Communicating Natural Programs to Humans and Machines

Samuel Acquaviva*  Yewen Pu*  Marta Kryven †  Theodoros Sechopoulos †
MIT  Autodesk Research  MIT  MIT

Catherine Wong †  Gabrielle E Ecanow  Maxwell Nye
MIT  MIT  MIT

Michael Henry Tessler  Joshua B. Tenenbaum
MIT  MIT

Abstract

The Abstraction and Reasoning Corpus (ARC) is a set of procedural tasks that tests an agent’s ability to flexibly solve novel problems. While most ARC tasks are easy for humans, they are challenging for state-of-the-art AI. What makes building intelligent systems that can generalize to novel situations such as ARC difficult? We posit that the answer might be found by studying the difference of language: While humans readily generate and interpret instructions in a general language, computer systems are shackled to a narrow domain-specific language that they can precisely execute. We present LARC, the Language-complete ARC: a collection of natural language descriptions by a group of human participants who instruct each other on how to solve ARC tasks using language alone, which contains successful instructions for 88% of the ARC tasks. We analyze the collected instructions as ‘natural programs’, finding that while they resemble computer programs, they are distinct in two ways: First, they contain a wide range of primitives; Second, they frequently leverage communicative strategies beyond directly executable codes. We demonstrate that these two distinctions prevent current program synthesis techniques from leveraging LARC to its full potential, and give concrete suggestions on how to build the next-generation program synthesizers.

1 Introduction

Humans solve a range of procedural tasks such as cooking, tying shoes, and programming. Although current AI systems achieve super-human proficiency at certain narrowly specified tasks [1,2], their reasoning is domain-specific and fails to generalize to novel situations [3]. The Abstraction and Reasoning Corpus (ARC) introduced by [4] presents a set of procedural tasks constructed expressly to benchmark fundamental capacities associated with human general intelligence, including abstraction, generalization, object categories, and procedural analogies [3][5][10]. Specifically, ARC requires one to infer a procedure consistent with a small number of abstract input-output examples and apply it to a new input to generate an unseen answer, see Figure[1].

How do we build systems that are capable of solving general, procedural tasks such as ARC? Traditional approaches of program synthesis [1][14] and semantic parsing [15][20] assume the tasks are DSL-closed – for any task, there exists a program, written in a predefined Domain Specific Language (DSL), that solves the task. The ARC benchmark is uniquely designed to be DSL-open

*and † denote equal contributions
Figure 1: Four ARC tasks, the goal is to correctly infer the unseen output from the given examples.

– it does not come with a predefined DSL capable of representing its tasks intuitively. This is both reasonable – most real life tasks, such as cooking and assembling furniture, are DSL-open – and challenging – how can one build an intelligent system that can solve tasks from few examples without a DSL? To illustrate, what might a DSL that would allow one to program all the ARC tasks in Figure 1 look like? This question is difficult to answer; a recent Kaggle competition found that the best AI systems solve at most 20% of the tasks, while [21] found that most humans easily solve over 80%.

Given that humans greatly outperform the best AI systems in solving ARC tasks, studying the human’s cognitive processes (for instance, which set of concepts do human use to represent these tasks?) can shed light on how to build similarly intelligent systems. As these thought processes are not observable directly, we study natural programs – instructions that humans give to each other, as a window into these latent cognitive processes. Like computer programs, these instructions can be reliably interpreted (by another human) to produce the intended output. Unlike computer programs, which must be stated in a specific style, natural programs can be stated in any form – such as verbal instructions or input-output examples – as long as another human can execute them. In this work, we study a particular form of natural programs, that of natural language instructions. We show that analyzing these natural programs – with explicit comparisons to computer programs – can both shed light on how humans communicate and interpret procedures [22, 25] and inform how one may build AI systems for challenging, DSL-open domains such as ARC.

Figure 2: Four LARC tasks, corresponding to those of Figure 1. The goal is to produce the correct output given only the language instructions. 88% of the ARC tasks can be communicated this way. What are some of the communicative strategies used by humans here?

We present the Language-complete Abstraction and Reasoning Corpus (LARC) of natural language instructions elicited from a two-player communication game, where 88% of the ARC tasks can be successfully communicated. LARC tasks are language-complete: The successful instructions contain all the relevant information, even in absence of the original input-output examples (see Figure 2). This is important in several ways: First, one can use LARC to study how humans use language to communicate abstract procedures, as humans clearly have the capacity to both generate and execute natural language instructions.

