A Appendix

The appendix serves as a complimentary document to the paper detailing the data collection process, analysis, and program synthesis. It should be used in conjunction with the following:

1. the LARC dataset and its annotation workflow, and bandit algorithm can be found in:
   `https://github.com/samacqua/LARC`
   Which contains the explore gui for the whole dataset entirely in browser (see Fig 10):
   `https://samacqua.github.io/LARC/explore`
2. alternatively, one can download the repo and run the explore gui offline:
   (a) point to the `LARC` root directory
   (b) run ‘`python3 -m http.server`‘
   (c) open ‘`localhost:8000/explore/`‘ in a chrome browser
3. program synthesis using language codes is at this URL :
   `https://github.com/theosech/ec/tree/language-guided_program_synthesis_for_larc`

A.1 The LARC Explorer GUI

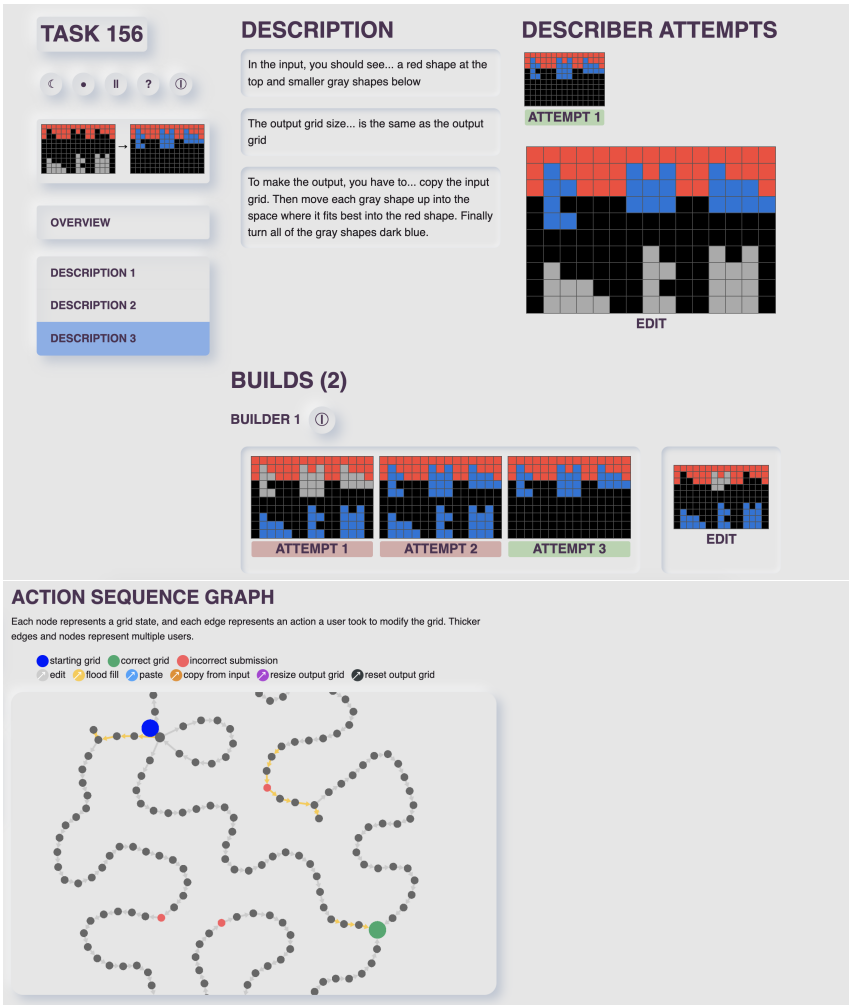

Figure 10: The explore interface for task 156 (top). action sequence graph of builder 1 (bot)

A.2 Consent Form and Annotation Workflow

Consent Form In this study, you will interpret descriptions of an abstract pattern that you observe in grids. By answering the following questions, you are participating in a study performed by cognitive scientists in [*author institution*]. If you have questions about this research, please contact [*author*] at [*author email*]. Your participation in this research is voluntary. You may decline to answer any or all of the following questions. You may decline further participation, at any time, without adverse consequences. Your anonymity is assured; the researchers who have requested your participation will not receive any personal identifying information about you. By clicking 'I AGREE' you indicate your consent to participate in this study.

Annotation Workflow Then, the user is given tutorials about communicating ARC tasks, and dynamically assigned a sequence of describe and/or build tasks until they have completed 45 minutes of work. Figure 11 shows the build and describe interface. For full workflow see `LARC/collection`.

