ReGAL: Refactoring Programs to Discover Generalizable Abstractions

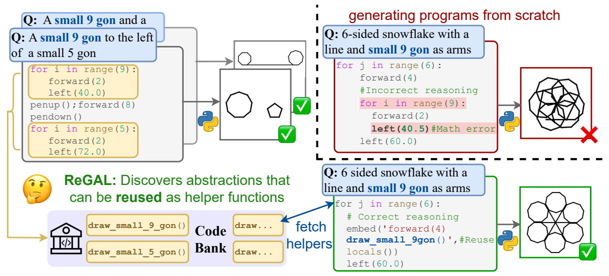
Elias Stengel-Eskin [*1] Archiki Prasad [*1] Mohit Bansal [1]

Abstract

While large language models (LLMs) are increasingly being used for program synthesis, they lack the global view needed to develop useful abstractions; they generally predict programs one at a time, often repeating the same functionality. Generating redundant code from scratch is both inefficient and error-prone. To address this, we propose **Re**factoring for **G**eneralizable **A**bstraction **L**earning (ReGAL), a gradient-free method for learning a library of *reusable* functions via code *refactorization*, i.e. restructuring code without changing its execution output. ReGAL learns from a small set of existing programs, iteratively verifying and refining its abstractions via execution. We find that the shared function libraries discovered by ReGAL make programs *easier to predict* across diverse domains. On three datasets (LOGO graphics generation, Date reasoning, and TextCraft, a Minecraft-based text-game), both open-source and proprietary LLMs improve in accuracy when predicting programs with ReGAL functions. For CodeLlama-13B, ReGAL results in absolute accuracy increases of 11.5% on graphics, 26.1% on date understanding, and 8.1% on TextCraft, outperforming GPT-3.5 in two of three domains. Our analysis reveals ReGAL's abstractions encapsulate frequently-used subroutines as well as environment dynamics.[1]

1. Introduction

An increasing range of tasks can be tackled by using a large language model (LLM) to generate an executable program for a given query; this paradigm has been applied in computer vision (Surís et al., 2023; Gupta et al., 2018; Cho et al., 2023), robotics (Ahn et al., 2022; Singh et al., 2023), tool use (Schick et al., 2023; Lu et al., 2023; Qin et al., 2023), and general reasoning (Lyu et al., 2023). In all these cases, the overall program generation framework is the same: an


Q: A small 9 gon and a
Q: A small 9 gon to the left of a small 5 gon
generating programs from scratch
Q: 6-sided snowflake with a line and small 9 gon as arms
ReGAL: Discovers abstractions that can be reused as helper functions
Code Bank
fetch helpers
Q: 6 sided snowflake with a line and small 9 gon as arms

```
for i in range(9):
   forward(2)
   left(40.0)
penup();forward(8)
pendown()
for i in range(5):
   forward(2)
   left(72.0)
```

```
for j in range(6):
    forward(4)
    #Incorrect reasoning
    for i in range(9):
      forward(2)
      left(40.5)#Math error
    left(60.0)
```

```
draw_small_9_gon()
draw_small_5_gon()
draw...
draw...
```

```
for j in range(6):
  # Correct reasoning
  embed('forward(4)
  draw_small_9gon()',#Reuse
  locals())
  left(60.0)
```

Figure 1. ReGAL trains by refactoring primitive-only programs into abstractions that are verified and stored. These abstractions have two benefits: **Reusability**: Rewriting the same code multiple times leads to errors; **Abstraction**: ReGAL makes prediction easier by allowing matching between the query and the abstractions.

individual query is given (along with an instructive prompt) to an LLM, which produces a program that, when executed, yields the desired result. Crucially, each program is generated *independently* (as shown in Fig. 1), with no reference to other queries or programs, and is composed of *primitive* operations, i.e. the domain language's built-in operations. This approach has two major and related limitations:

1) Lack of Reusability: Each program is designed as a one-off script to solve a given example but is not reused by other examples. This increases redundancy and can result in unnecessary errors: for two examples requiring a shared subroutine, the model might correctly generate the subroutine in one example and make a mistake in the other. For instance, in Fig. 1 (top) although the "primitive-only" model had previously generated nonagons, it uses the wrong interior angle (40.5). ReGAL's `draw_small_9gon()` function, on the other hand, executes correctly.

2) Lack of Abstraction: Shared abstractions can improve accuracy by making skills more accessible to the model. When generating from primitives alone, the model must interpret the query and generate the correct mapping from the query to multiple primitives, requiring more reasoning. The model's *overall* task becomes easier when it uses interpretable abstractions, as it is choosing a function name from a library instead of reasoning from scratch. In Fig. 1 (bottom) a model augmented with abstractions can match the sub-query *"a small 9 gon"* to `draw_small_9gon()`; with this part of the task simplified, the model reasons correctly about the remaining code, while the primitive-only model fails to correctly embed the shape in a loop.

[*]Equal contribution [1]UNC Chapel Hill. Correspondence to: Elias Stengel-Eskin <esteng@cs.unc.edu>.

[1]Code available at https://github.com/esteng/regal_program_learning

Both limitations can be traced to a *lack of global context* as the model sees each example separately, so it lacks a mechanism for developing reusable abstractions. This differs greatly from how humans write code: generally, developers might start solving individual tasks with one-off solutions, but quickly begin to develop a library of shared abstractions and code snippets for related problems, thereby reducing the redundancy of their code, promoting efficiency and readability (McConnell, 2004; Downey, 2012). Furthermore, functions can be verified: once we have tested a function, we can rely on it in the future – something that is harder to do for ever-changing one-off code snippets. Such abstraction and verification is only sensible if the code synthesis process takes place over the course of multiple examples. In other words, if presented with a single, one-off task, there is no reason not to write a one-off script.

While abstraction offers numerous benefits, it comes with the risk of over-fitting, where a function tailored to a specific example loses its generalizability. For instance, a function like `draw_9gon_snowflake()` may perfectly match one example but fails to generalize. Conversely, `draw_small_9gon()` is a more versatile function applicable in various contexts. The ability to produce novel programs using primitive operations needs to be balanced with the benefits of encoding subroutines into reusable abstractions (O'Donnell, 2015). A similar balance between flexibility and efficiency appears in a variety of domains, including language (O'Donnell, 2015; Yang, 2016), biology (Futuyma & Moreno, 1988), manufacturing (Flynn & Jacobs, 1987), and programming (Ellis et al., 2021). To strike this balance in LLM-based program synthesis, we propose **Re**factoring for **G**eneralizable **A**bstraction **L**earning (ReGAL). ReGAL refines abstractions iteratively by refactoring programs as well as verifying, correcting, and pruning abstractions such that overly specific or incorrect programs are improved upon or removed. ReGAL relies on two key elements: a small set of programs using primitive operations (i.e. *primitive programs*) and an execution environment (e.g., Python). Importantly, we show ReGAL can learn from LLM-generated programs without requiring any human annotations.

ReGAL follows a familiar train-test paradigm: during ReGAL's modular training phase (see Fig. 2), it iteratively refactors a small set of *(query, program)* examples to produce a library of useful abstractions. ReGAL uses an LLM to write helper functions for a batch of examples, which are verified against the expected result; successful helper functions are added to the library and the refactored program serves as an example of the function's usage. ReGAL can take success feedback into account to correct and debug errors, and it periodically edits the helper functions to make them more generalizable or – if they cannot be made more generic – prunes functions that are overly specific. Note that the training is gradient-free, relying on a frozen LLM to refactor programs. In the testing phase, an LLM agent is tasked with predicting programs for test queries. The agent has access to ReGAL's library of helper functions and demonstrations of how to use them.

We demonstrate the broad applicability of ReGAL by testing it on three diverse datasets: LOGO (Ellis et al., 2021; Wong et al., 2021), a program induction task; a date reasoning task (Srivastava et al., 2022) known to challenge LLMs (Suzgun et al., 2022); and TextCraft (Prasad et al., 2023), a text-based game for crafting Minecraft objects. Across these tasks, ReGAL significantly improves the accuracy of the predicted programs from various LLMs – especially open-source LLMs – over a baseline that predicts primitive programs (i.e. programs without ReGAL's abstractions). For instance, CodeLlama-13B's (Roziere et al., 2023) accuracy increases by 11.5%, 26.1%, and 8.1% on LOGO, Date, and TextCraft respectively, surpassing larger models like GPT-3.5 (cf. Sec. 5). In Sec. 6, we show that ReGAL's abstractions are reusable across examples, encapsulate key domain functionalities, and we include an error analysis further highlighting the features making ReGAL effective. Sec. 6.3 reveals that ReGAL can improve over the baseline with minimal training examples, yielding major improvements even with a 50% reduced training set.