1Humans were evaluated on a subset of the training tasks; the Kaggle competition used a private test set.

https://github.com/samacqua/LARC
these natural programs; Second, one can directly see what concepts an intelligent system must be aware of (such as colors and numbers); Third, as people readily generate natural programs, studying them will provide insights on building interactive systems.

We perform linguistic analysis on LARC, finding that humans readily leverage algorithmic concepts without being explicitly instructed to do so. These concepts range from domain general ones, such as loops, to domain-specific concepts such as flood-fill. However, natural programs in LARC are distinct from typical computer programs in two ways: (1) natural programs use a much wider range of concepts compared to a typical DSL; (2) natural programs contain clarifications and validations in greater quantity than directly executable procedures. We apply standard program synthesis algorithms on LARC, finding that while existing approaches can benefit from the additional language annotations, the two aforementioned distinctions pose significant challenges to standard program synthesis approaches. We conclude by providing concrete suggestions on how to build the next generation program synthesizers.

2 Communicating and Interpreting Programs

In programming, a programmer constructs a program in a suitable language, which is then executed on an interpreter, producing a behaviour. For instance, a person can instruct another person to carry out a certain task (Fig. 3 top-left), or directly program a machine to solve tasks using code (Fig. 3 top-right). A program synthesizer takes in an instruction, and reformulates it as code, insulating the person from the programming process (Fig. 3 bot). We treat all three as acts of programming.

How do we build systems that can be communicated naturally to solve challenging tasks? Typically, one follows a “DSL-first” approach, where one first defines a programming language and builds a corresponding interpreter capable of executing programs written in this language. Then, one naturalizes the initial DSL using synthesis, allowing end-users to describe tasks using natural language [15–18, 26, 27], or by giving examples [12, 13, 28]. While this DSL-first workflow has yielded impressive results, the DSL itself is also a single point of failure. It is difficult to design DSL with the right scope, so that it both expressive and non-redundant [29–31]. One must ensure that the DSL aligns reasonably to human instructions [32, 33], while simultaneously being efficient when used by the synthesizer [12, 34]. These challenges may explain why ARC, and other DSL-open domains (where procedural tasks are given in the absence of a narrow DSL), are difficult to tackle.

In this work, we adopt the Wizard-of-Oz approach [35–37] by using a human as an interpreter of natural language instructions (Fig 3 top-left). We define a natural program as instructions constructed by a person that can be interpreted by another person to produce a specific output. This program is natural—it can be understood by speakers of the language without a prior consensus—but behaves as a program, in that it produces a definitive output, which can be unambiguously checked for correctness. For instance, the original ARC [4] tasks are natural programs: Given a program consisting of input-output examples, a fellow human can readily interpret this program to produce an

---

4language here is to be understood loosely as any medium of communication between people
output on a new input, which can be checked for correctness. By starting with (linguistic) natural
programs, one can directly observe the set of concepts and strategies necessary to master a domain
(such as ARC), without committing to a specific interpreter.

![Figure 4](image)

**Figure 4:** a describer instructs a builder how to solve an ARC task using a natural program

3 LARC: Language-complete Abstraction and Reasoning Corpus

We present a dataset that augments the original ARC tasks from [4] with language-complete instruc-
tions: they can be demonstrably interpreted by other humans to correctly produce the intended outputs
without any additional contexts (i.e. in the absence of the original input-output examples). Thus,
LARC tasks (Fig. 2), like their counterparts in ARC, meet the definition of natural program while
containing only natural language descriptions. To collect this dataset, we introduce a communication
game: human describers produce linguistic instructions from the given input-output examples of ARC,
these instructions are then interpreted by human builders (in the absence of the original input-output)
on a new instance of the same task (Fig. 4). We deployed this experiment using a novel bandit
algorithm to efficiently collect verifiable natural programs. The final dataset augments 88% of the
original ARC tasks (354/400) with at least one verifiable natural program description that could
be successfully interpreted by another human participant to solve the task. Fig. 5(C-D) shows the
distribution of success rates for participants acting as describers and builders over time.