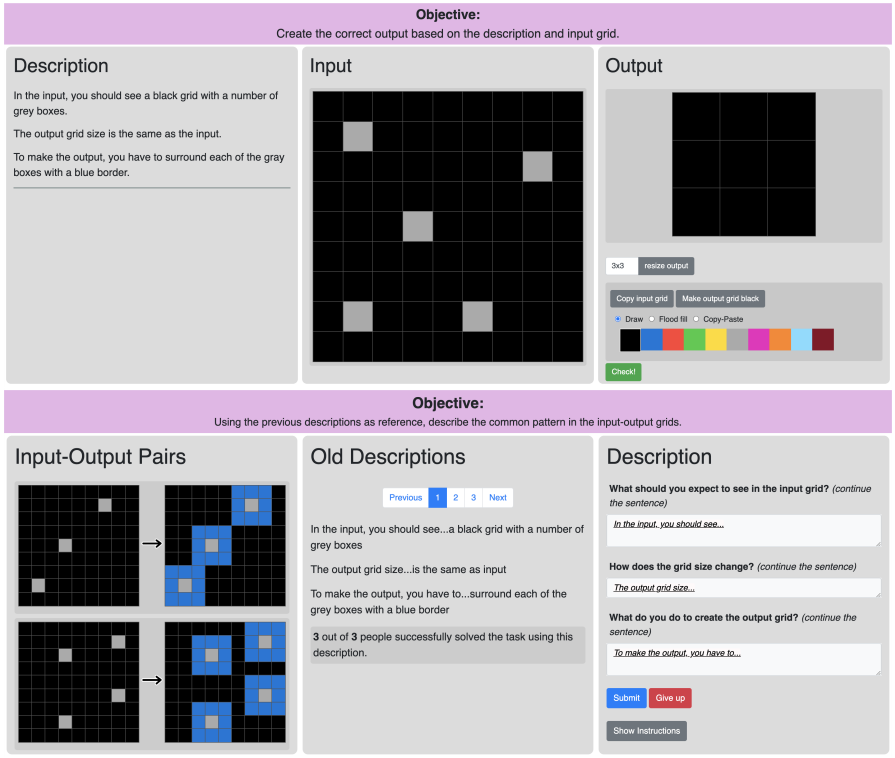

Figure 11: *A.* The builder interface. *B.* The describer interface.

A.3 LARC Linguistic Analysis Tagging Scheme

The tagged phrases can be found at `LARC/dataset/annotated_phrases.csv`

The phrases were codified by expert coders using a set of 17 binary tags. For each tag, a phrase can either be a positive instance (+) or a negative instance (-) [7]. The following table details the tags and coding scheme used:

Tag	Description	Examples
Procedure	Directly commands the builder to do something; If you were to delete it, the program will fail to execute.	(+) Fill each enclosed hole with yellow (-) look at the color that form the design in the input.
Metaphor	A metaphor can be an analogy or reference to human common sense knowledge – e.g. spiral.	(+) A random green pattern (+) A pattern like a long A
Clarification	A phrase made following a previous statement that attempts to clarify misinterpretations.	(+) Then, copy and paste each colored square in the input grid 4 times – once in each "quadrant" (+) (or 5 rows or whatever the number of rows is before it repeats). (+) Where there's a dark blue square, put orange squares directly above and below it (4 total).
Example	Gives a concrete instance.	(+) The opposite is also true (for example if it is light blue, change to dark red).
Array	Makes a comment about a collection of objects sharing some common property.	(+) Where there's a dark blue square, put orange squares directly above and below it (4 total). (+) Leave the magenta and light blue squares as they are; do not add anything to them if they are present.
Validation	After the builder executes a procedure, check if they got the right answer (i.e. asserts, test-cases, verification, or error handling).	(+) You should end up with all blue boxes touching each other (+) Fill in all of the black boxes to complete the pattern until there are no more black boxes.
Loop	Includes a looping procedure, such as the use of *while*, *for*, *until*, *for each*, or *repeat*.	(+) Continue coloring green until you reach the center of the grid. (+) Reduce the grid size so that one square is available for each group.
Start_Stop	Talks about the process or duration of some operations.	(+) start at the upper right corner (+) the red shape needs to move until it is touching the blue cube
Conditional	Of the form *if X then Y*.	(+) If they do not match, make the output square green.
Logic	Includes first-order logic, such as *same*, *and*, *or*, or *not*.	(+) The same size as the input (+) You will not use dark blue squares at all (-) A 4x4 pattern
Framing	Sets up the problem by offering a particular point of view, defining some objects to be referred to later.	(+) four colored area. (+) 1 or 2 squares filled in with the same color on a black background.

