Leveraging Debiased Cross-modal Attention Maps and Code-based Reasoning for Zero-shot Referring Expression Comprehension

Supplementary Material

Models	Venue	Cops-Ref val	Cops-Ref test
Supervised SOTA			
CM-Att-Erase [35]	CVPR20	-	80.40
Other zero-shot method			
ReCLIP [56]	ACL22	30.50	30.75
FGVP [67]	NIPS23	30.87	30.97
GDINO-T [34]	ECCV24	56.87	59.37
GVLP w/ ALBEF [49]	AAAI24	36.27	37.64
Ours			
GDINO-T w/ ALBEF	-	<u>60.33</u>	<u>62.60</u>
GDINO-T w/ BEiT-3-vqa	-	**61.98**	**64.25**

Table 6. **Comparisons with state-of-the-art methods on the val and test set of Cops-Ref.** Best and second best performance for zero-shot methods are in **bold** and <u>underline</u>, respectively.

7. Details of Program Generation and Execution

During the generation process, for each query, we fill the full-query description, the name of the referring subject, and those of other objects parsed by the LLM into the part of input of the prompt template, described in Sec. 12, thereby forming a complete prompt. We then configure the LLM and input the prompt. Typically, the LLM can correctly understand the output format requirements and returns a single code block wrapped with ` ```python ` and ` ``` `. The block defines a function with fixed name `reason` and parameters `Subject` and `Objects`, identical to the examples in Sec. 12.

During the execution process, we first use the built-in python regular expression to extract the string of the code block, in case the LLM does not strictly follow the prompt and outputs additional non-code information. We then use the built-in `exec(string)` function to define the `reason(Subject, Object)` function within an isolated namespace. Finally, within this namespace, we can dynamically invoke the function by its name using a variable call, passing in ”reason” as the function name along with the corresponding Subject and Object detection results as parameters. This enables automatic execution of the generated program.

8. Additional Experiment Results

Experiments on Cops-Ref dataset. We report additional evaluation results on the recent and semantically rich Cops-Ref dataset [9], which is specifically designed to benchmark models on multi-hop reasoning, complex queries involving logical operators (e.g., “and”, “or”, “same”, “not”) and ordinal relations (e.g., “first”). In comparison to the RefCOCO series, Cops-Ref features longer queries and a wider variety of categories, attributes, and relations.

The results in Tab. 6 indicate that our method consistently outperforms prior models on the Cops-Ref dataset. It also shows the flexibility of our approach, which, by leveraging language-based OVDs, can generalize across diverse datasets without requiring dataset-specific training for object detectors.

Model for $\mathcal{M}_{\text{v-bias}}$	RefCOCO	RefCOCO+	RefCOCOg	RefIt
w/o debias	60.03	53.11	63.41	51.59
ALBEF	**64.50**	**57.62**	**65.64**	**54.50**
DINOv2-L	62.93	53.61	63.62	52.07
MAE-H	62.65	55.35	64.05	52.42

Table 7. **Comparison on using different models to obtain $\mathcal{M}_{\text{v-bias}}$.** “RefIt” represents “ReferItGame”.

Attention Map	RefCOCO	RefCOCO+	RefCOCOg	RefIt
Using BEiT-3-vqa				
$\mathcal{M}_{\text{ori}}$	70.49	65.05	70.42	57.32
$\mathcal{M}^{+}_{\text{debias}}$	71.75	**66.65**	**71.60**	58.34
$\mathcal{M}^{+}_{\text{debias-l}}$	70.75	63.71	63.20	57.24
Average	**72.08**	65.63	63.75	**58.42**
Using ALBEF				
$\mathcal{M}_{\text{ori}}$	60.02	53.11	63.41	51.59
$\mathcal{M}^{+}_{\text{debias}}$	**64.50**	**57.62**	**65.64**	**54.50**
$\mathcal{M}^{+}_{\text{debias-l}}$	62.82	52.78	62.82	52.08
Average	64.34	55.29	64.35	54.02

Table 8. **Comparison on using attention maps from different debiasing strategies across two VLMs.** “RefIt” represents “ReferItGame”. “Average” means we use the mean value of $\mathcal{M}^{+}_{\text{debias}}$ and $\mathcal{M}^{+}_{\text{debias-l}}$ as the final attention map.