3.1 Human annotation details

We recruited 373 subjects via Amazon Mechanical Turk who were paid for 45 minutes of work. Fifty
individuals were excluded for failing to complete the task, so the final analysis included 323 subjects.
The study was approved by our institution’s Institutional Review Board, did not collect personally
identifiable information, and did not pose risks to participants. Subjects were paid $6.00 and a $0.25
bonus for every successful communication. Subjects averaged 5.5 communications, bringing their
expected hourly wage to $9.83. For interface and consent form see Appendix A.2.

3.2 Two-player communication game

For each task, a participant may be assigned one of two roles: a describer or a builder. The describer
plays the role of a human synthesizer, who reformulates input-output examples (of ARC) to natural
language descriptions. The builder plays a role of a human interpreter, who must construct the
correct output on a new input without access to the original examples (Fig 4). The description is
structured into three sections to incentivize consistency: (1) what the builder should expect to see
in the input, (2) the output grid size, and (3) what the builder should do to create the output (Fig 2).
After the description was submitted, we verify the describer’s own understanding by asking them to
build it, and discarding the submission if the describer fails. The describer was shown all previous
verified descriptions for a task, alleviating challenge of solving the task from scratch. Builders
construct/draw the output using actions defined in ARC, such as paint(color,x,y), copy/paste,
and floodfill. All drawing sequences are recorded and can be played back.

3.3 The Bandit Algorithm for Data Collection

Collecting valid linguistic natural programs requires significant human efforts: For each task (of
varying difficulties), natural programs must first be proposed by a number of describers, and then
validated by a number of builders, where both can make mistakes. Thus, A naive data-collection
process that simply collects a fixed number of descriptions and builds per task will be expensive. To
Figure 5:  A. Describer improves at verifying their own descriptions as they describe more tasks. B. Builders do not improve at constructing the correct outputs as they build more tasks (likely due to having no control over the qualities of their given descriptions). C. Rate of describers verifying their own descriptions (avg 75%). D. The rate of builders constructing the correct output, (avg 50%).

Figure 6: Words used in successfully built descriptions, sorted by their frequency in the corpus (total 642 unique words). The words were singularized. Colors names, numbers, and pronouns were grouped together.

address this challenge, we formulate the following bandit problem: multi-bandit – each of the 400 ARC tasks is a different bandit; infinite-arm – given a task, each natural language description (there are infinitely many) is a different arm; best-arm identification – once a natural program is proposed, we must validate it. We develop a novel bandit algorithm (Appendix B) to solve this problem, as to our best knowledge, no known bandit algorithm can be directly applied. For each MTurk participant, our bandit algorithm dynamically allocates a set of describing and building efforts for their session. As a result, the LARC dataset was annotated for $3667, whereas a naively collecting 20 annotations per task would cost at least $10,800.

4 Communication Strategies in Natural Programs

What are some strategies humans use to produce robustly interpretable instructions? To answer this question, we curate a linguistically tagged dataset of tagged phrases from successful descriptions under the lens of computer programs. We annotate these phrases with tags corresponding to general concepts from algorithms and core knowledge [38]. In total, we manually label 532 randomly sampled phrases (22% of the phrase corpus) using 17 conceptual tags (in which multiple tags can be applied to each phrase); Figure 7A. shows a frequency of these tags. For details see Appendix A.3

4.1 Similarities of Computer and Natural Programs

General Algorithmic Concepts LARC contains algorithmic concepts similar to those found in a typical programming language (i.e. python). For instance, tag_logic is a boolean check (i.e. “the box is blue”), tag_array references a set of similar objects (i.e. “you should see four red shapes”), and tag_loop is similar to loops (“keep going until ”). Humans generate these concepts without being directly instructed to do so, suggesting that humans reason about ARC tasks algorithmically.

Domain Specific Concepts Similar to a computer DSL, LARC contains concepts that distinguish it from other domains. We focus on the object system of core knowledge [38], defined by cohesion, persistence, and influence via contact, which the ARC corpus was explicitly designed to leverage. We find about half of the phrases referenced objects, and three quarters of these described spatial relations
4.2 Differences of Computer and Natural Programs

We outline two (related) ways natural programs differ from computer programs. First, instead of using a narrow DSL with few primitives, natural programs use a large, diverse set of primitive functions. Second, instead of stating a precise procedure verbatim, natural programs rely on a range of additional strategies to ensure that they can be interpreted precisely.