[7] marked by 1 and 0 respectively in the csv

Tag	Description	Examples
Spacial Relation	Any reference to a relative position in space to some other component. Positive examples include: under, reaches, touches, angle, outer, downward, parallel, near, after, in between, central, etc.	(+) The red shape next to the blue shape (+) Put yellow inside the green
Physical Interaction	Any reference to an imaginary force.	(+) The red object falls (+) Blue slides to the left towards red
Contact Transform	Influence via contact, i.e. any specialized version of physical interaction that involves *at least two objects* and *some type of contact causality*.	(+) Move X until contact with Y (+) Set X touching Y and turn it the color of Y (-) Red moves left one square
Affine Transform	Any reference to a affine transformation over an object, such as rotation, translation, etc.	(+) Rotate 90 degrees (+) Extend the square into a line
Visual-Graphical Transform	Any other visual or graphical modification other than a geometric one, such as coloring, flood-fill, or drawing a new shape.	(+) Make it gray (+) Draw a line
Object Detection	The localization of a cohesive, bounded object.	(+) The red shape (+) Move **it** to the left (+) The pattern

These tags can also be grouped hierarchically into the following categories:

Programmatic: procedure, array, validation, loop, start_stop, conditional, logic

Human/Mechanisms for Domain General Communication: metaphor, clarification, example, framing

Objects and Object Manipulation: spacial_relation, physical_interaction, contact_transform, geometric_transform, visual_graphical_transform, object_detection

A.4 THE ATTEMPTED LARC DSL

As LARC is DSL-open, we must first *construct* a suitable DSL before applying (symbolic) program synthesis approaches. Here is our attempt at constructing such a DSL. For each DSL primitives, we also list its corresponding pseudo-annotation comments. We hand-designed DSL a for the LARC domain consisting of 103 primitives (implemented as a set of polymorphically typed λ-calculus expressions) intended to be broadly and basically applicable to all tasks on the domain – the DSL operates over grids of pixels, and contains simple functions designed to repeatedly perform image transformations over pixel grids to produce an output grid. The complete DSL is available at the released code repository; below we provide representative example functions and the accompanying natural language glosses of their behavior used in the *pseudoannotations* generative procedure; as well as sampled program expressions and their generated pseudoannotations.

Example DSL Functions and Natural Language Gloss Function Annotations

DSL Function	*Natural Language Gloss*
blocks_to_original_grid	'place blocks onto input grid'
blocks_to_min_grid	'get the smallest grid containing the blocks'
first_of_sorted_object_list	'get the block with the smallest or greatest value of'
singleton_block	"
merge_blocks	"
filter_blocks	'remove the blocks that have'
map_blocks	'for every block'
filter_template_block	'find the main block'
reflect	'reflect'
move	'move'
center_block_on_tile	'move block to tile'
duplicate	'duplicate'
grow	'enlarge'
fill_color	'color the block'
fill_snakewise	'color the block in a snake pattern with'
replace_color	'replace colors'
remove_black_b	'remove the black background'
remove_color	'remove color from block'
box_block	'get smallest rectangle containing block'
wrap_block	'surround block with'
filter_block_tiles	'only keep tiles that'
map_block_tiles	'for each tile of block'
to_min_grid	"
to_original_grid_overlay	'place block on input grid'
get_height	'get height of block'
get_width	'get width of block'
get_original_grid_height	'get the height of the input grid'
get_original_grid_width	'get the width of the input grid'
get_num_tiles	'count the number of tiles of the block'
nth_primary_color	'find the nth most common color'
is_symmetrical	'is the block symmetrical'
is_rectangle	'is the block a rectangle'
has_min_tiles	'does the block have at least n tiles'
touches_any_boundary	'does the block touch any edge of the grid'
touches_boundary	'does the block touch the edge'
has_color	'does the block have color'
is_tile	'is the block a tile'
block_to_tile	"
get_block_center	'get the central tile of the block'
map_for_directions	'in every direction'
find_same_color_blocks	'find blocks based on shared color'