Reasoning Formulation	RefCOCO	RefCOCO+	RefCOCOg	RefIt
Using BEiT-3-vqa				
full	**71.75**	**66.65**	**71.60**	**58.34**
w/o rel. decompose	70.02	66.14	68.81	56.64
w/o discrimination	66.69	57.88	53.79	57.86
Using ALBEF				
full	**64.50**	**57.62**	**65.64**	**54.50**
w/o rel. decompose	61.82	56.98	62.42	52.88
w/o discrimination	59.26	49.34	48.71	54.44

Table 9. **Comparison on different designs of the general reasoning formulation.** “RefIt” represents “ReferItGame”.

Experiments on using vision-only models for visual bias. We also experiment with an alternative approach to obtaining the visual bias $\mathcal{M}_{\text{v-bias}}$, where it no longer derives from the VLMs with purely visual input, but instead from vision-only models such as DINOv2 [42] or MAE [20]. We anticipate that these models could provide more accurate object-centric semantic activations.

However, as shown in Tab. 7, under the same setting where $\mathcal{M}_{\text{ori}}$ is obtained from ALBEF, although $\mathcal{M}_{\text{v-bias}}$ from vision-only models also contributes to the subsequent debiasing process, its effectiveness is inferior to directly leveraging the visual bias from the VLM. We attribute this to the inherent characteristics of multi-modal models: while vision-only models may highlight the most visually salient regions, there still exists a semantic discrepancy between these regions and those implicitly activated by VLMs in the absence of textual input. We believe this is a promising direction for future investigation.

Effect of debiasing strategies. We conduct experiments to explore different debiasing strategies as the alternatives of removing bias towards visual modality in Sec. 3.2. We take the possible bias towards language modality into consideration. Specifically, we input the original text query and the masked image into the VLM to obtain the attention map of the language bias $\mathcal{M}_{\text{l-bias}}$. We then obtain the language-debiased attention map $\mathcal{M}_{\text{debias-l}}$ by computing their difference similar to Eq. (3), followed by filtering negative values to get $\mathcal{M}^{+}_{\text{debias-l}}$. We compare the results of combining $\mathcal{M}^{+}_{\text{debias-l}}$, $\mathcal{M}^{+}_{\text{debias}}$ and using them separately.

The results, presented in Tab. 8, shows that in comparison to using the vanilla map, debiasing for the language modality brings performance improvements in RefCOCO and ReferItGame while leads to performance declines in RefCOCO+ and RefCOCOg. We argue that it might be attributed to the spatial descriptions included in RefCOCO and ReferItGame, where the VLMs struggle to understand them in visual modality and tend to rely on the language modality to make predictions. Overall, its performance is inferior to that of debiasing only for the visual modality. When simply using their average as the final attention map, we observe a performance gain on RefCOCO and ReferItGame for BEiT-3-vqa while a slight decrease for ALBEF. These differences underscores the importance of delving into the causes of the bias and exploring more advanced debiasing approaches.

Effect of designs of the general reasoning formulation. We compare the performance of some variants of our designed reasoning formulation. As shown in Tab. 9, the absence of relation decomposition increases the difficulty for LLMs to comprehend complex relations and exacerbates the impact of small errors on the entire program, resulting in a clear accuracy drop. It emphasizes the importance of formulating the reasoning process in a general pattern and decomposing the understanding of complex queries into combinations of simpler ones, which enhances the explainability and robustness of the program. Another variant without discrimination of different kinds of proposals takes all detected proposals instead of referring subjects' proposals $\mathcal{P}^{\text{sub}}$ as the input of the reasoning program. It can be observed that this setting undoubtedly increase the difficulty of further reasoning due to irrelevant proposals. Performance on all datasets exhibit declines, particularly on the RefCOCOg dataset, which features longer text queries with more object names.

LLM	RefCOCO	RefCOCO+	RefCOCOg
Qwen-plus	0.05%	0.16%	0.22%
GPT-4o-mini	0.15%	0.17%	0.26%
Qwen2.5-Coder-32B	0.13%	0.10%	0.12%
Qwen2.5-Coder-14B	0.03%	0.08%	0.10%

Table 10. **Proportion of suspect programs containing hard-matching function.**

9. Further Analysis of Generated Programs

To generate soft matching functions and reduce the possible "hard-coding" caused by the hallucination of LLMs, some prompting strategies are adopted to guide LLMs in adhering to the principle of scoring based on the degree of matching. First, we explicitly instruct LLMs to reason based on the degree of matching instead of directly returning a predicted proposal. Second, as shown in Sec. 12, continuous matching and score smoothing are used throughout all in-context examples.