2. Related Work

Program Induction. Program induction involves learning a symbolic and programmatic mapping of inputs to outputs. Humans are adept at this kind of "rule-learning" (Marcus et al., 1999; Fürnkranz et al., 2012). ReGAL aims to learn a set of general functions that can be used to map inputs to outputs, i.e. a form of program induction. Ellis et al. (2021) present DreamCoder, a method for combining program induction and synthesis that uses a wake-sleep Bayesian learning method to learn programs. Wong et al. (2021) extend this work to incorporate language, using an alignment model as part of the joint model. Like Ellis et al. (2021), Grand et al. (2023) adopt a similar symbolic search procedure, but use an LLM to document abstractions. The symbolic search procedure used by this line of past work has relied on data structures for search that assume the domain language is λ-calculus (Lake et al., 2015; Ellis et al., 2021; Wong et al., 2021; Grand et al., 2023), which is not typically used for software development. In contrast, ReGAL uses an LLM-based search procedure, allowing us to use more flexible languages like Python, which are more commonly used by developers and better represented in pre-training data.

Program Synthesis and Tool Use. Tool use by LLMs (Schick et al., 2023; Mialon et al., 2023) refers to a form of program synthesis or semantic parsing where an LLM generates API calls to external tools (e.g. calculators, search functions, etc.). This formulation has also been applied to reasoning tasks (Lyu et al., 2023; Chen et al., 2022) as well as other domains such as computer vision (Surís et al., 2023; Gupta & Kembhavi, 2023; Cho et al., 2023), summarization (Saha et al., 2022), and robotics (Ahn et al., 2022; Singh et al., 2023; Huang et al., 2022; 2023). Past work has attempted to induce tools from examples. Cai et al. (2023) induce tools using an LLM for reasoning tasks from BigBench (Srivastava et al., 2022); unlike our work, their system generates one tool per task. While this can offer benefits for homogenous reasoning tasks (e.g. sorting words alphabetically), heterogenous tasks like the ones we explore require multiple functions. More akin to our work, Yuan et al. (2023) and Qian et al. (2023) induce multiple tools for vision and math tasks using an LLM-based framework which also includes retrieval-based parsing. In addition to focusing on different domains, we place an emphasis on efficiency, highlighting ReGAL's performance on open-source LLMs (both only consider proprietary models like GPT-3.5). We also differ in our focus on refactoring, and in the amount of information we provide to the refactoring model: unlike Yuan et al. (2023) and Qian et al. (2023), we do not provide in-context examples of the kinds of tools we want the model to create, investigating instead what abstractions the model builds without domain-specific guidance.

Induction in Interactive Domains. Wang et al. (2023) also induce functions in a Minecraft domain; however, theirs are written and stored based on one iteration. In contrast, our work refactors programs in a group and tests and refines them across the training process, showing generalization in multiple domains. Other prior work learns a library of abstractions for planning in embodied domains (Wong et al., 2023; Majumder et al., 2023). While we share a similar motivation, ReGAL operates in the space of generating executable programs instead of PDDL operators (Wong et al., 2023) or causal textual feedback (Majumder et al., 2023). Similarly, our work aligns with prior efforts in LLM-based task decomposition (Khot et al., 2023; Prasad et al., 2023), where skills are reused across multiple task instances. However, these approaches manually identify atomic skills and require the LLM to repeatedly execute skills from scratch. In contrast, ReGAL provides a way of automatically discovering such abstractions and reusing them via helper functions.

3. Methodology

In this section, we describe the overall pipeline of our method: **Re**factoring for **G**eneralizable **A**bstraction **L**earning (ReGAL). ReGAL consists of two phases: the *training* or induction stage where abstractions (i.e., helper functions) are learned, and the *testing* or synthesis stage, where abstractions are used to generate programs for test queries. During training, as illustrated in Fig. 2, ReGAL discovers reusable abstractions by generating candidate helper functions, validating their correctness, and debugging via editing and pruning of ineffective helper functions.

Given a set of demonstrations (q, p) of queries q and gold primitive programs p, we first preprocess the data to cluster examples based on query similarity, described in Sec. 3.1. The training stage then builds abstractions by refactoring primitive programs in batches (Sec. 3.2), while the testing stage solves new queries by generating programs that glue together the learned abstractions with primitive operations (Sec. 3.3). We use GPT-3.5 for all our training prompts; at test time we experiment with a range of LLMs, focusing on freely available open-source LLMs.

3.1. Preprocessing

Before training, we preprocess queries and programs (q, p) by (optionally) adding comments, clustering them into related batches, and sorting them by approximate difficulty.

Adding Comments. We optionally add comments to align the query with the primitive program, enabling the model to generate the correct abstractions. We present each (q, p) pair to GPT-3.5 independently, with a prompt asking the model to comment p based on q; we then verify that the commented code executes to the same result.

Clustering Data. In order to form abstractions that are *shared* between examples, the refactoring LLM requires a multi-instance scope, i.e. it must receive a batch of related (q, p) tuples at a time. We implement this by clustering examples using an embedding of the query q. Specifically, we embed each query using OpenAI's Ada embedding model (OpenAI, 2022) and hierarchically cluster the embeddings using Ward's agglomerative clustering algorithm (Ward Jr, 1963), implemented via Scikit-Learn (Pedregosa et al., 2011). This gives us a tree of related examples, which we topologically sort and group into batches of k related examples, where k is a hyperparameter (see Appendix C for all hyperparameter values).

Curriculum. Intuitively, shorter and easier programs should contain abstractions that can be reused in harder, more compositional programs, so we sort examples into a curriculum (Bengio et al., 2009). To approximate difficulty, we sort the batches based on the average length (in tokens) of their queries. See Appendix A.1 for preprocessing details.

3.2. Training

ReGAL's training data consists pairs of queries q and primitive programs p. The training phase outputs: 1) the *Code Bank* (C): the library of helper functions abstracted out during training and 2) the *Demo Bank* (D): examples of the functions being used. As shown in Fig. 2, the training phase is an iterative process where the LLM receives as input a batch of queries and primitive programs and then proposes helper functions that can be abstracted (Stage 1). For each batch, candidate helper functions are evaluated based on the