Natural Programs Invoke a Large Number of Concepts Since LARC is language-complete, analyzing the words used in LARC serves as a good proxy for the underlying concepts present in the ARC domain. Similar to [21], we find that humans use a wide range of concepts (Fig 6). This is a testament of the general capabilities of the human interpreter: the describers readily invoke these concepts from the builders, with the confidence that they can be correctly interpreted. Given a large number of concepts, effectively indicating the set of relevant concepts (for a given task) becomes nontrivial: While human describers and builders can make use of generic word such as ‘bump into’, computer programmers must be extremely careful in selecting the exact concept using a precise language (i.e. move_until_touches_block).

Natural Programs Communicate Information Beyond Procedures We study the relative frequencies of directly executable commands, tag_procedure, in contrast to not directly executable meta information such as tag_framing – comments about which concepts are relevant, tag_validation – checks to ensure correct execution, and tag_clarifications – restating the same procedure in different words. The most striking finding is that procedure, framing, and validation occur at roughly the same frequency (see Fig 7C). In contrast, only 14% of the codes are commented [39].

The high frequency of framing tags suggests that describers anticipate the large number of concepts that the builder can operate over, and carefully frame the instruction to invoke the appropriate ones. The describer often assumes the directly executable portion (i.e. tag_procedure) as inherently ambiguous, as suggested by frequent use of tag_validations and tag_clarifications following these procedures. Specifically, validation gives a check to the builder to test if their current interpretation is correct. Clarification amends the initial ambiguous explanation with another explanation, narrowing the number of possible interpretations. These are evidences that, unlike communication in computer programs over a narrow and unambiguous DSL, communication in natural programs are fundamentally expressive yet ambiguous, requiring extra efforts to maintain precision.
5 Executing Natural Programs using Program Synthesis

We evaluate whether current DSL-first program synthesis methods (Fig 3, bot) can execute natural programs as well as humans do. We consider three kinds of natural programs: (1) Input-output examples from the original ARC corpus (IO); (2) IO in conjunction with successful language instructions in LARC (IO+NL); And (3) language alone (NL-only) – same as the MTurk builder task.

5.1 Program Synthesis

In (symbolic) program synthesis [14, 19, 40], the synthesizer takes in a natural program, and reformulates it as code over a DSL to be executed. We have manually crafted a DSL based loosely on the concepts present in the LARC corpus and built its corresponding interpreter (see Appendix A.4). We present our best synthesis results here. For additional models (using a CNN encoder, a sequence decoder [19]) see A.5. Preliminary studies with codex and clip see A.6 and A.7.

Generate and Check Using IO

If the given natural program contains IO examples, the standard symbolic program synthesis approach [13,14] follows the generate and check strategy. Let natprog be a natural program, the synthesizer returns programs prog from a DSL from the following distribution:

\[ P_{\text{synth}}(\text{prog}|\text{natprog}) \propto P_{\text{gen}}(\text{prog}|\text{natprog}) \mathbb{1}[\text{prog} \vdash \text{IO}] \]

\( P_{\text{gen}} \) is the generative distribution: given a natural program, it proposes program prog from the DSL. \( \mathbb{1}[\text{prog} \vdash \text{IO}] \) is the checker: it validates prog by executing it on the interpreter, ensuring that \( \text{prog}(x) = y \) for all input-output pairs \((x, y) \in \text{IO}\). The key strength of this approach lies in its generalizability: If a proposed program can be checked against all IO examples, it is very likely to generalize to an new instance of the same task due to the inductive bias of the DSL.

Generation Models

Our \( P_{\text{gen}}(\text{prog}|\text{natprog}) \) generates programs in two parts: a neural model outputs a tree bigram over the grammar of the DSL [41], then a dedicated Ocaml enumerator deterministically enumerates programs from a probabilistic context free grammar fitted to this bigram distribution in decreasing probability [34]. For simplicity, we report results of unconditioned generators \( P_{\text{gen}}(\text{prog}) \) (i.e. a fitted prior) when language is absent, and language-conditioned models \( P_{\text{gen}}(\text{prog}|\text{NL}) \) when language is present. This way, we can use the same \( P_{\text{gen}}(\text{prog}|\text{NL}) \) model for both IO+NL and NL-only tasks in the test set, as it does not depend on IO. Similar to [42, 43], we first bootstrap our generative models with 10 “seed” programs, discovered uninformed enumeration.

