

SceneMotifCoder: Example-driven Visual Program Learning for Generating 3D Object Arrangements

Supplementary Material

We provide additional implementation details for the baseline approaches (Appendix A.1) and demonstration of how we can use OpenCLIP [1] for more fine-grained retrieval of matching objects (Appendix A.2). We also provide evaluation details (Appendix B), additional examples of generated 3D arrangements (Appendix C), and generated programs (Appendix D). Finally, we include the LLM prompts we used (Appendix E) and an example LLM session (Appendix F).

A. Implementation Details

A.1. Baseline Details

For MVDream, we use the default configuration without soft shading and lower the second stage resolution to 128 for computational efficiency. Since GraphDreamer’s public implementation only takes scene graphs as input and the prompts used in the paper are not available, we implement a text-to-scene-graph module using GPT-4 and follow the general strategy from the paper. We also use the hyperparameter settings in GraphDreamer’s demo scripts for all objects, except that we set center dispersion to 0.2 to allow the SDF for each object to be initialized at a reasonable distance from other objects. Similar to MVDream, we lower GraphDreamer’s 2nd stage resolution to 128 with batch size 1 to minimize VRAM usage. Generating all 202 arrangements took a combined 152 GPU hours on three Nvidia L40S GPUs for MVDream, and 263 hours on four L40S and two A40 GPUs for GraphDreamer. GraphDreamer often crashes due to out-of-memory errors for object arrangements that consist of more than three objects or contain larger objects. For these arrangements, we re-run on two A100 GPUs with 80GB VRAM, which took another 181 GPU hours. However, twelve input descriptions still crash due to out-of-memory errors.

A.2. Mesh Retrieval Approach

Our work retrieves meshes to instantiate object arrangements based on object category and bounding box dimensions. We tried incorporating OpenCLIP [1] to enable object appearance-based retrieval. However, since our evaluation is focused on the generated object layouts, our test descriptions reflect that and do not specify appearance attributes such as colour. As a result, the retrieved meshes do not differ significantly from the simpler approach when evaluated against the input text description. Therefore, we opted for the simpler approach in our final system. We note

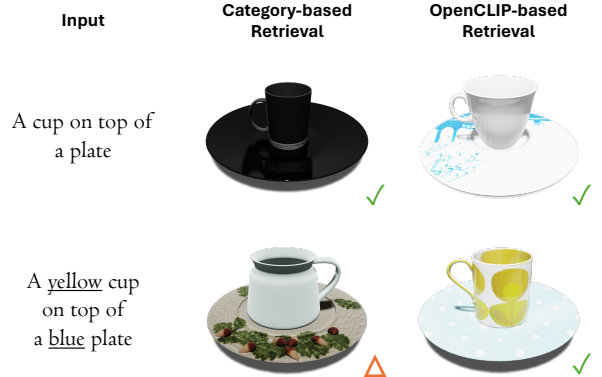


Figure 1. **Results using category-based and OpenCLIP-based retrieval.** First row: Input description does not specify object appearance. Both results are correct. This is the case for our test set descriptions. Second row: OpenCLIP-based retrieval is useful when the input description contains appearance attributes.

that in real life usage, users may have specific preferences to the appearances of the generated arrangements, making the feature-based retrieval module a worthwhile extension to our work. See Figure 1 for a demonstrative example.

B. Evaluation Details

B.1. Manual Verification Guidelines

We provide the guidelines the annotator received below.

Looking at two views of each generation result, judge the generation quality based on the following three criteria. Give a binary judgment (Y/N) for each.

- # Objs - Correct number of objects as specified in the text description. Can you reasonably say there are X instances of the object?
- Layout - Correct layout in terms of the relative positions between objects as specified in the text description. Ignoring the appearances and labels of the objects, are the objects arranged correctly? If you were to classify the result into one of the motif types, will it fall into the same type as the description?
- Plaus - Physical Plausibility. Is the generation result physically plausible? Can the result exist in real life?

B.2. Perceptual Study Instructions

We provide the instructions the study participants received below.

This study asks you to judge the quality of 3D object arrangements in terms of

- **Text Alignment** - does the object arrangement fit the text description? E.g., for “a stack of three books”, an arrangement with two books stacked on top of each other does not fit the description.
- **Realism** - is the object arrangement realistic? You should disregard the fine-grained style and textured appearance of the objects. E.g., a chair that changes shape across view points and a teddy bear with five legs are unrealistic.

Judge the arrangements solely on these two aspects. In each question you will see a text description followed by two object arrangements Left and Right, with a short video showing you different views of the arrangements. Select the better arrangement according to the two aspects above. Each question should take 5-10 sec. There are 60 questions in total.

C. Additional Qualitative Results

In Fig. 2 we see how a meta-program can be called with varying text descriptions to produce quite distinct, yet realistic arrangements. This is a concrete example of the reusability and generality of our framework.

Figure 3 provides additional qualitative comparisons of generated object arrangements with prior work. Figure 4 provides additional examples of arrangements generated using SceneMotifCoder. The results showcase a variety of motif types, exhibiting different spatial arrangements.

In addition, Fig. 5 shows how SceneMotifCoder’s generated arrangements can be used as part of generating new compositional arrangements. By treating previously generated arrangements as retrieval assets, SMC can invoke learned meta-programs to retrieve them and compose new arrangements with extra layers of complexity. The results show that SMC can generate new arrangements that are significantly more complex than the original ones within a few iterations, demonstrating SMC’s potential at generating complex and diverse arrangements.

D. Example Programs

In Tables 1 and 2, we provide example meta-programs generated by SceneMotifCoder, and compare them to two simpler prompting strategies:

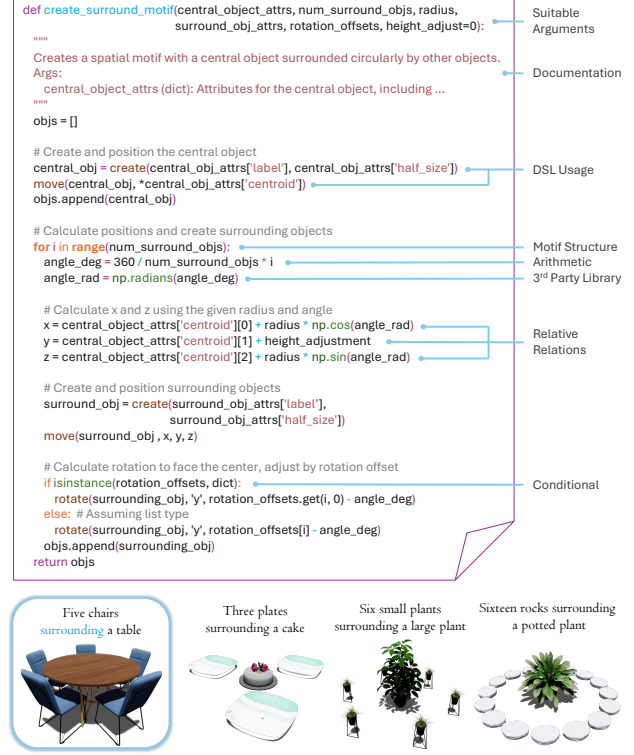


Figure 2. Meta-program for `surround` motif learned from the text and arrangement pair in blue box. The arrangements at the bottom are generated from the meta-program given the input text above each arrangement. Note the significant variations in object counts and types, all captured by human-interpretable and editable calls to the same meta-program.

- *direct-from-description*: the LLM is prompted directly for a program from a description of a spatial motif, without information from example arrangements.
- *direct-from-motif-program*: the LLM is prompted to generate a meta-program given a set of motif programs, without any observation and reasoning steps.

The two *direct-from-description* programs create naïve lists of repetitive statements with hard-coded values. They cannot adapt to changes in the input description and cannot be reused to generate other arrangements. While the *direct-from-motif-program* programs are more flexible, they make strong assumptions on the object poses in their program structures. These assumptions result in programs that lack generality, compared to the SceneMotifCoder meta-programs which are compact, human-readable, and admit many variations through setting of appropriate arguments in the function signature. Analyzing the programs shows that 58% of the *direct-from-motif-program* programs, compared to only 5% of meta-programs from SMC, have this issue.

These comparisons of concrete code examples help to demonstrate the value of the SceneMotifCoder visual program synthesis framework.

Example	Input	MVDream	GraphDreamer	LayoutPrompter	SMC (ours)
 A cup on a saucer	A potted plant on top of a book				
 A stack of seven plates	A stack of two cups				
 A row of three tables	A row of two potted plants				
 A letter P made of cups	A letter P made of plates				
 A row of three tables	A row of three jugs				
 A stack of seven plates	A stack of four books				
 A grid of six cups	A rectangle grid of four potted plants				
 A footrest in front of an armchair	A cup in front of a coffee maker				
 Five chairs surrounding a circular table	Three chairs circularly surrounding a table				

Figure 3. **Additional qualitative comparisons of generated 3D object arrangements.** Each blue box on the left is the text and arrangement pair used for learning a meta-program capturing the arrangement motif. The SceneMotifCoder arrangements better respect the input text in terms of number and type of objects present, and exhibit more plausible spatial configurations and object appearance.

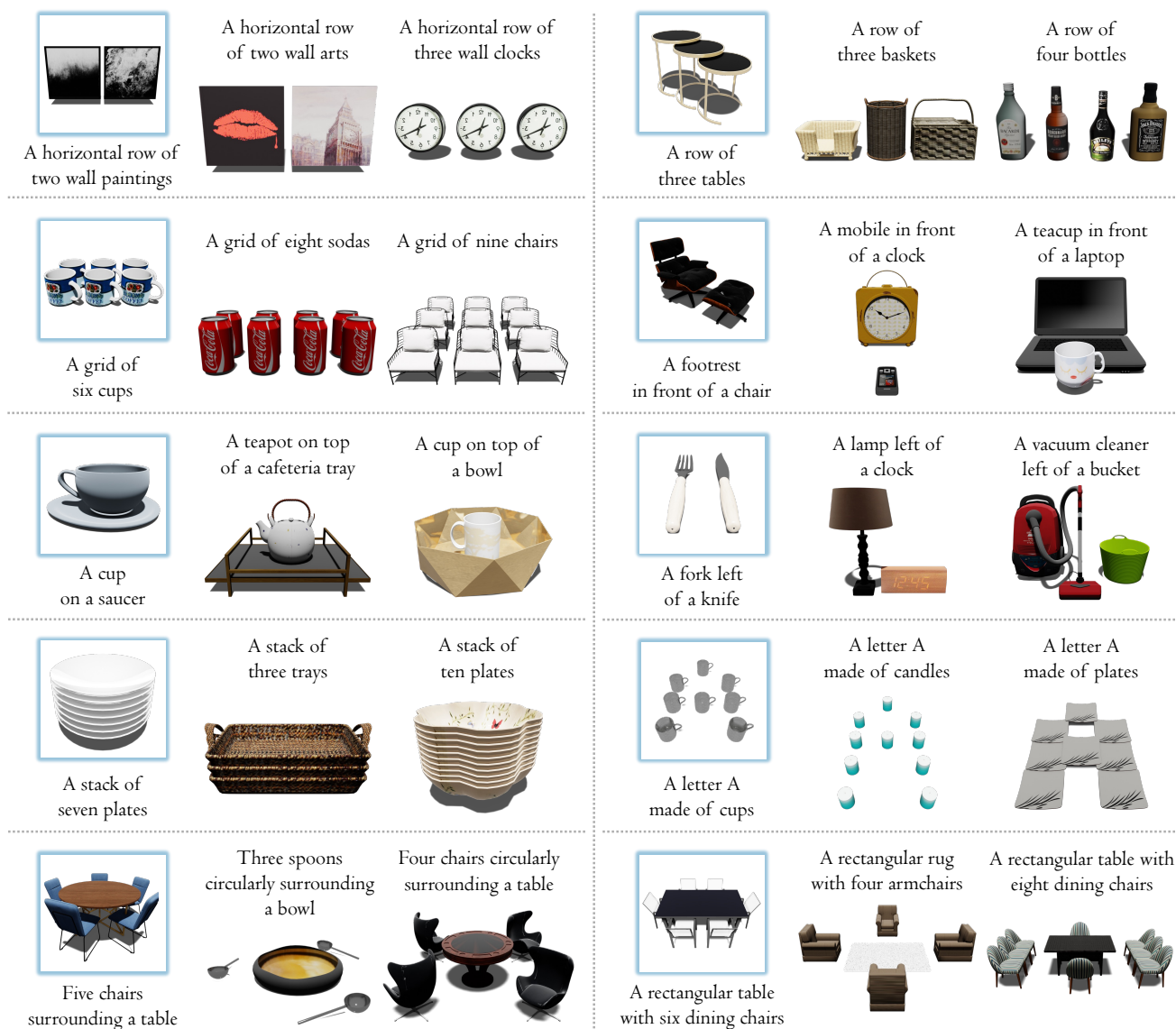


Figure 4. **Additional examples of generated object arrangements.** Our SceneMotifCoder framework generalizes the motifs exhibited by the text and arrangement example to produce diverse arrangements with differing object categories, counts, and spatial configurations.