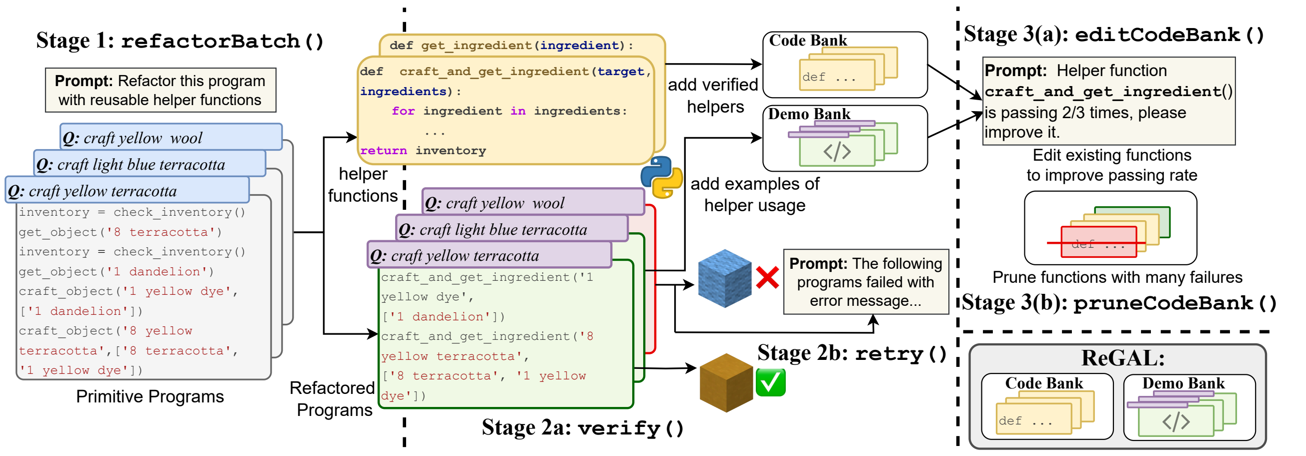

Stage 1: refactorBatch()
Prompt: Refactor this program with reusable helper functions
Q: craft yellow wool
Q: craft light blue terracotta
Q: craft yellow terracotta
inventory = check_inventory()
get_object('8 terracotta')
inventory = check_inventory()
get_object('1 dandelion')
craft_object('1 yellow dye', ['1 dandelion'])
craft_object('8 yellow terracotta',['8 terracotta', '1 yellow dye'])
Primitive Programs
def get_ingredient(ingredient):
def craft_and_get_ingredient(target, ingredients):
for ingredient in ingredients:
...
return inventory
helper functions
Refactored Programs
craft_and_get_ingredient('1 yellow dye', ['1 dandelion'])
craft_and_get_ingredient('8 yellow terracotta', ['8 terracotta', '1 yellow dye'])
Stage 2a: verify()
add verified helpers
Code Bank
def ...
Demo Bank
add examples of helper usage
Prompt: The following programs failed with error message...
Stage 2b: retry()
Stage 3(a): editCodeBank()
Prompt: Helper function craft_and_get_ingredient() is passing 2/3 times, please improve it.
Edit existing functions to improve passing rate
Prune functions with many failures
Stage 3(b): pruneCodeBank()
ReGAL:
Code Bank
Demo Bank

Figure 2. REGAL starts by refactoring a batch of primitive programs to develop a set of modified programs and helper functions (**Stage 1**). It then verifies the results of refactored programs, optionally retrying failed programs according to environment feedback. Useful helper functions are added to the Code Bank along with example usage added to the Demo Bank (**Stage 2**). Periodically, we edit and prune the Code Bank to improve its functions (**Stage 3**). At test time, REGAL agent has access to the Code Bank, the Demo Bank, and the remaining original programs. It is compared against a baseline agent which has access to a larger number of original programs.

correctness of the refactored programs that occur in (Stage 2a). After verification, failed examples are isolated into a second batch and re-attempted (Stage 2b). To improve the quality of the resultant Code Bank, we periodically edit helper functions after a fixed number of batches to improve their pass rate (over unit tests) and prune ineffective helper functions (Stage 3). At the end of training, the resultant library of helper functions – the code bank C – is stored for further use during testing along with successful demonstrations of refactored programs using helper functions – the demo bank D. Note that the training process can be repeated over multiple epochs. Below we describe each stage in detail, with the overall algorithm detailed in Algorithm 1.

Stage (1): Refactoring Examples (`refactorBatch`). The main module of the refactoring process takes as input a batch of examples, a set of instructions, and the current set of helper functions in the code bank (if any). It prompts the refactoring LLM for a set of new helper functions H^{new} along with refactored versions of each program that uses helpers from H^{new} when appropriate.

Stage (2a): Verification (`verify`). To avoid introducing errors, we need to verify the helper functions and refactored programs generated by the LLM by executing them and comparing the results to the original, i.e. determining if $\hat{p}() = p()$. The refactored program $(q, \hat{p})$ is stored as a demonstration for future use by the agent (cf. Sec. 3.3) if it passes verification. Only helper functions that pass verification are added to C. We also store a record of programs that *failed* verification, as these will be crucial in `editCodeBank()` and `pruneCodeBank()`, which improve existing functions and prune functions leading to failures.

Stage (2b): Feedback-based Retrial (`retry`). If a program fails to pass verification, we optionally retry the refactoring process. In a follow-up prompt, we present failed programs and their helper functions. We also include environment feedback from execution (i.e. the output or error message produced).[2] The refactoring LLM then produces a new version of each failed program; these are verified and their helpers are added to C if correct.

Stage (3a): Editing Code Bank (`editCodeBank`). From the `verify()` module, some helper functions fail to pass all unit tests because they contain incorrect abstractions; for example, a function like `draw_triangle()` might start with a hardcoded value for a small size, leading it to fail on medium triangles. To update such functions, we construct a prompt for each function in D that shows the LLM passing and failing unit tests and asks it to propose edits to the function; this occurs once every `editEvery` iterations, where `editEvery` is a hyperparameter. We replace a function if it passes more unit tests after editing.

Stage (3b): Pruning Code Bank (`pruneCodeBank`). In this module, we prune helper functions added to C that fail a majority of unit tests and cannot be improved further via editing. For each function, we derive a score based on the success rate of programs using the function; we set a threshold below which functions are pruned (shared by all domains). See Appendix A.2 for further details.

We use the dev set to select hyperparameter values, reported in Appendix C. All prompts can be found in Appendix D.

3.3. Testing

At test time, we deploy a program synthesis – or semantic parsing – *agent* that makes predictions for test examples, one at a time. Unlike related work on using learned tools

[2]We do not include the output for LOGO as it is an image.

(Yuan et al., 2023; Qian et al., 2023; Wong et al., 2023), we explore a variety of open-source LLMs, in addition to a black-box LLM (GPT-3.5). Following effective strategies in semantic parsing and in-context learning (Shin & Van Durme, 2022; Roy et al., 2022; Bogin et al., 2023; Liu et al., 2022; Yasunaga et al., 2023), for each test example, the agent constructs a prompt with in-context learning (ICL) examples retrieved from a training corpus, followed by a test query. The examples are retrieved from the training data using vector similarity between the training queries and the test query. Further details in Appendix A.3.

ReGAL-augmented Agent. Our agent has access to the *training data* and *code bank*, as well as examples of refactored programs in the *demo bank*. The ICL budget (10 examples for all experiments) is split between primitive training examples and refactored ones.[3] In addition to these demonstrations, the augmented agent retrieves up to 20 relevant helper functions from the code bank, where relevance is measured by the similarity between the query and the function name and description. These helper functions are concatenated into the prompt. The final input is a prompt containing the instructions, the retrieved helper functions, the mixed ICL examples, and the test query. To encourage the model to use helper functions, we include a ReAct-style prompt (Yao et al., 2023) that first asks the model to *think* about which functions might be relevant based on the query and then generate the code.[4]

4. Experimental Setup

4.1. Datasets

We explore three datasets: LOGO, Date understanding, and TextCraft. A common thread through these datasets is that they contain heterogenous problems requiring multiple helper functions as opposed to problems like sorting, which are challenging for LLMs but can be solved with a single function (Dziri et al., 2023). Statistics for the datasets are given in Table 4. Note that the number of primitives reflects functions not built into Python – all models also have access to all Python functions. See Appendix A.5 for further details about each dataset and its primitive operations.

LOGO. LOGO is based on the Logo Turtle graphics domain-specific language (Abelson & DiSessa, 1986), with which basic graphics can be drawn by controlling a pen (the "turtle") that draws as it moves through space, using commands like `forward(dist)` and `left(theta)`. The data we use is based on the Ellis et al. (2021)'s LOGO dataset, re-annotated by Wong et al. (2021). For easier prediction by LLMs, we parse the data into abstract syntax trees and write a set of rules for translating these into Python; we release this rewritten data. We use the "small" train/test splits (200/111) from Wong et al. (2021) and take 100 dev examples from the "large" train set.

Date. We use the date understanding task from BigBench-Hard (Srivastava et al., 2022; Suzgun et al., 2022), which consists of short word problems requiring date understanding. We obtain silver programs from Lyu et al. (2023)'s predictions. Specifically, we split their predicted programs from GPT-3.5 into train, dev, and test splits (66/113/180) and filter the train split by correctness.

TextCraft. To explore the utility of ReGAL in LLMs-as-agent settings (Liu et al., 2023), we use the TextCraft dataset (Prasad et al., 2023) that requires an agent to craft Minecraft items within a text-only environment (Côté et al., 2019). Each task instance in TextCraft consists of a goal (query) and a series of 10 crafting commands that contain recipes for related items including distractors. Unlike Prasad et al. (2023), who use TextCraft in an interactive setting where the LLM agent receives textual feedback from the environment at each step, we ask the agent to generate a single Python program for executing the entire task at once, making the task *more challenging*. We evaluate on the depth 2 split of the test set used in Prasad et al. (2023) while using a subset of depth 1 recipe examples for our dev set, giving us a train/dev/test split of 190/50/77.