Leveraging Language

We use a pre-trained model (T5, [44]) to represent language by taking an average of its encoded tokens. To encourage the learning of compositional relationships between language and program, we use pseudo-annotation, similar to recent methods that have leveraged synchronous grammars [18,33,43,45]. First, we provide linguistic comments for each primitive function in the program DSL (e.g. \texttt{flood_fill(color)} with \texttt{fill with the color}). Then, during training, we obtain additional paired language and program examples by substituting primitives of artificial programs with their corresponding comments. For more examples see Appendix A.4.

Distant Supervision

LARC, similar to SCONE [46], falls under the challenge of distant supervision: each training task only contains the correct output, but not the ground-truth program responsible for generating it. We adopt the iterative approach used in [19,44,42,43] to discover suitable programs during the training phase, by alternatively (1) generating a large sample of programs using \( P_{\text{gen}} \) and (2) fitting a better \( P_{\text{gen}} \) from good programs in the generated samples.

5.2 Results

We split the 400 tasks into 200 training tasks (with or without valid language descriptions) and 183 testing tasks (the remaining 200 filtered for having valid language deceptions). We then train the models for 10 hours each using iterative learning. We test on the 183 test tasks by first using the neural model to propose a bigram per task, then enumerating the bigram for 720 seconds. We keep this a tremendous engineering effort, consisting of 103 primitives compared to 33 of SCONE.

\begin{example}
\begin{verbatim}
(lambda (to_original_grid_overlay (remove_color(grid_to_block x) yellow) false)) becomes place block on input grid remove color from block yellow
\end{verbatim}
\end{example}
<table>
<thead>
<tr>
<th>training tasks discovered</th>
<th>no-pseudo</th>
<th>pseudo</th>
</tr>
</thead>
<tbody>
<tr>
<td>IO</td>
<td>15 / 200</td>
<td>-</td>
</tr>
<tr>
<td>IO + NL</td>
<td>13 / 200</td>
<td>21 / 200</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>testing tasks solved</th>
<th>no-pseudo</th>
<th>pseudo</th>
</tr>
</thead>
<tbody>
<tr>
<td>NL-only</td>
<td>1 / 183</td>
<td>0 / 183</td>
</tr>
<tr>
<td>IO</td>
<td>18 / 183</td>
<td>-</td>
</tr>
<tr>
<td>IO + NL</td>
<td>16 / 183</td>
<td>22 / 183</td>
</tr>
</tbody>
</table>

Table 1: Executing different kinds of natural programs (IO – Input-output examples from the original ARC corpus, IO+NL – IO in conjunction with successful language instructions in LARC, NL-only – same as the MTurk builder task) using program synthesis. Here, “pseudo” means the NL training has been pre-trained on generated synthetic language to code pairs. Train tasks discovered under distant supervision (top). Test tasks solved (bot).

Figure 8: Number of test tasks solved for the three kinds of natural programs, IO, NL, IO+NL, with and without pseudo annotations, as a function of enumeration time. There are error bars as the bigram enumerator is deterministic. It is possible (but not likely) that re-training these models will have an effect due to the randomness of sampling pseudo-annotated programs. All models vastly under performs when compared to a human, but natural programs consisting of NL+IO fairs best.

Quantitative Findings  IO+NL+pseudo performs best, solving 22/183 of the testing tasks. We believe this due to pseudo-annotation being able to generate an infinite number (albeit low quality) of artificial NL-prog pairs. We note that having the ability to check if a proposed program is correct under IO is crucial for the success of current program synthesizers, with no more than 1 task solved with NL-only. Like the validation phrases in LARC, the input-output examples in IO serve as a form of validation for the enumerative synthesizer. This finding corroborates with [47].

Qualitative Findings We investigate in what way does language affect synthesis. For each primitive in our DSL, we ask how many times more likely is it going to appear in correct programs generated with the language-conditioned bigram vs the unconditioned one. We plot this ratio on a log scale for all primitives that were used in ground-truth programs, see Figure 9. We note that for most of the frequently used primitives, the language-conditioned generator is more likely to generate the correct primitives than the unconditioned generator.

5.3 Challenges

The biggest challenge is scoping. Since LARC is DSL-open, we were in a vicious cycle of constantly adding more primitives and refactoring the DSL. Even now, we cannot guarantee our DSL can represent all LARC tasks. Second challenge is referencing: with 103 primitives, selecting the relevant primitives becomes crucial. Finally, current NL-to-code approaches – like the ones we used – assume a close, 1-to-1 paraphrase-like mapping between language and procedure, which misinterpret crucial framing and validation statements that occurs in abundance in LARC.

