    ↪  class, CLASS'),
has_rel(ID, REL, ID'), CLASS == airplane,
    ↪  CLASS' == brown.
bool(TO, yes) :- or(TO, TI0, TI1).
attr_value(TO, brown) :- has_rel(ID, REL,
    ↪  ID'), scene(TO), select(T1, TO,
    ↪ CLASS),
state(T1, ID), has_attr(ID, class, CLASS
    ↪ ), has_attr(ID', class, CLASS'),
has_rel(ID, REL, ID'), CLASS == airplane,
    ↪  CLASS' == brown.
ans(brown) :- end(TO), attr_value(TO,
    ↪ brown).
```

Listing 24: Filter Predicate Responses

```
%GPT4 Response
state(TO, ID) :- filter(TO, TI, ATTR,
    ↪ VALUE), state(TI, ID), has_attr(ID
    ↪ , ATTR, VALUE).
rel(TO, REL) :- choose_rel(TO, TI, CLASS,
    ↪  REL, REL', subject), state(TI, ID
    ↪ ), has_attr(ID, class, CLASS),
    ↪ has_rel(ID, REL, _).
rel(TO, REL') :- choose_rel(TO, TI,
    ↪ CLASS, REL, REL', subject), state(
    ↪ TI, ID), has_attr(ID, class, CLASS
    ↪ ), not has_rel(ID, REL, _),
    ↪ has_rel(ID, REL', _).

%GPT3.5 Response
state(TO, ID) :- filter(TO, TI, color,
    ↪ green), state(TI, ID), has_attr(ID
    ↪ , color, green).
state(TO, ID) :- filter(TO, TI, material,
    ↪  plastic), state(TI, ID), has_attr
    ↪ (ID, material, plastic).

%Mistral Response
has_rel(ID, REL) :- select(S, TI, ID),
    ↪ state(TI, ID'), has_attr(ID',
    ↪ class, CLASS), has_rel(ID', REL,
    ↪ ID).
has_rel(ID, REL) :- relate(S, TI, CLASS,
    ↪  REL, subject), state(TI, ID),
    ↪ has_attr(ID, class, CLASS),
    ↪ has_rel(ID, REL, ID').
has_rel(ID, REL) :- relate(S, TI, CLASS,
    ↪  REL, object), state(TI, ID),
    ↪ has_attr(ID, class, CLASS),
    ↪ has_rel(ID', REL, ID).
has_rel(ID, REL) :- relate_any(S, TI,
    ↪ REL, subject), state(TI, ID),
    ↪ has_rel(ID, REL, ID').
has_rel(ID, REL) :- relate_any(S, TI,
    ↪ REL, object), state(TI, ID),
    ↪ has_rel(ID', REL, ID).
state(S, ID) :- filter(S, TI, ATTR,
    ↪ VALUE), state(TI, ID), has_attr(ID
    ↪ , ATTR, VALUE).
attr(S, ATTR, VALUE) :- query(S, TI,
    ↪ ATTR), state(TI, ID), has_attr(ID,
    ↪  ATTR, VALUE).
bool(S, yes) :- attr(S, ATTR, VALUE),
    ↪ filter(S, TI, ATTR, VALUE).
```

```
bool(S, no) :- not attr(S, ATTR, VALUE),
    ↪  filter(S, TI, ATTR, VALUE).
rel(S, REL) :- has_rel(ID, REL), state(S,
    ↪  ID).
ans(yes) :- or(S, TI0, TI1), bool(TI0,
    ↪ yes), bool(TI1, yes).
ans(no) :- or(S, TI0, TI1), not bool(S,
    ↪ yes).
```

Listing 25: Exist Predicate Responses for GQA

```
%GPT4 Response
bool(TO, no) :- exist(TO, TI), not state
    ↪ (TI, _).
bool(TO, yes) :- exist(TO, TI), state(TI,
    ↪  _).
%GPT3.5 Response
state(TO, ID') :- filter(TO, TI, ATTR,
    ↪ VALUE), state(TI, ID), has_attr(ID
    ↪ ', ATTR, VALUE).
state(TO, ID') :- filter_any(TO, TI,
    ↪ VALUE), state(TI, ID), has_attr(ID
    ↪ ', ATTR, VALUE).
bool(TO, yes) :- exist(TO, TI), state(TI,
    ↪  ID').
%Mistral Response
has_rel(ID, color, brown) :-
state(TO, ID),
filter(TO2, TO, color, brown),
TO2 >= TO.
ans(yes) :- or(TO, 3, 7), end(TO).
```

Listing 26: Filter Large Predicate Responses

```
%GPT4 Response
state(T+1,ID) :- filter_large(T), state(
    ↪ T,ID), has_size(ID,large).
size(T+1,SIZE) :- query_size(T), state(T,
    ↪ ID), has_size(ID,SIZE).
%GPT3.5 Response
% Spatial relation rules
state(T+1,ID) :- relate_left(T), state(T,
    ↪ ID'), left_of(ID,ID').
state(T+1,ID) :- relate_behind(T), state
    ↪ (T,ID'), behind_of(ID,ID').

% Filtering rules
state(T+1,ID) :- filter_purple(T), state
    ↪ (T,ID), has_color(ID,purple).
state(T+1,ID) :- filter_metal(T), state(
    ↪ T,ID), has_material(ID,metal).
state(T+1,ID) :- filter_cylinder(T),
    ↪ state(T,ID), has_shape(ID,cylinder
    ↪ ).
state(T+1,ID) :- filter_large(T), state(
    ↪ T,ID), has_size(ID,large).
state(T+1,ID) :- filter_cube(T), state(T,
    ↪ ID), has_shape(ID,cube).

% Exist rule
bool(T+1,yes) :- exist(T), state(T,ID).
bool(T+1,no) :- exist(T), not bool(T+1,
    ↪ yes).
%Mistral Response
% Same material rule for unique step
state(T+1,ID) :- unique(T), state(T,ID),
    ↪  has_material(ID,MATERIAL), state(
    ↪ T,ID1), has_material(ID1,MATERIAL)
    ↪ , ID!=ID1.

% Same color rule for same_material step
state(T+1,ID) :- same_material(T), state
    ↪ (T,ID), has_color(ID,COLOR), state
    ↪ (T,ID1), has_color(ID1,COLOR), ID
    ↪ !=ID1.

% Filtering rule for brown color
state(T+1,ID) :- filter_brown(T), state(
    ↪ T,ID), has_color(ID,brown).

% Filtering rule for large size
state(T+1,ID) :- filter_large(T), state(
    ↪ T,ID), has_size(ID,large).

% Filtering rule for cube shape
state(T+1,ID) :- filter_cube(T), state(T,
    ↪ ID), has_shape(ID,cube).

% Filtering rule for small size
state(T+1,ID) :- filter_small(T), state(
    ↪ T,ID), has_size(ID,small).

% Filtering rule for cylinder shape
state(T+1,ID) :- filter_cylinder(T),
    ↪ state(T,ID), has_shape(ID,cylinder
    ↪ ).

% Query function for color
color(T+1,COLOR) :- query_color(T),
    ↪ state(T,ID), has_color(ID,COLOR).

% Derive answer (T must equal the last
    ↪ point in time)
ans(V) :- end(T), color(T,V).
```

Listing 27: Same Color Predicate Responses

```
%GPT4 Response
state(T+1,ID') :- same_color(T), state(T,
    ↪ ID), has_color(ID,COLOR),
    ↪ has_color(ID',COLOR), ID!=ID'.
%GPT3.5 Response
state(T+1,ID') :- same_color(T), state(T,
    ↪ ID), has_color(ID,COLOR),
    ↪ has_color(ID',COLOR), ID!=ID'.
%Mistral Response
% Safe rule for same_color/2
same_color(T+1,ID) :- same_color(T),
    ↪ state(T,ID'), has_color(ID,C),
    ↪ has_color(ID',C), ID!=ID'.

% Safe rule for query_shape/2
state(T+1,ID) :- query_shape(T),
    ↪ has_shape(ID,SHAPE).

% Safe rule for end/1
ans(SHAPE) :- end(T), state(T,ID),
    ↪ has_shape(ID,SHAPE).
```

Listing 28: Exist Predicate Responses for CLEVR

```
%GPT4 Response
state(T+1,ID) :- same_material(T), state
    ↪ (T,ID), has_material(ID,MATERIAL),
    ↪  obj(ID2,_,_,MATERIAL,_,_,_), ID!=
    ↪ ID2.
bool(T+1,yes) :- exist(T), state(T,ID).
bool(T+1,no) :- exist(T), not state(T,_).
    ↪
%GPT3.5 Response
% Rule to check if there are two objects
    ↪  with the same size and material
same_size_material(T+1,ID') :- same_size
    ↪ (T), state(T,ID), has_size(ID,SIZE
    ↪ ), has_size(ID',SIZE), ID!=ID',
    ↪ has_material(ID,MATERIAL),
    ↪ has_material(ID',MATERIAL).

% Rule to check if there is an object
    ↪ with the specified attributes
exist_attributes(T+1) :- exist(T), state
    ↪ (T,ID), has_size(ID,SIZE),
    ↪ has_color(ID,COLOR), has_material(
    ↪ ID,MATERIAL), has_shape(ID,SHAPE).

% Rule to derive the answer based on the
    ↪  existence of objects with the
    ↪ specified attributes
ans(yes) :- end(T), exist_attributes(T),
    ↪  not same_size_material(T).
ans(no) :- end(T), not ans(yes).
%Mistral Response
% New rules for handling existential
    ↪ quantification
state(T+1,ID) :- exist(T), state(T,ID),
    ↪ ID != 0.

% New rules for handling same shape
    ↪ constraint
same_shape(T+1) :- same_shape(T), state(
    ↪ T,ID1), has_shape(ID1,S), state(T,
    ↪ ID2), has_shape(ID2,S), ID1 != ID2
    ↪ .

% New rules for handling existential
    ↪ quantification with same shape
    ↪ constraint
state(T+1,ID) :- exist(T), same_shape(T),
    ↪  state(T,ID), has_shape(ID,S).