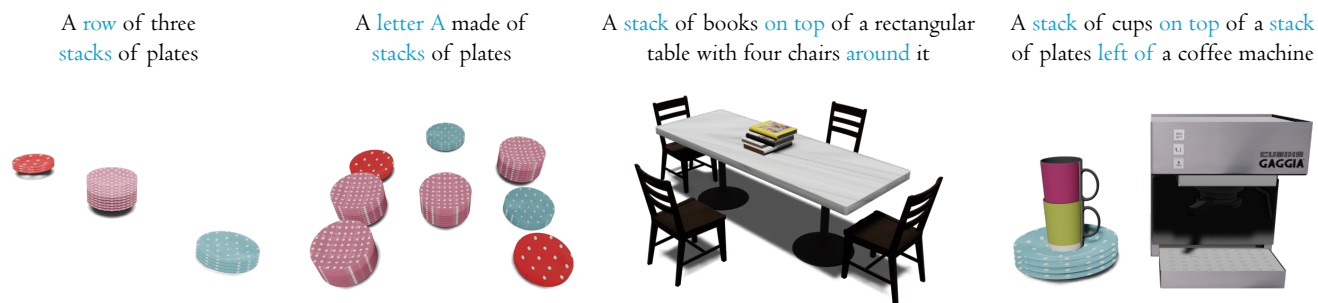


Figure 5. **Arrangements using compositional meta-programs.** SceneMotifCoder’s generated object arrangements can be used in generating new arrangements by treating them as retrieval assets. The generated arrangements exhibit compounded layers of complexity within a few iterations. Such compositions enable generation of more complex arrangements, opening the door to a wide range of possibilities.

direct-from-description

```
# Initialize a list to hold
↳ all created objects
objs = []

# Create four seat cushions
↳ with labels and half
↳ sizes, appending them to
↳ objs list
objs.append]
↳ (create("cushion1",
↳ [0.3, 0.1, 0.3]))
objs.append]
↳ (create("cushion2",
↳ [0.3, 0.1, 0.3]))
objs.append]
↳ (create("cushion3",
↳ [0.3, 0.1, 0.3]))
objs.append]
↳ (create("cushion4",
↳ [0.3, 0.1, 0.3]))

# Set initial position for
↳ the first cushion
move(objs[0], 0, 0.1, 0)

# Stack the second cushion
↳ on top of the first
move(objs[1], 0, 0.3, 0) #
↳ Additional 0.2 meters up
↳ from the first cushion's
↳ top surface

# Stack the third cushion on
↳ top of the second
move(objs[2], 0, 0.5, 0) #
↳ Additional 0.2 meters up
↳ from the second
↳ cushion's top surface

# Stack the fourth cushion
↳ on top of the third
move(objs[3], 0, 0.7, 0) #
↳ Additional 0.2 meters up
↳ from the third cushion's
↳ top surface
```

direct-from-motif-program

```
def create_pile(label_prefix,
↳ num_objects, half_size,
↳ base_position, y_gap,
↳ rotation_angle):
    """
    Creates a spatial motif of type
    ↳ 'pile' with the given properties.

    Args:
    - label_prefix (str): prefix for the
    ↳ label of each object in the pile.
    - num_objects (int): number of
    ↳ objects to create in the pile.
    - half_size (list[float]): list of
    ↳ three floats representing half
    ↳ the size (x, y, z) of each
    ↳ object.
    - base_position (tuple[float]):
    ↳ starting (x, y, z) position for
    ↳ the base of the pile.
    - y_gap (float): additional gap
    ↳ between each object in the pile
    ↳ on the y-axis.
    - rotation_angle (float): angle to
    ↳ rotate alternating objects, to
    ↳ add naturalism.

    Returns:
    - list: list containing each created
    ↳ and positioned object.
    """
    objs = []
    base_x, base_y, base_z =
    ↳ base_position

    for i in range(num_objects):
        # Calculate the y offset for each
        ↳ subsequent object to simulate
        ↳ the pile
        new_y = base_y + i *
        ↳ (half_size[1] * 2 + y_gap)

        # Create an object with a unique
        ↳ label based on index
        obj =
        ↳ create(f"{label_prefix}_{i+1}")
        ↳ half_size)

        # Move the object to its
        ↳ designated position
        move(obj, base_x, new_y, base_z)

        # Alternate rotation for a more
        ↳ natural-looking pile
        if i % 2 == 0:
            rotate(obj, 'z',
            ↳ rotation_angle)
        else:
            rotate(obj, 'z',
            ↳ -rotation_angle)

        # Append the positioned object to
        ↳ the list
        objs.append(obj)

    return objs
```

Meta-program from SceneMotifCoder

```
import random

def create_pile_motif(object_type, num_objects,
↳ base_position, size_variations,
↳ position_offsets, rotations=[], seed=None):
    """
    Creates a spatial motif of type 'pile' with
    ↳ configurable positions, sizes, and
    ↳ rotations.

    Args:
    - object_type (str or list[str]): The type
    ↳ label(s) for the objects in the pile.
    - num_objects (int): The number of objects
    ↳ to include in the pile.
    - base_position (list of float): The
    ↳ starting position [x, y, z] for the
    ↳ first object in the pile.
    - size_variations (list of list of float):
    ↳ The dimensions [width, height, depth]
    ↳ for each object.
    - position_offsets (list of list of float):
    ↳ Position offsets [x, y, z] relative to
    ↳ the base for each object.
    - rotations (list of tuples): Rotations to
    ↳ apply, each as (axis, angle, index).
    - seed (int, optional): Seed for the random
    ↳ number generator for reproducibility.

    Returns:
    - list: A list of objects representing the
    ↳ created pile.
    """
    if seed is not None:
        random.seed(seed)

    objs = []

    # Iterate over each object to be created
    for i in range(num_objects):
        # Determine the object type if a list
        ↳ is provided
        obj_type = object_type[i] if
        ↳ isinstance(object_type, list) else
        ↳ object_type

        # Calculate size and position for the
        ↳ current object
        obj_half_size = size_variations[i]
        pos_offset = position_offsets[i]
        position = [base_position[0] +
        ↳ pos_offset[0], base_position[1] +
        ↳ pos_offset[1], base_position[2] +
        ↳ pos_offset[2]]

        # Create the object
        obj = create(obj_type, obj_half_size)
        move(obj, *position)

        # Apply rotations if specified for this
        ↳ object
        for rotation in rotations:
            if rotation[2] == i: # Check if
            ↳ the rotation is meant for the
            ↳ current object
                rotate(obj, rotation[0],
                ↳ rotation[1])

        # Append the object to the list
        objs.append(obj)

    return objs
```

Table 1. Different programs for generating “a pile of four seat cushions”. The *direct-from-description* program is not reusable for generating new arrangements with different number of objects and types. The *direct-from-motif-program* program is reusable, but it makes a strong assumption on how to orient the objects (only alternating rotations around the z axis). In contrast, SceneMotifCoder’s program is less restrictive and takes into account for more potential object poses, which is especially important for the *pile* motif type.

direct-from-description

```
# List to keep track of all created
↳ objects
objs = []

# Create the table
table = create("table", [1.5, 0.1, 1])
objs.append(table)

# Move the table to the center of the
↳ scene (assume it is at coordinate
↳ (0, 0, 0))
move(table, 0, 0, 0)

# Create the chairs and place them
↳ around the table
# Chair size is arbitrarily chosen to
↳ be smaller than half the table size

# Create chairs
chairs = [create(f"chair_{i}", [0.3,
↳ 0.3, 0.3]) for i in range(8)]
objs.extend(chairs)

# Positions for chairs around the table
positions = [
    (1.8, 0.3, 0),    # right
    (-1.8, 0.3, 0),   # left
    (0, 0.3, 1.3),    # bottom
    (0, 0.3, -1.3),   # top
    (1.3, 0.3, 1.3),   # bottom-right
    (-1.3, 0.3, -1.3), # top-right
    (-1.3, 0.3, 1.3), # bottom-left
    (1.3, 0.3, -1.3)  # top-left
]

# Move and rotate chairs
for i, (x, y, z) in
↳ enumerate(positions):
    move(chairs[i], x, y, z)
    if i in [0, 1]: # right or left
        ↳ chair
        rotate(chairs[i], 'y', 90)
    elif i in [2, 3]: # bottom or top
        ↳ chair
        rotate(chairs[i], 'y', 0)
    elif i in [4, 7]: # corner chairs
        ↳ (right side)
        rotate(chairs[i], 'y', 45)
    elif i in [5, 6]: # corner chairs
        ↳ (left side)
        rotate(chairs[i], 'y', -45)
```

direct-from-motif-program

```
def create_rectangular_perimeter_motif(
↳ center_obj_label, center_obj_half_size,
↳ perimeter_obj_label, perimeter_obj_half_size,
↳ num_perimeter_objs, center_position):
    """
    Create a spatial motif where a central object
    ↳ is surrounded by a number of perimeter
    ↳ objects
    in a rectangular perimeter pattern.

    Args:
        center_obj_label (str): The label for the
        ↳ central object.
        center_obj_half_size (list[float]): The half
        ↳ size of the central object [hx, hy, hz].
        perimeter_obj_label (str): The label for the
        ↳ perimeter objects.
        perimeter_obj_half_size (list[float]): The
        ↳ half size of the perimeter objects [hx,
        ↳ hy, hz].
        num_perimeter_objs (int): The number of
        ↳ perimeter objects.
        center_position (list[float]): The position
        ↳ [x, y, z] of the central object.

    Returns:
        list: List of all created objects (central
        ↳ and perimeter).
    """
    objs = []

    # Create the central object
    center_obj = create(center_obj_label,
    ↳ center_obj_half_size)
    move(center_obj, center_position[0],
    ↳ center_position[1], center_position[2])
    objs.append(center_obj)

    # Calculate positions for perimeter objects
    center_x, center_y, center_z =
    ↳ center_position
    cx, cy, cz = center_obj_half_size
    px, py, pz = perimeter_obj_half_size
    perimeter_positions = []

    # Calculate the rectangular perimeter
    spacing_x = cx + px + 0.1 # Additional
    ↳ spacing
    spacing_z = cz + pz + 0.1 # Additional
    ↳ spacing

    for i in range(num_perimeter_objs):
        if i % 2 == 0: # Left and right sides
            sign = -1 if (i // 2) % 2 == 0 else 1
            x = center_x + sign * (spacing_x +
            ↳ px)
            z = center_z + (i // 4) * spacing_z
        else: # Front and back sides
            sign = -1 if ((i - 1) // 2) % 2 == 0
            ↳ else 1
            x = center_x + ((i - 1) // 4) *
            ↳ spacing_x
            z = center_z + sign * (spacing_z +
            ↳ pz)

        perimeter_positions.append((x, center_y,
        ↳ z))

    # Create and position the perimeter objects
    for pos in perimeter_positions:
        perimeter_obj =
        ↳ create(perimeter_obj_label,
        ↳ perimeter_obj_half_size)
        move(perimeter_obj, pos[0], pos[1],
        ↳ pos[2])
        objs.append(perimeter_obj)

    return objs
```