find_blocks_by_black_b	'find blocks based on if they are separated by the black background'
find_blocks_by_color	'find blocks based on if they are separated by the given color background'
find_blocks_by_inferred_b	'find blocks based on if they are separated by the background'
grid_to_block	''
split_grid	'split the grid in half'
find_tiles_by_black_b	'find the tiles based on if they are separated by the black background'
is_interior	'is the tile in the interior of a block'
is_exterior	'is the tile in the exterior of a block'
tile_touches_block	'does the tile touch the block'
tile_overlaps_block	'does the tile overlap the block'
tile_to_block	''
extend_towards_until	'extend tile towards a direction until the condition is met'
extend_towards_until_edge	'extend tile towards a direction until it touches the edge'
extend_until_touches_block	'extend tile towards a direction until it touches the edge'
move_towards_until	'move tile towards direction until condition is met'
move_towards_until_edge	'move tile towards direction until it touches edge'
move_until_touches_block	'move tile towards direction until it touches block'
move_until_overlaps_block	'move tile towards direction until it overlaps block'
get_tile_color	'get the color of the tile'
tiles_to_blocks	''
filter_tiles	'only keep tiles that'
map_tiles	'for every tile'
overlap_split_blocks	'overlap the split blocks based on colors'
splitblocks_to_blocks	''
color_logical	'take logical operation on colors using them as true and false'
land	'logical operator and'
lor	'logical operator or'
lxor	'logical operator xor'
negate_boolean	'not'
map_tbs	'for every block in template block scene'
make_colorpair	'make pair of colors'
north	'top'
south	'bottom'
west	'left'
east	'right'
north_east	'top right'
north_west	'top left'
south_east	'bottom right'
south_west	'bottom left'
0	'0'
1	'1'
2	'2'
3	'3'
4	'4'
5	'5'
6	'6'
7	'7'
8	'8'
9	'9'
true	''
false	''

invisible	'invisible'
black	'black'
blue	'blue'
red	'red'
green	'green'
yellow	'yellow'
grey	'grey'
pink	'pink'
orange	'orange'
teal	'teal'
maroon	'maroon'

Example Sampled Programs and Pseudoannotations

Sampled Program	*Natural Language Pseudoannotation*
(lambda (to_original_grid_overlay (remove_color (grid_to_block $0) yellow) false))	'place block on input grid remove color from block yellow'
(lambda (extend_towards_until_edge (block_to_tile (grid_to_block $0)) south_east) true))	'extend tile towards a direction until it touches the edge bottom right'
(lambda (blocks_to_min_grid (tiles_to_blocks (find_tiles_by_black_b $0)) true true))	'get the smallest grid containing the blocks find the tiles based on if they are separated by the black background'

Compared to SCONE [43], LARC poses a significantly greater challenge for distant supervision.

	domain	dsl size	language kind	number of instances
LARC	DSL-open	103	freeform text	354
SCONE: ALCHEMY	DSL-closed	24	step-by-step instruction	4560
SCONE: TANGRAMS	DSL-closed	14	step-by-step instruction	4989
SCONE: SCENE	DSL-closed	33	step-by-step instruction	4402

Table 3: Comparison of LARC to SCONE

A.5 Supplement to Sec. 5: Executing Natural Programs

Enumeration details For a task, we enumerate from the bi-gram distribution (proposed by the neural model) on a high-powered computing cluster for 720s; and with 24 CPUs in parallel.

Other Models: Neural Sequence Decoder We experiment with using a neural sequential decoder which can theoretically capture longer range dependencies. Specifically, we use GRU to decode a program one token at a time. In addition we mask the generated tokens to ensure the generated partial programs are syntactically correct (using the type system). We train using the distant supervision approach exactly as [19], with an epsilon-randomized beam search to balance exploiting the current policy and exploring low probability programs under the policy and take gradient steps on discovered programs using the meritocratic parameter update rule. We train using distant supervision on 24 CPUs for 10 hours of wall-clock time on the train split of 200 tasks. As we can see, the sequence

Neural Sequence Decoder		
	training tasks discovered	testing tasks solved
IO	6 / 200	2 / 183
IO + NL	7 / 200	0 / 183
NL	-	0 / 183

decoder cannot even recover the 10 seed programs during training, and performs poorly on the testing tasks compared to the bigram model. Consequently, we did not attempt pseudo-annotation on the sequence model.

Other Models: CNN encoding of IO We take our best model (IO+NL+pseudo) and additionally condition the neural model with a CNN encoder, rather than leaving it un-conditioned. We find that this model can discover 2 more programs during training and achieves identical outcome to the simpler model without CNN.

	train	test
IO+NL+pseudo	21/200	22/183
IO+NL+pseudo+CNN	23/200	22/183

In general, we find that the standard solution to distant supervision, although effective in SCONE, only discovers a few programs in LARC. This finding is unsurprising for the following reasons:

1. LARC is DSL-open whereas SCONE is not, thus, there is *no* guarantee that we will discover all LARC programs even if we enumerate an *infinite* number of programs.
2. In SCONE, every computer program is a sequence of 5 actions that transform the state of the world. A natural language utterance is collected for each of these actions. The language annotation (natural program) is the sequences of these 5 utterances. As a result there a tight alignment from utterance to actions (tokens in the DSL).
3. SCONE domains have an order of magnitude more tasks to learn from (through distant supervision).