% New rules for handling end of query
ans(yes) :- end(T), exist(T-1).
ans(no) :- end(T), not exist(T-1).
```

We show next responses from out batch experiment. Each theory used, Light, Medium and Heavy, is represented by Listings 29,30,31 for GQA and Listings 32,33,34 for CLEVR. In each of them, the rules learned using different batch sizes are presented ($b = 1, 2, 5, 10$). As the experiment with the Light theory and $b = 10$ did not produce results, it is not included. The high number of rules we receive when using $b = 1$ or $b = 2$ is expected, as the LLM only has to provide rules that can cover those examples, while for bigger batch sizes, we see that the number of rules diminishes drastically, as now the LLM must provide more general rules to pass the semantic check. Regarding the size of the theories used, smaller theories may be able to give more freedom to the LLM to try different approaches to solve a problem, but having more robust initial theory is seemingly better, as this can guide the LLM better to complete the parts missing.

Listing 29: Light Batch Experiment GQA

```
%Batch Size b=1 Response
state(TO, ID) :- exist(TO, TI), state(TI,
    ↪  ID).
bool(TO, yes) :- exist(TO, TI), state(TI,
    ↪  ID).
bool(TO, no) :- exist(TO, TI), not state
    ↪ (TI, _).
state(TO, ID) :- choose_attr(TO, TI,
    ↪ ATTR, V1, V2), state(TI, ID),
    ↪ has_attr(ID, ATTR, V1).
state(TO, ID) :- choose_attr(TO, TI,
    ↪ ATTR, V1, V2), not state(TI, ID1),
    ↪  state(TI, ID), has_attr(ID1, ATTR
    ↪ , V1), has_attr(ID, ATTR, V2).
attr_value(TO, V1) :- choose_attr(TO, TI,
    ↪  ATTR, V1, _), state(TO, ID),
    ↪ has_attr(ID, ATTR, V1).
attr_value(TO, V2) :- choose_attr(TO, TI,
    ↪  ATTR, _, V2), state(TO, ID),
    ↪ has_attr(ID, ATTR, V2).
state(TO, ID) :- filter_any(TO, TI, ATTR
    ↪ ), state(TI, ID), has_attr(ID,
    ↪ ATTR, _).
attr(TO, X) :- query(TO, TI, ATTR),
    ↪ state(TI, ID), has_attr(ID, ATTR,
    ↪ X).
{state(TO, ID) : state(TI, ID), has_attr
    ↪ (ID, ATTR, VALUE)} :- filter(TO,
    ↪ TI, ATTR, VALUE).
state(TO, ID) :- relate(TO, TI, CLASS,
    ↪ REL, Sub), state(TI, ID), has_attr
    ↪ (ID, class, CLASS), has_rel(ID,
    ↪ REL, Sub).
state(TO, Sub) :- relate(TO, TI, _, REL,
    ↪  subject), state(TI, Sub), has_rel
    ↪ (Sub, REL, _).
state(TO, ID) :- relate(TO, TI, CLASS,
    ↪ REL, subject), state(TI, Sub),
    ↪ has_attr(ID, class, CLASS),
    ↪ has_rel(ID, REL, Sub).
state(TO, ID) :- relate(TO, TI, _, REL,
    ↪ subject), state(TI, ID), has_rel(
    ↪ ID, REL, _).
attr_value(TO, X) :- query(TO, TI, ATTR),
    ↪  state(TI, ID), has_attr(ID, ATTR,
    ↪  X).
bool(TO, yes) :- and(TO, TI1, TI2), bool
    ↪ (TI1, yes), bool(TI2, yes).
bool(TO, no) :- and(TO, TI1, TI2), bool(
    ↪ TI1, no).
bool(TO, no) :- and(TO, TI1, TI2), bool(
    ↪ TI2, no).
bool(TO, yes) :- or(TO, TI1, TI2), bool(
    ↪ TI1, yes).
bool(TO, yes) :- or(TO, TI1, TI2), bool(
    ↪ TI2, yes).
bool(TO, no) :- or(TO, TI1, TI2), bool(
    ↪ TI1, no), bool(TI2, no).
state(TO, ID) :- verify_attr(TO, TI,
    ↪ ATTR, VALUE), state(TI, ID),
    ↪ has_attr(ID, ATTR, VALUE).
bool(TO, yes) :- verify_attr(TO, TI,
    ↪ ATTR, VALUE), state(TI, ID),
    ↪ has_attr(ID, ATTR, VALUE).
bool(TO, no) :- verify_attr(TO, TI, ATTR,
    ↪  VALUE), state(TI, ID), not
    ↪ has_attr(ID, ATTR, VALUE).
state(TO, ID) :- and(TO, TI1, TI2),
    ↪ state(TI1, ID), state(TI2, ID).
state(TO, ID) :- and(TO, TI1, TI2),
    ↪ state(TI1, ID), not state(TI2, _).
state(TO, ID) :- and(TO, TI1, TI2), not
    ↪ state(TI1, _), state(TI2, ID).

%Batch Size b=2 Response
state(TO, ID) :- exist(TO, TI), state(TI,
    ↪  ID).
bool(TO, yes) :- exist(TO, TI), state(TI,
    ↪  _).
bool(TO, no) :- exist(TO, TI), not state
    ↪ (TI, _).
state(TO, ID, Sub) :- relate(TO, TI, ID,
    ↪  R, Sub), state(TI, ID), has_rel(
    ↪ ID, R, Sub).
state(TO, Sub) :- relate(TO, TI, ID, R,
    ↪ Sub), state(TI, ID), has_rel(ID, R
    ↪ , Sub).

attr_value(TO, V) :- query(TO, TI, ATTR),
    ↪  state(TI, ID), has_attr(ID, ATTR,
    ↪  V).
bool(TO, yes) :- and(TO, TI1, TI2), bool
    ↪ (TI1, yes), bool(TI2, yes).
bool(TO, no) :- and(TO, TI1, TI2), bool(
    ↪ TI1, no).
bool(TO, no) :- and(TO, TI1, TI2), bool(
    ↪ TI2, no).
bool(TO, yes) :- or(TO, TI1, TI2), bool(
    ↪ TI1, yes).
bool(TO, yes) :- or(TO, TI1, TI2), bool(
    ↪ TI2, yes).
bool(TO, no) :- or(TO, TI1, TI2), bool(
    ↪ TI1, no), bool(TI2, no).

bool(TO, yes) :- verify_attr(TO, TI,
    ↪ ATTR, VALUE), state(TI, ID),
    ↪ has_attr(ID, ATTR, VALUE).
bool(TO, no) :- verify_attr(TO, TI, ATTR,
    ↪  VALUE), state(TI, ID), not
    ↪ has_attr(ID, ATTR, VALUE).

%Batch Size b=5 Response
state(TO, ID) :- negate(TO, TI, _),
    ↪ state(TI, ID), not state(TI, ID).
ans(no) :- end(TO), not exist(_, TO).
ans(yes) :- end(TO), exist(_, TO).
state(TO, ID) :- or(TO, TI1, TI2), state
    ↪ (TI1, ID).
state(TO, ID) :- or(TO, TI1, TI2), state
    ↪ (TI2, ID).
```

Listing 30: Medium Batch Experiment GQA

```
%Batch Size b=1 Response
state(TO, ID) :- exist(TO, TI), state(TI,
    ↪  ID).
ans(yes) :- exist(TO, _), state(TO, _).
ans(no) :- exist(TO, TI), not state(TO,
    ↪ _).
state(TO, ID) :- choose_attr(TO, TI,
    ↪ ATTR, V1, V2), state(TI, ID),
    ↪ has_attr(ID, ATTR, V1).
state(TO, ID) :- choose_attr(TO, TI,
    ↪ ATTR, V1, V2), state(TI, ID),
    ↪ has_attr(ID, ATTR, V2).
ans(V1) :- choose_attr(TO, _, ATTR, V1,
    ↪ _), state(TO, ID), has_attr(ID,
    ↪ ATTR, V1).
ans(V2) :- choose_attr(TO, _, ATTR, _,
    ↪ V2), state(TO, ID), has_attr(ID,
    ↪ ATTR, V2).
state(TO, ID) :- verify_attr(TO, TI,
    ↪ ATTR, VALUE), state(TI, ID),
    ↪ has_attr(ID, ATTR, VALUE).
ans(no) :- verify_attr(TO, TI, ATTR,
    ↪ VALUE), state(TO, ID), not
    ↪ has_attr(ID, ATTR, VALUE).
ans(yes) :- verify_attr(TO, TI, ATTR,
    ↪ VALUE), state(TO, ID), has_attr(ID
    ↪ , ATTR, VALUE).
state(TO, ID) :- relate_any(TO, TI, REL,
    ↪  subject), state(TI, ID), has_rel(
    ↪ ID, REL, _).
state(TO, ID) :- relate_any(TO, TI, REL,
    ↪  object), state(TI, ID), has_rel(_
    ↪ , REL, ID).
attr_value(TO, V) :- query(TO, TI, ATTR),
    ↪  state(TI, ID), has_attr(ID, ATTR,
    ↪  V).
state(TO, ID1, ID2) :- unique(TO, TI1),
    ↪ unique(TO, TI2), state(TI1, ID1),
    ↪ state(TI2, ID2), ID1 != ID2.
ans(no) :- two_different(TO, TI1, TI2,
    ↪ ATTR), state(TI1, ID1), state(TI2,
    ↪  ID2), has_attr(ID1, ATTR, V),
    ↪ has_attr(ID2, ATTR, V).
ans(yes) :- two_different(TO, TI1, TI2,
    ↪ ATTR), state(TI1, ID1), state(TI2,
    ↪  ID2), has_attr(ID1, ATTR, V1),
    ↪ has_attr(ID2, ATTR, V2), V1 != V2.
state(TO, ID1, ID2) :- unique(TO, TI1),
    ↪ unique(TO, TI2), state(TI1, ID1),
    ↪ state(TI2, ID2), ID1 != ID2.
ans(yes) :- two_same(TO, TI1, TI2, ATTR),
    ↪  state(TI1, ID1), state(TI2, ID2),
    ↪  has_attr(ID1, ATTR, V), has_attr(
    ↪ ID2, ATTR, V).
ans(no) :- two_same(TO, TI1, TI2, ATTR),
    ↪  state(TI1, ID1), state(TI2, ID2),
    ↪  has_attr(ID1, ATTR, V1), has_attr
    ↪ (ID2, ATTR, V2), V1 != V2.