4.2. Baselines

Baselines from Prior Work. We compare ReGAL against relevant external baselines from past work. However, note that multiple methodological differences in our work, like the use of ICL examples and the format of the output programming language, give our agent an inherent advantage over these baselines. Thus, we refer to these numbers primarily to highlight the strength of our baseline agent. For LOGO, we use the "offline synthesis" numbers reported by Grand et al. (2023), which resemble our train/test setting; however, we note that Grand et al. (2023) predict programs in their original Lisp format and use a different agent model. For the Date dataset, we run Lyu et al. (2023)'s Faithful-CoT method on our custom test split using `gpt-3.5-turbo`. While the output format and models used are the same, both our baseline and ReGAL use retrieved examples for in-context learning, while Lyu et al. (2023) do not. Furthermore, our ICL examples are based on filtered programs generated by Lyu et al. (2023), leading to better performance even from our baseline agent. Finally, for TextCraft we rerun Prasad et al. (2023)'s baseline – based on ReAct (Yao et al., 2023) – on the depth-2 test set of TextCraft. Here, we use `gpt-3.5-turbo-instruct`, as Prasad et al. (2023) found it to outperform `gpt-3.5-turbo`.

[3] We found it necessary to keep some primitive programs as ICL examples, as not all test queries can be handled by helper functions. We treat this ratio of primitive and refactored programs as a hyperparameter.

[4] Without these additional "thought" statements, we found the augmented agent rarely uses any helper functions.

	Method	Agent	Acc.
LOGO	LILO	Code-davinci	41.1
	Primitive Programs	CodeLlama-13B	45.6
	ReGAL Programs	CodeLlama-13B	**57.1**
Date	Faithful-CoT	GPT-3.5-turbo	83.3
	Primitive Programs	GPT-3.5-turbo	88.9
	ReGAL Programs	GPT-3.5-turbo	**90.2**
TC[†]	ReAct	GPT-3.5-turbo	25.6
	Primitive Programs	CodeLlama-34B	22.2
	ReGAL Programs	CodeLlama-34B	**30.8**

Table 1. Comparison to relevant past work in each domain. TC[†] denotes the TextCraft dataset.

	LOGO		Date		TextCraft	
Agent	**Prim.**	**ReGAL**	**Prim.**	**ReGAL**	**Prim.**	**ReGAL**
CodeLlama-7B	$34.5_{\pm 1.3}$	$34.5_{\pm 1.6}$	$52.4_{\pm 0.7}$	$55.2_{\pm 1.4}$	$12.8_{\pm 1.3}$	$16.7_{\pm 1.3}$
CodeLlama-13B	$45.6_{\pm 0.3}$	$\mathbf{57.1}_{\pm 0.6}$	$42.8_{\pm 2.0}$	$68.9_{\pm 1.6}$	$18.8_{\pm 0.7}$	$26.9_{\pm 2.2}$
CodeLlama-34B	$50.2_{\pm 0.8}$	$50.8_{\pm 0.6}$	$47.2_{\pm 1.5}$	$68.5_{\pm 2.1}$	$22.2_{\pm 0.7}$	$\mathbf{30.8}_{\pm 1.3}$
Lemur-70B	$44.1_{\pm 1.4}$	$56.8_{\pm 0.9}$	$68.2_{\pm 0.4}$	$70.5_{\pm 0.6}$	$15.7_{\pm 1.7}$	$23.5_{\pm 2.1}$
GPT-3.5-turbo	$36.9_{\pm 1.6}$	$49.3_{\pm 1.1}$	$88.9_{\pm 0.3}$	$\mathbf{90.2}_{\pm 0.5}$	$15.4_{\pm 1.3}$	$18.4_{\pm 2.0}$

Table 2. Accuracy of baseline agents predicting primitive programs (Prim.) and those augmented with ReGAL helper functions (3 random seeds). Across domains and models, ReGAL improves over a strong baseline programming agent with access to the same number of ICL examples.

Baseline Programming Agent. For a more direct comparison, we implement a baseline agent that has access to all the same data as ReGAL but does not use abstractions, thus directly testing the role of ReGAL abstractions in performance. Our baseline agent retrieves primitive programs from the training data; note that this is exactly *the same dataset* used for refactoring, i.e. the baseline LOGO agent retrieves demonstrations from the LOGO training examples. The input to the baseline agent is a prompt with the same overall instructions as the ReGAL agent (including a description of the primitives), the ICL examples, and the test query; the output is a program for the test query. We use a fixed budget of 10 ICL examples so that the baseline agent sees exactly as many demonstrations as the ReGAL agent.

5. Results

Comparison to External Baselines. Table 1 shows a comparison of the baseline and ReGAL agents the external baselines from prior work. We first note that ReGAL outperforms the baselines in all cases. Furthermore, because of the methodological differences detailed in Sec. 4, our baseline "primitive-only" agent – equipped with ICL examples and using a code-specific LLM – also outperforms past baselines on LOGO and Date. On TextCraft, the ReAct baseline from Prasad et al. (2023) has an advantage in that it receives environmental feedback, while our baseline does not. Nevertheless, even *without feedback* ReGAL outperforms ReAct. Thus, we compare primarily against our baseline agent, as this provides a direct measure of the impact of abstractions (rather than the other changes made).

ReGAL outperforms the baseline agent. Table 2 shows ReGAL's performance compared to the baseline agent using primitive programs (described in Sec. 4.2). Overall, for each model type, ReGAL generally outperforms the baseline by a wide margin; for example, ReGAL provides CodeLlama-13B a 11.5% boost on LOGO, allowing it to outperform much larger models. Across datasets, CodeLlama-13B generally benefits most from ReGAL abstractions. Table 2 also shows that large models also benefit

Ablation	LOGO	Date	TextCraft
ReGAL	55.0	77.0	34.12
– `retry`	48.3	51.9	30.43
– curriculum	36.3	56.6	28.78
– `pruneCodeBank`	52.0	65.5	25.26
– `editCodeBank`	53.3	69.6	27.35

Table 3. Ablations of each optional ReGAL component tested on dev splits with CodeLlama-13B. To remove the curriculum, we randomly shuffle example clusters instead of presenting them in order of shortest query to longest query.

from ReGAL, with large gains for Lemur-70B and GPT-3.5 on LOGO and TextCraft. Finally, the largest models are not necessarily the best: on LOGO and TextCraft, GPT-3.5 is outperformed by open-source models, especially after augmentation, e.g. CodeLlama-13B with ReGAL abstractions is able to outperform GPT-3.5 *without* abstractions by 19.2% (it also outperforms GPT-3.5 with abstractions). Thus, by running ReGAL's training process on only ∼200 examples, we can bring a much smaller open-source model's accuracy far beyond that of a (likely) much larger system.

Ablations. In Sec. 3.2 we describe ReGAL's multiple components; here we determine the utility of each by removing each one in isolation. Table 3 shows the results of these ablations. We use the CodeLlama-13B agent due to the size of ReGAL's impact on it across tasks. We average the performance across 3 seeds. Table 3 shows that each component contributes to performance, with drops when any is removed. Across datasets, the largest performance decreases come with the removal of retrials and with removal of the curriculum. Retrials can not only increase the number of useful helper functions but can also help increase the number of examples in the Demo Bank. Replacing the curriculum with a random ordering also severely hurts performance, e.g. leading to an 18.7% drop on LOGO.

ReGAL learns general, reusable functions. In Sec. 1, we stressed the importance of reusability. Specifically, generating programs without shared abstractions means that the model has to re-generate subprograms that could be reused across multiple test instances. We argue that REGAL improves over this paradigm by learning *shared* abstractions. The results in Table 2 indicate that REGAL offers large improvements over a baseline agent that lacks abstractions. Here, we verify that the abstractions learned are reusable, i.e. shared. Fig. 3 shows the number of times the top-5 most common REGAL functions are called in test programs produced by the CodeLlama-13B agent. Across all datasets, we see that the helper functions learned by REGAL are commonly reused, with the most relative reuse in TextCraft. Appendix B.1 shows examples of these common functions.