We also conduct a statistical experiment based on syntactic parsing to find possible hard matching functions. To be specific, if any comparison operator is present in a generated program and one of its operands is a constant, we label it as a suspicious 'hard-coding' instance. Results in Tab. 10 demonstrate that very few programs may perform hard matching. To generate more robust and flexible spatial reasoning programs, it may be worthwhile to explore the usage of more powerful LLMs and build up a close-loop code generation system with feedback and iterative refinement.

10. Descriptions of Utilized Models

10.1. Open-Vocabulary Detectors

GLIP. GLIP [29] is a language-guided detector reformulating detection as phrase-region matching. It performs early fusion of multi-modal features and is trained for phrase-region alignment. This design allows arbitrary text and image as input to locate corresponding regions. We use its GLIP-T (C) checkpoint pretrained on O365 and GoldG datasets (COCO images excluded). The detection threshold for filtering proposals is set to 0.4, while other settings are kept the same as default.

GDINO (GroundingDINO). GDINO [34] is a a dual-tower detector combining a Swin-Transformer [37] image backbone with a BERT-base text encoder [13]. It features performing cross-modal fusion at all stages and using subsentence for text feature representation. It also differs by leveraging a cross-modal decoder to computes text-proposal similarity via cross-attention. We use the public available GDINO-T-OGC checkpoint pretrained on O365,GoldG and Cap4M datasets (COCO images excluded) with a Swin-T backbone. We set the detection threshold to 0.2, while other settings are kept the same as default.

10.2. Vision Language Models

ALBEF. ALBEF [28] is a representative of the first paradigm of cross-modal fusion mentioned in Sec. 3.2. It employs cross-attention mechanisms for cross-modal fusion, where the output features of the image encoder interact with the features from last 6th layers of the text encoder (these 6th layers serve as the multi-modal encoder). [28, 49, 56]. In our experiments, we use the versions of ALBEF-14M and its corresponding vqa checkpoints. The 3rd layer of the multi-modal encoder is chosen to obtain the cross-modal attention maps in consistent with previous GradCAM-based methods.

TCL. TCL [65] improves ALBEF by integrating three contrastive modules to enhance the global-local fine-grained alignment within each modality as well as between two modalities. It also uses cross-attention to perform cross-modal fusion. We use its public available TCL-4M checkpoint for our experiment.

BEiT-3. BEiT-3 [60] is a typical VLM for the second fusion paradigm. It employs a unified multi-modal encoder leveraging shared self-attention layers for cross-modal fusion. It uses masked image modeling (MIM) as its core training objects instead of ITM, thus unable to be directly utilized for GradCAM. We uses two public available checkpoints, BEiT3-large-patch16-224 and BEiT3-large-patch16-480-vqa for the vanilla BEiT-3 and BEiT-3-vqa, respectively.

10.3. Large Language Models

For all LLMs, we set their temperatures for generation to 0.2 to balance generation diversity and stability, with other settings identical to their default configuration.

Qwen-plus. Qwen-plus [4] is a commercial LLM for general-purpose conversations. We obtain the generated programs via API calls [*].

GPT-4o-mini. GPT-4o-mini [24] is a lightweight variant of GPT-4, designed for faster general-purpose dialogue. We also obtain the generated programs via API calls [†].

Qwen2.5-Coder. Qwen2.5-Coder [23] is a code-specialized model. It demonstrates an excellent programming capability, supporting code completion for many programming languages (e.g., Python, Java). We use the public available version of two sizes, Qwen2.5-Coder-14B-Instruct and Qwen2.5-Coder-32B-Instruct for our experiments.

11. Complete Programs of Visualizations

In this section we demonstrate the complete generated programs whose core code snippets shown in Fig. 5. Please note that the input proposals are first transformed into the format of $(x_\mathrm{c}, y_\mathrm{c}, w_\mathrm{p}, h_\mathrm{p})$, which are the horizontal and vertical coordinates of the center of the proposal, and the width and height of the proposal, respectively.