Meta-program from SceneMotifCoder

```
def create_rectangular_perimeter_motif(
    central_label, central_half_size, central_position,
    surrounding_label, surrounding_half_size,
    count_long_sides, count_short_sides,
    long_side_offset, short_side_offset,
    rotation_long_sides, rotation_short_sides,
    y_offset):
    """
    Create a spatial motif of type 'rectangular_perimeter' with the
    ↳ specified attributes.

    Args:
        - central_label (str): Label/type of the central object (e.g.,
        ↳ 'table').
        - central_half_size (list of float): Half-size dimensions of the
        ↳ central object [x, y, z].
        - central_position (tuple of float): Position of the central object
        ↳ (x, y, z).
        - surrounding_label (str): Label/type of the surrounding objects
        ↳ (e.g., 'chair').
        - surrounding_half_size (list of float): Half-size dimensions of the
        ↳ surrounding objects [x, y, z].
        - count_long_sides (int): Number of surrounding objects on the longer
        ↳ sides (x-axis) of the central object.
        - count_short_sides (int): Number of surrounding objects on the
        ↳ shorter ends (z-axis) of the central object.
        - long_side_offset (tuple of float): Positional offsets (x, z) for
        ↳ objects on the longer sides.
        - short_side_offset (tuple of float): Positional offsets (x, z) for
        ↳ objects on the shorter ends.
        - rotation_long_sides (float): Rotation angle for objects on the
        ↳ longer sides.
        - rotation_short_sides (float): Rotation angle for objects on the
        ↳ shorter ends.
        - y_offset (float): Vertical offset to place all surrounding objects
        ↳ at a specified height.

    Returns:
        - list: List of all created objects.

    Example:
        # Example call to recreate the example program 1
        objs = create_rectangular_perimeter_motif(
            'table', [1.00012, 0.38418, 0.49848], (0.0, 0.0, 0.0), 'chair',
            ↳ [0.28401, 0.48686, 0.32049],
            ↳ 2, 4, (1.01483, 0.01433), (0.41141, 0.62215), -90.0, 180.0,
            ↳ 0.06209
        )
        """
    objs = []

    # Create and place the central object
    central_obj = create(central_label, central_half_size)
    move(central_obj, *central_position)
    objs.append(central_obj)

    # Create and place objects on the longer sides
    for i in range(count_long_sides // 2):
        for sign in [-1, 1]: # Use -1 and 1 to cover both sides
            surrounding_obj = create(surrounding_label,
            ↳ surrounding_half_size)
            # Compute x and z positions based on offsets and symmetry
            x_pos = central_position[0] + sign * long_side_offset[0]
            z_pos = central_position[2] + (i - count_long_sides // 4) *
            ↳ long_side_offset[1]
            move(surrounding_obj, x_pos, y_offset, z_pos)
            rotate(surrounding_obj, 'y', sign * rotation_long_sides)
            objs.append(surrounding_obj)

    # Create and place objects on the shorter ends
    for i in range(count_short_sides // 2):
        for sign in [-1, 1]: # Use -1 and 1 to cover both sides
            surrounding_obj = create(surrounding_label,
            ↳ surrounding_half_size)
            # Compute x and z positions based on offsets and symmetry
            x_pos = central_position[0] + (i - count_short_sides // 4) *
            ↳ short_side_offset[0]
            z_pos = central_position[2] + sign * short_side_offset[1]
            move(surrounding_obj, x_pos, y_offset, z_pos)
            if sign == 1:
                rotate(surrounding_obj, 'y', rotation_short_sides)
            objs.append(surrounding_obj)

    return objs
```

Table 2. Different programs for generating “a rectangular table with eight chairs around it”. The *direct-from-description* program is not reusable for generating new arrangements with different number of objects and types. The *direct-from-motif-program* program is reusable, but it assumes equal spacing between objects (hard-coded as 0.1) and evenly-distributed objects around the center object, limiting what it can generate. In contrast, SceneMotifCoder’s program is less restrictive and allows for a wide range of customization.

E. LLM Prompts

We provide the LLM prompts used for various phases in our SceneMotifCoder framework. We provide general prompts in Appendix E.1, prompts for generating motif programs from naïve programs in Appendix E.2, prompts for generating meta-programs from sets of motif-programs in Appendix E.3, inference prompts in Appendix E.4, and prompts for our ablation study in Appendix E.5. For the motif-program and meta-program generation, we have prompts that ask the LLM to make high-level observations about the input, to generate the motif or meta-program, validate the generated program, and feedback prompts to ask the LLM to iterate if the validation fails. For the meta-program generation, we also have reasoning prompts that ask the LLM to reason about the arguments and structure of the function to be generated.

E.1. General Prompts

We have two general prompts:

- System introduction prompt specifying the DSL used for our visual programs.
- Classification prompt for identifying the scene motif type given a description. This `classify` prompt is used both during inference and as part of the motif-program generation to determine the motif type that the generated program should be stored under.

```
# Define prompts used when prompting LLM.
system: >-
You are a Python programmer.
You have access to all Python built-in functions and
→ libraries that are available in Python 3.10.
To use a function or a library, you need to import it
→ first.
However, you are not allowed to use any third-party
→ libraries except numpy, which is available as np.
Everything you write will be content inside a
→ function, unless otherwise specified.
Do not write the function signature or any other code
→ outside the function, unless you are explicitly
→ asked to do so.
If the prompt asks you to respond with code only, you
→ should not include any example usage nor text
→ description in your response.
Write comments to describe your thinking process.
Your responses should strictly follow the requirements
→ specified in the prompts.
You will work with a domain-specific language (DSL)
→ for describing spatial motif of objects.
You will be asked to perform some tasks related to
→ spatial motif programs written using this DSL.
All functions in the DSL are valid Python functions.
Here are the functions that exists:
1. create(label: str, half_size: list[float]) -> Obj:
→ Create a new object with the given `label` and
→ `half_size`. The `half_size` is a list of three
→ floats representing the half of the size of the
→ object in the x, y, and z dimensions.
2. move(obj: Obj, x: float, y: float, z: float) ->
→ None: Move the object `obj` to the world
→ coordinate (x, y, z). The object's position is the
→ centroid of the object's bounding box.
3. rotate(obj: Obj, axis: str, angle: float) -> None:
→ Rotate the object `obj` around its local axis (x,
→ y, or z) by the given angle.
Units are in meters for x, y, z dimensions and degrees
→ for angle.
```

```
The world is in a right-handed coordinate system, that
→ is, when looking from the front, the x-axis is to
→ the right, the y-axis is up, and the z-axis is
→ towards the viewer.
When looking from above, the origin (0, 0, 0) is at
→ the top-left corner of the screen, and the x-axis
→ increases to the right, the z-axis increases
→ towards the bottom of the screen.
```

```
# -----
classify: >-
Based on your observations, classify the spatial motif
→ of objects "<DESCRIPTION>" into one of the
→ following motif types:
1. stack - multiple objects of the same type are
→ placed orderly on top of each other
2. pile - objects are placed on top of each other but
→ not in an orderly manner (when in doubt between
→ stack and pile, choose pile if the description
→ contains the word "pile" explicitly)
3. row - objects are placed next to each other in a
→ row
4. grid - objects are placed orderly in a grid, like a
→ chessboard
5. left_of - one object placed to the left of another
→ object
6. in_front_of - one object placed in front of another
→ object
7. on_top - one object placed on top of another object
8. surround - objects are placed around a central
→ object in a circular manner
9. wall_vertical_column - objects are placed in a
→ column from top to bottom vertically on a wall
10. wall_horizontal_row - objects are placed in a row
→ from left to right vertically on a wall
11. wall_grid - objects are placed in a grid orderly
→ and vertically on a wall (when in doubt between
→ grid and wall_grid, choose wall_grid if the
→ objects are placed on a wall)
12. letter - objects are placed to form a letter of
→ the alphabet
13. rectangular_perimeter - objects are placed around
→ the perimeter of a rectangular shape facing
→ inward
If the motif type is letter, concatenate the letter to
→ the end of the motif type (e.g., letter_A,
→ letter_B, etc.)
Respond with the name of the motif type only.
```

E.2. Prompts to Generate Motif Program from Naïve Program

When generating the motif program, our observational prompts instruct the LLM to make high-level observations about the following:

- the number and type of objects in the arrangement.
- the relative displacements between objects.
- spatial patterns such as symmetries.
- spatial patterns along the three coordinate axes.

```
# Observation
optimize_highlevel_count: >-
Below is a program about a spatial motif of
→ "<DESCRIPTION>".
Describe how many object types and how many are there
→ for each type.
Respond with a json-like text structure with the
→ object types as keys and the counts as values.
Here is the program:
```python
<PROGRAM>
```
optimize_highlevel_general_pattern: >-
Can you observe any pattern in this motif of objects
→ about "<DESCRIPTION>"? (Look for common sense
→ things like repeating layout, symmetry, etc.)
```

```

Respond with a detailed description of the pattern you
↳ observed in text.
optimize_highlevel_xyz_pattern: >-
Can you observe any pattern in the x, y, z coordinates
↳ of the objects in this motif?
Respond with a detailed description of the pattern you
↳ observed in text.
optimize_highlevel_xyz_displacements: >-
Can you find the relative displacements of each object
↳ to the next object in the x, y, z dimensions?
Base on these displacement values, group the objects
↳ into different subgroups.
The goal is to disentangle the overall motif into
↳ smaller, self-contained spatial patterns.
Assign each object to a subgroup such that the spatial
↳ pattern in the subgroup can easily be expressed by
↳ some programming constructs.
Each object should belong to exactly one subgroup.
# -----
# Generation
optimize_lowlevel: >-
Take a deep breath and follow the steps below
↳ carefully.
Using these observations you made, modify the original
↳ program such that the code structure represents
↳ the spatial pattern.
Here are the requirements for this task:
1. Do not create any functions
2. Store all created objects in a list named `objs`
3. The new program should represent the same scene as
↳ in the original program with the same number of
↳ objects.
4. The objects do not need to be in the same order as
↳ the original program, instead, opt for a more
↳ structured or natural order.
5. Do not list all object coordinates, rotation
↳ angles, or sizes in a list and access them in a
↳ loop using index.
This is not acceptable as the spatial pattern is
↳ lost.
Avoid doing this at all costs.
Use loops, arithmetic operators, if-else
↳ statements, and other programming constructs to
↳ encapsulate the patterns.
6. The coordinates and sizes can be slightly different
↳ from the original program, as long as the overall
↳ motif remains the same.
Please respond with code only.
# -----
# Validation
validate_naive_listing: >-
Below is a program that describes a spatial motif of
↳ objects.
Please analyze the program for the following
↳ criterion:
The program should define and manipulate the objects
↳ and their properties without using lists of
↳ specific object attributes for individual objects.
It is okay to store common values in variables and
↳ reuse them (e.g., dimensions, common properties).
However, the program should use programming constructs
↳ and arithmetic operations (like loops or
↳ calculations) to define and manipulate the
↳ positional and orientational properties of
↳ objects, rather than explicitly listing them.
For example, if there are four objects, the program
↳ should not have a list of four (x, y, z)
↳ coordinates or four rotation angles for each
↳ object.
Exclude the `objs` list from the analysis.
Answer in a json format:
{'valid': 'yes' or 'no', 'variable_names': a list of
↳ the names of the variables that violated the
↳ criterion}
Briefly explain your reasoning in two to three
↳ sentences after the json-like text structure.
Here is the program:
```python
<PROGRAM>

```

```

...