%Batch Size b=2 Response
state(TO, ID) :- relate(TO, TI, CLASS,
    ↪ REL, subject), state(TI, ID),
    ↪ has_attr(ID, class, CLASS),
    ↪ has_rel(ID, REL, ID2), object(ID2)
    ↪ .
state(TO, ID) :- relate(TO, TI, CLASS,
    ↪ REL, object), state(TI, ID2),
    ↪ has_attr(ID2, class, CLASS),
    ↪ has_rel(ID2, REL, ID), object(ID).

state(TO, ID) :- filter_any(TO, TI, ATTR
    ↪ ), state(TI, ID), has_attr(ID,
    ↪ ATTR, _).

state(TO, ID') :- relate_any(TO, TI, REL,
    ↪  subject), state(TI, ID), has_rel(
    ↪ ID', REL, ID), object(ID').
state(TO, ID') :- relate_any(TO, TI, REL,
    ↪  object), state(TI, ID), has_rel(
    ↪ ID, REL, ID'), object(ID').

attr_value(TO, V) :- query(TO, TI, ATTR),
    ↪  state(TI, ID), has_attr(ID, ATTR,
    ↪  V).
state(TO, ID) :- negate(TO, TI, _),
    ↪ state(TI, ID), not state(TI, ID).

bool(TO, yes) :- or(TO, TI1, TI2), state
    ↪ (TI1, _), state(TI2, _).
bool(TO, no) :- or(TO, TI1, TI2), not
    ↪ state(TI1, _), not state(TI2, _).
bool(TO, yes) :- or(TO, TI1, TI2), state
    ↪ (TI1, _), not state(TI2, _).
bool(TO, yes) :- or(TO, TI1, TI2), not
    ↪ state(TI1, _), state(TI2, _).

state(TO, ID) :- relate(TO, TI, CLASS,
    ↪ REL, object), state(TI, ID2),
    ↪ has_attr(ID2, class, CLASS),
    ↪ has_rel(ID2, REL, ID), object(ID).
state(TO, ID) :- relate(TO, TI, CLASS,
    ↪ REL, subject), state(TI, ID),
    ↪ has_attr(ID, class, CLASS),
    ↪ has_rel(ID, REL, ID2), object(ID2)
    ↪ .

{state(TO,ID): state(TI,ID)} = 1 :-
    ↪ unique(TO, TI).
bool(TO, yes) :- exist(TO, TI), state(TI,
    ↪  _).
bool(TO, no) :- exist(TO, TI), not state
    ↪ (TI, _).
state(TO, ID) :- relate(TO, TI, CLASS,
    ↪ REL, subject), state(TI, ID),
    ↪ has_attr(ID, class, CLASS),
    ↪ has_rel(ID, REL, ID2), object(ID2)
    ↪ .
state(TO, ID) :- relate(TO, TI, CLASS,
    ↪ REL, object), state(TI, ID2),
    ↪ has_attr(ID2, class, CLASS),
    ↪ has_rel(ID2, REL, ID), object(ID).

bool(TO, yes) :- verify_attr(TO, TI,
    ↪ ATTR, VAL), state(TI, ID),
    ↪ has_attr(ID, ATTR, VAL).
bool(TO, no) :- verify_attr(TO, TI, ATTR,
    ↪  VAL), state(TI, ID), not has_attr
    ↪ (ID, ATTR, VAL).

bool(TO, yes) :- and(TO, TI1, TI2), bool
    ↪ (TI1, yes), bool(TI2, yes).
bool(TO, no) :- and(TO, TI1, TI2), bool(
    ↪ TI1, no).
bool(TO, no) :- and(TO, TI1, TI2), bool(
    ↪ TI2, no).
state(TO, ID) :- choose_attr(TO, TI,
    ↪ ATTR, VAL1, VAL2), state(TI, ID),
    ↪ has_attr(ID, ATTR, VAL1).
state(TO, ID) :- choose_attr(TO, TI,
    ↪ ATTR, VAL1, VAL2), state(TI, ID),
    ↪ has_attr(ID, ATTR, VAL2).

attr(TO, VAL1) :- choose_attr(TO, TI,
    ↪ ATTR, VAL1, _), state(TI, ID),
    ↪ has_attr(ID, ATTR, VAL1).
attr(TO, VAL2) :- choose_attr(TO, TI,
    ↪ ATTR, _, VAL2), state(TI, ID),
    ↪ has_attr(ID, ATTR, VAL2).
state(TO, ID) :- choose_attr(TO, TI,
    ↪ ATTR, VAL1, VAL2), state(TI, ID),
    ↪ has_attr(ID, ATTR, VAL1).
state(TO, ID) :- choose_attr(TO, TI,
    ↪ ATTR, VAL1, VAL2), state(TI, ID),
    ↪ has_attr(ID, ATTR, VAL2).

attr(TO, VAL1) :- choose_attr(TO, TI,
    ↪ ATTR, VAL1, _), state(TI, ID),
    ↪ has_attr(ID, ATTR, VAL1).
attr(TO, VAL2) :- choose_attr(TO, TI,
    ↪ ATTR, _, VAL2), state(TI, ID),
    ↪ has_attr(ID, ATTR, VAL2).

bool(TO, yes) :- or(TO, TI1, TI2), bool(
    ↪ TI1, yes).
bool(TO, yes) :- or(TO, TI1, TI2), bool(
    ↪ TI2, yes).
bool(TO, no) :- or(TO, TI1, TI2), bool(
    ↪ TI1, no), bool(TI2, no).

%Batch Size b=5 Response
attr_value(TO, V) :- choose_attr(TO, TI,
    ↪  ATTR, V1, V2), state(TI, ID),
    ↪ has_attr(ID, ATTR, V1), V = V1.
attr_value(TO, V) :- choose_attr(TO, TI,
    ↪  ATTR, V1, V2), state(TI, ID),
    ↪ has_attr(ID, ATTR, V2), V = V2.

attr_value(TO, V) :- verify_attr(TO, TI,
    ↪  ATTR, V), state(TI, ID), has_attr
    ↪ (ID, ATTR, V).
bool(TO, yes) :- verify_attr(TO, TI,
    ↪ ATTR, V), state(TI, ID), has_attr(
    ↪ ID, ATTR, V).
bool(TO, no) :- verify_attr(TO, TI, ATTR,
    ↪  V), state(TI, ID), not has_attr(
    ↪ ID, ATTR, V).
```

```
attr(TO, V) :- query(TO, TI, ATTR), state(TI, ID), has_attr(ID, ATTR, V).

state(TO, ID) :- negate(TO, TI, ID), state(TI, ID), not state(TI, ID).

bool(TO, yes) :- exist(TO, TI), state(TI, _).
bool(TO, no) :- exist(TO, TI), not state(TI, _).

bool(TO, yes) :- and(TO, TI1, TI2), bool(TI1, yes), bool(TI2, yes).
bool(TO, no) :- and(TO, TI1, TI2), bool(TI1, no).
bool(TO, no) :- and(TO, TI1, TI2), bool(TI2, no).

bool(TO, yes) :- or(TO, TI1, TI2), bool(TI1, yes).
bool(TO, yes) :- or(TO, TI1, TI2), bool(TI2, yes).
bool(TO, no) :- or(TO, TI1, TI2), bool(TI1, no), bool(TI2, no).

%Batch Size b=10 Response
bool(TO, yes) :- and(TO, TI1, TI2), ans(TI1, yes), ans(TI2, yes).
bool(TO, no) :- and(TO, TI1, TI2), ans(TI1, no).
bool(TO, no) :- and(TO, TI1, TI2), ans(TI2, no).

bool(TO, yes) :- or(TO, TI1, TI2), ans(TI1, yes).
bool(TO, yes) :- or(TO, TI1, TI2), ans(TI2, yes).
bool(TO, no) :- or(TO, TI1, TI2), ans(TI1, no), ans(TI2, no).

ans(yes) :- verify_attr(TO, TI, ATTR, VALUE), state(TI, ID), has_attr(ID, ATTR, VALUE).
ans(no) :- verify_attr(TO, TI, ATTR, VALUE), state(TI, ID), not has_attr(ID, ATTR, VALUE).

ans(yes) :- exist(TO, TI), state(TI, _).
ans(no) :- exist(TO, TI), not state(TI, _).
```

Listing 31: Heavy Batch Experiment GQA

```
%Batch Size b=1 Response
state(TO, ID) :- exist(TO, TI), state(TI, ID).
bool(TO, yes) :- exist(TO, TI), state(TI, _).
bool(TO, no) :- exist(TO, TI), not state(TI, _).
ans(yes) :- end(TO), bool(TO, yes).
ans(no) :- end(TO), bool(TO, no).
bool(TO, yes) :- all_same(TO, TI, ATTR), state(TI, ID1), state(TI, ID2), ID1 != ID2, has_attr(ID1, ATTR, VAL), has_attr(ID2, ATTR, VAL).
bool(TO, no) :- all_same(TO, TI, ATTR), state(TI, ID1), state(TI, ID2), ID1 != ID2, has_attr(ID1, ATTR, VAL1), has_attr(ID2, ATTR, VAL2), VAL1 != VAL2.
state(TO, ID, left) :- choose_attr(TO, TI, hposition, left, right), state(TI, ID), has_attr(ID, hposition, left).
state(TO, ID, right) :- choose_attr(TO, TI, hposition, left, right), state(TI, ID), has_attr(ID, hposition, right).
ans(left) :- end(TO), state(TO, ID, left).
ans(right) :- end(TO), state(TO, ID, right).
state(TO, ID) :- unique(TO, TI), state(TI, ID).
attr_value(TO, V) :- query(TO, TI, ATTR), state(TI, ID), has_attr(ID, ATTR, V).
bool(TO, yes) :- verify_attr(TO, TI, ATTR, VAL), state(TI, ID), has_attr(ID, ATTR, VAL).
bool(TO, no) :- verify_attr(TO, TI, ATTR, VAL), state(TI, ID), not has_attr(ID, ATTR, VAL).
bool(TO, yes) :- two_different(TO, TI1, TI2, ATTR), state(TI1, ID1), state(TI2, ID2), ID1 != ID2, has_attr(ID1, ATTR, VAL1), has_attr(ID2, ATTR, VAL2), VAL1 != VAL2.
bool(TO, no) :- two_different(TO, TI1, TI2, ATTR), state(TI1, ID1), state(TI2, ID2), ID1 != ID2, has_attr(ID1, ATTR, VAL), has_attr(ID2, ATTR, VAL).
state(TO, ID) :- unique(TO, TI), state(TI, ID).
bool(TO, yes) :- two_same(TO, TI1, TI2, ATTR), state(TI1, ID1), state(TI2, ID2), ID1 != ID2, has_attr(ID1, ATTR, VAL), has_attr(ID2, ATTR, VAL).
bool(TO, no) :- two_same(TO, TI1, TI2, ATTR), state(TI1, ID1), state(TI2, ID2), ID1 != ID2, has_attr(ID1, ATTR, VAL1), has_attr(ID2, ATTR, VAL2), VAL1 != VAL2.