6. Analysis

6.1. What kinds of programs are discovered?

To further examine what kinds of helper functions are discovered, we examine the most frequent helper functions for each domain from Fig. 3, summarizing the results below. Refer to Appendix B.1 for function code. We find that distinct trends emerge across domains.

For LOGO, REGAL discovers functions that encapsulate different types of shapes. This is expected, as the LOGO data was generated with these functionalities in mind, i.e. the larger shapes are composed of objects like semicircles, pentagons, and circles. For Date, REGAL tends to encapsulate single operations, prioritizing interpretability with names like `get_date_one_year_ago()`. While seemingly less complex than LOGO's functions, this approach aligns with the importance of function naming in synthesis procedures, as highlighted by Grand et al. (2023). In TextCraft, the functions uncovered by REGAL are more complex and reflect the dynamics of the game. Specifically, the functions include conditional statements for checking ingredients, reflecting the fact that in TextCraft, having the correct crafting ingredients is a prerequisite for crafting an object (see `craft_and_get_ingredient()` in Fig. 2 and Table 10, which is taken from the learned code bank C).

6.2. Error Analysis

To better understand how REGAL aids program generation and also examine cases where it does not help, we perform a two-part error analysis. First, we examine cases where the REGAL-augmented program was correct and the baseline agent's primitive program was incorrect. We then examine the opposite set of cases, where the baseline program was correct but the REGAL program was incorrect.

Fig. 1 shows the first kind of comparison on LOGO using the CodeLlama-13B model, where we qualitatively show an actual example that highlights the benefits of reuse and abstraction. The baseline program makes an error in calculating the polygon's interior angle when generating the program from scratch. This is avoided by the REGAL agent,

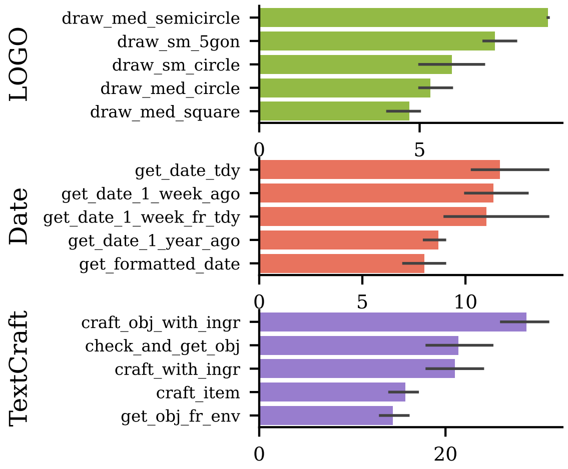


Figure 3. Function usage for CodeLlama-13B for the top-5 most common helpers. Functions are reused across programs.

which simply uses a verified helper to generate the polygon correctly. The example also illustrates the importance of abstraction: as queries become more complex, generating a solution from scratch becomes more challenging. The baseline program reasons incorrectly about code outside of the shape, failing to use `embed()` correctly. Meanwhile, the augmented program offloads reasoning about the shape to an easily-matched function, and is able to correctly use `embed()`. To quantify these trends, we manually inspect the output of the baseline CodeLlama-13B on LOGO on the 25 cases where the REGAL agent was correct, categorizing them into errors involving reasoning (first example in Fig. 1) and shape-internal errors (second example); we find 16 reasoning and 9 shape errors. We also examine REGAL's failure modes by manually inspecting all cases where the augmented agent failed and the baseline succeeded, again using CodeLlama-13B on LOGO; there are 13 such cases. We categorize them into three types:

- **Incorrect connector code**: (7 exs.) the program fails due to mistakes in the primitive operations or control flow.
- **Incorrect/undefined function**: (4 exs.) the code refers to non-existent functions, or incorrectly calls a function similar to the correct one.
- **Verification failure**: (2 exs.) the program was correct but the verification function gives a false negative.[5]

Thus, the most common error is a failure to predict primitive operations; here, the REGAL agent is at a disadvantage w.r.t. the baseline agent, since both have the same ICL budget. The baseline agent sees 10 ICL examples with *only* primitive code, while the REGAL agent sees 5 primitive examples and 5 Demo Bank examples.

[5]For example, for *"a small square next to a small 6 gon"* the agent generates the hexagon to the left of the square, where in the reference it is to the right.

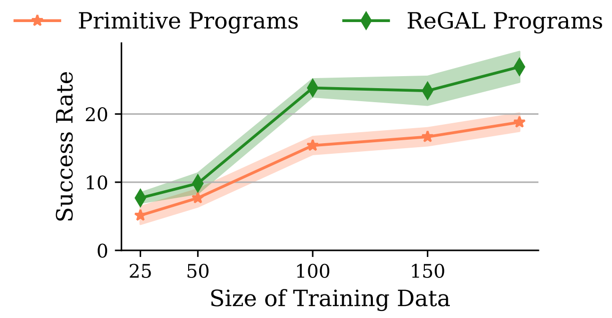


Figure 4. ReGAL programs yield a higher success rate (accuracy) compared to primitive programs on TextCraft for different sizes of training set X using CodeLlama-13B.

6.3. Sample Efficiency

As mentioned in Sec. 4.2, both baseline and ReGAL agents rely on demonstrations of queries and gold programs (X) to retrieve most similar ICL examples. Additionally, ReGAL uses the same demonstrations to learn helper functions in the code bank C. We now study how the performance of both agents scales with the size of annotated gold programs in train set X using the CodeLlama-13B model on TextCraft. From Fig. 4, we observe that the ReGAL agent consistently outperforms the baseline agent as we vary the number of training examples. Notably, helper functions learned by ReGAL yield a 2.56% improvement with as few as 25 demonstrations and an 8.45% improvement with nearly half the size of the train set used in Sec. 5. Additionally, the performance of both the baseline and ReGAL agents improves as the number of demonstrations increases. This is expected as both agents benefit from the retrieval of demonstrations similar to the test query as the training set becomes larger and consequently more diverse.

7. Discussion and Conclusion

Fixed vs. Variable Costs. In Sec. 5, ReGAL was especially effective for open-source LLMs like CodeLlama. This result is encouraging, as it indicates that we can bring freely available and open-source models up to at least the same performance as a proprietary, closed-source model (if not more) using ReGAL abstractions. Thus, we can convert a variable cost – running an LLM on test data, which scales linearly with the size of the test set – into the fixed cost of running ReGAL to learn a library of helper functions.

Connections to Semantic Parsing. Executable semantic parsing (Winograd, 1972; Zelle & Mooney, 1996) typically involves mapping queries to a domain-specific language (DSL); the DSL specifies a set of abstractions that are useful for a particular goal, e.g. SQL operations for querying databases, or spatial abstractions for robotics. These DSLs are typically defined by a human; one way to view ReGAL is as a way of *learning* a DSL on top of an extremely general set of primitives. One of the benefits of ReGAL is its generality: on three different domains, it is able to learn useful abstractions without human intervention, while, in a standard semantic parsing setting, these abstractions would need to be manually specified.

Connections to Hierarchical Reinforcement Learning. One way to view the functions discovered by ReGAL is as low-level policies – or skills – composed of primitive actions. In this view, our system is similar to hierarchical reinforcement learning (HRL; Barto & Mahadevan, 2003), where tasks are factorized into skill selection and skills themselves. In hierarchical frameworks, there is typically a high-level controller policy that selects lower-level policies. In our setting, the agent LLM acts as a controller, selecting from a set of skills, while ReGAL's training stage is responsible for discovering a useful set of skills; this is akin to option discovery (Sutton et al., 1999), where closed-loop policies for specific skills are learned from a reward signal. While ReGAL has a similar hierarchical structure, it differs from HRL in that ReGAL's skills are symbolic, interpretable, and editable, as opposed to HRL skill policies, which typically have none of these features.

Limitations. As mentioned in connection to HRL, the functions ReGAL learns are code-based. This can make them less flexible than functions parameterized by neural networks (e.g. Andreas et al. (2016)), especially in domains where the environment can change dynamically. For instance, consider a navigation domain where an agent needs to get to the kitchen in various homes; depending on the primitive actions, a function that succeeds in one home will likely fail in another. However, ReGAL's verification-based pruning means that no function would be discovered for this navigation task. Relatedly, not every domain has reusable abstractions, and not every example stands to benefit from them. In many cases, the primitives for a domain are the right level of abstraction for that domain, e.g. if they already form a DSL. The abstractions are also not always ideal; in Appendix B.1 we see that ReGAL's abstractions are not necessarily the same as those a human would choose, e.g. a human would likely write a function like `draw_5gon(size)` rather than `draw_small_5gon()`.