We conclude that collecting simpler, more fine-grained tasks as in SCONE would confer significant benefits to solving LARC, notwithstanding the DSL-open challenge.

A.6 Synthesis with codex

We conduct a exploratory study where we took the 7 tasks solved by the NL+IO specification (in addition to just IO), and see whether github's co-pilot auto-complete tool (built on codex) can correctly infer the right program using only language as prompt. The prompt is constructed by giving a few hundred of pseudo-annotation - program pairs as context (see A.4), followed by a real NL prompt, and asking co-pilot to rephrase it in the LARC DSL:

```
# English: not is the block symmetrical color the block maroon
# Program: (lambda (to_min_grid (grid_to_block $0) (negate_boolean  ...

# English: overlap the split blocks based on colors split the grid ...
# Program: (lambda (overlap_split_blocks (split_grid $0 false) (lambda ...

... 400 of such pairs ...

# English: copy only the biggest shape into the output grid
# Program:
```

The top-10 generated candidates are then executed to see if they can generate the correct output for the given task. See Figure below.

tasks	NL	correct program	example copilot program	# of syntactically valid out of 10 samples	# of correct
0520fde7.json	look at both the left and right parts of the input grid. 'You will notice that the left and right parts are 3x3. For each square that is colored on both the left and right parts, color the output grid with red on the new 3x3.	(lambda (overlap_split_blocks (split_grid $0 false) (lambda (lambda (color_logical $0 $1 red land)))))	(lambda (to_original_grid_overlay (grid_to_block $0) (is_symmetrical (grid_to_block $0) true)))	2	0
0b148d64.json	copy the pattern that is a different color	(lambda (to_min_grid (remove_color (remove_black_b (grid_to_block $0)) (nth_primary_color (grid_to_block $0) 1)) false))	(lambda (to_original_grid_overlay (grid_to_block $0) (has_color (grid_to_block $0) pink)))	4	0
1f85a75f.json	Copy only the biggest shape into the output grid	(lambda (to_min_grid (first_of_sorted_object_list (find_same_color_blocks $0 true false) (lambda (get_num_tiles $0)) false) false))	lambda (copy_shape_to_output_grid (first_of_sorted_object_list (find_same_color_blocks $0 false false) (lambda $0) true)))	0	-
23b5c85d.json	change the grid size with the colored object with the smallest area and fill the same color	(lambda (to_min_grid (first_of_sorted_object_list (find_same_color_blocks $0 true false) (lambda (get_num_tiles $0)) (negate_boolean false)) false))	(lambda (to_original_grid_overlay (fill_color (grid_to_block $0) (grid_to_block $0)) true))	1	0
5582e5ca.json	determine which color is used for the majority of the squares in the input grid, then use that color for the entire output grid.	(lambda (to_original_grid_overlay (fill_color (merge_blocks (find_blocks_by_inferred_b $0 true false) true) (nth_primary_color (grid_to_block $0) 0)) true))	(lambda (to_original_grid_overlay (find_majority_color $0) (lambda (lambda $0))))	0	-
6f8cd79b.json	create a light blue border around the grid. So, any squares that are touching the edges are filled in with light blue. The rest of the squares inside the light blue border remain black	(lambda (to_original_grid_overlay (fill_color (filter_block_tiles (grid_to_block $0) (lambda (is_exterior $0 false))) teal) true))	(lambda (to_original_grid_overlay (map_block_tiles (grid_to_block $0) (lambda (block_to_tile (set_color (reflect (grid_to_block $1) true) light_blue)))) true))	0	-
72ca375d.json	copy one of the figures and just that figure.	(lambda (to_min_grid (singleton_block (filter_blocks (find_same_color_blocks $0 true false) (lambda (is_symmetrical $0 false)))) false))	(lambda (to_original_grid_overlay (copy_grid $0) (lambda (lambda (has_color (fill_color (grid_to_block $0) black))))))	0	-

Figure 12: synthesizing programs using copilot (yes, this is a screenshot of a google sheet)

As we can see, while co-pilot suggests programs that look similar to a correct one stylistically, most are syntactically invalid. For instance, it often invents primitives that do not even exist in our DSL, such as "copy_shape_to_output_grid". Further, none of the syntactically correct programs can produce the intended output either. This is to be expected, as we use a DSL that has not been seen before in any existing corpus of code (on github), and we should not expect codex to perform well

naively. Taking a general model (such as codex) and specializing it to a specific context (LARC) will be exciting future research.