%Batch Size b=2 Response
bool(TO,no) :- negate(TO, TI0, TI1), state(TI0, ID), state(TI1, ID).
bool(TO,yes) :- negate(TO, TI0, TI1), state(TI1, ID), not state(TI0, ID).
bool(TO,yes) :- exist(TO, TI), state(TI, _).
bool(TO,no) :- exist(TO, TI), not state(TI, _).
attr_value(TO, V) :- query(TO, TI, ATTR),
    ↪  state(TI, ID), has_attr(ID, ATTR,
    ↪  V).
state(TO, ID) :- relate(TO, TI, CLASS,
    ↪ REL, object), state(TI, ID2),
    ↪ has_attr(ID, class, CLASS),
    ↪ has_rel(ID2, REL, ID).
state(TO, ID) :- relate(TO, TI, CLASS,
    ↪ REL, subject), state(TI, ID2),
    ↪ has_attr(ID2, class, CLASS),
    ↪ has_rel(ID, REL, ID2).
state(TO, ID) :- relate(TO, TI, CLASS,
    ↪ REL, object), state(TI, ID2),
    ↪ has_attr(ID2, class, CLASS),
    ↪ has_rel(ID2, REL, ID).

bool(TO, yes) :- verify_attr(TO, TI,
    ↪ ATTR, VAL), state(TI, ID),
    ↪ has_attr(ID, ATTR, VAL).
bool(TO, no) :- verify_attr(TO, TI, ATTR,
    ↪  VAL), state(TI, ID), not has_attr
    ↪ (ID, ATTR, VAL).
state(TO, ID1, ID2) :- two_different(TO,
    ↪  TI1, TI2, ATTR), state(TI1, ID1),
    ↪  state(TI2, ID2), has_attr(ID1,
    ↪ ATTR, VAL1), has_attr(ID2, ATTR,
    ↪ VAL2), VAL1 != VAL2.

bool(TO, yes) :- two_different(TO, TI1,
    ↪ TI2, ATTR), state(TI1, ID1), state
    ↪ (TI2, ID2), has_attr(ID1, ATTR,
    ↪ VAL1), has_attr(ID2, ATTR, VAL2),
    ↪ VAL1 != VAL2.
bool(TO, no) :- two_different(TO, TI1,
    ↪ TI2, ATTR), state(TI1, ID1), state
    ↪ (TI2, ID2), not bool(TO, yes).

state(TO, ID) :- relate(TO, TI, CLASS,
    ↪ REL, subject), state(TI, ID),
    ↪ has_rel(ID, REL, ID2), has_attr(
    ↪ ID2, class, CLASS).
state(TO, ID2) :- relate(TO, TI, CLASS,
    ↪ REL, subject), state(TI, ID),
    ↪ has_rel(ID, REL, ID2), has_attr(
    ↪ ID2, class, CLASS).

bool(TO, yes) :- verify_attr(TO, TI,
    ↪ ATTR, VAL), state(TI, ID),
    ↪ has_attr(ID, ATTR, VAL).
bool(TO, no) :- verify_attr(TO, TI, ATTR,
    ↪  VAL), state(TI, ID), not has_attr
    ↪ (ID, ATTR, VAL).
state(TO, ID, right) :- choose_attr(TO,
    ↪ TI, hposition, right, _), state(TI
    ↪ , ID), has_attr(ID, hposition,
    ↪ right).
state(TO, ID, left) :- choose_attr(TO,
    ↪ TI, hposition, _, left), state(TI,
    ↪  ID), has_attr(ID, hposition, left
    ↪ ).
ans(right) :- end(TO), state(TO, _,
    ↪ right).
ans(left) :- end(TO), state(TO, _, left).
    ↪
state(TO, ID, brown) :- choose_attr(TO,
    ↪ TI, color, brown, _), state(TI, ID
    ↪ ), has_attr(ID, color, brown).
state(TO, ID, silver) :- choose_attr(TO,
    ↪  TI, color, _, silver), state(TI,
    ↪ ID), has_attr(ID, color, silver).
ans(brown) :- end(TO), state(TO, _,
    ↪ brown).
ans(silver) :- end(TO), state(TO, _,
    ↪ silver).
state(TO, ID) :- filter_any(TO, TI, ATTR
    ↪ ), state(TI, ID), has_attr(ID,
    ↪ ATTR, _).

bool(TO, yes) :- verify_rel(TO, TI,
    ↪ CLASS, REL, object), state(TI, ID)
    ↪ , has_rel(ID, REL, ID2), has_attr(
    ↪ ID2, class, CLASS).
bool(TO, no) :- verify_rel(TO, TI, CLASS,
    ↪  REL, object), state(TI, ID), not
    ↪ has_rel(ID, REL, ID2), has_attr(
    ↪ ID2, class, CLASS).

bool(TO, yes) :- verify_attr(TO, TI,
    ↪ ATTR, VAL), state(TI, ID),
    ↪ has_attr(ID, ATTR, VAL).
bool(TO, no) :- verify_attr(TO, TI, ATTR,
    ↪  VAL), state(TI, ID), not has_attr
    ↪ (ID, ATTR, VAL).

%Batch Size b=5 Response
state(TO, ID) :- relate(TO, TI, CLASS,
    ↪ REL, subject), state(TI, ID),
    ↪ has_attr(ID, class, CLASS),
    ↪ has_rel(ID, REL, ID2), object(ID2)
    ↪ .
state(TO, ID) :- relate(TO, TI, CLASS,
    ↪ REL, object), state(TI, ID2),
    ↪ has_attr(ID2, class, CLASS),
    ↪ has_rel(ID2, REL, ID), object(ID).

bool(TO, yes) :- verify_attr(TO, TI,
    ↪ ATTR, VALUE), state(TI, ID),
    ↪ has_attr(ID, ATTR, VALUE).
bool(TO, no) :- verify_attr(TO, TI, ATTR,
    ↪  VALUE), state(TI, ID), not
    ↪ has_attr(ID, ATTR, VALUE).

attr_value(TO, V) :- query(TO, TI, ATTR),
    ↪  state(TI, ID), has_attr(ID, ATTR,
    ↪  V).
bool(TO, yes) :- all_same(TO, TI, ATTR),
    ↪  state(TI, ID1), state(TI, ID2),
    ↪ ID1 != ID2, has_attr(ID1, ATTR,
    ↪ VAL), has_attr(ID2, ATTR, VAL).
bool(TO, no) :- all_same(TO, TI, ATTR),
    ↪ state(TI, ID1), state(TI, ID2),
    ↪ ID1 != ID2, has_attr(ID1, ATTR,
    ↪ VAL1), has_attr(ID2, ATTR, VAL2),
    ↪ VAL1 != VAL2.

state(TO, ID) :- relate_any(TO, TI, REL,
    ↪  object), state(TI, ID2), has_rel(
    ↪ ID2, REL, ID).

attr(TO, V) :- choose_attr(TO, TI, ATTR,
    ↪  VAL1, VAL2), state(TI, ID),
    ↪ has_attr(ID, ATTR, VAL1), V = VAL1
    ↪ .
attr(TO, V) :- choose_attr(TO, TI, ATTR,
    ↪  VAL1, VAL2), state(TI, ID),
    ↪ has_attr(ID, ATTR, VAL2), V = VAL2
    ↪ .
bool(TO, yes) :- verify_rel(TO, TI,
    ↪ CLASS, REL, subject), state(TI, ID
    ↪ ), has_rel(ID, REL, ID2), has_attr
    ↪ (ID2, class, CLASS).
bool(TO, no) :- verify_rel(TO, TI, CLASS,
    ↪  REL, subject), state(TI, ID), not
    ↪  has_rel(ID, REL, ID2), has_attr(
    ↪ ID2, class, CLASS).

bool(TO, yes) :- verify_rel(TO, TI,
    ↪ CLASS, REL, object), state(TI, ID2
    ↪ ), has_rel(ID2, REL, ID), has_attr
    ↪ (ID, class, CLASS).
bool(TO, no) :- verify_rel(TO, TI, CLASS,
    ↪  REL, object), state(TI, ID2), not
    ↪  has_rel(ID2, REL, ID), has_attr(
    ↪ ID, class, CLASS).

bool(TO, no) :- negate(TO, TI0, TI1),
    ↪ bool(TI0, yes).
bool(TO, yes) :- negate(TO, TI0, TI1),
    ↪ bool(TI1, yes).

%Batch Size b=10 Response
attr_value(TO, V) :- query(TO, TI, ATTR),
    ↪  state(TI, ID), has_attr(ID, ATTR,
    ↪  V).

bool(TO, yes) :- exist(TO, TI), state(TI,
    ↪  _).
bool(TO, no) :- exist(TO, TI), not state
    ↪ (TI, _).

bool(TO, yes) :- verify_attr(TO, TI,
    ↪ ATTR, VAL), state(TI, ID),
    ↪ has_attr(ID, ATTR, VAL).
bool(TO, no) :- verify_attr(TO, TI, ATTR,
    ↪  VAL), state(TI, ID), not has_attr
    ↪ (ID, ATTR, VAL).
```

Listing 32: Light Batch Experiment CLEVR

```
%Batch Size b=1 Response
ans(yes) :- not ans(no).
ans(no) :- not exist(_).

exist(T) :- state(T,ID), not not_exist(T,
    ↪ ID).
not_exist(T,ID) :- state(T,ID),
    ↪ filter_property(PROP), not
    ↪ property(T,ID,PROP).

property(T,ID,size(SIZE)) :- has_size(ID,
    ↪ SIZE), filter_size(T,SIZE).
property(T,ID,color(COLOR)) :- has_color
    ↪ (ID,COLOR), filter_color(T,COLOR).
property(T,ID,material(MATERIAL)) :-
    ↪ has_material(ID,MATERIAL),
    ↪ filter_material(T,MATERIAL).
property(T,ID,shape(SHAPE)) :- has_shape
    ↪ (ID,SHAPE), filter_shape(T,SHAPE).
exist(T) :- state(T,ID), filter_property
    ↪ (PROP), property(T,ID,PROP), not
    ↪ not_exist(T,ID).
not_exist(T,ID) :- state(T,ID),
    ↪ filter_property(PROP), not
    ↪ property(T,ID,PROP).
ans(no) :- end(T), not exist(T).
filter_color(T,COLOR) :- scene(T), color
    ↪ (COLOR).
color(cyan).

exist(T) :- state(T,ID), filter_color(T,
    ↪ COLOR), has_color(ID,COLOR).

ans(yes) :- exist(_).
ans(no) :- not exist(_).
filter_size(T,SIZE) :- scene(T), size(
    ↪ SIZE).
size(small).
size(medium).
size(large).

unique(T) :- state(T,ID), not non_unique
    ↪ (T,ID).
non_unique(T,ID) :- state(T,ID), state(T,
    ↪ ID2), ID != ID2, filter_property(
    ↪ PROP), property(T,ID,PROP),
    ↪ property(T,ID2,PROP).

query_size(T) :- unique(T), has_size(ID,
    ↪ SIZE), state(T,ID).
ans(SIZE) :- query_size(T), unique(T),
    ↪ has_size(ID,SIZE), state(T,ID).
filter_material(T,MATERIAL) :- scene(T),
    ↪  material(MATERIAL).
material(rubber).
material(metal).

query_material(T) :- unique(T),
    ↪ has_material(ID,MATERIAL), state(T
    ↪ ,ID).
ans(MATERIAL) :- query_material(T),
    ↪ unique(T), has_material(ID,
    ↪ MATERIAL), state(T,ID).
filter_shape(T,SHAPE) :- scene(T), shape
    ↪ (SHAPE).
shape(sphere).
shape(cube).
shape(cylinder).

query_shape(T) :- unique(T), has_shape(
    ↪ ID,SHAPE), state(T,ID).
ans(SHAPE) :- query_shape(T), unique(T),
    ↪  has_shape(ID,SHAPE), state(T,ID).
filter_shape(T,SHAPE) :- filter_cylinder
    ↪ (T), SHAPE = cylinder.
filter_material(T,MATERIAL) :-
    ↪ filter_metal(T), MATERIAL = metal.

query_color(T) :- unique(T), has_color(
    ↪ ID,COLOR), state(T,ID).
ans(COLOR) :- query_color(T), unique(T),
    ↪  has_color(ID,COLOR), state(T,ID).