- *"green van 7 o clock"* (Fig. 5 (a)): Listing 1
- *"a vase that is shorter and rounder"* (Fig. 5 (b)): Listing 3
- *"no hat guy"* (Fig. 5 (c)): Listing 4
- *"a flower vase that is round at the bottom and a narrow cylinder at the top"* (Fig. 5 (d)): Listing 2

12. Full Prompt for Generating Programs

We demonstrate the complete prompt including the general formulation and examples in Sec. 12

[*]https://help.aliyun.com/zh/model-studio/developer-reference/use-qwen-by-calling-api

[†]https://openai.com/blog/openai-api

```python
import numpy as np
def reason(Subject, Objects):
    # Reasoning step by step
    # 1. The query can be decomposed into 2 sub-queries: "green van", "van 7 o clock".
    van_lst = Subject

    # 2. "green van": "green" is an incalculable attribute, so we just skip this sub-query.

    # 3-1. "van 7 o clock" means we can reason and calculate depending on the position of each van
     relative to the 7 o'clock direction.
    # 3-2. Calculate the angle of each van relative to the center of the image.
    center_x, center_y = 320, 240
    van_angle_lst = []
    for van in van_lst:
        dx, dy = van[0] - center_x, van[1] - center_y
        angle = np.arctan2(dy, dx) * 180 / np.pi
        if angle < 0:
            angle += 360
        van_angle_lst.append(angle)

    # 3-3. Normalize all the angles to make them comparable.
    van_angle_lst = [x/max(van_angle_lst) for x in van_angle_lst]

    # 3-4. "7 o clock" corresponds to an angle around 210 degrees (considering 0 degree at 3 o'
    clock and increasing counter-clockwise).
    target_angle = 210
    van_angle_diff_lst = [np.abs(angle - target_angle) for angle in van_angle_lst]
    van_angle_diff_lst = [x/max(van_angle_diff_lst) for x in van_angle_diff_lst]

    # 3-5. "AT 7 o clock" means the score should be negatively correlated with the angle
    difference.
    for van in van_lst:
        van[-1] -= van_angle_diff_lst[van_lst.index(van)]

    # 4. Return the final Subject info list.
    return van_lst
```

Listing 1. **Full Program for Fig. 5 (a).**

```python
import numpy as np
def reason(Subject, Objects):
    # Reasoning step by step.
    # 1. The query can be decomposed into 2 sub-queries: "vase round at the bottom", "vase narrow
    cylinder at the top".
    vase_lst = Subject
    # 2-1. "vase round at the bottom" is not a calculable attribute, skip it.
    # 2-2. "vase narrow cylinder at the top" is also not a calculable attribute, skip it.
    # Since both sub-queries describe attributes that are not calculable based on the provided
    information, no changes to scores.
    # 3. Return the final Subject list.
    return vase_lst
```

Listing 2. **Full Program for Fig. 5 (d).**

```python
import numpy as np
def reason(Subject, Objects):
    # Reasoning step by step.
    # 1. The query can be decomposed into 2 sub-queries: "vase that is shorter", "vase that is rounder".
    vase_lst = Subject
    # 2-1. "vase that is shorter" means we reason and calculate depending on the height of each vase.
    vase_h_lst = [vase[3] for vase in vase_lst]
    if max(vase_h_lst) != min(vase_h_lst): # avoid division by zero
        vase_h_lst = [(max(vase_h_lst) - h) / (max(vase_h_lst) - min(vase_h_lst)) for h in vase_h_lst]
    else:
        vase_h_lst = [1.0] * len(vase_h_lst)
    # 2-2. Noting we want the vase to be SHORTER, so the score should be positively correlated with the inverse of height.
    for vase in vase_lst:
        vase[-1] += vase_h_lst[vase_lst.index(vase)]
    # 3-1. "vase that is rounder" means we reason and calculate depending on the width-to-height/height-to-width ratio of each vase.
    vase_ratio_lst = [min(vase[2], vase[3]) / max(vase[2], vase[3]) for vase in vase_lst]
    if max(vase_ratio_lst) != min(vase_ratio_lst): # avoid division by zero
        vase_ratio_lst = [(ratio - min(vase_ratio_lst)) / (max(vase_ratio_lst) - min(vase_ratio_lst)) for ratio in vase_ratio_lst]
    else:
        vase_ratio_lst = [1.0] * len(vase_ratio_lst)
    # 3-2. Noting we want the vase to be ROUNDER, which means a ratio close to 1, so the score should be positively correlated with the ratio.
    for vase in vase_lst:
        vase[-1] += vase_ratio_lst[vase_lst.index(vase)]
    # 4. Return the final Subject list.
    return vase_lst
```