Feedback
optimize_lowlevel_feedback_syntax: >-
I could not run the program you provided.
<FEEDBACK>
Please try again.
optimize_lowlevel_feedback_naive_listing: >-
Your program creates an motif that accurately
↳ recreates the one in the original program.
However, you listed some attributes of all objects in
↳ a list and accesses them in a loop.
<FEEDBACK>
Please remove such statements and use loops,
↳ arithmetic operators, if-else statements to
↳ capture the object attribute patterns
optimize_lowlevel_feedback_num_objs: >-
The number of objects in the original program does not
↳ match the number of objects in your program.
<FEEDBACK>
Please try again.
optimize_lowlevel_feedback_centroids: >-
Objects in the original program are not accurately
↳ placed in your program.
<FEEDBACK>
Please try again.
optimize_lowlevel_feedback_bounding_boxes: >-
Objects in the original program are not accurately
↳ created in your program.
<FEEDBACK>
The objects in your program either have incorrect
↳ dimensions or are not oriented correctly.
Please try again.

```

### E.3. Prompts to Generate Meta-program from Motif Program

When generating the meta-program, our observational prompts instruct the LLM to make high-level observations about the following:

- commonalities between the motif-programs.
- differences between the motif-programs.

To ensure the generated meta-program is robust and re-useable, we use reasoning prompts to instruct the LLM to consider:

- reasons why the motif-programs all belong to the same motif
- what arguments should be extracted
- what structure should the meta-program function have

```

Observation
generalize_high_level_commonalities: >-
Take a deep breath and follow the steps below
↳ carefully.
Here are "<NUM_PROGRAMS>" programs that describe the
↳ same type of spatial motif: "<MOTIF_TYPE>".
Each program may have different objects and different
↳ coordinates, but they all fall under the same type
↳ of spatial motif.
Here are the programs and the spatial motifs they
↳ create:
<ALL_PROGRAMS>
For each program, analyze and observe its structure
↳ and how the structure coorelates with the spatial
↳ motif it creates.
Pay extra attention to how the objects are placed
↳ relative to each other.
Can you identify the commonalities among these
↳ programs?
If there is only one program, hypothesize what the
↳ commonalities could be if there were multiple
↳ programs of the same type.

```



For the alphabet\_letter motif type, only consider the  
 ↳ commonalities among the programs that form the  
 ↳ same letter.  
 Respond with a detailed description of the pattern you  
 ↳ observed in text.

**generalize\_high\_level\_differences:** >-

What are the differences among these programs?  
 Specifically, how are the objects placed differently  
 ↳ in each program?  
 For each program, observe and describe the differences  
 ↳ in the program structure that lead to the  
 ↳ different spatial motifs.  
 If there is only one program, hypothesize how the  
 ↳ program could be written differently to create a  
 ↳ different spatial motif of the same type.  
 Your hypothesized programs should not deviate from the  
 ↳ concrete, non-hypothetical programs you analyzed,  
 ↳ and should focus on the number of objects, their  
 ↳ positions, rotations, and sizes as the main axes  
 ↳ of variation.  
 The hypothesized programs should be close to the  
 ↳ original program in terms of the spatial motif  
 ↳ they create (e.g., a stack of 3 objects instead of  
 ↳ 4, or a row of 4 objects instead of 3, etc.)  
 Respond with a detailed description of the differences  
 ↳ of each program in text.  
 Refer to the programs by their order in the previous  
 ↳ prompt.

# -----

# Reasoning

**generalize\_high\_level\_motif\_reason:** >-

Despite these differences, these programs all fall  
 ↳ under the same type of spatial motif:  
 ↳ "<MOTIF\_TYPE>".  
 Here is a list of all available motif types:  
 1. stack - multiple objects of the same type are  
 ↳ placed orderly on top of each other  
 2. pile - objects are placed on top of each other but  
 ↳ not in an orderly manner (when in doubt between  
 ↳ stack and pile, choose pile if the description  
 ↳ contains the word "pile" explicitly)  
 3. row - objects are placed next to each other in a  
 ↳ row  
 4. grid - objects are placed orderly in a grid, like a  
 ↳ chessboard  
 5. left\_of - one object placed to the left of another  
 ↳ object  
 6. in\_front\_of - one object placed in front of another  
 ↳ object  
 7. on\_top - one object placed on top of another object  
 8. surround - objects are placed around a central  
 ↳ object in a circular manner  
 9. wall\_vertical\_column - objects are placed in a  
 ↳ column from top to bottom vertically on a wall  
 10. wall\_horizontal\_row - objects are placed in a row  
 ↳ from left to right vertically on a wall  
 11. wall\_grid - objects are placed in a grid orderly  
 ↳ and vertically on a wall (when in doubt between  
 ↳ grid and wall\_grid, choose wall\_grid if the  
 ↳ objects are placed on a wall)  
 12. letter - objects are placed to form a letter of  
 ↳ the alphabet  
 13. rectangular\_perimeter - objects are placed around  
 ↳ the perimeter of a rectangular shape facing  
 ↳ inward  
 Based on your observations, what are the reasons that  
 ↳ these programs fall under the same type of spatial  
 ↳ motif?  
 Also, identify the closest incorrect motif type and  
 ↳ explain why it is not the correct motif type.  
 Respond with a detailed description of the reasons in  
 ↳ text.

**generalize\_low\_level\_arguments:** >-

With this conversation in mind, you are now tasked to  
 ↳ write a python function (a meta-program) that can  
 ↳ be used to create various spatial motifs of  
 ↳ "<MOTIF\_TYPE>".  
 A user should be able to call this meta-program with  
 ↳ different arguments to create different spatial  
 ↳ motifs of this type.

Using the provided programs as examples, the  
 ↳ meta-program should be able to create any spatial  
 ↳ motif of this type with slight variations in the  
 ↳ number of objects and their attributes.  
 Apart from the objects in the examples, the  
 ↳ meta-program should be able to create new spatial  
 ↳ motifs of different object types and attributes.  
 Use your analysis on the commonalities and differences  
 ↳ among the example programs to guide you in writing  
 ↳ the meta-program.  
 Keep in mind that the final meta-program should be  
 ↳ able to recreate the given example spatial motifs  
 ↳ as closely as possible.  
 Avoid taking plain lists of object attributes, such as  
 ↳ coordinates, sizes, and rotations, as arguments,  
 ↳ as this defeats the purpose of using a  
 ↳ meta-program to capture fundamental spatial  
 ↳ patterns.  
 I will guide you through the process of writing the  
 ↳ function step by step.  
 First, what are the arguments that the meta-program  
 ↳ should take to create a spatial motif of this  
 ↳ type?  
 Respond with a detailed description of the arguments  
 ↳ in text.

**generalize\_low\_level\_structure:** >-

You have identified the arguments needed for the  
 ↳ meta-program.  
 Now, think about the structure of the meta-program.  
 How should the meta-program be structured to  
 ↳ generalize the spatial motif of "<MOTIF\_TYPE>"?  
 Aim to encapsulate the commonalities and differences  
 ↳ among the example programs in the structure of the  
 ↳ meta-program.  
 Think about how the arguments should be used in the  
 ↳ meta-program to create the spatial motif.  
 What programming constructs should be used to  
 ↳ encapsulate the commonalities and differences  
 ↳ among the example programs?  
 Respond with a detailed description of the structure  
 ↳ of the meta-program in text.

# -----

# Generation

**generalize\_low\_level:** >-

Take a deep breath and follow the steps below  
 ↳ carefully.  
 You have identified the arguments and the structure of  
 ↳ the meta-program.  
 Now, write the meta-program that can create various  
 ↳ spatial motifs of "<MOTIF\_TYPE>".  
 Here is a meta-program of the same motif type you  
 ↳ created in the past:  
 ```python  
 <PAST_META_PROGRAM>
 ```  
 You can use this as a reference to write the new  
 ↳ meta-program, but you can also write it from  
 ↳ scratch.  
 Refer to the arguments you identified and the  
 ↳ structure you described to write the meta-program.  
 Remember that the meta-program should be able to  
 ↳ recreate the spatial motifs of the example  
 ↳ programs as closely as possible.  
 Avoid taking plain lists of object attributes, such as  
 ↳ coordinates, sizes, and rotations, as arguments,  
 ↳ as this defeats the purpose of using a  
 ↳ meta-program to capture fundamental spatial  
 ↳ patterns.  
 Include a docstring at the beginning of the  
 ↳ meta-program to describe the purpose of the  
 ↳ function and the arguments it takes.  
 Use comments to explain your code step by step.  
 Please respond with code only.

# -----

# Validation

**generalize\_low\_level\_batch\_recreate:** >-

Using the meta-program you wrote, what are the  
 ↳ function calls needed to recreate each of the  
 ↳ example programs you analyzed previously?

```

Write one function call for each example program, that
→ recreates the spatial motif of the program as
→ closely as possible.
Ignore the programs you hypothesized and only write
→ function calls for the example programs I
→ provided.
Response with a json-like text structure with the
→ example program order integers as keys and the
→ function calls as values.
For example, {"1": "function_call(program1_args)",
→ "2": "function_call(program2_args)", ...}

Feedback
generalize_low_level_feedback: >-
The meta-program you wrote could not recreate the
→ spatial motif of the example programs as closely
→ as possible.
Here are the feedback(s) for the example program(s)
→ that were not recreated accurately:
<FEEDBACK>
The issue(s) may be related to the meta-program
→ structure or the arguments used in the function
→ calls.
Please analyze the feedback and modify the
→ meta-program if necessary.
Remember that the meta-program should be able to
→ recreate the spatial motifs of the example
→ programs as closely as possible.
Also, remember to edit the docstring and comments in
→ the meta-program to reflect the changes you made.
Then, respond with the code of the modified
→ meta-program only.
If you think the meta-program is correct, and the
→ function calls are incorrect, then you can respond
→ with the same meta-program code.
I will ask you to provide the function calls again
→ after this where you can correct the function
→ calls.
generalize_refine_comments: >-
You have successfully written the meta-program to
→ generalize the spatial motif of "<MOTIF_TYPE>".
The latest function calls you provided are able to
→ recreate the spatial motifs of the example
→ programs.
Please update the docstring and comments in the
→ meta-program to provide as much information as
→ possible about the expected arguments.
If the program contains well-defined parts that can be
→ explained, include comments to describe these
→ parts - how they contribute to the spatial motif
→ and how they differ from other parts.
At inference time, the meta-program will be the sole
→ source of information for recreating this spatial
→ motif.
Therefore, these comments will be crucial for
→ understanding how the meta-program should be used.
Include at least one example function call in the
→ comments to show how the meta-program should be
→ called.
Please respond with code only.

```

## E.4. Inference Prompts

```

inference: >-
Here is a meta-program that generalizes a spatial
→ arrangement of type "<MOTIF_TYPE>":
...python
<META_PROGRAM>
...
And here is a description of a spatial motif of the
→ same type:
<DESCRIPTION>
Your task is to call the meta-program with the
→ necessary arguments to recreate the spatial motif
→ described in the description as closely as
→ possible.
Read the docstring and comments in the meta-program to
→ understand how to use it.

```