---

7 if we can magically select 10, the search space is $10^5$ instead of $103^5$ for a program of length 5.
Figure 9: Relative odds of using the correct primitive for a task, language-conditioned vs unconditioned generation. Number in parenthesis denotes the total number of times a primitive is used.

6 Related Works

Task oriented dialogue systems LARC as a dataset belongs to the family of task oriented dialogue systems [35–37, 48, 49]. One can view natural programs in LARC as a single-turn, task-oriented dialogue, where the describer gives a naturalistic instruction with a specific, check-able task in mind. Further, LARC uses the Wizard-of-Oz style of data collection – leveraging a human interpreter without committing to building a working system – a common framework to collect data in dialogue systems. LARC differs from these existing datasets mainly in the diversity of its tasks (Section 4) which contain a wide range of abstract concepts rather than being limited to specific domains such as database manipulations [37][49].

Embodied instruction following Embodied instruction following consisting of an embodied agent (often a avatar in a video game) being able to carry out a sequence of commands when prompted with natural language instructions [27][50–53]. These commands can often be hierarchical [17][50][53], which are naturally represented as programs. LARC again differs from these works due to the range of abstract concepts, whereas aforementioned works typically follows a DSL-closed assumption.

As a result of a narrower range of concepts, a paraphrasal strategy that simply translate natural language into code has been fairly successful in prior works that aim to build an instruction following system [27][49][51]. LARC gives strong evidence that additional grounding strategies need to be modeled to truly capture the richness of natural language instructions (for instance, consider the set of strategies used in Fig 2).

7 Conclusion and Future Works

We present LARC, a DSL-open yet Language-complete dataset, highlighting the difference of between human-to-human and human-to-machines communications. By annotating successful communications (dataset of linguistically-tagged-phrases), we find that humans communicate using a wide range of concepts and communicative strategies, which are difficult to interpret using existing techniques. We hope LARC can help different communities (AI, Programming Language, Cognitive Science, etc) understand and build intelligent, communicative systems. Specifically, we believe that defining concepts upfront (DSL-first) is not scalable. Instead, they should be learned and taught (by end-users). To fully harness the power of natural language, we need to look beyond the simplistic notion that language having a 1-1 relationship with direct execution, and entertain different communicative strategies [54]. We believe datasets [51][55][56] that share the properties – namely, DSL-open and language-complete – are crucial to bridging the gaps between human-human and human-machine communications. Lastly, it will be beneficial to adapt foundational models [57][59] – with some conventional understandings of language, vision, and code – towards specific domains.
Limitations and Potential Negative Impacts  LARC consists of a single, constrained task format in a highly controlled setting. The long-term goal of this work is to ‘reverse-engineer’ how humans think and communicate, and such systems raise concerns regarding value alignments of users, for instance, non-experts operating safety-critical equipment using natural language.

References


**Checklist**

1. For all authors...
   (a) Do the main claims made in the abstract and introduction accurately reflect the paper’s contributions and scope? [Yes]
   (b) Did you describe the limitations of your work? [Yes]
   (c) Did you discuss any potential negative societal impacts of your work? [Yes]
   (d) Have you read the ethics review guidelines and ensured that your paper conforms to them? [Yes]

2. If you are including theoretical results...
   (a) Did you state the full set of assumptions of all theoretical results? [N/A]
   (b) Did you include complete proofs of all theoretical results? [N/A]

3. If you ran experiments (e.g. for benchmarks)...
   (a) Did you include the code, data, and instructions needed to reproduce the main experimental results (either in the supplemental material or as a URL)? [Yes] See Supplement
(b) Did you specify all the training details (e.g., data splits, hyperparameters, how they were chosen)? [Yes] See Supplement

(c) Did you report error bars (e.g., with respect to the random seed after running experiments multiple times)? [No] Each synthesis study takes 30 hours and is expensive, and we are not expressly claiming results of the form “our approach is good” but only providing suggestions on what may/maynot work

(d) Did you include the total amount of compute and the type of resources used (e.g., type of GPUs, internal cluster, or cloud provider)? [Yes] See Supplement

