Communicating Natural Programs to Humans and Machines

Samuel Acquaviva MIT
Yewen Pu Autodesk Research
Marta Kryven* MIT
Catherine Wong* MIT
Gabrielle E Ecanow † MIT
Maxwell Nye † MIT
Theodoros Sechopoulos † MIT
Michael Henry Tessler MIT
Joshua B. Tenenbaum MIT

Abstract

The Abstraction and Reasoning Corpus (ARC) is a set of 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. How do we build intelligent systems that can generalize to novel situations and understand human instructions in domains such as ARC? We posit that the answer may be found by studying how humans communicate to each other in solving these tasks. We present LARC, the *Language-annotated ARC*: a collection of natural language descriptions by a group of human participants, unfamiliar both with ARC and with each other, who instruct each other on how to solve ARC tasks. LARC contains successful instructions for 88% of the ARC tasks. We analyze the collected instructions as 'natural programs', finding that most natural program concepts have analogies in typical computer programs. However, unlike how one precisely programs a computer, we find that humans both anticipate and exploit ambiguities to communicate effectively. We demonstrate that a state-of-the-art program synthesis technique, which leverages the additional language annotations, outperforms its language-free counterpart.

1 Introduction

A long-term goal of Artificial Intelligence (AI) is to build agents that can flexibly solve new problems, understand human instructions, and communicate with human collaborators. Although current AI systems achieve super-human proficiency at certain narrowly specified tasks [1,2], their reasoning is often highly domain-specific, and fails to generalize, explain, or adapt flexibly to novel and out-of-domain situations [3]. The *Abstraction and Reasoning Corpus* (ARC) introduced by [4] presents a set of tasks constructed expressly to benchmark fundamental capacities associated with human intelligence, including abstract reasoning and generalization [4]. The ARC tasks are presented *inductively*: solving them entails inferring a pattern consistent with a small number of abstract input-output examples and applying it to a new input to generate an unseen answer. Cognitive research suggests that humans leverage a rich array of general reasoning strategies on tasks like these, including prior knowledge about object categories; the ability to infer structured rules from small numbers of examples; and the explanatory capacity to communicate these rules to themselves and to others [3,5–10]. We focus on ARC because it is demonstrably challenging for machines: results

*and † denote equal contributions

Preprint. Under review.

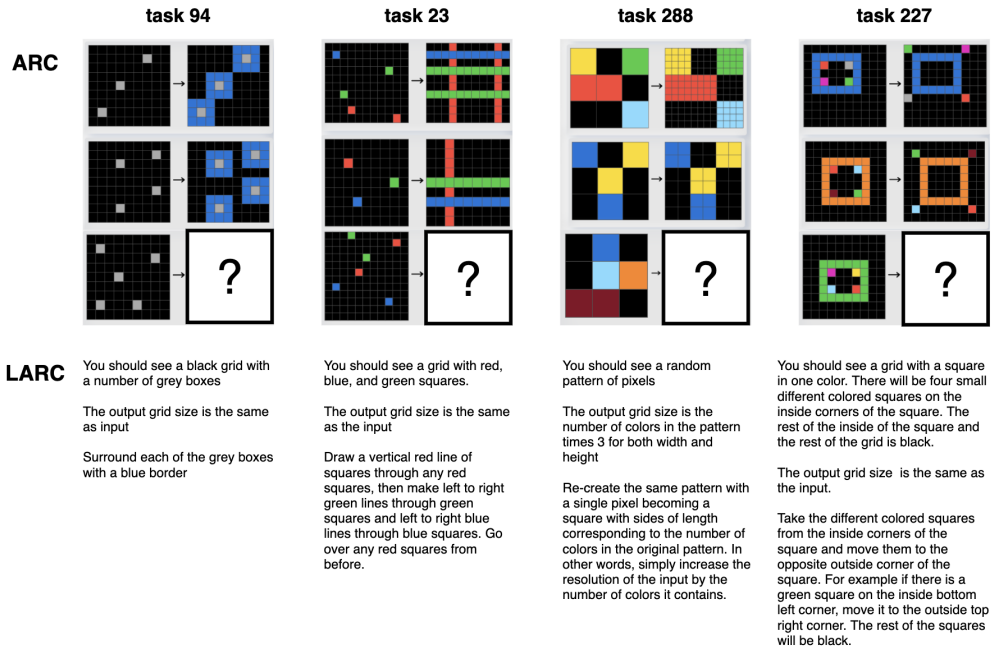

Figure 1: Tasks from the ARC dataset (above) and the corresponding natural programs from LARC (below). Given input-output examples (2 shown here per task), the ARC task is to extrapolate the output on a new input. In LARC, a **describer** describes the rule present in the examples so that a different person, a **builder**, can generate the correct output using the description alone.

from a recent Kaggle competition [2] find that the best AI systems solve at most 20% of the tasks, while a study benchmarking human performance found that most humans easily solve over 80% [11]. [3]

How can we build intelligent systems that achieve human-level performance on challenging and structured domains (like ARC), with or without additional human guidance? The internal mental representations humans use to solve a given task are not directly observable; instead, we look to *natural programs* – instructions that humans give to each other – as a window into the cognitive representations that inform downstream behavior. Much like computer programs, these instructions contain expressions that can be reliably interpreted and 'executed' (by another human) to produce intended outputs. Unlike computer programs, which must be stated in a specific style, natural programs can be stated in any form – verbal instructions, hand gestures, doodles – as long as another human can successfully execute them. In this work, we study a particular form of natural programs, that of *natural language instructions*, building on a long tradition in cognitive, linguistic, and computational sciences that seeks to model the structure and semantics of natural language using formal symbolic systems [12, 13]. We suggest that analyzing human-interpretable linguistic instructions as natural programs – with explicit comparisons to formal programming languages – can both shed light on how humans generate, communicate, and interpret structured information [14–17] and inform how AI systems approach and interface with human users on challenging domains.

We present the **Language-annotated Abstraction and Reasoning Corpus** (LARC), which augments the original inductive tasks in [4] with *human-provided natural programs* expressed in natural language (Figure 1) elicited from a two-player communication game. We find that language provides an effective channel for communicating the structure of each task: human participants can successfully communicate at least 88% of the tasks in our dataset using language descriptions alone (i.e., without any of the original inductive examples). A **linguistic analysis** on this dataset (Sec. 4) comparing natural programs to computer programs reveals that, like computer programs, humans relied on general algorithmic concepts, such as conditional statements and loops, and on ARC-specific concepts, such as objects. However, unlike computer programs, the majority of human phrases provided examples,

[2] The solutions are hosted on GitHub: https://github.com/top-quarks/ARC-solution

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

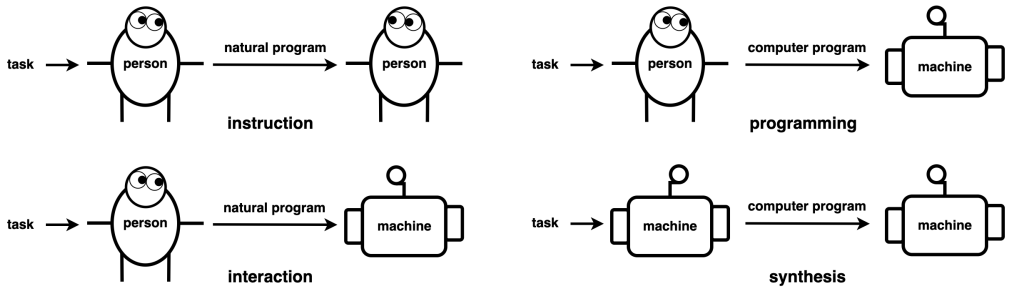

Figure 2: Four ways humans and machines communicate with each other: instruction (top-left), programming (top-right), interaction (bot-left), and synthesis (bot-right). In the case of synthesis, the machine serves both as the programmer and the interpreter as a form of self-communication.

clarifications, and verifications, providing context to the instructions. This suggests that humans both anticipate and exploit ambiguity when communicating natural programs. Finally, we present results showing that this language can be leveraged by learned **program synthesis** models to improve search-based synthesis in the absence of any ground-truth program supervision on the ARC domain.