A.7 Description Pairing Study with Clip

We conduct a exploratory study whether the CLIP model [49], which computes a similarity score between image and captions, can perform the simple task of correctly pair a test input grid with its corresponding description in LARC. Performance on this simple binary classification task is a reasonable upper-bound on how large pre-trained models (such as CLIP, Flamingo [47], or DALLE [48]) would work on LARC out of the box.

Specifically, we sampled 1000 instances of $(test\_input\_grid, paired\_description, distractor\_description)$ where the paired description comes from the same LARC task, and the distractor description is randomly chosen from a different task. See Figure 13.

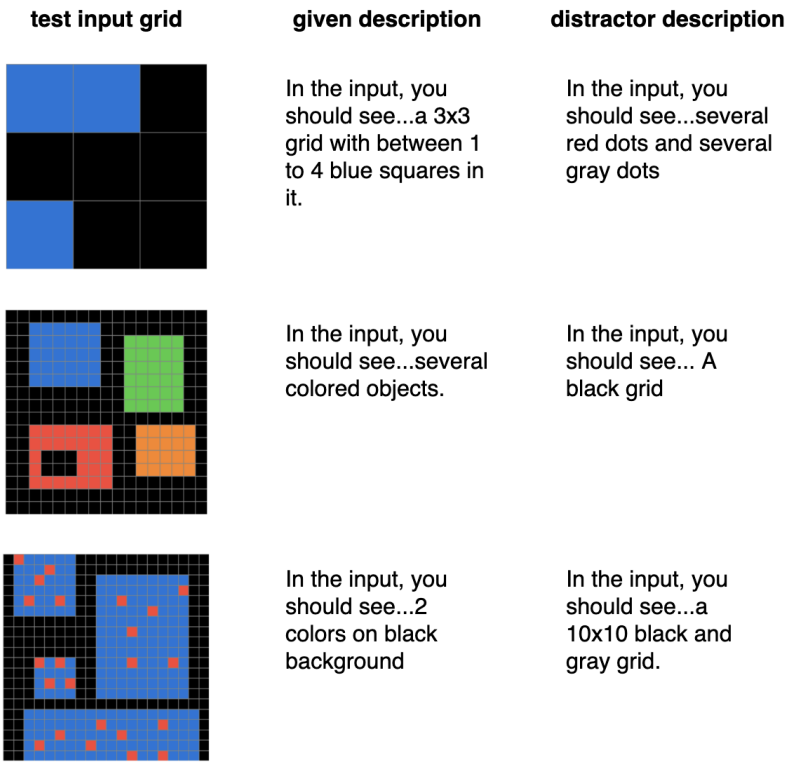

Figure 13: a few instances of the pairing task

We find that CLIP was able to correctly pair the input-grid with its description (having a higher similarity score than the distractor) 64% of the times (randomly guessing will have 50%). To understand how it is making the pairing, we replaced all occurrences of a color word (such as 'red' or 'black') with the dummy word 'COLOR'. After this substitution, the performance drops to 56%. In conclusion, there are certainly values in using a large pre-trained model that builds a joint representation between language and images. However, achieving only 64% accuracy on a extremely simplified, binary classification task, where most of the benefits comes from low level concepts such as color, motivates further research endeavours towards actually solving LARC using pure neural approaches – i.e. *generating* a correct, pixel perfect output grid from input-grid and language alone.

B Appendix : multi-bandit, infinite-arm, best-arm identification

Imagine there are N different mAgIcAl casinos, where each has an infinite number of slot machines (arms). While each individual arm has its own probability p (Bernoulli) of generating an outcome of either 0 or 1, the arms are related to each other depending on the casinos they belong to. Some casinos are easier than others, in a sense that for some, it is easier to find a "good" arm whereas for others, most arms will have a small chance of success. Moreover, each casino i has one (or multiple) best arm, whose probability of generating a 1 is p_i^*. Your job is to identify the best arm within each casino. This is in essence the multi-bandit, infinite-arm, best-arm identification problem.