Conclusion. We introduce ReGAL, a gradient-free approach to learning abstractions from a small set of examples. Our experimental results show that abstractions from ReGAL improve the accuracy of programs predicted by a variety of LLMs across three diverse domains. Furthermore, ReGAL abstractions are reusable and general, allowing them to be applied across examples for a given task. In our analysis, we find that the functions learned by ReGAL codify commonly-used subroutines as well as task dynamics. Our error analysis indicates that ReGAL's improvements come both from function reuse as well as simplification of the reasoning involved in program prediction.

8. Broader Impacts

Our work aims to learn symbolic functions given a set of demonstrations; this has the potential to improve LLM predictions not only in terms of accuracy but also in terms of interpretability and trustworthiness. Unlike the mechanisms of an LLM itself, a Python function is natively interpretable by a human and can be debugged. Furthermore, results obtained by executing such a function are inherently faithful, in that we can identify the exact trace of operations that generated the result (Lyu et al., 2023). Our work does not have more potential for negative use than typical LLM-based systems and is subject to the biases inherent to these models and the datasets they are trained on (Weidinger et al., 2021). As with any system generating code, particular caution should be taken before executing snippets with the potential to damage the execution environment (Ruan et al., 2023).

Acknowledgements

We thank Yichen Jiang, Justin Chen, Jaehong Yoon, and Swarnadeep Saha for their valuable feedback on the paper. This work was supported by NSF-AI Engage Institute DRL-2112635, DARPA Machine Commonsense (MCS) Grant N66001-19-2-4031, and Accelerate Foundation Models Research program. The views contained in this article are those of the authors and not of the funding agency.

References

Abelson, H. and DiSessa, A. *Turtle geometry: The computer as a medium for exploring mathematics*. MIT press, 1986.

Ahn, M., Brohan, A., Brown, N., Chebotar, Y., Cortes, O., David, B., Finn, C., Fu, C., Gopalakrishnan, K., Hausman, K., Herzog, A., Ho, D., Hsu, J., Ibarz, J., Ichter, B., Irpan, A., Jang, E., Ruano, R. J., Jeffrey, K., Jesmonth, S., Joshi, N., Julian, R., Kalashnikov, D., Kuang, Y., Lee, K.-H., Levine, S., Lu, Y., Luu, L., Parada, C., Pastor, P., Quiambao, J., Rao, K., Rettinghouse, J., Reyes, D., Sermanet, P., Sievers, N., Tan, C., Toshev, A., Vanhoucke, V., Xia, F., Xiao, T., Xu, P., Xu, S., Yan, M., and Zeng, A. Do as i can and not as i say: Grounding language in robotic affordances. In *arXiv preprint arXiv:2204.01691*, 2022.

Andreas, J., Rohrbach, M., Darrell, T., and Klein, D. Neural module networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 39–48, 2016.

Barto, A. G. and Mahadevan, S. Recent advances in hierarchical reinforcement learning. *Discrete event dynamic systems*, 13(1-2):41–77, 2003.

Bengio, Y., Louradour, J., Collobert, R., and Weston, J. Curriculum learning. In *Proceedings of the 26th annual international conference on machine learning*, pp. 41–48, 2009.

Bogin, B., Gupta, S., Clark, P., and Sabharwal, A. Leveraging code to improve in-context learning for semantic parsing. *arXiv preprint arXiv:2311.09519*, 2023.

Bowers, M., Olausson, T. X., Wong, L., Grand, G., Tenenbaum, J. B., Ellis, K., and Solar-Lezama, A. Top-down synthesis for library learning. *Proceedings of the ACM on Programming Languages*, 7(POPL):1182–1213, 2023.

Cai, T., Wang, X., Ma, T., Chen, X., and Zhou, D. Large language models as tool makers. *arXiv preprint arXiv:2305.17126*, 2023.

Chen, W., Ma, X., Wang, X., and Cohen, W. W. Program of thoughts prompting: Disentangling computation from reasoning for numerical reasoning tasks. *arXiv preprint arXiv:2211.12588*, 2022.

Cho, J., Zala, A., and Bansal, M. Visual programming for text-to-image generation and evaluation. *Thirty-seventh Conference on Neural Information Processing Systems (NeurIPS)*, 2023.

Côté, M.-A., Kádár, A., Yuan, X., Kybartas, B., Barnes, T., Fine, E., Moore, J., Hausknecht, M., El Asri, L., Adada, M., et al. Textworld: A learning environment for text-based games. In *Computer Games: 7th Workshop, CGW 2018, Held in Conjunction with the 27th International Conference on Artificial Intelligence, IJCAI 2018, Stockholm, Sweden, July 13, 2018, Revised Selected Papers 7*, pp. 41–75. Springer, 2019.

Downey, A. *Think python*. " O'Reilly Media, Inc.", 2012.

Dziri, N., Lu, X., Sclar, M., Li, X. L., Jian, L., Lin, B. Y., West, P., Bhagavatula, C., Bras, R. L., Hwang, J. D., et al. Faith and fate: Limits of transformers on compositionality. *arXiv preprint arXiv:2305.18654*, 2023.

Ellis, K., Wong, L., Nye, M., Sablé-Meyer, M., Morales, L., Hewitt, L., Cary, L., Solar-Lezama, A., and Tenenbaum, J. B. Dreamcoder: Bootstrapping inductive program synthesis with wake-sleep library learning. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pp. 835–850, 2021.

Flynn, B. B. and Jacobs, F. R. Applications and implementation: an experimental comparison of cellular (group technology) layout with process layout. *Decision Sciences*, 18(4):562–581, 1987.

Fürnkranz, J., Gamberger, D., and Lavrač, N. *Foundations of rule learning*. Springer Science & Business Media, 2012.

Futuyma, D. J. and Moreno, G. The evolution of ecological specialization. *Annual review of Ecology and Systematics*, 19(1):207–233, 1988.

Grand, G., Wong, L., Bowers, M., Olausson, T. X., Liu, M., Tenenbaum, J. B., and Andreas, J. Learning interpretable libraries by compressing and documenting code. In *Intrinsically-Motivated and Open-Ended Learning Workshop@ NeurIPS2023*, 2023.

Gupta, S., Shah, R., Mohit, M., Kumar, A., and Lewis, M. Semantic parsing for task oriented dialog using hierarchical representations. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pp. 2787–2792, 2018.

Gupta, T. and Kembhavi, A. Visual programming: Compositional visual reasoning without training. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 14953–14962, 2023.

Huang, W., Abbeel, P., Pathak, D., and Mordatch, I. Language models as zero-shot planners: Extracting actionable knowledge for embodied agents. *arXiv preprint arXiv:2201.07207*, 2022.

Huang, W., Wang, C., Zhang, R., Li, Y., Wu, J., and Fei-Fei, L. Voxposer: Composable 3d value maps for robotic manipulation with language models. In *Conference on Robot Learning*, pp. 540–562. PMLR, 2023.

Khot, T., Trivedi, H., Finlayson, M., Fu, Y., Richardson, K., Clark, P., and Sabharwal, A. Decomposed prompting: A modular approach for solving complex tasks. In *The Eleventh International Conference on Learning Representations*, 2023.

Lake, B. M., Salakhutdinov, R., and Tenenbaum, J. B. Human-level concept learning through probabilistic program induction. *Science*, 350(6266):1332–1338, 2015.

Liu, J., Shen, D., Zhang, Y., Dolan, B., Carin, L., and Chen, W. What makes good in-context examples for GPT-3? In Agirre, E., Apidianaki, M., and Vulić, I. (eds.), *Proceedings of Deep Learning Inside Out (DeeLIO 2022): The 3rd Workshop on Knowledge Extraction and Integration for Deep Learning Architectures*, pp. 100–114, Dublin, Ireland and Online, May 2022. Association for Computational Linguistics.

Liu, X., Yu, H., Zhang, H., Xu, Y., Lei, X., Lai, H., Gu, Y., Ding, H., Men, K., Yang, K., et al. Agentbench: Evaluating llms as agents. *arXiv preprint arXiv:2308.03688*, 2023.