In sum, we present the following contributions:

1. We introduce the **natural programming** framework for using human instructions to inform machine behavior in difficult domains. We present the **Language-annotated Abstraction and Reasoning Corpus**, gathered with a human communication game.
2. We present a **linguistic analysis** of the corpus that compares human natural language instructions to concepts from computer programming. We find that while most concepts in natural programs have analogies in computer programs, unlike computer programs, humans both account for and leverage ambiguities to communicate effectively.
3. We present a simple method for leveraging compositional relationships between natural language and programs in search-based **program synthesis**, showing that incorporating language this way can yield significant improvements on the ARC tasks.

2 Communication Using Natural and Computer Programs

Humans instruct each others on a range of procedural tasks, from cooking (step-by-step recipes) to detailed technical tasks (manuals with diagrams). In this setting, a person first solves the task, then describes the solution using instructions. These instructions are then interpreted by another person, producing the desired output (Fig. 2 top-left). This two-person communication process can also serve as a common framework for how we typically use machines to solve tasks. For instance, one might directly *program* a machine to solve tasks (Fig. 2 top-right), or have the machine *interacts* with an end-user (Fig. 2 bot-left) in collaboration, or develop program synthesis algorithms that solve tasks alone (Fig. 2 bot-right). In all four instances, one agent functions as a *programmer*, who constructs a *program* for a different agent, the *interpreter*, who executes the program.

How do we build machines that are programmable, interact well with end-users, and are capable of synthesizing solutions for challenging tasks? Typically, one follows a "DSL-first" approach. In the DSL-first approach, one first defines a *domain-specific language* (DSL), so that a skilled programmer may program a machine interpreter. To build an interactive system on top of a DSL, one can *naturalize* the initial DSL through language annotations [18–23], and translate them into programs for the interpreter. Similarly, program synthesis systems that automatically solves tasks [24–26] typically start with a well defined DSL, then attempt to search for a program within the DSL that satisfies the task specification. While this DSL-first workflow has yielded impressive results, the DSL itself is also a single point of failure. A key challenge for designing any DSL is determining its *scope*: a good DSL should make it easy to construct programs for the domain, without bloating the language with redundant concepts [27–29]. For systems that seek to understand human natural language, extensive effort is often necessary to ensure that the DSL aligns reasonably to human

instructions [30, 31]. Finally, for program synthesis systems, the choice of DSL has a drastic impact on whether the synthesizer can find a solution within a reasonable computation budget [24, 32].

In this work, rather than immediately constructing a formal DSL, we suggest that *starting* with human instructions (Fig 2 (top-left)) can yield important benefits for machine systems on otherwise difficult domains. We define a **natural program** as a set of 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[4] without a prior consensus–but behaves as a *program*, in that it produces a definitive output, which can be checked for correctness. Our work builds on prior work studying program-like natural language used for building shared algorithmic representations [15] and understanding cooperative communications [17], although such studies have often worked with much simpler domains than the one we examine here. Contrary the DSL-first approach, by starting with natural programs, one can focus exclusively on understanding the sets of concepts underlying a domain (such as ARC) without having to first building and committing to an interpreter that is capable of understanding these concepts. Further, as natural programs are readily constructed and effectively interpreted by humans, studying natural programs will provide insights on how to build interactive and program synthesis systems.

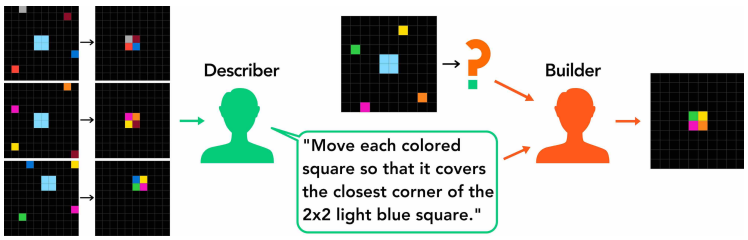

Figure 3: a **describer** instructs a **builder** how to solve an ARC task using a natural program

3 The Language-Annotated Abstraction and Reasoning Corpus (LARC)

We present a dataset that augments the original inductive ARC tasks from [4] with human *natural language* instructions that satisfy the **natural programs** definition from Sec. 2; they can be demonstrably interpreted by other human users to correctly produce intended outputs on a provided input, without any additional context (including the original inductive examples). This ensures that the language functions as a complete specification sufficient for each task. To collect this dataset, we introduce a **communication game** framing (Sec. 3.2) in which human **describers** produce linguistic instructions for unseen downstream human **builders** asked to solve the same tasks; and deploy this experiment using a novel bandit algorithm to efficiently collect verifiable natural programs [5].

The final dataset augments 88% of the original tasks (354/400) with at least one verifiable *natural program* description that could be successfully interpreted by another human participant in order to solve the task. The bandit algorithm described in Sec. 3.3 redelegates human annotators and solvers to each task based on joint estimates of the current correctness of a task's linguistic description and the total time spent on each task; Fig. 4(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. 50 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. User interface and consent form are in the supplement materials.

[4] language here is to be understood loosely as any medium of communication, including gestures and diagrams

[5] For the dataset and its collection procedure, see `https://github.com/samacqua/LARC`

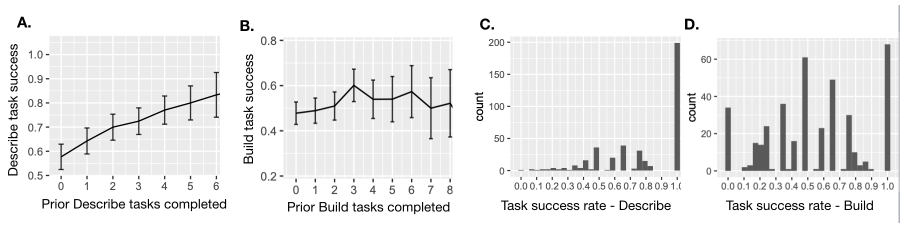

Figure 4: *A.* Describer improves at verifying their own descriptions as a they describe more tasks. *B.* Builders do not improve at constructing the correct outputs as they build more tasks (likely due to having no control over the qualities of their given descriptions). *C.* The rate of describers verifying their own descriptions. The average accuracy per task was 75%. *D.* The success rate of builders constructing the correct output. The average accuracy per task was task was 50%.

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 sees all input-outputs examples for a task, but not the test-input. The describer is instructed to solve the task and provide instructions to the builder, who can see only the test-input (and their received instructions) and must construct the correct output (See Figure 3). The description is structured into three sections to incentivize some consistency in the information conveyed: (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 (Figure 1). After the description was submitted, we verify the describer's understanding by revealing to them the test-input and asking them to produce the correct output. If the describer fails the verification task, the submitted natural program is deemed incorrect and discarded. Since we are primarily interested in communicating rather than solving ARC tasks (as opposed to [11]), each describer was shown all previous verified descriptions for a task, allowing the describer to focus on constructing an informative natural program. This results in *generations* of descriptions, forming a chain of improving natural programs written by a group of humans in collaboration.

3.3 The Bandit Algorithm for Data Collection

Collecting valid natural programs requires a significant human effort. For each task, natural programs must first be *proposed* by a number of describers, and then *validated* by a number of builders, where both the describers and builders can make mistakes. In addition, the tasks vary in difficulty, requiring different amounts of effort per task. Thus, a naive data-collection process, that simply collects a fixed number of descriptions and builds per task would be expensive.