%Batch Size b=2 Response
end(T) :- not exist(T+1), scene(T).
ans(no) :- end(T), not exist(T).
filter_color(T,C,ID) :- state(T,ID),
    ↪ has_color(ID,C).

exist(T) :- filter_color(T,_,_).

bool(T,true) :- exist(T).
bool(T,false) :- not exist(T), state(T,_
    ↪ ).

ans(V) :- end(T), bool(T,V).
filter_color(T,C,ID) :- state(T,ID),
    ↪ has_color(ID,C).

exist(T) :- filter_color(T,_,_).
material(T,MATERIAL,ID) :- state(T,ID),
    ↪ has_material(ID,MATERIAL).
shape(T,SHAPE,ID) :- state(T,ID),
    ↪ has_shape(ID,SHAPE).

unique(T,ID) :- filter_color(T,C,ID),
    ↪ not other_color(T,C,ID).
unique(T,ID) :- filter_size(T,S,ID), not
    ↪  other_size(T,S,ID).
unique(T,ID) :- filter_material(T,M,ID),
    ↪  not other_material(T,M,ID).
unique(T,ID) :- filter_shape(T,S,ID),
    ↪ not other_shape(T,S,ID).

other_color(T,C,ID) :- filter_color(T,C,
    ↪ ID2), ID != ID2, filter_color(T,C,
    ↪ ID).
other_size(T,S,ID) :- filter_size(T,S,
    ↪ ID2), ID != ID2, filter_size(T,S,
    ↪ ID).
other_material(T,M,ID) :-
    ↪ filter_material(T,M,ID2), ID !=
    ↪ ID2, filter_material(T,M,ID).
other_shape(T,S,ID) :- filter_shape(T,S,
    ↪ ID2), ID != ID2, filter_shape(T,S,
    ↪ ID).

query_material(T) :- unique(T,ID),
    ↪ has_material(ID,MATERIAL).
query_shape(T) :- unique(T,ID),
    ↪ has_shape(ID,SHAPE).

ans(MATERIAL) :- query_material(T),
    ↪ material(T,MATERIAL,_).
ans(SHAPE) :- query_shape(T), shape(T,
    ↪ SHAPE,_).
filter_material(T,MATERIAL,ID) :- state(
    ↪ T,ID), has_material(ID,MATERIAL).

unique(T,ID) :- filter_material(T,M,ID),
    ↪  not other_material(T,M,ID).

query_color(T) :- unique(T,ID),
    ↪ has_color(ID,COLOR).

ans(COLOR) :- query_color(T), color(T,
    ↪ COLOR,_).

filter_color(T,C,ID) :- state(T,ID),
    ↪ has_color(ID,C).

color(T,COLOR,ID) :- state(T,ID),
    ↪ has_color(ID,COLOR).

unique(T,ID) :- filter_color(T,C,ID),
    ↪ not other_color(T,C,ID).

ans(MATERIAL) :- query_material(T),
    ↪ material(T,MATERIAL,_).
unique_material(T,ID) :- material(T,
    ↪ MATERIAL,ID), not other_material(T
    ↪ ,MATERIAL,ID).
unique_shape(T,ID) :- shape(T,SHAPE,ID),
    ↪  not other_shape(T,SHAPE,ID).

query_material(T) :- unique_material(T,
    ↪ ID).
query_shape(T) :- unique_shape(T,ID).

ans(yes) :- query_material(T),
    ↪ query_shape(T).
same_material(T) :- unique(T,ID),
    ↪ has_material(ID,M), object(ID2),
    ↪ ID != ID2, has_material(ID2,M).
same_shape(T) :- unique(T,ID), has_shape
    ↪ (ID,S), object(ID2), ID != ID2,
    ↪ has_shape(ID2,S).
filter_size(T,SIZE,ID) :- state(T,ID),
    ↪ has_size(ID,SIZE).
size(T,SIZE,ID) :- state(T,ID), has_size
    ↪ (ID,SIZE).
query_size(T) :- unique(T,ID), has_size(
    ↪ ID,SIZE).
ans(SIZE) :- query_size(T), size(T,SIZE,
    ↪ _).

%Batch Size b=5 Response
exist(T) :- state(T,ID), not not_exist(
    ↪ ID,T).
not_exist(ID,T) :- state(T,ID),
    ↪ filter_color(T1,COLOR), has_color(
    ↪ ID,COLOR'), COLOR != COLOR', T1 <
    ↪ T.
not_exist(ID,T) :- state(T,ID),
    ↪ filter_size(T1,SIZE), has_size(ID,
    ↪ SIZE'), SIZE != SIZE', T1 < T.
not_exist(ID,T) :- state(T,ID),
    ↪ filter_material(T1,MATERIAL),
    ↪ has_material(ID,MATERIAL'),
    ↪ MATERIAL != MATERIAL', T1 < T.
not_exist(ID,T) :- state(T,ID),
    ↪ filter_shape(T1,SHAPE), has_shape(
    ↪ ID,SHAPE'), SHAPE != SHAPE', T1 <
    ↪ T.

ans(no) :- exist(T), not state(T,_).
ans(yes) :- exist(T), state(T,_).
ans(no) :- not exist(T), end(T).
ans(yes) :- exist(T), end(T), state(T,_).
    ↪
color(T,COLOR) :- query_color(T), state(
T,ID), has_color(ID,COLOR).
material(T,MATERIAL) :- query_material(T), state(T,ID), has_material(ID,MATERIAL).
shape(T,SHAPE) :- query_shape(T), state(T,ID), has_shape(ID,SHAPE).
size(T,SIZE) :- query_size(T), state(T,ID), has_size(ID,SIZE).

equal_color(T1,T2) :- color(T1,COLOR), color(T2,COLOR).
equal_material(T1,T2) :- material(T1,MATERIAL), material(T2,MATERIAL).
equal_shape(T1,T2) :- shape(T1,SHAPE), shape(T2,SHAPE).
equal_size(T1,T2) :- size(T1,SIZE), size(T2,SIZE).

ans(COLOR) :- equal_color(T1,T2), color(T1,COLOR), end(T).
ans(MATERIAL) :- equal_material(T1,T2), material(T1,MATERIAL), end(T).
ans(SHAPE) :- equal_shape(T1,T2), shape(T1,SHAPE), end(T).
ans(SIZE) :- equal_size(T1,T2), size(T1,SIZE), end(T).
```

Listing 33: Medium Batch Experiment CLEVR