Lu, P., Peng, B., Cheng, H., Galley, M., Chang, K.-W., Wu, Y. N., Zhu, S.-C., and Gao, J. Chameleon: Plug-and-play compositional reasoning with large language models. *arXiv preprint arXiv:2304.09842*, 2023.

Lyu, Q., Havaldar, S., Stein, A., Zhang, L., Rao, D., Wong, E., Apidianaki, M., and Callison-Burch, C. Faithful chain-of-thought reasoning. *arXiv preprint arXiv:2301.13379*, 2023.

Majumder, B. P., Mishra, B. D., Jansen, P., Tafjord, O., Tandon, N., Zhang, L., Callison-Burch, C., and Clark, P. Clin: A continually learning language agent for rapid task adaptation and generalization. *arXiv preprint arXiv:2310.10134*, 2023.

Marcus, G. F., Vijayan, S., Bandi Rao, S., and Vishton, P. M. Rule learning by seven-month-old infants. *Science*, 283 (5398):77–80, 1999.

McConnell, S. *Code complete*. Pearson Education, 2004.

Mialon, G., Dessì, R., Lomeli, M., Nalmpantis, C., Pasunuru, R., Raileanu, R., Rozière, B., Schick, T., Dwivedi-Yu, J., Celikyilmaz, A., et al. Augmented language models: a survey. *arXiv preprint arXiv:2302.07842*, 2023.

O'Donnell, T. J. *Productivity and reuse in language: A theory of linguistic computation and storage*. MIT Press, 2015.

OpenAI. New and improved embedding model, 2022. URL https://openai.com/blog/new-and-improved-embedding-model.

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., and Duchesnay, E. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

Prasad, A., Koller, A., Hartmann, M., Clark, P., Sabharwal, A., Bansal, M., and Khot, T. Adapt: As-needed decomposition and planning with language models. *arXiv preprint arXiv:2311.05772*, 2023.

Qian, C., Han, C., Fung, Y., Qin, Y., Liu, Z., and Ji, H. Creator: Tool creation for disentangling abstract and concrete reasoning of large language models. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pp. 6922–6939, 2023.

Qin, Y., Liang, S., Ye, Y., Zhu, K., Yan, L., Lu, Y., Lin, Y., Cong, X., Tang, X., Qian, B., et al. Toolllm: Facilitating large language models to master 16000+ real-world apis. *arXiv preprint arXiv:2307.16789*, 2023.

Roy, S., Thomson, S., Chen, T., Shin, R., Pauls, A., Eisner, J., and Van Durme, B. Benchclamp: A benchmark for evaluating language models on semantic parsing. *arXiv preprint arXiv:2206.10668*, 2022.

Roziere, B., Gehring, J., Gloeckle, F., Sootla, S., Gat, I., Tan, X. E., Adi, Y., Liu, J., Remez, T., Rapin, J., et al. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*, 2023.

Ruan, Y., Dong, H., Wang, A., Pitis, S., Zhou, Y., Ba, J., Dubois, Y., Maddison, C., and Hashimoto, T. Identifying the risks of LM agents with an LM-emulated sandbox. In *NeurIPS 2023 Foundation Models for Decision Making Workshop*, 2023.

Saha, S., Zhang, S., Hase, P., and Bansal, M. Summarization programs: Interpretable abstractive summarization with neural modular trees. In *The Eleventh International Conference on Learning Representations*, 2022.

Schick, T., Dwivedi-Yu, J., Dessì, R., Raileanu, R., Lomeli, M., Zettlemoyer, L., Cancedda, N., and Scialom, T. Toolformer: Language models can teach themselves to use tools. *arXiv preprint arXiv:2302.04761*, 2023.

Shin, R. and Van Durme, B. Few-shot semantic parsing with language models trained on code. In *Proceedings of the 2022 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pp. 5417–5425, 2022.

Singh, I., Blukis, V., Mousavian, A., Goyal, A., Xu, D., Tremblay, J., Fox, D., Thomason, J., and Garg, A. ProgPrompt: Generating situated robot task plans using Large Language Models. In *2023 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 11523–11530. IEEE, 2023.

Srivastava, A., Rastogi, A., Rao, A., Shoeb, A. A. M., Abid, A., Fisch, A., Brown, A. R., Santoro, A., Gupta, A., Garriga-Alonso, A., et al. Beyond the imitation game: Quantifying and extrapolating the capabilities of language models. *arXiv preprint arXiv:2206.04615*, 2022.

Surís, D., Menon, S., and Vondrick, C. ViperGPT: Visual inference via Python execution for reasoning. *arXiv preprint arXiv:2303.08128*, 2023.

Sutton, R. S., Precup, D., and Singh, S. Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning. *Artificial intelligence*, 112(1-2): 181–211, 1999.

Suzgun, M., Scales, N., Schärli, N., Gehrmann, S., Tay, Y., Chung, H. W., Chowdhery, A., Le, Q. V., Chi, E. H., Zhou, D., et al. Challenging big-bench tasks and whether chain-of-thought can solve them. *arXiv preprint arXiv:2210.09261*, 2022.

Wang, G., Xie, Y., Jiang, Y., Mandlekar, A., Xiao, C., Zhu, Y., Fan, L., and Anandkumar, A. Voyager: An open-ended embodied agent with large language models. *arXiv preprint arXiv:2305.16291*, 2023.

Ward Jr, J. H. Hierarchical grouping to optimize an objective function. *Journal of the American statistical association*, 58(301):236–244, 1963.

Weidinger, L., Mellor, J., Rauh, M., Griffin, C., Uesato, J., Huang, P.-S., Cheng, M., Glaese, M., Balle, B., Kasirzadeh, A., et al. Ethical and social risks of harm from language models. *arXiv preprint arXiv:2112.04359*, 2021.

Winograd, T. Understanding natural language. *Cognitive psychology*, 3(1):1–191, 1972.

Wolf, T., Debut, L., Sanh, V., Chaumond, J., Delangue, C., Moi, A., Cistac, P., Rault, T., Louf, R., Funtowicz, M., Davison, J., Shleifer, S., von Platen, P., Ma, C., Jernite, Y., Plu, J., Xu, C., Scao, T. L., Gugger, S., Drame, M., Lhoest, Q., and Rush, A. M. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pp. 38–45, Online, October 2020. Association for Computational Linguistics.

Wong, L., Ellis, K. M., Tenenbaum, J., and Andreas, J. Leveraging language to learn program abstractions and search heuristics. In *International Conference on Machine Learning*, pp. 11193–11204. PMLR, 2021.

Wong, L., Mao, J., Sharma, P., Siegel, Z. S., Feng, J., Korneev, N., Tenenbaum, J. B., and Andreas, J. Learning adaptive planning representations with natural language guidance. *arXiv preprint arXiv:2312.08566*, 2023.

Yang, C. *The price of linguistic productivity: How children learn to break the rules of language*. MIT press, 2016.

Yao, S., Zhao, J., Yu, D., Du, N., Shafran, I., Narasimhan, K. R., and Cao, Y. React: Synergizing reasoning and acting in language models. In *The Eleventh International Conference on Learning Representations*, 2023.

Yasunaga, M., Chen, X., Li, Y., Pasupat, P., Leskovec, J., Liang, P., Chi, E. H., and Zhou, D. Large language models as analogical reasoners. *arXiv preprint arXiv:2310.01714*, 2023.

Yuan, L., Chen, Y., Wang, X., Fung, Y. R., Peng, H., and Ji, H. Craft: Customizing llms by creating and retrieving from specialized toolsets. *arXiv preprint arXiv:2309.17428*, 2023.

Zelle, J. M. and Mooney, R. J. Learning to parse database queries using inductive logic programming. In *Proceedings of the national conference on artificial intelligence*, pp. 1050–1055, 1996.

Appendix

A. Methods

A.1. Preprocessing

Adding comments. To add comments, we first use a zero-shot prompt to break the query down into its constituent parts; for example, a LOGO query like *"Place 4 small semicircles in a row"* is broken down into *"1. place semicircles 2. small semicircles 3. in a row 4. 4 semicircles*. We then include this decomposition in a prompt asking the model to add comments to the code. After adding comments, we verify the code first with exact match (excluding comment strings) and then use execution accuracy if exact match fails.