To address this issue, we propose the following multi-bandit, infinite-arm, best-arm identification problem: **multi-bandit** – each ARC task is a different bandit, there are 400 bandits total. **infinite-arm** – for each task, each natural program is a different arm, and there are infinitely many natural language descriptions (arms); **best-arm identification** – once a natural program is proposed, we validate it by asking builders to build it. To our best knowledge, there are no known bandit algorithms that solve this problem. We use [33] to solve the infinite-arm, best-arm identification problem and a heuristic that ranks the uncertainties of a particular bandit's best arm's to address the multi-bandit problem (See Appendix). Our bandit algorithm dynamically allocates all MTurk participants to a set of appropriate efforts, which could either be generating a new description for an ARC task, or to validate an existing description, given by a previous describer. The LARC dataset was collected for an overall cost of $3667, and we estimate that a naive collection scheme would cost at least $10,800, assuming subjects would be paid the same hourly rate to produce 20 annotations per task.

4 Linguistic Analysis of Natural Program Phrases

How do people actually use language in our dataset in order to produce robustly interpretable descriptions for each task? In this section, we describe results from a manual *linguistic analysis* of verified natural program phrases in our dataset – natural language phrases that were correctly interpreted by another human to produce a correct output. We focus on two domains of conceptual content: language that conveys concepts from **Core Knowledge** [34] (such as object cohesiveness and

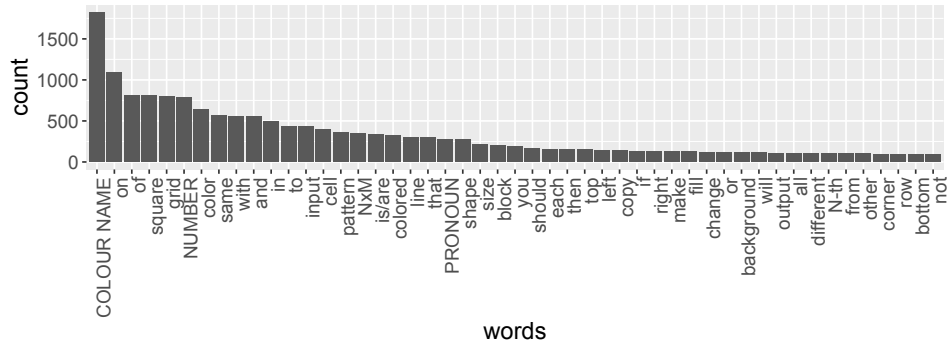

Figure 5: Words used in successfully built descriptions, sorted by their frequency in the corpus. The words were singularized. Colors names, numbers, and pronouns were grouped together.

physical interactions, hypothesized to offer innate inductive priors for human reasoning); and language that conveys concepts consistent with **programming languages** (such as loops and assertions).

Our analysis codes individual phrases with manual *tags* corresponding to relevant concepts from both domains, derived from an initial analysis for a random subset of the dataset. The details on the expert tagging procedure appears in the Supplement. 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); Fig 6A. shows a frequency of these tags across the labeled phrases.

4.1 Core Knowledge in Natural Programs

Spelke's theory of Core Knowledge systems [34] posits that humans posses innate inductive priors, which support flexible and efficient learning. Specifically, the Object system, one of the most well studied of the Core Knowledge systems, is defined by the principles of *cohesion, persistence*, and *influence via contact*. The ARC task corpus was designed to leverage Core Knowledge principles [4]:

> The ARC priors are designed to be as close as possible to Core Knowledge priors,
> ... a fair ground for comparing human intelligence and artificial intelligence (47).

Our tags reflect the core knowledge of objects in the following way: **object_detection** implies the localization of any cohesive, bounded objects; **contact_transform** refers to descriptions that can be interpreted as one object imparting a change onto another object through contact; **physical_interaction** refers to object persistence in physical scenarios – objects colliding, occluding each other, or attracting each other by an imparted 'magnetic force' or 'gravity'. About a half of the phrases referenced **objects**, three quarters of which described spatial relations of one object relative to another (Fig.6B). Fig.6D shows the relative frequencies of operations performed on objects. While objects are prevalent in LARC, most operations on objects were of the category **visual_graphical_transform** – such as recolouring and creating new objects, and **affine_transform** – such as scaling and displacement. In contrast, only 5% of phrases could also be read as influence via **physical_interaction**. This suggests that Objects behaviours in ARC are rooted more in abstract graphical operations, than in physical interactions, possibly because that it is difficult to represent time in the input-ouput format of ARC, which makes depicting physical interactions challenging.

4.2 Programmatic Concepts in Natural Programs

We study how natural programs can be understood as computer programs. First, we study the relative frequencies of executable commands – **procedure**, in contrast to "meta information", which can be analogized roughly as: **framing** – comments and setting up which library to use, **validation** – checks and assertions to ensure correct execution, and **example** – test cases. Notably, only about a third of the LARC phrases are **procedure**, while the tags **framing** and **validation** occur at roughly the same frequency (see Fig.6 C). This is in stark contrast with computer programs, where 86% of the executable functions are not commented [35]. This suggests that natural programs are more declarative (what should happen) rather than procedural (how to make it happen).

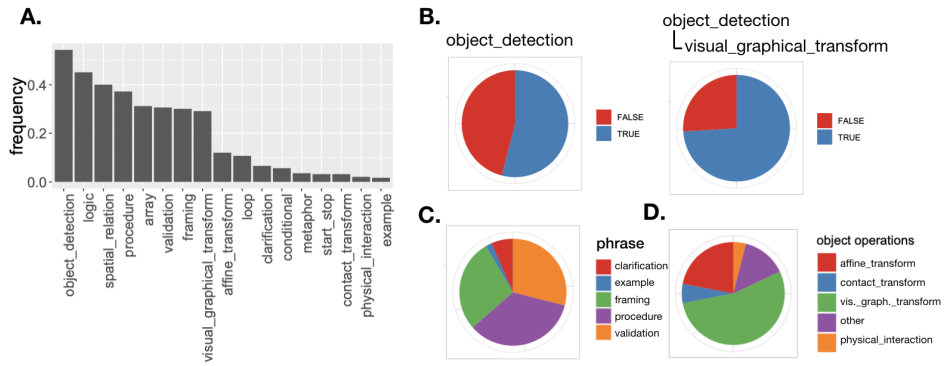

Figure 6: **A.** The frequencies of all tags occurring in human phrases. Each phrase can have multiple tags. **B.** More than half of the phrases described objects, of which, 75% described spatial relations. **C.** Relative frequencies of code (procedures) against non-code (example, framing, clarification, validation). The fraction of framing, procedure, and clarification occur at roughly the same frequency. **D.** Relative frequencies of core knowledge topics in phrases that referenced objects. More than a half of phrases referred to object modifications (such as extending, colouring, merging or dividing an object), and about a quarter described an affine transform (such as translating, rotating, or scaling). A small number of phrases could be read as influence via contact, or physical interaction (about 0.05 each)

The high frequency of **framing** tags in the LARC corpus suggests that humans carefully establish context and resolve uncertainty over which programmatic concepts may be relevant: natural programs spend roughly a third of the times *identifying* which library function to run, and how to *parse* the current task. This corroborates a key claim in [11], where they state that unlike a typical interpreter that understands only a handful of programmatic concepts, the human interpreter contains a multitude of concepts useful in solving ARC tasks. Thus, if we were to build a rich system capable of solving complex tasks such as ARC with a multitude of concepts, it might be challenging to refer to these concepts via function names in a way similar to computer programs. Instead, we must communicate with such a system carefully to ensure that the right concepts are in context at the right time.