```

Refer to the example function call in the meta-program
→ documentation to understand how the meta-program
→ should be called, if available.
Use common sense to infer the arguments for ambiguous
→ arguments, such as object dimensions, positions,
→ and rotations.
When in doubt, refer back to the example function call
→ in the meta-program documentation.
Ensure the arguments make the new spatial motif
→ physically possible without intersecting or
→ floating objects and make the objects contact
→ tightly, if applicable.
I will run a postprocessing step to refine the spatial
→ motif after you provide the function call to me.
Remember that x, y, z dimensions are in meters and
→ rotation angles are in degrees.
+ x is to the right, + y is up, and + z is front (towards
→ the viewer).
Most object rotations hence are around the y-axis, as
→ it is the vertical axis.
Assume the objects are large so make sure they are
→ spaced out enough to not intersect.
When placing objects, consider how human would
→ organize them in real life.
For example, large objects are usually placed at the
→ bottom and smaller objects on top.
Please respond with code - the function call with the
→ inferred arguments only.
inference_feedback: >-
I could not run the meta-program using the function
→ call you provided.
<FEEDBACK>
Please try again.

retrieval_mesh_rotations: >-
I am retrieving meshes of objects from a
→ human-authored object dataset to instantiate a
→ spatial motif of "<DESCRIPTION>".
The labels of the objects are "<OBJECT_LABELS>".
How likely would be the retrieved mesh in the correct
→ orientation?
Use common sense knowledge to reason about this.
Consider whether the objects are commonly oriented in
→ both upright and sideways positions in real life.
If there are multiple common orientations, then the
→ retrieved mesh is likely to be in the incorrect
→ orientation.
Respond in a json-like text structure with the object
→ labels as keys and two probabilities ("correct",
→ "incorrect") as values 1
...json
{"obj1": {"correct": X, "incorrect": 1-X}, "obj2":
→ ...}
...
where X is between 0 and 1.
The two probabilities "correct", "incorrect" indicate
→ how likely the retrieved mesh would be in the
→ correct orientation for the motif.
For objects that are hinged on a wall, always consider
→ the mesh would be in the incorrect orientation.
Include a brief two to three sentences explanation for
→ each object after the json-like text structure.
spatial_optimization_touch: >-
I have instantiated a spatial motif of "<DESCRIPTION>"
→ using meshes retrieved from a human-authored
→ object dataset.
How common is it for the individual objects to be
→ placed in tight contact with each other in this
→ motif?
Consider whether this motif describes a decorative
→ arrangement or a functional arrangement.
A decorative arrangement is one that is intentionally
→ designed to be arranged in a specific way for
→ aesthetic purposes.
If the motif is a decorative arrangement, then
→ touching is less common.
Otherwise, use common sense to reason about whether
→ the objects are typically placed in tight contact
→ with each other in real life.

```

```

Answer using percentage to indicate how common each
→ version is.
Answer in a json-like text structure with two
→ probabilities like below:
```json
{"touch": X, "no_touch": 1-X}
```
where X, between 0 and 1, is the probability of the
→ version fitting the common sense definition.
Briefly explain your reasoning in two to three
→ sentences after the json-like text structure.

wnsynsetkeys: >-
Here is a list of wordnet synset keys.
<WNSYNSETKEYS>
What is the best key for the following object labels?
<OBJECT_LABELS>
Response with a json-like list of synset keys in the
→ same order as the object labels like below:
```json
{"wnsynsetkeys": ["key1", "key2", "key3"]}
```
Use "none" if none of the keys are suitable for an
→ object.

invalid_response: >-
<FEEDBACK>
Please try again.

```

## E.5. Ablation Prompts

Simple prompts for generating a meta-program directly from a description of a spatial motif (without example arrangement), directly from a set of motif programs (without any reasoning), and through in-context learning.

```

ablation_direct: >-
Using the DSL, write a program about a spatial motif
→ of "<DESCRIPTION>".
Please respond with code only.
Do not define any function.
Your code should be a list of statements that can be
→ executed sequentially and directly without any
→ modifications or extra function calls.
Please respond with code only.
ablation_generalize: >-
Here are different programs that describe the same
→ type of spatial motif: "<MOTIF_TYPE>".
Each program may have different objects and different
→ coordinates, but they all fall under the same type
→ of spatial motif.
Here are the programs and the spatial motifs they
→ create:
<ALL_PROGRAMS>
Now, write a meta-program that can generalize the
→ spatial motif of this type using the provided
→ programs as examples.
The meta-program should be able to create any spatial
→ motif of this type with slight variations in the
→ number of objects and their attributes.
Identify the arguments that the meta-program should
→ take and the structure of the meta-program to
→ generalize the spatial motif.
Put the arguments in the function signature and write
→ the meta-program as a callable function.
The function should return the list of objects created
→ in the spatial motif.
Here is a meta-program of the same motif type you
→ created in the past:
```python
<PAST_META_PROGRAM>
```
You can use this as a reference to write the new
→ meta-program, but you can also write it from
→ scratch.
Include a docstring at the beginning of the
→ meta-program to describe the purpose of the
→ function and the arguments it takes.

```

```

Use comments to explain your code step by step.
Please respond with code only.
ablation_incontext: >-
Using the DSL, write a program about a spatial motif
→ of "<DESCRIPTION>".
Here are programs that describe the same type of
→ spatial motif: "<MOTIF_TYPE>".
Use these programs as examples to guide you in writing
→ the program:
<ALL_PROGRAMS>
Please respond with code only.

```

## F. Example Session

We show an example session generating meta-program given arrangement corresponding to “a stack of seven plates”. We provide both the prompts to the LLM and the responses from the LLM for different phases of learning to generate the meta-program. Note that the session is fully automatic and does not require any user intervention.

### F.1. Naïve Program Extraction

Initial naïve program extracted directly from the example arrangement (does not require a LLM).

```

Description: a stack of seven plates
Naive program extracted from input arrangement
objs = []
obj_1_half_size = [0.08909, 0.0143, 0.08853]
obj_1_centroid = [0.0, 0.0, 0.0]
obj_1 = create('plate', obj_1_half_size)
move(obj_1, obj_1_centroid[0], obj_1_centroid[1],
→ obj_1_centroid[2])
objs.append(obj_1)
obj_2_half_size = [0.08909, 0.0143, 0.08853]
obj_2_centroid = [0.0, -0.00757, 0.0]
obj_2 = create('plate', obj_2_half_size)
move(obj_2, obj_2_centroid[0], obj_2_centroid[1],
→ obj_2_centroid[2])
objs.append(obj_2)
obj_3_half_size = [0.08909, 0.0143, 0.08853]
obj_3_centroid = [0.0, -0.01514, 0.0]
obj_3 = create('plate', obj_3_half_size)
move(obj_3, obj_3_centroid[0], obj_3_centroid[1],
→ obj_3_centroid[2])
objs.append(obj_3)
obj_4_half_size = [0.08909, 0.0143, 0.08853]
obj_4_centroid = [0.0, -0.02272, -0.0]
obj_4 = create('plate', obj_4_half_size)
move(obj_4, obj_4_centroid[0], obj_4_centroid[1],
→ obj_4_centroid[2])
objs.append(obj_4)
obj_5_half_size = [0.08909, 0.0143, 0.08853]
obj_5_centroid = [0.0, -0.03029, -0.0]
obj_5 = create('plate', obj_5_half_size)
move(obj_5, obj_5_centroid[0], obj_5_centroid[1],
→ obj_5_centroid[2])
objs.append(obj_5)
obj_6_half_size = [0.08909, 0.0143, 0.08853]
obj_6 = create('plate', obj_6_half_size)
move(obj_6, obj_6_centroid[0], obj_6_centroid[1],
→ obj_6_centroid[2])
objs.append(obj_6)
obj_7_half_size = [0.08909, 0.0143, 0.08853]
obj_7_centroid = [0.0, -0.04544, 0.0]
obj_7 = create('plate', obj_7_half_size)
move(obj_7, obj_7_centroid[0], obj_7_centroid[1],
→ obj_7_centroid[2])
objs.append(obj_7)

```

### F.2. Motif Program Generation

High level observations of naïve program.

## Asking LLM to make high level observations of the naïve program using prompt `optimize_highlevel_count`

Below is a program about a spatial motif of  
↪ "`<DESCRIPTION>`".  
Describe how many object types and how many are there  
↪ for each type.  
Respond with a json-like text structure with the object  
↪ types as keys and the counts as values.  
Here is the program:  
```python  
<PROGRAM>
```

### Response from LLM (json)

```
{
 "plate": 7
}
```

## Asking LLM to make high level observations of the naïve program using prompt `optimize_highlevel_general_pattern`

Can you observe any pattern in this motif of objects  
↪ about "`<DESCRIPTION>`"? (Look for common sense things  
↪ like repeating layout, symmetry, etc.)  
Respond with a detailed description of the pattern you  
↪ observed in text.

### Response from LLM (text)

The pattern observed in the "stack of seven plates" can  
↪ be described as follows:

1. **Object Consistency**: All seven objects in the  
↪ stack are of the same type, i.e., "plate." They all  
↪ share identical dimensions, with a half-size of  
↪ [0.08909, 0.0143, 0.08853].
2. **Vertical Stacking**: The plates are vertically  
↪ stacked along the y-axis. This observation can be  
↪ made from the changes in the 'y' component of the  
↪ centroids of each plate.
3. **Even Spacing**: The centroids of the plates are  
↪ evenly spaced along the negative y-axis. Each  
↪ subsequent plate is positioned 0.00757 meters below  
↪ the previous one. This incremental change in the  
↪ y-coordinate confirms a precise stacking  
↪ arrangement.
4. **Fixed x and z coordinates**: The 'x' and 'z'  
↪ coordinates of the centroids of all the plates  
↪ remain constant at 0.0. This indicates the stack is  
↪ perfectly aligned along these axes, ensuring that  
↪ the stack is upright and not tilted or offset in any  
↪ horizontal direction.

Given these observations, the motif can be described as  
↪ a neat, symmetrical vertical stack of seven uniform  
↪ plates, each spaced evenly from one another.

## Asking LLM to make high level observations of the naïve program using prompt `optimize_highlevel_xyz_pattern`

Can you observe any pattern in the x, y, z coordinates  
↪ of the objects in this motif?  
Respond with a detailed description of the pattern you  
↪ observed in text.

### Response from LLM (text)

The x, y, and z coordinates of the objects in this motif  
↪ exhibit the following patterns:

```
x-coordinate:
```

```
- The x-coordinate for the centroids of all plates is
↪ 0.0.
- This consistent value indicates that there is no
↪ horizontal shifting in the x-direction, meaning all
↪ plates are aligned vertically without any horizontal
↪ displacement.
```

```
y-coordinate:
- The y-coordinates show a clear pattern with a
↪ consistent decrease of 0.00757 meters for each
↪ subsequent plate.
- The initial y-coordinate is 0.0 for the first plate,
↪ and the y-coordinates for the following plates are:
 - 0.0 for the 1st plate
 - -0.00757 for the 2nd plate
 - -0.01514 for the 3rd plate
 - -0.02272 for the 4th plate
 - -0.03029 for the 5th plate
 - -0.03786 for the 6th plate
 - -0.04544 for the 7th plate
- This indicates that the plates are evenly spaced along
↪ the vertical axis (negative y-direction) by a
↪ distance of 0.00757 meters.
```

```
z-coordinate:
- The z-coordinate for the centroids of all plates is
↪ consistently 0.0.
- This indicates there is no depth displacement along
↪ the z-axis, meaning the stack is aligned in the
↪ depth direction as well, with no forward or backward
↪ shifting.
```

```
Pattern Summary:
- Vertical Alignment: All plates are perfectly
↪ aligned in the vertical axis (y-axis) with an even
↪ spacing.
- No Horizontal Displacement: There is no change in
↪ the x-coordinate, ensuring horizontal alignment.
- No Depth Displacement: There is no change in the
↪ z-coordinate, ensuring depth alignment.
```

In conclusion, the coordinates exhibit a clear pattern  
↪ of a vertically stacked arrangement with precise  
↪ spacing and no horizontal or depth displacement,  
↪ ensuring a neat and uniform stack of plates.

## Asking LLM to make high level observations of the naïve program using prompt `optimize_highlevel_xyz_displacements`

Can you find the relative displacements of each object  
↪ to the next object in the x, y, z dimensions?  
Base on these displacement values, group the objects  
↪ into different subgroups.  
The goal is to disentangle the overall motif into  
↪ smaller, self-contained spatial patterns.  
Assign each object to a subgroup such that the spatial  
↪ pattern in the subgroup can easily be expressed by  
↪ some programming constructs.  
Each object should belong to exactly one subgroup.

### Response from LLM (text)

```
Relative Displacements:
The relative displacements of each object to the next
↪ object in the stack are as follows:
```

```
- Displacements:
 - Object 1 to Object 2: (0.0, -0.00757, 0.0)
 - Object 2 to Object 3: (0.0, -0.00757, 0.0)
 - Object 3 to Object 4: (0.0, -0.00757, 0.0)
 - Object 4 to Object 5: (0.0, -0.00757, 0.0)
 - Object 5 to Object 6: (0.0, -0.00757, 0.0)
 - Object 6 to Object 7: (0.0, -0.00757, 0.0)
```

Each of these displacements is identical, indicating a  
↪ uniform vertical displacement between consecutive  
↪ objects.