Listing 3. **Full Program for Fig. 5 (b).**

```python
import numpy as np
def reason(Subject, Objects):
    # Reasoning step by step.
    # 1. The query can be decomposed into 1 sub-query: "no hat guy".
    person_lst = Subject
    # 2. "no hat guy" means we reason and calculate depending on the distance between each person and any hat.
    hat_lst = Objects["hat"]
    if len(hat_lst) == 0: # if no "hat", all persons are considered to have no hat.
        for person in person_lst:
            person[-1] += 1
    else:
        # 3. Calculate the distance of each person and its nearest hat, then normalize.
        person_hat_dis_lst = []
        for person in person_lst:
            person_hat_dis = min([np.sqrt((person[0]-hat[0])**2+(person[1]-hat[1])**2) for hat in hat_lst])
            person_hat_dis_lst.append(person_hat_dis)
        person_hat_dis_lst = [x/max(person_hat_dis_lst) for x in person_hat_dis_lst]
        # 4. Noting we want the person WITHOUT hat, the score should be positively correlated with the distance.
        for person in person_lst:
            person[-1] += person_hat_dis_lst[person_lst.index(person)]
    # 5. Return the final Subject list.
    return person_lst
```

Listing 4. **Full Program for Fig. 5 (c).**

~~~
Your task is to generate an executable Python program based on the following input.
"Query": A sentence or phrase referring to a unique "Subject" instance.
"Subject_name": The name of the unique referent "Subject" in the "Query".
"Object_names": A list of names access to argument "Objects".
The program should perform the following steps: 1)Decompose the query into several independent sub-queries focusing on "Subject" if it contains multiple relationships or attributes. 2)For each sub-query that contains calculable relationships or attributes, convert its spatial or logical relationships and attributes into computational code based on coordinates. Then normalize and transform the computation results to obtain scores representing the degree of match between the "Subject" and the sub-query. 3)Add the scores of all sub-queries to the original score of each "Subject" instance.
The program should return the list of "Subject" instances.

The programs you generate will take the following two kwargs as input:
Subject: [[x1,y1,w1,h1,s1],..., [xn,yn,wn,hn,sn]] # [x,y,w,h,s] represent the horizontal coordinate and vertical coordinate of the center, the width, the height, and the matching score of each instance(initialize as 0), respectively.
Objects: {{
    "name of the first object": [[x1,y1,w1,h1,s1],...,[xm,ym,wm,hm,sm]],
    ...
}}

Here are some examples of input information and output programs:
[1]
"Query": "right sofa behind, farthest away from a dog"
"Subject_name": "sofa"
"Object_names": ["dog"]
```python
import numpy as np
def reason(Subject, Objects): # the names of the function and the arguments should be fixed.
    # Reasoning step by step
    # 1. This query can be decomposed into 3 sub-queries: "right sofa", "sofa behind", "sofa farthest away from a dog".
    sofa_lst = Subject
    # 2-1. "right sofa" means we reason and calculate depending on each sofa's normalized x-value.
    sofa_x_lst = [sofa[0] for sofa in sofa_lst]
    sofa_x_lst = [x/max(sofa_x_lst) for x in sofa_x_lst]
    # 2-2. Noting the x-value goes larger as it goes nearer to the right and we want to be near the right, the score should be positively correlated with x-value.
    for sofa in sofa_lst:
        sofa[-1] += sofa_x_lst[sofa_lst.index(sofa)]
    # 3. "sofa behind" means we want the sofa in the back of the scene, however, we can't reason on this sub-query because the depth information is not provided.
    # 4-1. "sofa farthest away from a dog" means we reason and calculate depending on the distance between each sofa and dog.
    # Word for word, use "dog" in "Object_names": ["dog"] above to access argument Objects.
    dog_lst = Objects["dog"]
    if len(dog_lst) == 0: # if no "dog", skip.
        pass
    else:
        # 4-2. Calculate the distance of each sofa and its nearest dog, then normalize.
        sofa_dog_dis_lst = []
        for sofa in sofa_lst:
            sofa_dog_dis = min([np.sqrt((sofa[0]-dog[0])**2+(sofa[1]-dog[1])**2) for dog in dog_lst])
            sofa_dog_dis_lst.append(sofa_dog_dis)
        sofa_dog_dis_lst = [x/max(sofa_dog_dis_lst) for x in sofa_dog_dis_lst]
        # 4-3. "FARTHEST" means largest distance, so the score should be positively correlated with the distance.
        for sofa in sofa_lst:
            sofa[-1] += sofa_dog_dis_lst[sofa_lst.index(sofa)]
    # 5. Return the final Subject list.
    return sofa_lst
```
~~~

~~~
[2]
"Query": "drawing second from bottom"
"Subject_name": "drawing"
"Object_names": []
```python
import numpy as np
def reason(Subject, Objects):
    # Reasoning step by step
    # 1. The query can’t be decomposed into multiple sub-queries, so we reason directly.
    drawing_lst = Subject
    # 2. "FROM BOTTOM" means y axis from largest to smallest, so we can sort the instances by y axis in descending order.
    drawing_lst = sorted(drawing_lst, key=lambda x: x[1], reverse=True) # sort the instances by y axis in descending order.
    # 3. The target instance is the second one in the sorted list, whose index is 1, so we can assign a higher score to it.
    for i in range(len(drawing_lst)):
        drawing_lst[i][-1] += 1.0/(np.abs(i-1)+1)
    # 4. Return the final Subject list.
    return drawing_lst
```

[3]
"Query": "bear in the corner with gray shirt, looks dirty"
"Subject_name": "bear"
"Object_names": ["gray shirt"]
```python
import numpy as np
def reason(Subject, Objects):
    # Reasoning step by step.
    # 1. The query can be decomposed into 3 sub-queries: "bear in the corner", "bear with gray shirt", "bear looks dirty".
    bear_lst = Subject
    # 2-1. "bear in the corner" means we reason and calculate by the distance of each bear and its nearest corner.
    corners = [
        (0, 0), # top left
        (640, 0), # top right
        (0, 480), # bottom left
        (640, 480) # bottom right
    ] # Noting the y-value goes larger as y goes down!
    # 2-2. Calculate the distance of each bear and the its nearest corner, then normalize.
    bear_cor_dis_lst = []
    for bear in bear_lst:
        bear_cor_dis = min([np.sqrt((x-bear[0])**2+(y-bear[1])**2) for x,y in corners])
        bear_cor_dis_lst.append(bear_cor_dis)
    bear_cor_dis_lst = [x/max(bear_cor_dis_lst) for x in bear_cor_dis_lst]
    # 2-3. Noting we want the bear IN the corner, so the score should be negatively correlated with the distance.
    for bear in bear_lst:
        bear[-1] -= bear_cor_dis_lst[bear_lst.index(bear)]
    # 3-1. "bear with gray shirt" means we reason and calculate depending on the "bear" and "gray shirt" instances.
    # Word for word, use "gray shirt" in "Object_names": ["gray shirt"] above to access argument Objects.
    gray_shirt_lst = Objects["gray shirt"]
    if len(gray_shirt_lst) == 0: # if no "gray shirt", skip.
        pass
~~~