4. If you are using existing assets (e.g., code, data, models) or curating/releasing new assets...

(a) If your work uses existing assets, did you cite the creators? [Yes] ARC [4]

(b) Did you mention the license of the assets? [Yes] See Supplement

(c) Did you include any new assets either in the supplemental material or as a URL? [Yes] See Supplement

(d) Did you discuss whether and how consent was obtained from people whose data you’re using/curating? [Yes] See Supplement

(e) Did you discuss whether the data you are using/curating contains personally identifiable information or offensive content? [Yes] See Supplement. We do not collect personally identifiable or sensitive information

5. If you used crowdsourcing or conducted research with human subjects...

(a) Did you include the full text of instructions given to participants and screenshots, if applicable? [Yes] See Supplement

(b) Did you describe any potential participant risks, with links to Institutional Review Board (IRB) approvals, if applicable? [Yes] See Supplement

(c) Did you include the estimated hourly wage paid to participants and the total amount spent on participant compensation? [Yes] See Supplement

Acknowledgements

The authors would like to thank Eric Lu for inspiring the wonderful communication game that catalyzed our work.
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

![The LARC Explorer GUI](image)

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.

![Diagram](image_url)

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 (-). The following table details the tags and coding scheme used:

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

*marked by 1 and 0 respectively in the csv
<table>
<thead>
<tr>
<th>Tag</th>
<th>Description</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spacial Relation</td>
<td>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.</td>
<td>(+) The red shape next to the blue shape</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(+) Put yellow inside the green</td>
</tr>
<tr>
<td>Physical Interaction</td>
<td>Any reference to an imaginary force.</td>
<td>(+) The red object falls</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(+) Blue slides to the left towards red</td>
</tr>
<tr>
<td>Contact Transform</td>
<td>Influence via contact, i.e. any specialized version of physical interaction that involves at least two objects and some type of contact causality.</td>
<td>(+) Move X until contact with Y</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(+) Set X touching Y and turn it the color of Y</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(-) Red moves left one square</td>
</tr>
<tr>
<td>Affine Transform</td>
<td>Any reference to an affine transformation over an object, such as rotation, translation, etc.</td>
<td>(+) Rotate 90 degrees</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(+) Extend the square into a line</td>
</tr>
<tr>
<td>Visual-Graphical Transform</td>
<td>Any other visual or graphical modification other than a geometric one, such as coloring, flood-fill, or drawing a new shape.</td>
<td>(+) Make it gray</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(+) Draw a line</td>
</tr>
<tr>
<td>Object Detection</td>
<td>The localization of a cohesive, bounded object.</td>
<td>(+) The red shape</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(+) Move it to the left</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(+) The pattern</td>
</tr>
</tbody>
</table>

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.

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

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
'take logical operation on colors using them as true and false'
color_logical

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
Example Sampled Programs and Pseudoannotations

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

Compared to SCONE [46], LARC poses a significantly greater challenge for distant supervision.

<table>
<thead>
<tr>
<th></th>
<th>domain</th>
<th>dsl size</th>
<th>language kind</th>
<th>number of instances</th>
</tr>
</thead>
<tbody>
<tr>
<td>LARC</td>
<td>DSL-open</td>
<td>103</td>
<td>freeform text</td>
<td>354</td>
</tr>
<tr>
<td>SCONE: ALCHEMY</td>
<td>DSL-closed</td>
<td>24</td>
<td>step-by-step instruction</td>
<td>4560</td>
</tr>
<tr>
<td>SCONE: TANGRAMS</td>
<td>DSL-closed</td>
<td>14</td>
<td>step-by-step instruction</td>
<td>4989</td>
</tr>
<tr>
<td>SCONE: SCENE</td>
<td>DSL-closed</td>
<td>33</td>
<td>step-by-step instruction</td>
<td>4402</td>
</tr>
</tbody>
</table>

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 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.

<table>
<thead>
<tr>
<th></th>
<th>training tasks discovered</th>
<th>testing tasks solved</th>
</tr>
</thead>
<tbody>
<tr>
<td>IO</td>
<td>6 / 200</td>
<td>2 / 183</td>
</tr>
<tr>
<td>IO + NL</td>
<td>7 / 200</td>
<td>0 / 183</td>
</tr>
<tr>
<td>NL</td>
<td>-</td>
<td>0 / 183</td>
</tr>
</tbody>
</table>

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 an 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.