```

Grouping into Subgroups:
Given the uniform relative displacement pattern, the
↳ entire set of objects can be grouped into one single
↳ subgroup. This is due to the consistent and
↳ repetitive nature of the displacement, which can be
↳ expressed succinctly using a loop construct in a
↳ programming context.

Subgroup Assignment:
- **Subgroup 1**: All objects (Object 1 to Object 7)

Expressed Subgroup Pattern:
- The spatial pattern in this subgroup is a uniform
↳ vertical displacement of 0.00757 meters along the
↳ negative y-axis between consecutive objects.

Programming Context:
- This can be expressed using a loop to create and move
↳ each object in the subgroup with the uniform
↳ displacement.

```

## Determining motif type for storing into program library.

Asking LLM to classify motif type for the description using prompt classify

Based on your observations, classify the spatial motif  
↳ of objects "<DESCRIPTION>" into one of the following  
↳ motif types:

1. stack - multiple objects of the same type are placed  
↳ orderly on top of each other
2. pile - objects are placed on top of each other but  
↳ not in an orderly manner (when in doubt between  
↳ stack and pile, choose pile if the description  
↳ contains the word "pile" explicitly)
3. row - objects are placed next to each other in a row
4. grid - objects are placed orderly in a grid, like a  
↳ chessboard
5. left\_of - one object placed to the left of another  
↳ object
6. in\_front\_of - one object placed in front of another  
↳ object
7. on\_top - one object placed on top of another object
8. surround - objects are placed around a central object  
↳ in a circular manner
9. wall\_vertical\_column - objects are placed in a column  
↳ from top to bottom vertically on a wall
10. wall\_horizontal\_row - objects are placed in a row  
↳ from left to right vertically on a wall
11. wall\_grid - objects are placed in a grid orderly and  
↳ vertically on a wall (when in doubt between grid and  
↳ wall\_grid, choose wall\_grid if the objects are  
↳ placed on a wall)
12. letter - objects are placed to form a letter of the  
↳ alphabet
13. rectangular\_perimeter - objects are placed around  
↳ the perimeter of a rectangular shape facing inward

If the motif type is letter, concatenate the letter to  
↳ the end of the motif type (e.g., letter\_A, letter\_B,  
↳ etc.)

Respond with the name of the motif type only.

## Response from LLM (text)

stack

## Motif program generation.

Asking LLM to transform the naïve program into a motif program based on the observations made by the LLM using the prompt optimize\_lowlevel

Take a deep breath and follow the steps below carefully.  
Using these observations you made, modify the original  
↳ program such that the code structure represents the  
↳ spatial pattern.  
Here are the requirements for this task:

1. Do not create any functions
  2. Store all created objects in a list named `objs`
  3. The new program should represent the same scene as in  
↳ the original program with the same number of  
↳ objects.
  4. The objects do not need to be in the same order as  
↳ the original program, instead, opt for a more  
↳ structured or natural order.
  5. Do not list all object coordinates, rotation angles,  
↳ or sizes in a list and access them in a loop using  
↳ index.  
This is not acceptable as the spatial pattern is lost.  
Avoid doing this at all costs.  
Use loops, arithmetic operators, if-else statements,  
↳ and other programming constructs to encapsulate the  
↳ patterns.
  6. The coordinates and sizes can be slightly different  
↳ from the original program, as long as the overall  
↳ motif remains the same.
- Please respond with code only.

## Response from LLM (python)

```

objs = []
half_size = [0.08909, 0.0143, 0.08853]
initial_y = 0.0
displacement_y = -0.00757

for i in range(7):
 y_position = initial_y + i * displacement_y
 obj = create('plate', half_size)
 move(obj, 0.0, y_position, 0.0)
 objs.append(obj)

```

## Motif program validation.

Asking LLM to validate the motif program using the prompt validate\_naive\_listing.

Below is a program that describes a spatial motif of  
↳ objects.  
Please analyze the program for the following criterion:  
The program should define and manipulate the objects and  
↳ their properties without using lists of specific  
↳ object attributes for individual objects.  
It is okay to store common values in variables and reuse  
↳ them (e.g., dimensions, common properties).  
However, the program should use programming constructs  
↳ and arithmetic operations (like loops or  
↳ calculations) to define and manipulate the  
↳ positional and orientational properties of objects,  
↳ rather than explicitly listing them.  
For example, if there are four objects, the program  
↳ should not have a list of four (x, y, z) coordinates  
↳ or four rotation angles for each object.  
Exclude the `objs` list from the analysis.  
Answer in a json format:  
{'valid': 'yes' or 'no', 'variable\_names': a list of the  
↳ names of the variables that violated the criterion}  
Briefly explain your reasoning in two to three sentences  
↳ after the json-like text structure.  
Here is the program:  
```python  
<PROGRAM>
```

## Response from LLM (json)

```

{
 "valid": "yes",
 "variable_names": []
}

```

All validation checks were passed after iteration 1. The program provided by the LLM is the stored in the program library with the classified motif type of stack.



If the validation did not pass, then the `optimize_lowlevel_feedback` prompts are used to refine the program based on the validation test that didn't pass.

### F.3. Meta-program Generation

Asking LLM to make observations of *commonalities* in a set of motif program with motif type `stack` using the prompt `generalize_high_level_commonalities`

```
Take a deep breath and follow the steps below carefully.
Here are "<NUM_PROGRAMS>" programs that describe the
↳ same type of spatial motif: "<MOTIF_TYPE>".
Each program may have different objects and different
↳ coordinates, but they all fall under the same type
↳ of spatial motif.
Here are the programs and the spatial motifs they
↳ create:
<ALL_PROGRAMS>
For each program, analyze and observe its structure and
↳ how the structure coorelates with the spatial motif
↳ it creates.
Pay extra attention to how the objects are placed
↳ relative to each other.
Can you identify the commonalities among these programs?
If there is only one program, hypothesize what the
↳ commonalities could be if there were multiple
↳ programs of the same type.
For the alphabet_letter motif type, only consider the
↳ commonalities among the programs that form the same
↳ letter.
Respond with a detailed description of the pattern you
↳ observed in text.
```

#### Response from LLM (text)

```
Based on the analysis of Program 1, which describes a
↳ "stack" spatial motif, we can identify several
↳ commonalities that characterize this type of spatial
↳ arrangement. Here are the observed commonalities and
↳ patterns:
```

```
Commonalities of the Stack Motif:
```

- \*\*Object Type Consistency\*\*:**
  - All objects in the motif are of the same type. In  
↳ this case, all objects are identified as 'plate'.
  - ↳ This consistency ensures uniformity in the  
↳ appearance and arrangement of the stack.
- \*\*Uniform Object Dimensions\*\*:**
  - The objects in the stack all have the same  
↳ dimensions, represented by the 'half\_size'  
↳ parameter. This uniformity in size contributes to  
↳ the regular and orderly appearance of the stack.
- \*\*Vertical Alignment\*\*:**
  - The objects are aligned vertically along the  
↳ y-axis. This is characteristic of a stack, where  
↳ objects are placed directly on top of one another  
↳ in a straight line.
- \*\*Even Spacing\*\*:**
  - There is a consistent vertical displacement between  
↳ consecutive objects in the stack. In Program 1,  
↳ the displacement is uniform with a negative  
↳ y-coordinate increment of -0.00757 meters. This  
↳ regular spacing ensures that the stack maintains  
↳ a tidy and organized appearance.
- \*\*Fixed Horizontal and Depth Positions\*\*:**

- The x and z coordinates of the objects remain  
↳ constant. In Program 1, these coordinates are  
↳ both 0.0 for all objects, indicating that there  
↳ is no horizontal or depth displacement. This  
↳ fixed positioning along the x and z axes  
↳ maintains the straight vertical alignment  
↳ characteristic of a stack.

```
Hypothesized Commonalities for Multiple Programs:
If there were multiple programs describing the "stack"
↳ motif, we might observe similar commonalities:
```

- \*\*Object Type\*\*:**
  - Multiple programs could feature different types of  
↳ objects (e.g., books, boxes), but within each  
↳ program, the objects would be consistently of the  
↳ same type.
- \*\*Uniform Dimensions\*\*:**
  - Even if different programs use different objects,  
↳ the dimensions of the objects within each program  
↳ would remain uniform.
- \*\*Vertical Spacing\*\*:**
  - The vertical displacement between objects might  
↳ vary between programs, but within each program,  
↳ the displacement would be consistent and regular.
- \*\*Vertical Alignment\*\*:**
  - Each program would align the objects vertically  
↳ along the y-axis, maintaining this key  
↳ characteristic of a stack.
- \*\*Fixed x and z Coordinates\*\*:**
  - While different programs might place the stack at  
↳ different positions in the world, within each  
↳ program, the x and z coordinates would be fixed  
↳ for all stacked objects to enforce the vertical  
↳ alignment.