```
    else:
        # 3-2. Calculate the distance of each bear and its nearest shirt, then normalize.
        bear_shirt_dis_lst = []
        for bear in bear_lst:
            bear_shirt_dis = min([np.sqrt((bear[0]-shirt[0])**2+(bear[1]-shirt[1])**2) for shirt
    in gray_shirt_lst])
            bear_shirt_dis_lst.append(bear_shirt_dis)
        bear_shirt_dis_lst = [x/max(bear_shirt_dis_lst) for x in bear_shirt_dis_lst]
        # 3-3. Noting we want the bear WITH shirt, the score should be negatively correlated with
    the distance.
        for bear in bear_lst:
            bear[-1] -= bear_shirt_dis_lst[bear_lst.index(bear)]
    # 4. "bear looks dirty": "dirty" is not a calculable attribute, skip it.
    # 5. Return the final Subject list.
    return bear_lst
```

Here are some tips:
# Remember the program's input args have a fixed format and content. NEVER assume any infomation
    not being provided!!!
# You should skip all incalculable sub-queries ONLY describing attributes like color, material,
    behavior and pattern.
# You should try to convert calculable attributes, such as relative relationships, orientation,
    size, distance, etc., into soft score.
# The top/up left, top/up right, bottom/down left, bottom/down right corner of the image is always
     (0,0), (640,0), (0,480), (640,480).
# DO NOT include unnecessary information such as type annotations in the program.
# Your answer should ONLY contain a valid Python program of fixed function name and arguments.

"Query": {Query}
"Subject_name": {Subject}
"Object_names": [{Objects}]
Please think step by step and return a python program without any additional information.
```

Listing 5. **Full prompt of the general formulation and examples across all datasets.**
```