```
%Batch Size b=1 Response
state(T+1,ID) :- exist(T), state(T,ID).
bool(T+1,yes) :- exist(T), state(T,ID).
bool(T+1,no) :- exist(T), not state(T,_).
state(T+1,ID) :- filter_metal(T), state(T,ID), has_material(ID,metal).
bool(T+1,yes) :- exist(T), state(T,_).
bool(T+1,no) :- exist(T), not state(T,_).
ans(V) :- end(T), bool(T,V).
state(T+1,ID) :- filter_yellow(T), state(T,ID), has_color(ID,yellow).
bool(T+1,yes) :- exist(T), state(T,_).
bool(T+1,no) :- exist(T), not state(T,_).
ans(V) :- end(T), bool(T,V).
state(T+1,ID) :- filter_red(T), state(T,ID), has_color(ID,red).
bool(T+1,yes) :- exist(T), state(T,_).
bool(T+1,no) :- exist(T), not state(T,_).
ans(V) :- end(T), bool(T,V).
state(T+1,ID) :- filter_cube(T), state(T,ID), has_shape(ID,cube).
bool(T+1,yes) :- exist(T), state(T,_).
bool(T+1,no) :- exist(T), not state(T,_).
ans(V) :- end(T), bool(T,V).
state(T+1,ID) :- filter_small(T), state(T,ID), has_size(ID,small).
bool(T+1,yes) :- exist(T), state(T,_).
bool(T+1,no) :- exist(T), not state(T,_).
ans(V) :- end(T), bool(T,V).
state(T+1,ID) :- filter_rubber(T), state(T,ID), has_material(ID,rubber).
bool(T+1,yes) :- exist(T), state(T,_).
bool(T+1,no) :- exist(T), not state(T,_).
ans(V) :- end(T), bool(T,V).
state(T+1,ID) :- filter_gray(T), state(T,ID), has_color(ID,gray).
bool(T+1,yes) :- exist(T), state(T,_).
bool(T+1,no) :- exist(T), not state(T,_).
ans(V) :- end(T), bool(T,V).
state(T+1,ID) :- filter_blue(T), state(T,ID), has_color(ID,blue).
state(T+1,ID) :- unique(T), state(T,ID), not state(T,ID2), ID != ID2, ID2 = ID+1.
shape(T+1,SHAPE) :- query_shape(T), state(T,ID), has_shape(ID,SHAPE).
ans(V) :- end(T), shape(T,V).
shape(T+1,SHAPE) :- query_shape(T), state(T,ID), has_shape(ID,SHAPE).
ans(V) :- end(T), shape(T,V).
color(T+1,COLOR) :- query_color(T), state(T,ID), has_color(ID,COLOR).
state(T+1,ID) :- filter_cylinder(T), state(T,ID), has_shape(ID,cylinder).
color(T+1,COLOR) :- query_color(T), state(T,ID), has_color(ID,COLOR).
state(T+1,ID) :- filter_brown(T), state(T,ID), has_color(ID,brown).
material(T+1,MATERIAL) :- query_material(T), state(T,ID), has_material(ID,MATERIAL).
ans(V) :- end(T), material(T,V).
state(T+1,ID) :- filter_sphere(T), state(T,ID), has_shape(ID,sphere).
int(T+1,COUNT) :- count(T), state(T,ID), COUNT = #count{ ID : state(T,ID) }.
ans(V) :- end(T), int(T,V).
state(T+1,ID) :- filter_purple(T), state(T,ID), has_color(ID,purple).
state(T+1,ID) :- filter_sphere(T), state(T,ID), has_shape(ID,sphere).
state(T+1,ID) :- same_shape(T), state(T,ID1), has_shape(ID1,SHAPE), has_shape(ID,SHAPE), ID != ID1.
state(T+1,ID) :- unique(T), state(T,ID), not state(T,ID2), ID != ID2, ID2=ID.
state(T+1,ID) :- same_shape(T), state(T,ID1), has_shape(ID1,SHAPE), has_shape(ID,SHAPE), ID != ID1.
state(T+1,ID) :- unique(T), state(T,ID), not state(T,ID2), ID != ID2, ID2=ID.
state(T+1,ID) :- same_shape(T), state(T,ID), has_shape(ID,SHAPE), state(T,ID2), has_shape(ID2,SHAPE), ID != ID2.
state(T+1,ID) :- unique(T), state(T,ID), not state(T,ID2), ID != ID2, ID2=ID.
state(T+1,ID) :- same_color(T), state(T,
    ↪ ID1), has_color(ID1,COLOR),
    ↪ has_color(ID,COLOR), ID != ID1.
state(T+1,ID) :- filter_cyan(T), state(T,
    ↪ ID), has_color(ID,cyan).
state(T+1,ID) :- same_shape(T), state(T,
    ↪ ID1), has_shape(ID1,SHAPE),
    ↪ has_shape(ID,SHAPE), ID != ID1.
material(T+1,MATERIAL) :- query_material
    ↪ (T), state(T,ID), has_material(ID,
    ↪ MATERIAL).
state(T+1,ID) :- filter_green(T), state(
    ↪ T,ID), has_color(ID,green).
state(T+1,ID) :- relate_left(T), state(T,
    ↪ ID'), left_of(ID,ID').
state(T+1,ID) :- same_material(T), state
    ↪ (T,ID1), has_material(ID1,MATERIAL
    ↪ ), has_material(ID,MATERIAL), ID
    ↪ != ID1.
state(T+1,ID) :- filter_small(T), state(
    ↪ T,ID), has_size(ID,small).
state(T+1,ID) :- filter_gray(T), state(T,
    ↪ ID), has_color(ID,gray).
state(T+1,ID) :- filter_cylinder(T),
    ↪ state(T,ID), has_shape(ID,cylinder
    ↪ ).
state(T+1,ID) :- relate_right(T), state(
    ↪ T,ID'), right_of(ID,ID').
state(T+1,ID) :- filter_large(T), state(
    ↪ T,ID), has_size(ID,large).
state(T+1,ID) :- relate_behind(T), state
    ↪ (T,ID'), behind_of(ID,ID').
color(T+1,COLOR) :- query_color(T),
    ↪ state(T,ID), has_color(ID,COLOR).
state(T+1,ID) :- filter_red(T), state(T,
    ↪ ID), has_color(ID,red).
state(T+1,ID) :- filter_metal(T), state(
    ↪ T,ID), has_material(ID,metal).
state(T+1,ID) :- unique(T), state(T,ID),
    ↪  not state(T,ID2), ID != ID2, ID2
    ↪ = ID.
state(T+1,ID) :- relate_front(T), state(
    ↪ T,ID'), in_front_of(ID,ID').
state(T+1,ID) :- filter_cyan(T), state(T,
    ↪ ID), has_color(ID,cyan).
shape(T+1,SHAPE) :- query_shape(T),
    ↪ state(T,ID), has_shape(ID,SHAPE).
state(T+1,ID) :- relate_front(T), state(
    ↪ T,ID'), in_front_of(ID,ID').
state(T+1,ID) :- filter_cyan(T), state(T,
    ↪ ID), has_color(ID,cyan).
shape(T+1,SHAPE) :- query_shape(T),
    ↪ state(T,ID), has_shape(ID,SHAPE).
ans(V) :- end(T), shape(T,V).
state(T+1,ID) :- unique(T), state(T,ID),
    ↪  not state(T,ID2), ID != ID2, ID2=
    ↪ ID.
shape(T+1,SHAPE) :- query_shape(T),
    ↪ state(T,ID), has_shape(ID,SHAPE).
bool(T+1,yes) :- equal_shape(T,T'),
    ↪ shape(T,SHAPE), shape(T',SHAPE).
bool(T+1,no) :- equal_shape(T,T'), not
    ↪ bool(T+1,yes).
state(T+1,ID) :- filter_gray(T), state(T,
    ↪ ID), has_color(ID,gray).
shape(T+1,SHAPE) :- query_shape(T),
    ↪ state(T,ID), has_shape(ID,SHAPE).
bool(T+1,yes) :- equal_shape(T,T'),
    ↪ shape(T,SHAPE), shape(T',SHAPE).
bool(T+1,no) :- equal_shape(T,T'), shape
    ↪ (T,SHAPE), shape(T',SHAPE2), SHAPE
    ↪  != SHAPE2.
material(T+1,MATERIAL) :- query_material
    ↪ (T), state(T,ID), has_material(ID,
    ↪ MATERIAL).
bool(T+1,yes) :- equal_material(T,T'),
    ↪ material(T,MATERIAL), material(T',
    ↪ MATERIAL).
bool(T+1,no) :- equal_material(T,T'),
    ↪ material(T,MATERIAL), material(T',
    ↪ MATERIAL2), MATERIAL != MATERIAL2.
bool(T+1,no) :- equal_color(T,T'), not
    ↪ bool(T+1,yes).
state(T+1,ID) :- filter_sphere(T), state
    ↪ (T,ID), has_shape(ID,sphere).
color(T+1,COLOR) :- query_color(T),
    ↪ state(T,ID), has_color(ID,COLOR).
bool(T+1,yes) :- equal_color(T,T'),
    ↪ color(T,COLOR), color(T',COLOR).
bool(T+1,no) :- equal_color(T,T'), color
    ↪ (T,COLOR), color(T',COLOR2), COLOR
    ↪  != COLOR2.

%Batch Size b=2 Response
% Existence check rules
exist(T+1) :- exist(T), state(T,ID).
bool(T+1,yes) :- exist(T), state(T,_).
bool(T+1,no) :- exist(T), not state(T,_).
    ↪

% Filtering rules for color
state(T+1,ID) :- filter_red(T), state(T,
    ↪ ID), has_color(ID,red).
% Filtering rules for material
state(T+1,ID) :- filter_metal(T), state(
    ↪ T,ID), has_material(ID,metal).

% Filtering rules for color
state(T+1,ID) :- filter_cyan(T), state(T,
    ↪ ID), has_color(ID,cyan).
state(T+1,ID) :- filter_rubber(T), state
    ↪ (T,ID), has_material(ID,rubber).
state(T+1,ID) :- filter_large(T), state(
    ↪ T,ID), has_size(ID,large).
% Filtering rules for size
state(T+1,ID) :- filter_small(T), state(
    ↪ T,ID), has_size(ID,small).

% Existence check rules
bool(T+1,yes) :- exist(T), state(T,_).
bool(T+1,no) :- exist(T), not state(T,_).
    ↪

% Filtering rules for color
state(T+1,ID) :- filter_blue(T), state(T,
    ↪ ID), has_color(ID,blue).
state(T+1,ID) :- filter_cube(T), state(T,
    ↪ ID), has_shape(ID,cube).
state(T+1,ID) :- filter_green(T), state(
    ↪ T,ID), has_color(ID,green).
```

```
state(T+1,ID) :- filter_yellow(T), state(T,ID), has_color(ID,yellow).
state(T+1,ID) :- filter_purple(T), state(T,ID), has_color(ID,purple).
% Filtering rules for color
state(T+1,ID) :- filter_gray(T), state(T,ID), has_color(ID,gray).
state(T+1,ID) :- filter_purple(T), state(T,ID), has_color(ID,purple).

% Unique object identification
state(T+1,ID) :- unique(T), state(T,ID), not another_object(T,ID).
another_object(T,ID) :- state(T,ID1), state(T,ID), ID1 != ID.

% Query functions for material
material(T+1,MATERIAL) :- query_material(T), state(T,ID), has_material(ID,MATERIAL).

% Derive answer for material
ans(MATERIAL) :- end(T), material(T,MATERIAL).
% Query functions for shape
shape(T+1,SHAPE) :- query_shape(T), state(T,ID), has_shape(ID,SHAPE).

% Derive answer for shape
ans(SHAPE) :- end(T), shape(T,SHAPE).
% Filtering rules for shape
state(T+1,ID) :- filter_sphere(T), state(T,ID), has_shape(ID,sphere).

% Query functions for color
color(T+1,COLOR) :- query_color(T), state(T,ID), has_color(ID,COLOR).

% Derive answer for color
ans(COLOR) :- end(T), color(T,COLOR).
% Filtering rules for shape
state(T+1,ID) :- filter_cylinder(T), state(T,ID), has_shape(ID,cylinder).

% Derive answer for color
ans(COLOR) :- end(T), color(T,COLOR).

% Query functions for shape
shape(T+1,SHAPE) :- query_shape(T), state(T,ID), has_shape(ID,SHAPE).

% Derive answer for shape
ans(SHAPE) :- end(T), shape(T,SHAPE).
% Same-color relations
state(T+1,ID') :- same_color(T), state(T,ID), has_color(ID,COLOR), has_color(ID',COLOR), ID!=ID'.

% Same-material relations
state(T+1,ID') :- same_material(T), state(T,ID), has_material(ID,MATERIAL), has_material(ID',MATERIAL), ID!=ID'.
% Same-size relations
state(T+1,ID') :- same_size(T), state(T,ID), has_size(ID,SIZE), has_size(ID',SIZE), ID!=ID'.

% Same-shape relations
state(T+1,ID') :- same_shape(T), state(T,ID), has_shape(ID,SHAPE), has_shape(ID',SHAPE), ID!=ID'.
% Filtering rules for color
state(T+1,ID) :- filter_brown(T), state(T,ID), has_color(ID,brown).

% Query functions for size
size(T+1,SIZE) :- query_size(T), state(T,ID), has_size(ID,SIZE).

% Derive answer for size
ans(SIZE) :- end(T), size(T,SIZE).

% Same-material relations
state(T+1,ID') :- same_material(T), state(T,ID), has_material(ID,MATERIAL), has_material(ID',MATERIAL), ID!=ID'.

% Query functions for material
material(T+1,MATERIAL) :- query_material(T), state(T,ID), has_material(ID,MATERIAL).