We find that natural programs are *ambiguous*, as suggested by frequent **validations** and **clarifications** (restating the same procedure in different words). Specifically, **validation** is meant to actively help the interpreter in forming a correct interpretation. This is in contrast to conventional computer programs that use assertions, which catches an incorrect interpretation by crashing the interpreter. The **clarification** amends the initial ambiguous explanation, with another, also ambiguous explanation, resulting in an interpretations that are consistent with both. We posit that a better analogy to natural programs are not imperative programs, but program synthesis [25]. In program synthesis, the user specifies a rough sketch of what the correct program should look like (**framing**, **procedure**), leaving the details ambiguous. To resolve these ambiguities, user provides a list of constraints (**validation**) that must be satisfied for the program to be correct. Overall, we conclude that humans share (to some degree) knowledge of a large, domain-general language, and readily anticipate and exploit ambiguities to effectively communicate in this language.

5 Leveraging Language for Program Synthesis

Our empirical results (Sec. 3) and linguistic analysis (Sec. 4) suggest that humans flexibly use and interpret structured, program-like information in natural language. In this section we show that this language can be used to guide learning in *machine* models in the same domain. Given the parallels between human language, programs, and program synthesis suggested by Sec. 3-4, we focus on learned *program synthesis* models, where a learned synthesizer infers computer programs for a given task that can be executed by a machine (Fig 2 (bot-right)).

Most prior work in learned program synthesis has focused on solving tasks specified inductively by a set of input/output examples [32, 36–38]. Here we show how leveraging relationships between language and programs can inform a key learning problem in program synthesis: learning effective *search models* for programs that solve tasks in a provided domain-specific language (DSL) [27, 37, 39, 40]. We show that simple techniques for incorporating linguistic information can improve learning

performance. Our results support the conclusions of recent studies that leverage the structure and compositionality of human language to inform programmatic representations [20,41–45] .

5.1 Language-guided program search

To explore the use of natural language to inform **search models** in program synthesis, we build on recent neurally-guided search algorithms, which train neural models to predict distributions over solution programs to improve search [27,37,38,40,41,46,47]. We focus on learning in the *distant supervision* setting – domains without initial ground-truth programs (like ARC), where natural language is likely to provide the most value as an alternative source of prior information. In all of our experiments, we train models using the iterative approach used in [32,48,49] that alternates between searching for solutions with a current trained model, and using the solutions to supervise learning for the next training iteration.

We implement two techniques for incorporating natural language to inform training. Both are general methods that could be used with any neurally-guided synthesis model (e.g. [37,38]) that conditions on a vector-based task encoding. All language-guided and baseline models are trained using a base neural search architecture from [32,49] (a variant on the DeepCoder model in [37]), which conditions on a task embedding and predicts a distribution over tree bigrams in a DSL. All model, training details and a full code release are available in the Supplement.

1. **Language-guided search with pre-trained language model encoding**: our most basic language-guided search model simply uses a pre-trained language model to encode the language annotations for each task, which we provide as additional input during training to the neural synthesizer. We implement this as the standard first approach for incorporating language to guide neural program search. In our experiments we use a pre-trained T5 Transformer model to encode language [50], taking a mean over contextual token embeddings to produce a single sentence embedding for each task based on its annotations.
2. **Language-guided search with pre-trained language model encoding and *pseudoannotation generation*:** methods like (1) face a challenging learning problem in the distant supervision setting – even with pre-trained language representations, we aim to learn compositional relationships between language and an unknown program DSL from only distant supervision. Therefore, we introduce an additional technique for leveraging relationships between language and programs – we annotate each function in the program DSL with a phrase that gives a natural language 'gloss' of its behavior (e.g. annotating a `flood_fill(color)` function with the gloss *fill with the color*). During training, we use these to generate additional paired language and program examples, by generating programs from the DSL and mapping the flattened program trees to their token-wise natural language glosses to produce a full "pseudo natural language annotation".

   This approach is similar to recent methods that have leveraged engineered or learned synchronous grammars [23,31,49,51] for data augmentation. We find our approach to be simple but effective: we see natural language glosses as an easy way for human engineers to provide additional information about how language corresponds to code, similar to linguistic comments for each function. Our full set of function annotations appears in the supplement, along with example programs and their pseudo-annotations.

We compare these language-guided approaches to two non-linguistic baselines. Any of these language-guided techniques could also augment multimodal search (eg. by concatenating CNN and linguistic features, or through a mixture of experts); we train them separately here to evaluate the specific contributions of linguistic data. All training details on these models are also available in the supplement.

1. **Fitted bigram enumeration**: we fit a PCFG using the inside-outside algorithm over tree bigrams in the DSL to the solution programs [52]. This provides a fitted prior over tasks learned from the domain, but does not use task-conditional information.
2. **CNN-guided search**: we train a 4-layer CNN to encode the ARC I/O example grids as input. This provides an alternative task encoding feature embedding to the language embeddings.

We evaluate search performance in the distant supervision setting, using a matched time budget for search per task at each iteration (720s) and fixed number of training iterations overall (n=5). All

Table 1: Tasks solved by language and non-language-guided models, with learned search and learned libraries. Iterative distant supervision training (n=5 iterations). To compare performance with different amounts of initial supervision, we initialize models with (*left*) 720s of initial search, yielding 11 initial solved tasks; (*middle*) 1 hour and 26 solved tasks; and (*right*) 8 hours and 57 solved tasks.

Model	**Total solved (720s init. enum)**	**Total solved (1hr init. enum)**	**Total solved (8hr init. enum)**
No language – initial enumeration from the base DSL	11/400	26/400	57/400
No language – fitted bigram baseline	33/400	44/400	58/400
No language – CNN task encoder for neural search	36/400	43/400	64/400
Language – T5 language encoder for neural search	29/400	41/400	64/400
Language – T5 + pseudoannotation training	**59/400**	**66/400**	**70/400**

models are initialized using a basic DSL containing primitives to select and transform grids of colored pixels. As with prior work [32, 48, 49] we initialize iterative search by first enumerating for a fixed budget of time from the DSL before beginning iterative training. We vary this initial amount of time in order to better explore how and when different methods are able to best bootstrap learning with varied initial amounts of solution tasks.

Table 1 shows that across all initial enumeration settings – including our weakest enumeration setting, in which we search for 720s in the DSL to initialize models with only *11 initial tasks* – the language-guided model trained with "pseudoannotations" achieves significant performance gains, both over the baselines and over the initial solution state. In contrast, the basic language-supervised model becomes more effective only with greater amounts of starting supervision.

We see leveraging *function* annotations compositionally and generatively – as in the pseudoannotation generation procedure – as an important avenue for future work in introducing language to guide machine behavior. We suggest that language can provide a powerful interface between humans and machines when used not only to describe individual tasks, but the behavior of program components, leveraging the natural productivity of both human and machine languages to learn relationships between domain-general, pre-trained linguistic priors and domain-specific, structured programming languages with little ground truth supervision.

6 Discussion

We proposed **natural programs**, instructions that can readily be interpreted by other humans in order to produce a verifiable output, as a pathway to building intelligent systems that can generalize to novel situations and understand human instructions. To this end, we curated and presented the **Language-Annotated Abstraction and Reasoning Corpus** (LARC), a new dataset containing natural language descriptions of tasks from the ARC domain [4] that can be used to study both human and machine behavior in a difficult inductive reasoning setting. We outlined a **novel data collection procedure** (Sec. 3.3) to efficiently collect natural program annotations, using a bandit algorithm to delegate data collection and framing individual annotations as a *communication game*.

Our **linguistic analysis** (Sec. 4) of communication strategies suggests that there are rich connections between the natural language participants use in this setting and formal concepts from machine-interpretable programming languages, though we also find important differences between natural language and most current programming languages. We conclude by presenting results from **language-guided program synthesis** experiments (Sec. 5), which suggest that natural language programs can be used to significantly improve search-based program synthesis in the ARC domain, with no other ground-truth program data.