You can take observations in the casinos, where each observation involves selecting a casino, and trying one of its arms (either one of the arms you already tried, or trying a new one out of its infinite possibilities), observing an outcome of either 0 or 1. We seek an online algorithm that, given any observation budget, propose a set of N arms. Let $p_1 \dots p_N$ denote the ground-truth Bernoulli parameters of the proposed arms. We seek to minimize the following regret:

$$L = \sum_i (p_i^* - p_i)$$

Where each term $p_i^* - p_i$ is the "gap" between the proposed arm and the best arm in a given casino.

B.1 Application to LARC

Our goal is to collect a working natural program for each of the 400 ARC tasks. Natural programs are difficult to collect, because it involves both: 1) obtaining a natural program from a describer and 2) validating this natural program by having a builder build from it. Thus, rather than exhaustively studying each task to estimate its difficulty, we are content with just getting a "good enough" natural program for each task. In another words, given a certain annotation budget, we want to find a single good natural program for each of the 400 tasks.

If we take the 400 tasks as 400 casinos, then each casino would have an intrinsic difficulty, which corresponds to how easy it is to communicate a particular task. Within each task, there are an infinitely many possible natural programs (i.e. all natural language strings), which correspond to the infinite-arm aspect. For each task, we are interested in finding as good of a description as we can, which correspond to the best-arm identification aspect.

Specifically, we are seeking an online algorithm that *at any budget* can propose a set of natural programs, and this set of proposed programs should improve with added budget (budget here is synonymous with total participants' time). To use the bandit algorithm in conjunction with the annotation process, we divide the 45 minutes of a participant's time into several "units" of participation, where each unit can be assigned to one of two jobs: 1) The participant can either give a new description to an ARC task, then immediately build from it (in the form of describer verification) or 2) The participant can be given an existing description of a task, and build from it to to assess if it is a good description. See Figure 14. We estimate how many minutes would this particular unit take, and dynamically allocate additional units until the full 45 minutes are exhausted.

B.2 Reinforcement Learning Formulation

A great way to formalize a bandit problem is casting it as an instance of a Markov Decision Process:

A **state** consists of all the observations (the 0, 1 outcomes) on all the arms thus far. Let there be N bandits/casinos, then the observation is a collection of all casinos' outcomes $C_1 \dots C_N$ where for each casino C_i, we have observation for its K arms that we already sampled: $c_i^1 \dots c_i^K$. Each arm's observation, c_i^j is simply a tuple (A, B) where A denotes the number of 1s observed from arm c_i^j and B denotes the number of 0s. Thus, the space of observation is $O(N \times K \times (A + B))$. See Figure 15.

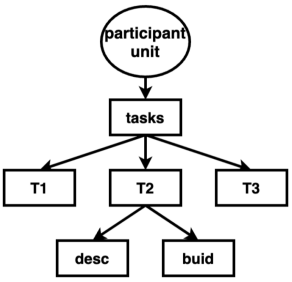

Figure 14: How a "unit" of a participant's time can be utilized

There are two kinds of **actions** – the *arm-selection* action, and the *best-arm-proposal* action. Arm selection consists of a tuple (i, j) where i selects a casino, and j selects from which of the arms within that casino to sample an additional observation. We will use $j = 1 \dots K$ to denote sampling from the K arms within a particular bandit i, and use $j = 0$ to denote sampling a *new* arm from bandit i. When the interaction budget is exhausted, the agent must make a best-arm-proposal action, in which the agent picks one sampled arm from each casino to be calculated in the regret. For arm proposal, we use a simple heuristic that selects the arm with the highest estimated mean using a beta distribution with (1,1) prior. For the remainder of this section, **action** will refer exclusively to arm-selection.

Transition modifies the **state** to include the new observation. See Figure 15.

Reward is the sum of the Bernoulli parameters for the set of proposed arms. $p_1 + \cdots + p_N$.

```
original state:
[(2,0), (1,1), (0,1)]   # casino1, has 3 arms, 3rd arm has 0 success and 1 fail
[(1,1), (1,0)]          # casino2, has 2 arms, first arm has 1 success and 1 fail
[ ]                     # casino3, has 0 arms so-far

taking the 2 following actions:
action (3,0) observed a 0, and action (2,1) observed a 1

resulting state after those 2 actions:
[(2,0), (1,1), (0,1)]   # casino1, has 3 arms, 3rd arm has 0 success and 1 fail
[(2,1), (1,0)]          # casino2, has 2 arms, first arm has 2 success and 1 fail
[(0,1)]                 # casino3, has 1 arm, first arm has 0 success and 1 fail
```