% Derive answer for material
ans(MATERIAL) :- end(T), material(T,MATERIAL).
state(T+1,ID) :- relate_right(T), state(T,ID'), right_of(ID,ID').
state(T+1,ID) :- filter_sphere(T), state(T,ID), has_shape(ID,sphere).
state(T+1,ID) :- filter_purple(T), state(T,ID), has_color(ID,purple).
state(T+1,ID) :- filter_yellow(T), state(T,ID), has_color(ID,yellow).
state(T+1,ID) :- filter_rubber(T), state(T,ID), has_material(ID,rubber).
state(T+1,ID') :- same_shape(T), state(T,ID), has_shape(ID,SHAPE), has_shape(ID',SHAPE), ID!=ID'.
size(T+1,SIZE) :- query_size(T), state(T,ID), has_size(ID,SIZE).
ans(SIZE) :- end(T), size(T,SIZE).
color(T+1,COLOR) :- query_color(T), state(T,ID), has_color(ID,COLOR).
ans(COLOR) :- end(T), color(T,COLOR).
state(T+1,ID) :- relate_front(T), state(T,ID'), in_front_of(ID,ID').
state(T+1,ID) :- relate_behind(T), state(T,ID'), behind_of(ID,ID').
state(T+1,ID) :- filter_small(T), state(T,ID), has_size(ID,small).
state(T+1,ID) :- filter_cylinder(T), state(T,ID), has_shape(ID,cylinder).
material(T+1,MATERIAL) :- query_material(T), state(T,ID), has_material(ID,MATERIAL).
ans(MATERIAL) :- end(T), material(T,
    ↪ MATERIAL).
size(T+1,SIZE) :- query_size(T), state(T,
    ↪ ID), has_size(ID,SIZE).
ans(SIZE) :- end(T), size(T,SIZE).
% Query functions for size
size(T+1,SIZE) :- query_size(T), state(T,
    ↪ ID), has_size(ID,SIZE).

% Derive answer for size
ans(SIZE) :- end(T), size(T,SIZE).

% Same-material relations
state(T+1,ID') :- same_material(T),
    ↪ state(T,ID), has_material(ID,
    ↪ MATERIAL), has_material(ID',
    ↪ MATERIAL), ID!=ID'.

% Query functions for color
color(T+1,COLOR) :- query_color(T),
    ↪ state(T,ID), has_color(ID,COLOR).

% Attribute comparison for color
bool(T+1,yes) :- equal_color(T,T'),
    ↪ color(T,C), color(T',C), C=C'.
bool(T+1,no) :- equal_color(T,T'), not
    ↪ bool(T+1,yes).
state(T+1,ID) :- filter_cube(T), state(T,
    ↪ ID), has_shape(ID,cube).
state(T+1,ID) :- filter_sphere(T), state
    ↪ (T,ID), has_shape(ID,sphere).
material(T+1,MATERIAL) :- query_material
    ↪ (T), state(T,ID), has_material(ID,
    ↪ MATERIAL).
bool(T+1,yes) :- equal_material(T,T'),
    ↪ material(T,M), material(T',M), M=M
    ↪ '.
bool(T+1,no) :- equal_material(T,T'),
    ↪ not bool(T+1,yes).

%Batch Size b=5 Response
bool(T+1,yes) :- exist(T), state(T,ID),
    ↪ has_color(ID,COLOR).
bool(T+1,no) :- exist(T), not bool(T+1,
    ↪ yes).

bool(T+1,yes) :- exist(T), state(T,ID),
    ↪ has_size(ID,SIZE).
bool(T+1,no) :- exist(T), not bool(T+1,
    ↪ yes).

state(T+1,ID) :- filter_green(T), state(
    ↪ T,ID), has_color(ID,green).
state(T+1,ID) :- filter_large(T), state(
    ↪ T,ID), has_size(ID,large).
state(T+1,ID) :- filter_blue(T), state(T,
    ↪ ID), has_color(ID,blue).
state(T+1,ID) :- filter_purple(T), state
    ↪ (T,ID), has_color(ID,purple).
state(T+1,ID) :- filter_small(T), state(
    ↪ T,ID), has_size(ID,small).
```

Listing 34: Heavy Batch Experiment CLEVR

```
%Batch Size b=1 Response
state(T+1,ID) :- filter_cube(T), state(T,
    ↪ ID), has_shape(ID,cube).
bool(T+1,yes) :- exist(T), state(T,ID).
bool(T+1,no) :- exist(T), not state(T,_).
    ↪
state(T+1,ID) :- filter_cube(T), state(T,
    ↪ ID), has_shape(ID,cube).
bool(T+1,yes) :- exist(T), state(T,ID).
bool(T+1,no) :- exist(T), not state(T,_).
    ↪
state(T+1,ID) :- filter_cube(T), state(T,
    ↪ ID), has_shape(ID,cube).
bool(T+1,yes) :- exist(T), state(T,ID).
bool(T+1,no) :- exist(T), not state(T,_).
    ↪
state(T+1,ID) :- filter_rubber(T), state
    ↪ (T,ID), has_material(ID,rubber).
state(T+1,ID) :- filter_metal(T), state(
    ↪ T,ID), has_material(ID,metal).
state(T+1,ID) :- filter_green(T), state(
    ↪ T,ID), has_color(ID,green).
shape(T+1,SHAPE) :- query_shape(T),
    ↪ state(T,ID), has_shape(ID,SHAPE).
state(T+1,ID) :- filter_purple(T), state
    ↪ (T,ID), has_color(ID,purple).
state(T+1,ID) :- filter_cylinder(T),
    ↪ state(T,ID), has_shape(ID,cylinder
    ↪ ).
state(T+1,ID) :- filter_sphere(T), state
    ↪ (T,ID), has_shape(ID,sphere).
state(T+1,ID) :- filter_blue(T), state(T,
    ↪ ID), has_color(ID,blue).
material(T+1,MATERIAL) :- query_material
    ↪ (T), state(T,ID), has_material(ID,
    ↪ MATERIAL).
state(T+1,ID) :- same_material(T), state
    ↪ (T,ID1), has_material(ID1,MATERIAL
    ↪ ), has_material(ID,MATERIAL), ID!=
    ↪ ID1.
state(T+1,ID) :- filter_large(T), state(
    ↪ T,ID), has_size(ID,large).
state(T+1,ID) :- filter_blue(T), state(T,
    ↪ ID), has_color(ID,blue).
state(T+1,ID) :- same_shape(T), state(T,
    ↪ ID1), has_shape(ID1,SHAPE),
    ↪ has_shape(ID,SHAPE), ID!=ID1.
bool(T+1,yes) :- exist(T), state(T,ID).
bool(T+1,no) :- exist(T), not state(T,_).
    ↪
state(T+1,ID) :- filter_cyan(T), state(T,
    ↪ ID), has_color(ID,cyan).
state(T+1,ID) :- same_material(T), state
    ↪ (T,ID1), has_material(ID1,MATERIAL
    ↪ ), has_material(ID,MATERIAL), ID!=
    ↪ ID1.
state(T+1,ID) :- filter_cylinder(T),
    ↪ state(T,ID), has_shape(ID,cylinder
    ↪ ).
state(T+1,ID) :- relate_behind(T), state
    ↪ (T,ID'), behind_of(ID,ID').
state(T+1,ID) :- filter_yellow(T), state
    ↪ (T,ID), has_color(ID,yellow).
state(T+1,ID) :- relate_front(T), state(
    ↪ T,ID'), in_front_of(ID,ID').
state(T+1,ID) :- filter_sphere(T), state
```

```
    ↪ (T,ID), has_shape(ID,sphere).
state(T+1,ID) :- filter_brown(T), state(
    ↪ T,ID), has_color(ID,brown).
material(T+1,MATERIAL) :- query_material
    ↪ (T), state(T,ID), has_material(ID,
    ↪ MATERIAL).
bool(T+1,yes) :- equal_material(T,T'),
    ↪ material(T,MATERIAL), material(T',
    ↪ MATERIAL), MATERIAL=MATERIAL'.
bool(T+1,no) :- equal_material(T,T'),
    ↪ not bool(T+1,yes).
state(T+1,ID) :- filter_rubber(T), state
    ↪ (T,ID), has_material(ID,rubber).
shape(T+1,SHAPE) :- query_shape(T),
    ↪ state(T,ID), has_shape(ID,SHAPE).
bool(T+1,yes) :- equal_shape(T,T'),
    ↪ shape(T,SHAPE), shape(T',SHAPE),
    ↪ SHAPE=SHAPE'.
bool(T+1,no) :- equal_shape(T,T'), not
    ↪ bool(T+1,yes).