We see the communication strategies used by humans to solve tasks as a rich source of structured knowledge for program-inspired cognitive analyses (such as [3, 9]), and for developing AI systems augmented with natural language to guide machine behavior (e.g. semantic parses and models of structured search and representation learning [18–20, 45, 49, 53, 54]). We propose that studying the *differences* between natural programs and current state-of-the-art programming languages, such as the framing and validation devices used to resolve ambiguity, can lend valuable guidance for developing

future machine learning systems. Developing systems that leverage human-like use of language can lead to improving the effectiveness of language-based interfaces, and to more flexible formal representations with enhanced ability to generalize across domains.

Limitations The ARC domain consists of a single, constrained task format that assays human intelligence in a highly controlled setting. While we also present synthesis models informed by natural language, our synthesis model leaves important aspects of human language, such as giving and controlling ambiguities, unexplored. Future cognitive and computational work should extend these findings to other domains – especially tasks that directly invoke 'core' competencies and explore tighter couplings between program synthesis and natural language.

Potential negative impact The long-term goal of this work is to 'reverse-engineer' how humans think and communicate, in order to jointly inform cognitive research – how do people solve and learn from structured tasks – and computational systems, that can learn and collaborate with humans. While this work takes constrained steps in these directions, it assumes both of these aims as important end goals. Any system that seeks to accurately use and interpret human language raises concerns regarding safe deployment for downstream applications, for instance, non-experts operating safety-critical equipment using natural language. Further, AI could exploit ambiguity in human language to write legally-binding agreements that can be read in different ways.

Acknowlegements

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

References

[1] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *arXiv preprint arXiv:1712.01815*, 2017.

[2] Adam Lerer, Hengyuan Hu, Jakob Foerster, and Noam Brown. Improving policies via search in cooperative partially observable games. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 7187–7194, 2020.

[3] Brenden M Lake, Tomer D Ullman, Joshua B Tenenbaum, and Samuel J Gershman. Building machines that learn and think like people. *Behavioral and brain sciences*, 40, 2017.

[4] François Chollet. On the measure of intelligence. *arXiv preprint arXiv:1911.01547*, 2019.

[5] Michelene TH Chi, Robert Glaser, and Marshall J Farr. *The nature of expertise*. Psychology Press, 2014.

[6] Harry F Harlow. The formation of learning sets. *Psychological review*, 56(1):51, 1949.

[7] Brenden M Lake and Steven T Piantadosi. People infer recursive visual concepts from just a few examples. *Computational Brain & Behavior*, 3(1):54–65, 2020.

[8] Frederic Charles Bartlett. *Remembering: A study in experimental and social psychology*. Cambridge University Press, 1932.

[9] Lucas Tian, Kevin Ellis, Marta Kryven, and Josh Tenenbaum. Learning abstract structure for drawing by efficient motor program induction. *Advances in Neural Information Processing Systems*, 33, 2020.

[10] Tania Lombrozo. The structure and function of explanations. *Trends in cognitive sciences*, 10(10):464–470, 2006.

[11] Aysja Johnson, Wai Keen Vong, Brenden M Lake, and Todd M Gureckis. Fast and flexible: Human program induction in abstract reasoning tasks. *arXiv preprint arXiv:2103.05823*, 2021.

[12] Paul H Portner and Barbara H Partee. *Formal semantics: The essential readings*, volume 7. John Wiley & Sons, 2008.

[13] Jerry A Fodor. *The language of thought*, volume 5. Harvard University Press, 1975.

[14] Elizabeth S Spelke, Karen Breinlinger, Janet Macomber, and Kristen Jacobson. Origins of knowledge. *Psychological review*, 99(4):605, 1992.

[15] W. McCarthy, R.D. Hawkins, C. Holdaway, H. Wang, and J Fan. Learning to communicate about shared procedural abstractions. In *Proceedings of the 43rd Annual Conference of the Cognitive Science Society*, 2021.

[16] Herbert H Clark, Robert Schreuder, and Samuel Buttrick. Common ground at the understanding of demonstrative reference. *Journal of verbal learning and verbal behavior*, 22(2):245–258, 1983.

[17] Herbert H Clark and Deanna Wilkes-Gibbs. Referring as a collaborative process. *Cognition*, 22(1):1–39, 1986.

[18] Yoav Artzi and Luke Zettlemoyer. Weakly supervised learning of semantic parsers for mapping instructions to actions. *Transactions of the Association for Computational Linguistics*, 1:49–62, 2013.

[19] Yoav Artzi, Dipanjan Das, and Slav Petrov. Learning compact lexicons for ccg semantic parsing. 2014.

[20] Xi Ye, Qiaochu Chen, Isil Dillig, and Greg Durrett. Optimal neural program synthesis from multimodal specifications. *arXiv preprint arXiv:2010.01678*, 2020.

[21] Yushi Wang, Jonathan Berant, and Percy Liang. Building a semantic parser overnight. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 1332–1342, 2015.

[22] Sida I Wang, Percy Liang, and Christopher D Manning. Learning language games through interaction. *arXiv preprint arXiv:1606.02447*, 2016.

[23] Alana Marzoev, Samuel Madden, M Frans Kaashoek, Michael Cafarella, and Jacob Andreas. Unnatural language processing: Bridging the gap between synthetic and natural language data. *arXiv preprint arXiv:2004.13645*, 2020.

[24] Kevin Ellis, Maxwell Nye, Yewen Pu, Felix Sosa, Josh Tenenbaum, and Armando Solar-Lezama. Write, execute, assess: Program synthesis with a repl. In *Advances in Neural Information Processing Systems*, pages 9165–9174, 2019.

[25] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. Combinatorial sketching for finite programs. In *ACM Sigplan Notices*, volume 41, pages 404–415. ACM, 2006.

[26] Yewen Pu, Kevin Ellis, Marta Kryven, Josh Tenenbaum, and Armando Solar-Lezama. Program synthesis with pragmatic communication. *Advances in Neural Information Processing Systems*, 33, 2020.

[27] Sumit Gulwani, José Hernández-Orallo, Emanuel Kitzelmann, Stephen H Muggleton, Ute Schmid, and Benjamin Zorn. Inductive programming meets the real world. *Communications of the ACM*, 58(11):90–99, 2015.

[28] Bobby R Bruce, Tianyi Zhang, Jaspreet Arora, Guoqing Harry Xu, and Miryung Kim. Jshrink: in-depth investigation into debloating modern java applications. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 135–146, 2020.

[29] César Soto-Valero, Nicolas Harrand, Martin Monperrus, and Benoit Baudry. A comprehensive study of bloated dependencies in the maven ecosystem. *Empirical Software Engineering*, 26(3):1–44, 2021.

[30] Percy Liang. Learning executable semantic parsers for natural language understanding. *Communications of the ACM*, 59(9):68–76, 2016.

[31] Richard Shin, Christopher H Lin, Sam Thomson, Charles Chen, Subhro Roy, Emmanouil Antonios Platanios, Adam Pauls, Dan Klein, Jason Eisner, and Benjamin Van Durme. Constrained language models yield few-shot semantic parsers. *arXiv preprint arXiv:2104.08768*, 2021.

[32] Kevin Ellis, Catherine Wong, Maxwell Nye, Mathias Sable-Meyer, Luc Cary, Lucas Morales, Luke Hewitt, Armando Solar-Lezama, and Joshua B Tenenbaum. Dreamcoder: Growing generalizable, interpretable knowledge with wake-sleep bayesian program learning. *arXiv preprint arXiv:2006.08381*, 2020.