Figure 15: an example transition where there are 3 casinos

B.3 A Heuristically Defined Agent

To the best of our knowledge, there is no bandit algorithm that address the specific bandit problem we are solving. However [50] solves the infinitely many armed bandit problem for a single bandit, where they explicitly model the difficulty of the underlying bandit. We take their algorithm as inspiration. Note that [50] prescribe a solution to the regret-minimization problem, which is not exactly best-arm-identification. However, in the limit, the two are equivalent as minimizing regret is

equivalent to finding the optimal arm. We will first state the result of [50], which applies to the case of a single casino/bandit, then extend it to the case of multi-bandit.

arm selection Suppose we know that we want to generate an action in casino i. [50] proposed the following rule for selecting which arm to interact with. Let β be the difficulty parameter of the task, defined as: $P(p^* - p_j < \epsilon) = \Theta(\epsilon^\beta)$. Which is to say, if you were to sample a new arm with ground truth parameter p_j, the probability that this arm lies within ϵ of the optimal arm, is approximately ϵ^β. For instance, if $\beta = 1$, the task is very difficult as ϵ^1 is a tiny number, meaning it is almost impossible for you to sample an arm p_j that is ϵ close to optimum. Conversely, if $\beta = 0$, the task is very simple, as $\epsilon^0 = 1$, so any arm you sample will be optimal.

[50] states that, if you let M be the total number of observations on a bandit, and K be the total number of arms currently sampled, if $K \leq M^\beta$, then you should sample a new arm. Otherwise, you should perform the standard UCB algorithm on the set of existing arms. In our bandit RL environment, M and K are well defined, but how do we estimate β? We use the following heuristic to estimate difficulty: Let j be the best arm in the current casino w.r.t. its sampled mean $\tilde{p_j}$, then we define $\beta = 1 - \tilde{p_j}$. For instance, if the best arm has a sampled mean of 0.9, then we are in an "easy" casino, and the difficulty will be $1 - 0.9 = 0.1$, which is fairly close to 0, implying we should *NOT* be sampling new arms, as the best arm we have currently is likely to be good. Conversely, if the best arm has a sampled mean of 0.1, then we are in a "difficult" casino, where we stand a better chance of finding a good arm by sampling more arms.

casino selection To adopt the infinitely-many arm algorithm to a multi-bandit setting, we use the following heuristic: selecting the casino where we have the least information about p^* of a casino. In practice, we rank all K arms based on their sampled mean, and take the top-half of the arms, and aggregate a beta distribution of the total number of 1s and 0s of these arms, and use the variance of the beta distribution as a proxy for uncertainty. For instance, if a casino whose top-half arms have in total many observations, and most of them are 1s, then we are certain about its p^*. Conversely, if a casino whose top-half arms have few observations, and it is an even split of 1s and 0s, we are unsure of its p^*.

B.4 Simulated Evaluation

With both arm selection and casino selection, we have a functioning agent. We can evaluate this agents' performance against several baseline agents in the bandit RL environment to verify that it is indeed more efficient. We consider the following baseline agents, **rand** is the random agent that select an action at random, **tile** is the agent that tries to evenly spread out the observation budget, **tile-inf** is the agent that uses the infinitely many arm algorithm, and tries to spread the budget evenly across casinos, **cas-inf(ours)** is the agent that selects the casino using uncertainty of p^*, and use infinitely many arm algorithm.

The algorithms performance over 100 casinos with a total of 600 interaction budgets is in Figure 16

As one can see, for the simulated environment, which makes several simplifications, such as not taking in the generation aspect of description making, and modeling difficulty of a casino as a truncated gaussian, our proposed bandit algorithm out-performs the other baselines. The implementation of the bandit environment and the bandit policies can be found at `LARC/bandit`

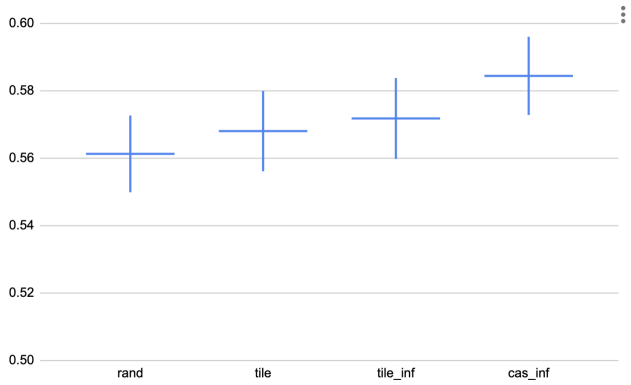

Figure 16: performance of various bandit policies, of 100 casinos and a budget of 600, averaged across 100 repetitions. horizontal bar is average, whiskers indicate standard deviation