<table>
<thead>
<tr>
<th>tasks</th>
<th>NL</th>
<th>correct program</th>
<th>example copilot program</th>
<th># of syntactically valid out of 10 samples</th>
<th># of correct</th>
</tr>
</thead>
<tbody>
<tr>
<td>0520667.json</td>
<td>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.</td>
<td>(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))))</td>
<td>2</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0948d64.json</td>
<td>copy the pattern that is a different color</td>
<td>(lambda (to_min_grid (remove_color (remove_block $b (grid_to_block $0)) (nth primary_color (grid_to_block $0) $0) $0) false))</td>
<td>(lambda (to_original_grid_overlay (grid_to_block $0) (has_color (grid_to_block $0) pink)))</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>185875f.json</td>
<td>Copy only the biggest shape into the output grid</td>
<td>(lambda (to_min_grid (first_of_sorted_object_list (find_same_color_blocks $0 false false) (lambda (get_num_ties $0))) (false false)))</td>
<td>lambda (copy_shape_to_output_grid (first_of_sorted_object_list (find_same_color_blocks $0 false false) (lambda (true true)))</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>235c65d.json</td>
<td>change the grid size with the colored object with the smallest area and fill the same color</td>
<td>(lambda (to_min_grid (first_of_sorted_object_list (find_same_color_blocks $0 true false false) (lambda (get_num_ties $0))) (false false))</td>
<td>(lambda (to_original_grid_overlay (fill_color (grid_to_block $0) (grid_to_block $0) false)))</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>5582d5ca.json</td>
<td>determine which color is used for the majority of the squares in the input grid, then use that color for the entire output grid.</td>
<td>(lambda (to_original_grid_overlay (fill_color (merge_blocks (find_blocks_by_inherited $b $0 true false true) (nth primary_color (grid_to_block $0) $0))) $0) true))</td>
<td>(lambda (to_original_grid_overlay (find_majority_color $0) (lambda (lambda (true))))</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>689c79e.json</td>
<td>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</td>
<td>(lambda (to_original_grid_overlay (fill_color (filter:tile:blocks (grid_to_block $0) (lambda (is:exerior $0 false))) $false)))</td>
<td>lambda (to_original_grid_overlay (map:tile:blocks (grid_to_block $0) (lambda (block:to:tile (set:color (reflet:grid (grid_to_block $1) true) light:blue)) $false)))</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>72ca375d.json</td>
<td>copy one of the figures and just that figure.</td>
<td>(lambda (to_min_grid (singleton:blocks (filter:blocks (find_same_color_blocks $0 true false) (lambda (is:symmetrical $0 false))) false)))</td>
<td>(lambda (to_original_grid_overlay (copy:grid $0) (lambda (lambda (has_color (fill_color (grid_to_block $0) block)))</td>
<td>0</td>
<td>-</td>
</tr>
</tbody>
</table>

Figure 12: synthesizing programs using co-pilot (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 [60], 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 [58], or DALLE [59]) 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].

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.

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

Imagine there are N different magic 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 $p_1 \ldots p_N$ 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 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.

Figure 14: How a “unit” of a participant’s time can be utilized
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 $C_1 \ldots C_N$ where for each casino $C_i$, we have observation for its $K$ arms that we already sampled: $c^j_i$. Each arm’s observation, $c^j_i$ is simply a tuple $(A, B)$ where $A$ denotes the number of 1s observed from arm $c^j_i$ and $B$ denotes the number of 0s. Thus, the space of observation is $O(N \times K \times (A + B))$. See Figure 15.

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 $j = 1 \ldots K$ 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
- $[(0,1)]$ # casino3, has 1 arm, 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 [61] 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 [61] 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 [61], 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 $i$. [61] proposed the following rule for selecting which arm to interact with. Let $\beta$ 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 $\epsilon$ of the optimal arm, is approximately $\epsilon^{\beta}$. For instance, if $\beta = 1$, the task is very difficult as $\epsilon^1$ is a tiny number, meaning it is almost impossible
for you to sample an arm $p_j$ that is $\epsilon$ close to optimum. Conversely, if $\beta = 0$, the task is very simple, as $\epsilon^0 = 1$, so any arm you sample will be optimal.

[61] 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 $K \leq M^\alpha$, 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 $\beta$? 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]

![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](image)

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