[33] Yizao Wang, Jean-Yves Audibert, and Rémi Munos. Infinitely many-armed bandits. In *Advances in Neural Information Processing Systems*, 2008.

[34] Elizabeth S. Spelke and Katherine D. Kinzler. Core knowledge. *Developmental Science*, 10(1):89–96, 2007.

[35] Yuan Huang, Nan Jia, Junhuai Shu, Xinyu Hu, Xiangping Chen, and Qiang Zhou. Does your code need comment? *Software: Practice and Experience*, 50(3):227–245, 2020.

[36] Tessa Lau, Steven A Wolfman, Pedro Domingos, and Daniel S Weld. Programming by demonstration using version space algebra. *Machine Learning*, 53(1-2):111–156, 2003.

[37] Matej Balog, Alexander L Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. Deepcoder: Learning to write programs. *ICLR*, 2016.

[38] Emilio Parisotto, Abdel-rahman Mohamed, Rishabh Singh, Lihong Li, Dengyong Zhou, and Pushmeet Kohli. Neuro-symbolic program synthesis. *arXiv preprint arXiv:1611.01855*, 2016.

[39] Oleksandr Polozov and Sumit Gulwani. Flashmeta: A framework for inductive program synthesis. *ACM SIGPLAN Notices*, 50(10):107–126, 2015.

[40] Jacob Devlin, Jonathan Uesato, Surya Bhupatiraju, Rishabh Singh, Abdel-rahman Mohamed, and Pushmeet Kohli. Robustfill: Neural program learning under noisy i/o. *ICML*, 2017.

[41] Xi Ye, Qiaochu Chen, Isil Dillig, and Greg Durrett. Benchmarking multimodal regex synthesis with complex structures. *arXiv preprint arXiv:2005.00663*, 2020.

[42] Maxwell Nye, Luke Hewitt, Joshua Tenenbaum, and Armando Solar-Lezama. Learning to infer program sketches. *ICML*, 2019.

[43] Illia Polosukhin and Alexander Skidanov. Neural program search: Solving programming tasks from description and examples. *arXiv preprint arXiv:1802.04335*, 2018.

[44] Aditya Desai, Sumit Gulwani, Vineet Hingorani, Nidhi Jain, Amey Karkare, Mark Marron, and Subhajit Roy. Program synthesis using natural language. In *Proceedings of the 38th International Conference on Software Engineering*, pages 345–356, 2016.

[45] Shashank Srivastava, Igor Labutov, and Tom Mitchell. Joint concept learning and semantic parsing from natural language explanations. In *Proceedings of the 2017 conference on empirical methods in natural language processing*, pages 1527–1536, 2017.

[46] Daniel A Abolafia, Mohammad Norouzi, Jonathan Shen, Rui Zhao, and Quoc V Le. Neural program synthesis with priority queue training. *arXiv preprint arXiv:1801.03526*, 2018.

[47] Xujie Si, Yuan Yang, Hanjun Dai, Mayur Naik, and Le Song. Learning a meta-solver for syntax-guided program synthesis. In *International Conference on Learning Representations*, 2019.

[48] Eyal Dechter, Jon Malmaud, Ryan P Adams, and Joshua B Tenenbaum. Bootstrap learning via modular concept discovery. In *Twenty-Third International Joint Conference on Artificial Intelligence*, 2013.

[49] Wong Catherine, Levin Ellis, Jacob Andreas, and Joshua Tenenbaum. Leveraging natural language for program search and abstraction learning. *Thirty-eighth International Conference on Machine Learning*, 2021.

[50] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *arXiv preprint arXiv:1910.10683*, 2019.

[51] Robin Jia and Percy Liang. Data recombination for neural semantic parsing. *arXiv preprint arXiv:1606.03622*, 2016.

[52] Karim Lari and Steve J Young. The estimation of stochastic context-free grammars using the inside-outside algorithm. *Computer speech & language*, 4(1):35–56, 1990.

[53] Jacob Andreas, Dan Klein, and Sergey Levine. Learning with latent language. *arXiv preprint arXiv:1711.00482*, 2017.

[54] Richard Shin, Illia Polosukhin, and Dawn Song. Improving neural program synthesis with inferred execution traces. In *NeurIPS*, pages 8931–8940, 2018.

[55] Tom B Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*, 2020.

[56] Andrea Madotto, Zihan Liu, Zhaojiang Lin, and Pascale Fung. Language models as few-shot learner for task-oriented dialogue systems. *arXiv preprint arXiv:2008.06239*, 2020.

[57] Ought. Elicit: The ai research assistant, 2021.

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] See Limitations, Sec. 6.
   (c) Did you discuss any potential negative societal impacts of your work? [Yes] See Potential negative impact, Sec. 6.
   (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...
   (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, containing both the human experiment and synthesis modeling details and code.
   (b) Did you specify all the training details (e.g., data splits, hyperparameters, how they were chosen)? [Yes] See supplement for details on experimental details on synthesis. We also release command line data to reproduce all experiments.
   (c) Did you report error bars (e.g., with respect to the random seed after running experiments multiple times)? [N/A] We use a fixed random seed and do not batch over the training tasks for all experiments, to minimize variance.
   (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 for experimental details on synthesis.
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]
   (b) Did you mention the license of the assets? [No] Most assets are open source and available online.
   (c) Did you include any new assets either in the supplemental material or as a URL? [Yes] We introduced the Language-annotated Abstraction and Reasoning Corpus, which is included as a URL in Sec. 3.
   (d) Did you discuss whether and how consent was obtained from people whose data you're using/curating? [Yes] All participants gave informed consent. See Sec. 3; the full experimental framework (including the consent form) is released in the code repository along with the supplement.
   (e) Did you discuss whether the data you are using/curating contains personally identifiable information or offensive content? [Yes] This dataset does not include personally identifiable information or offensive content. See Sec. 3.
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] We provide the detailed text of the instructions and screenshots of the experimental interface in the supplement, as well as a full release of the experimental framework.
   (b) Did you describe any potential participant risks, with links to Institutional Review Board (IRB) approvals, if applicable? [Yes] We do not believe our experiment has potential risks. This experiment was approved by an IRB; we include the full IRB approval in the de-anonymized release of the supplement.
   (c) Did you include the estimated hourly wage paid to participants and the total amount spent on participant compensation? [Yes] See details in Sec. 3.

7 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 an explore gui that allows one to browse the entire dataset within the browser.
2. program synthesis using language codes is at this URL :
   `https://anonymous.4open.science/r/ec-28F5`

7.1 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 7 shows the build and describe interface. For full workflow see `LARC/collection`.

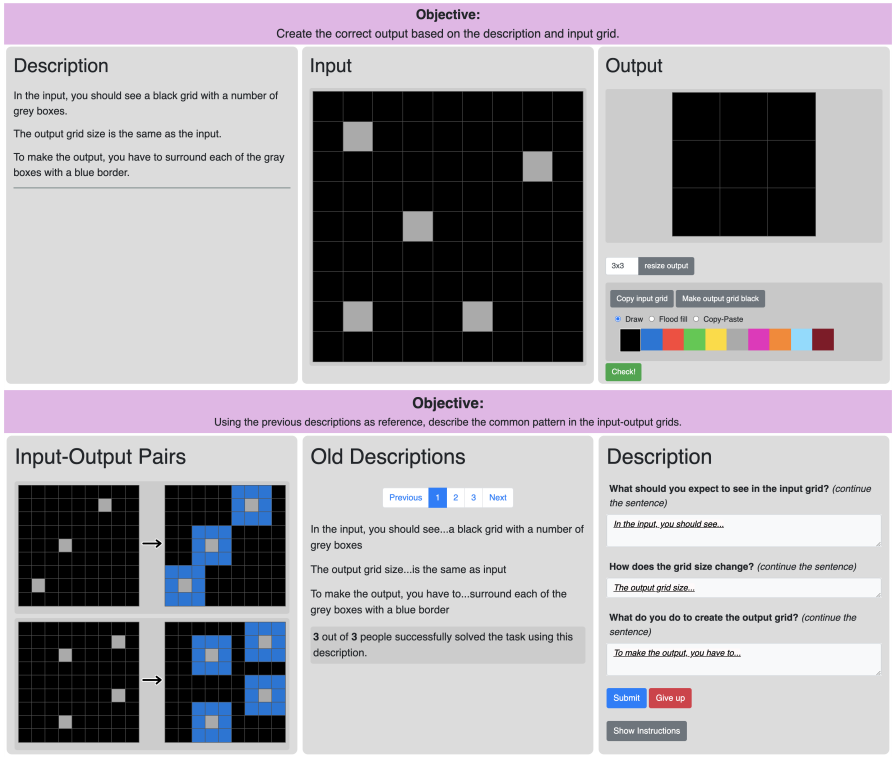

Figure 7: *A.* The builder interface. *B.* The describer interface.