%Batch Size b=2 Response
state(T+1,ID) :- filter_rubber(T), state
    ↪ (T,ID), has_material(ID,rubber).
bool(T+1,yes) :- exist(T), state(T,ID).
bool(T+1,no) :- exist(T), not bool(T+1,
    ↪ yes).
state(T+1,ID) :- filter_red(T), state(T,
    ↪ ID), has_color(ID,red).
state(T+1,ID) :- filter_yellow(T), state
    ↪ (T,ID), has_color(ID,yellow).
state(T+1,ID) :- filter_cube(T), state(T,
    ↪ ID), has_shape(ID,cube).
material(T+1,MATERIAL) :- query_material
    ↪ (T), state(T,ID), has_material(ID,
    ↪ MATERIAL).
shape(T+1,SHAPE) :- query_shape(T),
    ↪ state(T,ID), has_shape(ID,SHAPE).
state(T+1,ID) :- filter_metal(T), state(
    ↪ T,ID), has_material(ID,metal).
state(T+1,ID) :- filter_cube(T), state(T,
    ↪ ID), has_shape(ID,cube).
state(T+1,ID) :- filter_purple(T), state
    ↪ (T,ID), has_color(ID,purple).
state(T+1,ID) :- filter_sphere(T), state
    ↪ (T,ID), has_shape(ID,sphere).
state(T+1,ID) :- filter_cylinder(T),
    ↪ state(T,ID), has_shape(ID,cylinder
    ↪ ).
state(T+1,ID) :- filter_cyan(T), state(T,
    ↪ ID), has_color(ID,cyan).
state(T+1,ID) :- filter_cylinder(T),
    ↪ state(T,ID), has_shape(ID,cylinder
    ↪ ).
state(T+1,ID) :- filter_cylinder(T),
    ↪ state(T,ID), has_shape(ID,cylinder
    ↪ ).
state(T+1,ID) :- filter_brown(T), state(
    ↪ T,ID), has_color(ID,brown).
state(T+1,ID) :- filter_gray(T), state(T,
    ↪ ID), has_color(ID,gray).
state(T+1,ID') :- same_shape(T), state(T,
    ↪ ID), has_shape(ID,SHAPE),
    ↪ has_shape(ID',SHAPE), ID!=ID'.
material(T+1,MATERIAL) :- query_material
    ↪ (T), state(T,ID), has_material(ID,
    ↪ MATERIAL).
state(T+1,ID) :- filter_purple(T), state
    ↪ (T,ID), has_color(ID,purple).
state(T+1,ID) :- filter_sphere(T), state
    ↪ (T,ID), has_shape(ID,sphere).
state(T+1,ID') :- same_material(T),
    ↪ state(T,ID), has_material(ID,
    ↪ MATERIAL), has_material(ID',
    ↪ MATERIAL), ID!=ID'.
state(T+1,ID) :- filter_cylinder(T),
    ↪ state(T,ID), has_shape(ID,cylinder
    ↪ ).
state(T+1,ID) :- relate_behind(T), state
    ↪ (T,ID'), behind_of(ID,ID').
bool(T+1,yes) :- exist(T), state(T,ID),
    ↪ has_color(ID,red).
bool(T+1,no) :- exist(T), not state(T,ID
    ↪ ), has_color(ID,red).
state(T+1,ID) :- same_shape(T), state(T,
    ↪ ID), state(T',ID'), has_shape(ID,
    ↪ SHAPE), has_shape(ID',SHAPE), ID!=
    ↪ ID', T' < T.
state(T+1,ID) :- relate_front(T), state(
    ↪ T,ID'), in_front_of(ID,ID').
state(T+1,ID) :- filter_yellow(T), state
    ↪ (T,ID), has_color(ID,yellow).
state(T+1,ID) :- filter_rubber(T), state
    ↪ (T,ID), has_material(ID,rubber).
shape(T+1,SHAPE) :- query_shape(T),
    ↪ state(T,ID), has_shape(ID,SHAPE).
bool(T+1,yes) :- equal_shape(T,T'),
    ↪ shape(T,SHAPE), shape(T',SHAPE).
bool(T+1,no) :- equal_shape(T,T'), not
    ↪ bool(T+1,yes).

%Batch Size b=5 Response
state(T+1,ID) :- filter_cyan(T), state(T,
    ↪ ID), has_color(ID,cyan).
state(T+1,ID) :- filter_purple(T), state
    ↪ (T,ID), has_color(ID,purple).
state(T+1,ID) :- filter_metal(T), state(
    ↪ T,ID), has_material(ID,metal).

bool(T+1,yes) :- exist(T), state(T,ID).
bool(T+1,no) :- exist(T), not state(T,_).
    ↪
state(T+1,ID) :- filter_yellow(T), state
    ↪ (T,ID), has_color(ID,yellow).
state(T+1,ID) :- filter_rubber(T), state
    ↪ (T,ID), has_material(ID,rubber).
state(T+1,ID) :- filter_cube(T), state(T,
    ↪ ID), has_shape(ID,cube).
state(T+1,ID) :- filter_small(T), state(
    ↪ T,ID), has_size(ID,small).
state(T+1,ID) :- filter_large(T), state(
    ↪ T,ID), has_size(ID,large).
size(T+1,SIZE) :- query_size(T), state(T,
    ↪ ID), has_size(ID,SIZE).
state(T+1,ID) :- filter_rubber(T), state
    ↪ (T,ID), has_material(ID,rubber).
material(T+1,MATERIAL) :- query_material
    ↪ (T), state(T,ID), has_material(ID,
    ↪ MATERIAL).
state(T+1,ID) :- filter_sphere(T), state
```

```
    ↪ (T,ID), has_shape(ID,sphere).
shape(T+1,SHAPE) :- query_shape(T),
    ↪ state(T,ID), has_shape(ID,SHAPE).
state(T+1,ID) :- filter_cube(T), state(T,
    ↪ ID), has_shape(ID,cube).
state(T+1,ID) :- filter_sphere(T), state
    ↪ (T,ID), has_shape(ID,sphere).

state(T+1,ID') :- same_color(T), state(T,
    ↪ ID), has_color(ID,COLOR),
    ↪ has_color(ID',COLOR), ID!=ID'.
state(T+1,ID') :- same_material(T),
    ↪ state(T,ID), has_material(ID,
    ↪ MATERIAL), has_material(ID',
    ↪ MATERIAL), ID!=ID'.

material(T+1,MATERIAL) :- query_material
    ↪ (T), state(T,ID), has_material(ID,
    ↪ MATERIAL).
state(T+1,ID) :- filter_cylinder(T),
    ↪ state(T,ID), has_shape(ID,cylinder
    ↪ ).
state(T+1,ID) :- filter_sphere(T), state
    ↪ (T,ID), has_shape(ID,sphere).
state(T+1,ID) :- filter_red(T), state(T,
    ↪ ID), has_color(ID,red).
state(T+1,ID) :- filter_rubber(T), state
    ↪ (T,ID), has_material(ID,rubber).
state(T+1,ID) :- same_material(T), state
    ↪ (T,ID), has_material(ID,MATERIAL),
    ↪  has_material(ID',MATERIAL), ID!=
    ↪ ID'.
state(T+1,ID) :- same_shape(T), state(T,
    ↪ ID), has_shape(ID,SHAPE),
    ↪ has_shape(ID',SHAPE), ID!=ID'.
state(T+1,ID) :- same_size(T), state(T,
    ↪ ID), has_size(ID,SIZE), has_size(
    ↪ ID',SIZE), ID!=ID'.
state(T+1,ID) :- filter_purple(T), state
    ↪ (T,ID), has_color(ID,purple).
state(T+1,ID) :- filter_cylinder(T),
    ↪ state(T,ID), has_shape(ID,cylinder
    ↪ ).

state(T+1,ID) :- relate_behind(T), state
    ↪ (T,ID'), behind_of(ID,ID').

bool(T+1,yes) :- equal_material(T,T'),
    ↪ material(T,M), material(T',M), M=M
    ↪ '.
bool(T+1,no) :- equal_material(T,T'),
    ↪ not bool(T+1,yes).
shape(T+1,SHAPE) :- query_shape(T),
    ↪ state(T,ID), has_shape(ID,SHAPE).

bool(T+1,yes) :- equal_shape(T,T'),
    ↪ shape(T,SHAPE), shape(T',SHAPE),
    ↪ SHAPE=SHAPE'.
bool(T+1,no) :- equal_shape(T,T'), not
    ↪ bool(T+1,yes).

state(T+1,ID) :- relate_front(T), state(
    ↪ T,ID'), in_front_of(ID,ID').
state(T+1,ID) :- relate_behind(T), state
    ↪ (T,ID'), behind_of(ID,ID').

state(T+1,ID) :- filter_cylinder(T),
    ↪ state(T,ID), has_shape(ID,cylinder
    ↪ ).
state(T+1,ID) :- filter_sphere(T), state
    ↪ (T,ID), has_shape(ID,sphere).

%Batch Size b=10 Response
state(T+1,ID) :- filter_rubber(T), state
    ↪ (T,ID), has_material(ID,rubber).
state(T+1,ID) :- filter_yellow(T), state
    ↪ (T,ID), has_color(ID,yellow).
state(T+1,ID) :- filter_purple(T), state
    ↪ (T,ID), has_color(ID,purple).
state(T+1,ID) :- filter_cube(T), state(T,
    ↪ ID), has_shape(ID,cube).
state(T+1,ID) :- filter_red(T), state(T,
    ↪ ID), has_color(ID,red).
state(T+1,ID) :- filter_cyan(T), state(T,
    ↪ ID), has_color(ID,cyan).
state(T+1,ID) :- filter_sphere(T), state
    ↪ (T,ID), has_shape(ID,sphere).
shape(T+1,SHAPE) :- query_shape(T),
    ↪ state(T,ID), has_shape(ID,SHAPE).
material(T+1,MATERIAL) :- query_material
    ↪ (T), state(T,ID), has_material(ID,
    ↪ MATERIAL).
state(T+1,ID) :- filter_sphere(T), state
    ↪ (T,ID), has_shape(ID,sphere).
state(T+1,ID) :- filter_cube(T), state(T,
    ↪ ID), has_shape(ID,cube).
state(T+1,ID) :- filter_metal(T), state(
    ↪ T,ID), has_material(ID,metal).
state(T+1,SIZE) :- query_size(T), state(
    ↪ T,ID), has_size(ID,SIZE).
state(T+1,MATERIAL) :- query_material(T),
    ↪  state(T,ID), has_material(ID,
    ↪ MATERIAL).
state(T+1,SHAPE) :- query_shape(T),
    ↪ state(T,ID), has_shape(ID,SHAPE).
state(T+1,ID) :- filter_cylinder(T),
    ↪ state(T,ID), has_shape(ID,cylinder
    ↪ ).
state(T+1,ID') :- same_shape(T), state(T,
    ↪ ID), has_shape(ID,SHAPE),
    ↪ has_shape(ID',SHAPE), ID!=ID'.
state(T+1,ID') :- same_material(T),
    ↪ state(T,ID), has_material(ID,
    ↪ MATERIAL), has_material(ID',
    ↪ MATERIAL), ID!=ID'.
state(T+1,ID') :- same_size(T), state(T,
    ↪ ID), has_size(ID,SIZE), has_size(
    ↪ ID',SIZE), ID!=ID'.
state(T+1,ID') :- same_color(T), state(T,
    ↪ ID), has_color(ID,COLOR),
    ↪ has_color(ID',COLOR), ID!=ID'.
state(T+1,ID) :- same_size(T), state(T,
    ↪ ID), has_size(ID,SIZE), has_size(
    ↪ ID',SIZE), ID!=ID', state(T,ID').
state(T+1,ID) :- same_material(T), state
    ↪ (T,ID), has_material(ID,MATERIAL),
    ↪  has_material(ID',MATERIAL), ID!=
    ↪ ID', state(T,ID').
state(T+1,ID) :- relate_front(T), state(
    ↪ T,ID'), in_front_of(ID,ID').
```

```
state(T+1,ID) :- relate_behind(T), state
    ↪ (T,ID'), behind_of(ID,ID').
bool(T+1,yes) :- equal_material(T,T'),
    ↪ material(T,V), material(T',V'), V=
    ↪ V'.
bool(T+1,no) :- equal_material(T,T'),
    ↪ not bool(T+1,yes).

bool(T+1,yes) :- equal_shape(T,T'),
    ↪ shape(T,V), shape(T',V'), V=V'.
bool(T+1,no) :- equal_shape(T,T'), not
    ↪ bool(T+1,yes).
```

We leave out the rules learnt on the random removal experiment, as they may be hard to interpret, but the reader can inspect them in the logs folder provided.