```
Conclusion:
```

```
The commonalities observed in the "stack" motif
↳ emphasize uniformity in object type and size,
↳ regular vertical spacing, and maintaining a
↳ consistent vertical alignment without horizontal or
↳ depth displacement. These patterns ensure that the
↳ spatial arrangement of objects reflects a neat and
↳ orderly stack.
```

Asking LLM to make observations of *differences* in a set of motif program with motif type `stack` using the prompt `generalize_high_level_differences`

```
What are the differences among these programs?
Specifically, how are the objects placed differently in
↳ each program?
For each program, observe and describe the differences
↳ in the program structure that lead to the different
↳ spatial motifs.
If there is only one program, hypothesize how the
↳ program could be written differently to create a
↳ different spatial motif of the same type.
Your hypothesized programs should not deviate from the
↳ concrete, non-hypothetical programs you analyzed,
↳ and should focus on the number of objects, their
↳ positions, rotations, and sizes as the main axes of
↳ variation.
The hypothesized programs should be close to the
↳ original program in terms of the spatial motif they
↳ create (e.g., a stack of 3 objects instead of 4, or
↳ a row of 4 objects instead of 3, etc.)
Respond with a detailed description of the differences
↳ of each program in text.
Refer to the programs by their order in the previous
↳ prompt.
```

#### Response from LLM (text)

Since we have only one concrete program to analyze,  
 ↳ let's hypothesize how it could be written  
 ↳ differently to create different variations of the  
 ↳ same spatial motif, "stack." We will focus on  
 ↳ varying the number of objects, their positions, and  
 ↳ possibly other factors such as rotations (though  
 ↳ rotations might be less relevant for the stack  
 ↳ motif).

### ### Hypothesized Variations:

1. **\*\*Changing the Number of Objects\*\*:**  
 - Instead of a stack of seven plates, we can create a  
 ↳ stack with a different number of plates. For  
 ↳ instance:

```
```python
objs = []
half_size = [0.08909, 0.0143, 0.08853]
initial_y = 0.0
displacement_y = -0.00757

for i in range(5): # Stack of 5 plates instead of 7
    y_position = initial_y + i * displacement_y
    obj = create('plate', half_size)
    move(obj, 0.0, y_position, 0.0)
    objs.append(obj)
...

- **Difference**: The main difference is the
  ↳ reduction in the number of objects from 7 to 5,
  ↳ leading to a shorter stack.
```

2. ****Changing the Vertical Displacement**:**
 - By modifying the vertical displacement, we can
 ↳ create a stack with different spacing between
 ↳ objects:

```
```python
objs = []
half_size = [0.08909, 0.0143, 0.08853]
initial_y = 0.0
displacement_y = -0.01 # Increased spacing between
 ↳ plates

for i in range(7):
 y_position = initial_y + i * displacement_y
 obj = create('plate', half_size)
 move(obj, 0.0, y_position, 0.0)
 objs.append(obj)
...

- **Difference**: The increased vertical displacement
 ↳ creates a stack with more space between each
 ↳ plate compared to the original program.
```

3. **\*\*Introducing Randomness\*\*:**  
 - Adding a slight randomness to the displacement  
 ↳ could simulate a less orderly but still  
 ↳ essentially vertical stack:

```
```python
import random

objs = []
half_size = [0.08909, 0.0143, 0.08853]
initial_y = 0.0
displacement_y = -0.00757

for i in range(7):
    random_offset = random.uniform(-0.001, 0.001)
    y_position = initial_y + i * displacement_y +
      ↳ random_offset
    obj = create('plate', half_size)
    move(obj, 0.0, y_position, 0.0)
    objs.append(obj)
...

- **Difference**: Adding a slight random offset to
  ↳ the y-position for each plate introduces minor
  ↳ variations, making it look less perfect while
  ↳ still maintaining the overall stack structure.
```

4. ****Using Different Object Sizes**:**
 - Introducing variety in object sizes while
 ↳ maintaining the stack structure:

```
```python
objs = []
initial_y = 0.0
displacement_y = -0.00757

for i in range(7):
 half_size = [0.08909, 0.0143 + i * 0.002,
 ↳ 0.08853] # Incremental increase in size
 y_position = initial_y + i * displacement_y
 obj = create('plate', half_size)
 move(obj, 0.0, y_position, 0.0)
 objs.append(obj)
...

- **Difference**: Varying the size slightly with each
 ↳ subsequent object while maintaining the stack
 ↳ adds a gradient or stepped effect to the stack
 ↳ without deviating from the vertical alignment.
```

### ### Summary of Differences:

- **\*\*Number of Objects\*\*:** Varying the quantity alters the  
 ↳ height and the visual appearance of the stack.
- **\*\*Vertical Displacement\*\*:** Adjusting the spacing  
 ↳ between objects changes the density/compression of  
 ↳ the stack.
- **\*\*Randomness\*\*:** Introducing slight variations in  
 ↳ positioning to make the stack appear less  
 ↳ mechanically perfect.
- **\*\*Object Sizes\*\*:** Varying object sizes can add a  
 ↳ dynamic look while retaining the core stack motif.

Each of these variations adheres to the overarching  
 ↳ "stack" motif but introduces subtle differences in  
 ↳ how the objects are placed relative to each other.  
 ↳ These variations maintain the essence of a vertical  
 ↳ arrangement while exploring different aspects of  
 ↳ spatial positioning and object properties.

## Meta-program reasoning.

Asking LLM to explain why all motif program belong to the motif type stack using the prompt  
 generalize\_high\_level\_motif\_reason

Despite these differences, these programs all fall under  
 ↳ the same type of spatial motif: "<MOTIF\_TYPE>".  
 Here is a list of all available motif types:

1. stack - multiple objects of the same type are placed  
 ↳ orderly on top of each other
2. pile - objects are placed on top of each other but  
 ↳ not in an orderly manner (when in doubt between  
 ↳ stack and pile, choose pile if the description  
 ↳ contains the word "pile" explicitly)
3. row - objects are placed next to each other in a row
4. grid - objects are placed orderly in a grid, like a  
 ↳ chessboard
5. left\_of - one object placed to the left of another  
 ↳ object
6. in\_front\_of - one object placed in front of another  
 ↳ object
7. on\_top - one object placed on top of another object
8. surround - objects are placed around a central object  
 ↳ in a circular manner
9. wall\_vertical\_column - objects are placed in a column  
 ↳ from top to bottom vertically on a wall
10. wall\_horizontal\_row - objects are placed in a row  
 ↳ from left to right vertically on a wall
11. wall\_grid - objects are placed in a grid orderly and  
 ↳ vertically on a wall (when in doubt between grid and  
 ↳ wall\_grid, choose wall\_grid if the objects are  
 ↳ placed on a wall)
12. letter - objects are placed to form a letter of the  
 ↳ alphabet
13. rectangular\_perimeter - objects are placed around  
 ↳ the perimeter of a rectangular shape facing inward

Based on your observations, what are the reasons that  
→ these programs fall under the same type of spatial  
→ motif?  
Also, identify the closest incorrect motif type and  
→ explain why it is not the correct motif type.  
Respond with a detailed description of the reasons in  
→ text.

## Response from LLM (text)

### Reasons These Programs Fall Under the "Stack" Motif:

1. **Object Consistency**:
  - All objects in each program are of the same type.  
→ For example, in the provided program, all objects  
→ are 'plate'. This consistency ensures that the  
→ overall structure is uniform.
2. **Vertical Alignment**:
  - Objects are placed in a vertical line with  
→ consistent alignment along the y-axis. Each  
→ object is positioned directly above or below the  
→ previous one, creating an orderly vertical  
→ configuration.
3. **Uniform Spacing**:
  - There is a consistent vertical displacement between  
→ consecutive objects. This regular spacing further  
→ reinforces the orderly arrangement typical of a  
→ stack.
4. **Fixed x and z Coordinates**:
  - The x and z coordinates of all objects remain  
→ constant, ensuring that there is no horizontal or  
→ depth displacement. This fixed positioning along  
→ the x and z axes maintains the strict vertical  
→ alignment required for a stack.
5. **Orderliness**:
  - The objects are placed in an orderly manner,  
→ directly on top of each other. There is no  
→ randomness or displacement that would suggest  
→ disorganized placement.

### Closest Incorrect Motif Type:

The closest incorrect motif type is **"pile"**.

### Why "Pile" is Not Correct:

1. **Disorganized Placement**:
  - The "pile" motif is characterized by objects placed  
→ on top of each other but not in an orderly  
→ manner. This typically implies some degree of  
→ randomness or irregular displacement among the  
→ objects.
  - In the analyzed and hypothesized programs, the  
→ placement of objects is precise, with no  
→ randomness introduced that would suggest a  
→ disorganized pattern. Even when slight randomness  
→ is added in one of the hypothesized variations,  
→ the overall structure remains far too regular and  
→ ordered to be considered a pile.
2. **Lack of Explicit Disorder**:
  - Descriptions of the motif type "pile" often  
→ explicitly include the word "pile," indicating a  
→ conscious choice to describe a disorganized  
→ arrangement.
  - The provided programs have a clear, structured  
→ approach to object placement, consistently using  
→ precise coordinates and displacements to maintain  
→ order, which does not align with the definition  
→ of a "pile."

### Summary:

These programs fall under the "stack" motif because they  
→ involve multiple objects of the same type placed in  
→ an orderly vertical arrangement with consistent  
→ spacing and alignment. The closest incorrect motif  
→ type, "pile," is not appropriate because it implies  
→ disorganized placement, which contradicts the  
→ precise and orderly nature of the stack motif  
→ observed in the programs. The consistency in object  
→ type, uniform spacing, vertical alignment, and lack  
→ of random disorder are key reasons for classifying  
→ these programs as "stack" rather than any other  
→ motif type.

Asking LLM to reason about how to gener-  
ate a meta-program for the motif type stack by  
first identifying the *arguments* using the prompt  
generalize\_low\_level\_arguments

With this conversation in mind, you are now tasked to  
→ write a python function (a meta-program) that can be  
→ used to create various spatial motifs of  
→ "<MOTIF\_TYPE>".

A user should be able to call this meta-program with  
→ different arguments to create different spatial  
→ motifs of this type.

Using the provided programs as examples, the  
→ meta-program should be able to create any spatial  
→ motif of this type with slight variations in the  
→ number of objects and their attributes.

Apart from the objects in the examples, the meta-program  
→ should be able to create new spatial motifs of  
→ different object types and attributes.

Use your analysis on the commonalities and differences  
→ among the example programs to guide you in writing  
→ the meta-program.

Keep in mind that the final meta-program should be able  
→ to recreate the given example spatial motifs as  
→ closely as possible.

Avoid taking plain lists of object attributes, such as  
→ coordinates, sizes, and rotations, as arguments, as  
→ this defeats the purpose of using a meta-program to  
→ capture fundamental spatial patterns.

I will guide you through the process of writing the  
→ function step by step.

First, what are the arguments that the meta-program  
→ should take to create a spatial motif of this type?  
Respond with a detailed description of the arguments in  
→ text.

## Response from LLM (text)

To create a versatile meta-program that can generate  
→ various spatial motifs of the "stack" type, we need  
→ to carefully consider the arguments that the  
→ function should accept. These arguments should allow  
→ for flexibility in creating different stacks while  
→ adhering to the core patterns observed in the  
→ provided examples. Here are the key arguments:

### Arguments for the Meta-Program:

1. **Object Type ('label')**:
  - The type of object to be created, which will  
→ determine the label for each object in the stack  
→ (e.g., 'plate', 'box', etc.).
2. **Number of Objects ('num\_objects')**:
  - The total number of objects to be included in the  
→ stack. This allows flexibility in defining the  
→ height or length of the stack by varying the  
→ number of constituent objects.
3. **Initial Position ('initial\_position')**:
  - A list or tuple representing the initial world  
→ coordinates (x, y, z) for the bottom or starting  
→ object in the stack.

4. **Vertical Displacement** (`displacement\_y`)\*\*:
  - The consistent vertical distance between
    - ↳ consecutive objects in the stack. This will
    - ↳ determine the regular spacing between the objects
    - ↳ along the y-axis.
5. **Object Size** (`half\_size`)\*\*:
  - The dimensions of each object in the stack,
    - ↳ represented as a list of three floats
    - ↳ [half\_width, half\_height, half\_depth]. This
    - ↳ ensures that all objects in the stack have
    - ↳ uniform sizes.
6. **Random Offset** (`random\_offset`)\*\* (optional):
  - An optional argument specifying the range of random
    - ↳ variation in the vertical displacement for each
    - ↳ object. This can add slight randomness to the
    - ↳ positioning, creating a less mechanically perfect
    - ↳ stack if desired.

#### ### Additional Considerations:

- **Rotation**: While rotations might be less relevant
  - ↳ for a perfectly vertical stack, we could include an
  - ↳ optional argument for rotating objects around one or
  - ↳ more axes, though this will be kept simple and
  - ↳ optional.
- **Axis of Stack**: For future extensibility, we might
  - ↳ consider allowing the stack to be along different
  - ↳ axes, though for now, we focus on the y-axis for
  - ↳ simplicity.

#### ### Summary of Arguments:

Here is a refined list:

1. `label (str)`: Type of object to create (e.g.,
  - ↳ 'plate').
2. `num_objects (int)`: Number of objects in the stack.
3. `initial_position (list[float])`: Initial position
  - ↳ [x, y, z] of the first object.
4. `displacement_y (float)`: Vertical displacement
  - ↳ between objects.
5. `half_size (list[float])`: Dimensions of each object
  - ↳ as [half\_width, half\_height, half\_depth].
6. `random_offset (float, optional)`: Range for random
  - ↳ variation in vertical displacement.

The next step will be to proceed to writing the  
 ↳ meta-program using these arguments to generate the  
 ↳ desired spatial motifs.

Asking LLM to reason about how to generate a  
 meta-program for the motif type stack by first  
 identifying the *function structure* using the prompt  
 generalize\_low\_level\_structure

You have identified the arguments needed for the  
 ↳ meta-program.  
 Now, think about the structure of the meta-program.  
 How should the meta-program be structured to generalize  
 ↳ the spatial motif of "<MOTIF\_TYPE>"?  
 Aim to encapsulate the commonalities and differences  
 ↳ among the example programs in the structure of the  
 ↳ meta-program.  
 Think about how the arguments should be used in the  
 ↳ meta-program to create the spatial motif.  
 What programming constructs should be used to  
 ↳ encapsulate the commonalities and differences among  
 ↳ the example programs?  
 Respond with a detailed description of the structure of  
 ↳ the meta-program in text.

## Response from LLM (text)

### Structure of the Meta-Program:

To generalize the spatial motif of "stack," the  
 ↳ meta-program needs to encapsulate the commonalities  
 ↳ and differences observed in the example programs.  
 ↳ This involves using the identified arguments  
 ↳ effectively and employing appropriate programming  
 ↳ constructs to create a stack of objects. Here's a  
 ↳ detailed description of the structure of the  
 ↳ meta-program:

#### ### 1. Function Definition:

The function should be defined with the arguments  
 ↳ identified earlier, including the main ones (label,  
 ↳ num\_objects, initial\_position, displacement\_y,  
 ↳ half\_size) and optional ones (random\_offset).

#### ### 2. List Initialization:

Initialize an empty list `objs` to store the created  
 ↳ objects, ensuring that the created objects can be  
 ↳ accessed and manipulated as needed.

#### ### 3. Loop to Create Objects:

Use a loop structure to iterate through the number of  
 ↳ objects (`num\_objects`). This loop will handle the  
 ↳ creation and placement of each object in the stack.

#### ### 4. Object Creation:

Within the loop, create the object using the  
 ↳ `create(label, half\_size)` function. Each object  
 ↳ will have the same type and dimensions as specified  
 ↳ by the arguments.

#### ### 5. Position Calculation:

Calculate the position for each object. This involves  
 ↳ using the initial position (`initial\_position`) and  
 ↳ applying the vertical displacement incrementally.

- **Base Position**: Start with the initial position for
  - ↳ the first object.
- **Incremental Displacement**: For each subsequent
  - ↳ object, add the vertical displacement
  - ↳ (`displacement\_y`) to the y-coordinate of the
  - ↳ previous object's position.
- **Optional Random Offset**: Apply a random offset to
  - ↳ the vertical displacement if the `random\_offset`
  - ↳ argument is provided, simulating a less perfect
  - ↳ stack.

#### ### 6. Moving Objects:

Use the `move()` function to place each object at the  
 ↳ calculated position. Ensure that the x and z  
 ↳ coordinates remain constant for the stack along the  
 ↳ y-axis.

#### ### 7. Append to List:

Append each created and moved object to the `objs` list.

#### ### 8. Return the List:

After the loop completes, return the `objs` list  
 ↳ containing all the objects in the stack.

#### ### Programming Constructs:

- **Loop (for loop)**: To iterate through the number of
  - ↳ objects.
- **Arithmetic Operations**: For calculating the
  - ↳ positions.
- **Conditional Handling**: To apply optional random
  - ↳ offsets.
- **List Operations**: For storing and returning the
  - ↳ objects.

#### ### Pseudo-Code:

Below is a pseudo-code representation to help visualize  
 ↳ the structure:

```
```python
def create_stack(label, num_objects, initial_position,
↳ displacement_y, half_size, random_offset=None):
    objs = []
    initial_x, initial_y, initial_z = initial_position
```

```

for i in range(num_objects):
    # Calculate vertical position with optional
    ↪ random offset
    y_position = initial_y + i * displacement_y
    if random_offset:
        y_position += random.uniform(-random_offset,
        ↪ random_offset)

    # Create and move the object
    obj = create(label, half_size)
    move(obj, initial_x, y_position, initial_z)

    # Append to the list
    objs.append(obj)

return objs
...

### Summary:
The meta-program is structured to use the provided
↪ arguments to create a stack of objects. It leverages
↪ a loop for object creation and placement, arithmetic
↪ operations for position calculations, optional
↪ randomness for variability, and list operations to
↪ manage the created objects. This structure ensures
↪ the generality and flexibility needed to recreate
↪ the given example spatial motifs and generate new
↪ variations within the "stack" motif.

```

Meta-program generation.

Asking LLM to generate a meta-program for the motif type stack based on the reasoning it just did using the prompt generalize_low_level. The LLM is asked to also use docstring and include comments.

Take a deep breath and follow the steps below carefully. You have identified the arguments and the structure of ↪ the meta-program. Now, write the meta-program that can create various ↪ spatial motifs of "<MOTIF_TYPE>". Here is a meta-program of the same motif type you ↪ created in the past:

```

'''python
<PAST_META_PROGRAM>
'''

```

You can use this as a reference to write the new ↪ meta-program, but you can also write it from ↪ scratch. Refer to the arguments you identified and the structure ↪ you described to write the meta-program. Remember that the meta-program should be able to ↪ recreate the spatial motifs of the example programs ↪ as closely as possible. Avoid taking plain lists of object attributes, such as ↪ coordinates, sizes, and rotations, as arguments, as ↪ this defeats the purpose of using a meta-program to ↪ capture fundamental spatial patterns. Include a docstring at the beginning of the meta-program ↪ to describe the purpose of the function and the ↪ arguments it takes. Use comments to explain your code step by step. Please respond with code only.

Response from LLM (python)

```

import random

def create_stack(label, num_objects, initial_position,
↪ displacement_y, half_size, random_offset=None):
    """
    Create a stack of objects with the specified
    ↪ parameters.

    Args:
    label (str): Type of object to create (e.g.,
    ↪ 'plate').

```

```

num_objects (int): Number of objects in the stack.
initial_position (list[float]): Initial position [x,
↪ y, z] of the first object.
displacement_y (float): Vertical displacement
↪ between objects.
half_size (list[float]): Dimensions of each object
↪ as [half_width, half_height, half_depth].
random_offset (float, optional): Range for random
↪ variation in vertical displacement.

```

```

Returns:
list: A list of created objects.
"""
objs = []
initial_x, initial_y, initial_z = initial_position

```

```

for i in range(num_objects):
    # Calculate vertical position with optional
    ↪ random offset
    y_position = initial_y + i * displacement_y
    if random_offset is not None:
        y_position += random.uniform(-random_offset,
        ↪ random_offset)

```

```

    # Create and move the object
    obj = create(label, half_size)
    move(obj, initial_x, y_position, initial_z)

    # Append the object to the list
    objs.append(obj)

```

```

return objs

```

Meta-program validation.

Asking LLM to create function calls to the meta-program it just created for each of the motif programs, using the prompt generalize_low_level.batch.recreate.

Using the meta-program you wrote, what are the function ↪ calls needed to recreate each of the example ↪ programs you analyzed previously? Write one function call for each example program, that ↪ recreates the spatial motif of the program as ↪ closely as possible. Ignore the programs you hypothesized and only write ↪ function calls for the example programs I provided. Response with a json-like text structure with the ↪ example program order integers as keys and the ↪ function calls as values. For example, {"1": "function_call(program1_args)", "2": ↪ "function_call(program2_args)", ...}

Response from LLM (json)

```

{
  "1": "create_stack('plate', 7, [0.0, 0.0, 0.0],
    ↪ -0.00757, [0.08909, 0.0143, 0.08853])"
}

```

All validation checks were passed after iteration 1. If the validation did not pass, the LLM is asked to try to correct for any errors using the generalize_low_level.feedback prompt.

Meta-program documentation. Asking LLM to refine the comments in the meta-program using the prompt generalize_refine_comments

You have successfully written the meta-program to ↪ generalize the spatial motif of "<MOTIF_TYPE>". The latest function calls you provided are able to ↪ recreate the spatial motifs of the example programs. Please update the docstring and comments in the ↪ meta-program to provide as much information as ↪ possible about the expected arguments.

If the program contains well-defined parts that can be
 ↪ explained, include comments to describe these parts
 ↪ - how they contribute to the spatial motif and how
 ↪ they differ from other parts.
 At inference time, the meta-program will be the sole
 ↪ source of information for recreating this spatial
 ↪ motif.
 Therefore, these comments will be crucial for
 ↪ understanding how the meta-program should be used.
 Include at least one example function call in the
 ↪ comments to show how the meta-program should be
 ↪ called.
 Please respond with code only.

Response from LLM (python)

```
import random

def create_stack(label, num_objects, initial_position,
  ↪ displacement_y, half_size, random_offset=None):
    """
    Create a stack of objects with the specified
    ↪ parameters.

    Args:
    label (str): Type of object to create (e.g.,
    ↪ 'plate').
    num_objects (int): Number of objects in the stack.
    initial_position (list[float]): Initial position [x,
    ↪ y, z] of the first object.
    displacement_y (float): Vertical displacement
    ↪ between objects.
    half_size (list[float]): Dimensions of each object
    ↪ as [half_width, half_height, half_depth].
    random_offset (float, optional): Range for random
    ↪ variation in vertical displacement. Default is
    ↪ None.

    Returns:
    list: A list of created objects.

    Example Call:
    objs = create_stack('plate', 7, [0.0, 0.0, 0.0],
    ↪ -0.00757, [0.08909, 0.0143, 0.08853])

    This example recreates a stack of seven plates with
    ↪ the given initial position, displacement, and
    ↪ dimensions.
    """

    # Initialize an empty list to store the created
    ↪ objects
    objs = []

    # Extract initial position components for
    ↪ readability
    initial_x, initial_y, initial_z = initial_position

    # Iterate over the number of objects to be created
    ↪ in the stack
    for i in range(num_objects):
        # Calculate vertical position with optional
        ↪ random offset
        y_position = initial_y + i * displacement_y
        if random_offset is not None:
            y_position += random.uniform(-random_offset,
            ↪ random_offset)

        # Create the object with the specified label and
        ↪ dimensions
        obj = create(label, half_size)

        # Move the object to its calculated position
        move(obj, initial_x, y_position, initial_z)

        # Append the created and moved object to the
        ↪ list
        objs.append(obj)
```

```
# Return the list of created objects
return objs
```

The meta-program provided by the LLM is then stored in the meta-program library with the motif type of stack.

References

- [1] Gabriel Ilharco, Mitchell Wortsman, Ross Wightman, Cade Gordon, Nicholas Carlini, Rohan Taori, Achal Dave, Vaishal Shankar, Hongseok Namkoong, John Miller, Hannaneh Hajishirzi, Ali Farhadi, and Ludwig Schmidt. OpenCLIP, 2021.