7.2 LARC Linguistic Analysis Tagging Scheme

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 (-) [6]. 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.

[6] 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

The tagged phrases can be found at `LARC/dataset`

7.3 Supplement to Sec. 5: Leveraging Language for Program Synthesis

Here we describe model and experimental details for the program synthesis experiments in Sec. 5. The full DSL, model implementations, and commands to replicate the experiments here are available at `https://anonymous.4open.science/r/ec-28F5/Readme.md`.

7.3.1 Supplement to Sec. 5.1: Model and training details for Experiment 1

All neural search models use a base model architecture shown to be effective in the small data, minimal supervision setting we study here, drawn from [32, 49] (a variant on the DeepCoder model in [37]), which conditions on a task embedding and predicts a distribution over tree bigrams in a DSL. In particular, this model can be described as an *amortized inference* model $Q(\rho|t, \mathcal{L})$ where ρ is a program, t is a task-specific vector encoding of task features (such as language or I/O examples), and $\mathcal{L}$ is a DSL containing function primitives. This is parameterized by two modular components:

1. *A domain-specific task encoder* $E(t)$. This encodes task and program specific information (eg. images or language) that are input to the neural model. This task encoder architecture is defined domain-specifically based on the form of the task examples (e.g. a CNN for the graphics domain). It outputs a fixed dimensional embedding for any given task as *input* to the model. The encoder component varies across each of our models, and is described in more detail below.
2. *A conditional model over programs* $Q(\rho|E(t))$. This component receives the task encoding as input and outputs a distribution over programs. Following [32], this is a 2-layer fully-connected MLP (with 64 hidden units and a final tanh activation layer) that outputs a fixed-dimensional real-valued tensor encoding a distribution over programs in the library $\mathcal{L}$ as output. The real-valued tensor corresponds to weights over program primitives conditioned on their local context in the syntax tree of the program, consisting of the parent node in the syntax tree and which argument is being generated. This functions as a 'bigram transition model' over trees that encodes the likelihood of transitions from one primitive to the next. Q returns this as a $(|\mathcal{L}|+1) \times (|\mathcal{L}|+2) \times A$-dimensional tensor, where A is the maximum arity of any primitive in the library.

This parameterization supports fast sampling of programs during conditional synthesis: the neural model runs once per task (to encode the task examples and produce the bigram transition model) and the resulting parameterization can then be used to sample programs during synthesis (e.g. by enumerating programs by expanding trees (as 'bigrams' over parent and children primitives) ranked in order of their likelihood starting from the program root.)

Following [32], the neural model is trained to optimize the following MAP inference objective on the solved training tasks (where $(\mathcal{L}, \theta_{\mathcal{L}})$ is the base PCFG prior fit to the solved programs, as in [32]), as well as (in the case of the pseudoannotation model trained with a generative model), sampled programs and accompanying feature vectors:

$$\underset{\text{MAP}=\mathrm{E}_{x\sim(\mathcal{L},\theta_{\mathcal{L}})}}{\mathcal{L}} \left[\log Q \left(\arg\max_{\rho} \mathrm{P}[\rho|x,\mathcal{L},\theta_{\mathcal{L}}] \;\middle|\; x \right) \right] \tag{1}$$

Using this base model, we now describe the specific encoders $E(t)$ we implement for the models described in 5:

1. **Language-guided search with pre-trained language model encoding**: This uses a large-scale language model as an encoder over descriptions: in particular, an encoder $E(\mathbf{d})$ where $\mathbf{d}$ is a set of all *sentences* in the 'output' descriptions for each task. We find that using a modified tokenization heuristic to pre-process the language data into sentences that are encoded individually (rather than encoding the entire, multi-sentence annotation produced by each annotator during the communication experiment at once) is more effective. We use the `t5-small` parameterization of the T5 pre-trained transformer model from [50], which produces contextual embeddings for each token in the sentence; we take a mean-reduction over all tokens to produce a single embedding vector per sentence, and then an additional mean-reduction over all sentences in $\mathbf{d}$ to produce a final 512-dimensional embedding for each task.

2. **Language-guided search with pre-trained language model encoding and *pseudoannotation generation*:** This uses the same encoder model $E(\mathbf{d})$ as model (1). However, as described in the main text, this model incorporates additional sampled 'pseudoannotation' language and programs $(\mathbf{d'}, \rho')$ generated using a simple generative model based on linguistic function annotations of each function in the DSL $\mathcal{L}$. In particular, we annotate each function $l \in \mathcal{L}$ with a corresponding natural language gloss d_l that describes its behavior (example annotations are shown in Table X below.) To generate new training data, we sample programs ρ' from the fitted PCFG prior $(\mathcal{L}, \theta_{\mathcal{L}})$ over training tasks (similar to the generative steps used in [32, 49], and then generate a corresponding pseudoannotation **d'** by taking a left-order traversal of the functions in ρ' to produce a flattened program tree, and then mapping each function to its corresponding gloss. These naturalistic pseudoannotations are provided interchangeably as training data – that is, we encode $E(\mathbf{d}')$ using a pre-trained language model, as if it were a true human-provided annotation, to produce a 512-dimensional embedding for each true task annotation and pseudo annotation training datapoint.

3. **CNN-guided search**: This baseline uses a CNN encoder $E(\mathbf{x})$ to encode the I/O examples $\mathbf{x}$ in the ARC tasks. This baseline is a re-implementation of the CNN encoder used previously in [32, 49]: it uses a 4-layer CNN consisting of 4 stacked convolution blocks; except for the initial input (which takes in a full grid for each task as input), each convolutional layer has a 64 input/output channels with a kernel size of 3, and is followed by a RELU and max-pooling (dim=2) block. We concatenate the input-output grids before encoding, and mean-reduce over the encodings over all I/O examples in $\mathbf{x}$ to produce a 64-dimensional embedding for each task.

4. **Language-guided search with pre-trained language model encoding and few-shot learned semantic parsing features (not included in main paper):** We also experiment with using the function annotations from (2) to explore one additional means of relating language and programs with minimal superivsion, inspired by recent work [31, 55, 56] suggesting that very large-scale language models (such as GPT-3) can "few shot" parse structured patterns from a few linguistic examples. We develop a simple approach to use GPT-3 as a 'binary classifier' over individual functions in the DSL (using [57]). Using functions present in solution programs as positive examples (and randomly sampled other tasks as noisy negative examples), we present a large-scale language model with paired {*task annotation, positive/negated natural language function gloss*} examples and ask it to infer the appropriate pattern for unsolved task annotations (e.g. to classify the `flood_fill(color)` from above, we ask GPT-3 to predict the phrase *fill with color* or *do not fill with color* based on a task annotation.) While these predictions could be in principle used directly place a distribution over functions to guide search, we find that they are somewhat noisy and instead better provided as an additional feature vector $E_2(\mathbf{d})$ – concatenated with the full task annotation encoding $E(\mathbf{d})$ – and provided as input to the same base model as above producing a distribution over programs. For a given task annotation, encoder $E_2(\mathbf{d})$ produces a length 6 feature vector. Each of the 6 features correspond to a specific DSL function, and specifically the feature value is the probability GPT-3 assigns to the corresponding function's positive natural language gloss. To generate every such prediction, a maximum of 20 paired {*task annotation, positive/negated natural language function gloss*} examples are selected based on a measure of semantic similarity with the task annotation (as measured by GPT-3) to be classified and are used to create the few-shot classification language-model prompt (see Figure 8 for an example prompt).
   In the preliminary pilot experiments we find that this approach does not outperform (1). We hypothesize this is due to the small sample size of the initial solved tasks and because we only use 6 features when in principle we could create a feature for every one of the 103 DSL elements (assuming every element appears in at least one of the initial solved task programs). Although the $E_2(\mathbf{d})$ features did not lead to more task solved than in (1), we find that that the few-shot learned classifiers are better than chance at classifying the natural language function gloss (as either positive or negated) of unseen task annotations. Specifically, the leave-one out AUC scores for the initial 57 solved tasks were 0.92, 0.71, 0.96, 0.76, 0.80 and 0.84 for each of the 6 DSL functions used to create the $E_2(\mathbf{d})$ features (although note the small sample size from which these were calculated). Based on the above we believe it is a promising avenue for future work.

Example DSL Functions and Natural Language Gloss Function Annotations	
DSL Function	*Natural Language Gloss*
map_blocks	'for every block'
filter_block_tiles	'only keep tiles that'
fill_with_color	'color the block'
blocks_to_min_grid	'get the smallest grid containing the blocks'
has_at_least_n_tiles	'does the block have at least n tiles'
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'

The other baseline model, **Fitted bigram enumeration**, only produces a prior $P[\rho]$ over programs. This uses a PCFG fit using the inside-outside algorithm over tree bigrams in the DSL to the solution programs [52], but does not use task-conditional information.

Program DSL and function annotations: All experiments used a hand-designed DSL for the ARC 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.

Search details: We train all models using the iterative training procedure previously reported in [32, 49]: at each iteration, we alternate between a training step in which models are fit to the set of existing solved tasks from prior iterations (along with any data augmented examples from the generative model), and a search step in which the models are used to guide search for a fixed time budget per task. All experiments for were conducted on a high-powered computing cluster using a fixed training budget of wall-clock search time per task of 720s for all models and baselines in a given domain (determined via hyperparameter search using the baseline model per domain); and with 24 CPUs per experiment.

As reported in the main text, all iterative training began with an initial, unbiased search for a fixed budget of time from the starting DSL to yield an initial set of solved tasks (with accompanying programs) used to begin iterative learning; we varied this initial search duration to explore training under different amounts of starting supervision, as reported in Sec 5.1.

```
Classify the following as one of:

1. do not color the block
2. color the block

natural_language: copy the significant pattern.
Tag: do not color the block

natural_language: remove the yellow rectangles and replace the color to match the existing pattern.
Tag: do not color the block

natural_language:  look at both the left and right parts of the input grid. You will notice that the left and right p
arts 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.
Tag: do not color the block

natural_language: copy the input grid. Any shapes which only contain one single square should be colored blue.
Tag: color the block

natural_language: find the color with the most spaces colored and put that pattern into the 3x3.
Tag: do not color the block

natural_language:  copy and paste the left-side of the input (the yellow side) onto the output grid. Then, draw the g
reen pattern on the output grid on top of the yellow (do not copy and paste because the black will overwrite the yell
ow). Your output grid should now have green and yellow on a black background. Then, recolor each green and yellow squ
are pink, so your output is only pink and black.
Tag: do not color the block

natural_language: place the input grid in the top 3x3 space and create a mirror image in the bottom 3x3 space.
Tag: do not color the block

natural_language: remove the yellow rectangles and replace the all of the yellow squares with the appropriate color i
n order to create a pattern that is both horizontally and vertically symmetrical.
Tag: do not color the block

natural_language: copy the same colors in the same position. For every single grid in the input, it should be changed
to a 3x3 of color each in the output.
Tag: do not color the block

natural_language: fill the black spots with the exact same pattern that need to be filled in the original grid.
Tag: do not color the block

natural_language: make the pattern and if there is any more on the sides or any blue, fill in with black.
Tag: do not color the block

natural_language: create a left-right mirror of pattern of the orange shape.
Tag: do not color the block

natural_language: fill the top of the grid with the same as original. And fill the bottom of the grid with a mirror i
mage of the original.
Tag: do not color the block

natural_language: place the input design all the way on the left half and then duplicate the exact same design and co
lors on the right half. The final grid is two of the original grid next to each other.
Tag: do not color the block

natural_language: compare the squares in the top and bottom 4x4 square. If they match (both black or both color) make
the output square black. If they do not match, make the output square green.
Tag: do not color the block

natural_language: make the pattern mirror as left-right with the same height and twice the row.
Tag: do not color the block

natural_language: change the pattern to match the one colored square at the bottom and fill in the one square at the
bottom to black.
Tag: color the block

natural_language: copy the pattern exactly to the new empty section. There should be two of the same pattern side-by-
side.
Tag: do not color the block

natural_language: copy the original pattern to the left half of the grid, then mirror it on the right, like an inkblo
t.
Tag: do not color the block

natural_language: copy the pattern as you see it on the bottom half of the grid into 3x4. Once done, flip the pattern
up to the top half of the 3x4 grid.
Tag: do not color the block

natural_language: replicate one of the patterns in one of the corners. Bottom right, bottom left, top right, or top l
eft.
Tag:
```

Figure 8: GPT-3 prompt for classifying natural language annotation "replicate one of the patters in one of the corners. Bottom right, bottom, left, top right, or top left." for DSL function fill_color

8 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.

8.1 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 9. We estimate how many minutes would this particular unit take, and dynamically allocate additional units until the full 45 minutes are exhausted.

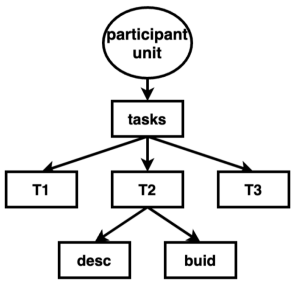

Figure 9: How a "unit" of a participant's time can be utilized

8.2 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 10.

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 10.

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

```
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 10: an example transition where there are 3 casinos

8.3 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 [33] 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 [33] 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 [33], 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. [33] 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.

[33] 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^*.

8.4 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 11

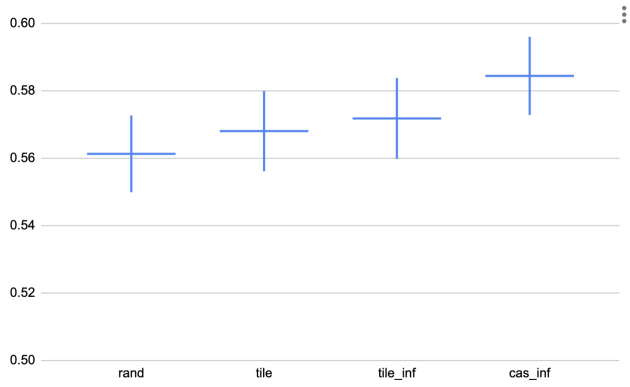

Figure 11: performance of various bandit policies, of 100 casinos and a budget of 600, averaged across 100 repetitions. horizontal bar is average, whiskers indicate standard deviation

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`