A.2. Training

Code Bank Editing. Our Code Bank editing prompt asks the model to produce a CoT-style output, first specifying *why* the failing unit tests failed and then proposing an edit for the function. We then execute the stored demonstrations for that function with the new version; if there are more passing cases after refactoring, we replace the function. If the new function's signature differs from the old, we use a simple prompt to refactor the unit tests to accommodate the new function.

Code Bank Pruning. For each function, given a set of passing programs P and failing programs F, we compute a score $s = |P| - \sum_{p \in F} 1/n_p$, where n_p is the number of functions used in p. In other words, the function receives $+1$ for each passing program it participates in, and a negative score inversely proportional to the number of functions in the program (since naively, the failure could be attributed to any of the functions). Functions are pruned if they have been used a sufficient number of times and s falls below a threshold θ (set to 0 for all experiments).

A.3. Testing

In our test-time agent, we use ChromaDB for indexing and retrieval[6] with OpenAI Ada embeddings. ICL examples are retrieved from the training data and from the Demo Bank using query similarity. We limit the number of Code Bank functions to 20, using the similarity between the query and the function name for retrieval. The Code Bank is pruned once before testing.

A.4. Models

For GPT-3.5, we use the `gpt-3.5-turbo` version (0613). All CodeLlama models use the `CodeLlama-*-Instruct-hf` versions, and we use the `lemur-70b-v1` version of Lemur.

[6] https://github.com/chroma-core/chroma/

For the latter two open-source models, we use the checkpoints from HuggingFace (Wolf et al., 2020).

Algorithm 1 ReGAL: Training Algorithm

```
Input: $X = (q, p)$ // Training data: (query, program)
Params: editEvery, pruneEvery, threshold $\theta$
Output: CodeBank $C$, DemoBank $D$
$C \leftarrow \varnothing, D \leftarrow \varnothing$ // Initialization, i.e., no refactoring
// Preprocessing data via clustering and sorting by difficulty
$\mathcal{P} \leftarrow \texttt{preprocessAndGroupData}(X)$
for index $g$, batch $\mathcal{G} \in \mathcal{P}$ do
    // Refactor programs in group $\mathcal{G}$ based on the current CodeBank $C$. Returns new programs and helper functions.
    $(p_1^{new}, h_1), \ldots, (p_k^{new}, h_k) = \texttt{refactorBatch}(\mathcal{G}, C)$
    $H^{new} \leftarrow \{h_1, \cdots, h_k\}$ // Set of new helper functions $H$
    // Verifying that the gold program and the refactored program yield the same result when executed via indicator $\delta^{new}$.
    $\delta_{1:k}^{new} \leftarrow \texttt{verify}(H, C, \{p_i^{new}\}_{i=1}^{k}, \{p_i\}_{i=1}^{k})$
    for $i \in \{i : \delta_i^{new} = False\}$ do
        $(p_i^{retry}, h_i^{retry}) \leftarrow \texttt{retry}(p_i, p_i^{new}, C)$
        $\delta_i^{new} \leftarrow \texttt{verify}(h_i^{retry} \cup H, C, p^{new}, p)$
        if $\delta_i^{new} = True$ then // Update if retry succeeds
            $p_i^{new} \leftarrow p_i^{retry}$
            $H^{new}[i] \leftarrow h_i^{retry}$
    // update CodeBank $C$ with successful helper functions
    $C \leftarrow C + H^{new}[i]$ for $i \in \{i : \delta_i^{new} = True\}$
    // update DemoBank $D$ with refactored programs
    for $i \in \{1, \ldots, k\}$ do
        $D \leftarrow D + (p_i^{new}, \delta_i^{new})$
    // edit and prune CodeBank
    if $g \pmod{\text{editEvery}} = 0$ then
        $C \leftarrow \texttt{editCodeBank}(C, D)$
    if $g \pmod{\text{pruneEvery}} = 0$ then
        $C, D \leftarrow \texttt{pruneCodeBank}(C, D, \theta)$
return $C, D$
```

A.5. Data

Dataset	Train	Dev	Test	# Primitives
LOGO	200	100	111	9
Date	66	113	180	4
TextCraft	190	50	77	3

Table 4. Dataset statistics. We list the number of primitive functions in the programs (aside from built-in Python functions).

A.5.1. LOGO

LOGO data is generated from a synchronous text-code grammar, and pairs procedurally-generated language commands like *"three small triangles in a row"* with a corresponding Turtle graphics program; however, the original LOGO dataset is expressed in a Lisp-style functional syntax. While

Algorithm 2 REGAL: Testing Algorithm

```
Input: $Q, C, D, X$ // Test queries Q, Code Bank C, Demo Bank D, Training data $X$ = (query, program)
Params: ICL Budget $M$, ICL Percentage $r$
Output: Predicted programs $\hat{P}$
$M^{demo} \leftarrow r * M$
$M^{train} \leftarrow M - M_{demo}$
$\hat{P} \leftarrow \varnothing$
for $q \in X$ do
    $H \leftarrow \mathtt{retrieve}(q, C, 20)$ // retrieve up to 20 helper functions conditioned on the query
    $X^{demo} \leftarrow \mathtt{retrieve}(q, D, M^{demo})$ // retrieve helper demos from D
    $X^{train} \leftarrow \mathtt{retrieve}(q, X, M^{train})$ // retrieve primitive demos from X
    $I \leftarrow \mathtt{createPrompt}(H, X^{demo}, X^{train})$
    $\hat{p} \leftarrow LLM(I)$ // generate program
    $\hat{P} \leftarrow \hat{P} + \hat{p}$
return $\hat{P}$
```

this facilitates the application of helpful data structures for efficient code search (Ellis et al., 2021; Bowers et al., 2023), object-oriented languages like Python are far more common in practice. As a result, they are represented more in LLM pretraining data, which has been shown to contribute to parsing performance (Bogin et al., 2023). To account for this, we write an AST-based parsing script to translate the LOGO dataset into Python.

Primitives. Table 5 describes the primitives available in the LOGO library. Note that these are in addition to all Python primitives. We also provide all agents with several hard-coded values for long loops and small steps so that they can draw round shapes. `HALF_INF` is the number of steps required to draw a semicircle. `EPS_DIST` is a small distance, and `EPS_ANGLE` is a small angle.

Primitive	Description
`forward(dist)`	move forward `dist` units
`left(theta)`	rotate left by `theta` degrees
`right(theta)`	rotate right by `theta` degrees
`penup()`	lift the pen (stop drawing)
`pendown()`	put the pen down (start drawing)
`teleport(x, y, theta)`	move to position (`x`, `y`) with angle `theta`
`heading()`	get the current angle of the turtle
`isdown()`	check if the pen is down
`embed(program, vars)`	runs the code in `program` using the current context `vars` and teleports back to the original position.

Table 5. LOGO Primitives

A.5.2. DATE UNDERSTANDING

Date understanding involves both mathematical reasoning and parsing. Each question poses a word problem that requires reasoning about dates and times. For example, problems ask questions like: *"On May 9th, 2017 Jane bought 40 eggs. She ate one per day. Today she ran out of eggs. What is the date 10 days ago in MM/DD/YYYY?"*. These kinds of questions are especially hard for LLMs to answer directly. Lyu et al. (2023) approached this task as a program prediction task, wherein an LLM predicts a Python script that gives the answer to the question when executed. This paradigm is especially helpful for Date, as there are existing Python libraries that can perform math on dates, such as `datetime` and `dateutil`. While predicting programs with these libraries results in strong performance as compared to LLM-based reasoning, Lyu et al. (2023) method predicts programs one at a time, leaving the benefits of shared subroutines on the table. We use the programs predicted by Lyu et al. (2023) as a starting point for our refactoring process. Table 6 describes the Python libraries called by Lyu et al. (2023)'s programs, which we treat as the primitives for Date.

Primitive	Description
`date()`	returns a `date` object
`time()`	returns a `time` object
`relativedelta(time)`	performs addition/subtraction of `time`, which can be days, weeks, months, or years.
`strftime(format)`	prints the date in the specified format

Table 6. Date Primitives

A.5.3. TEXTCRAFT

TextCraft consists of goal queries paired with crafting recipes. Recipes are presented with distractors, making the parsing task challenging. Furthermore, the agent must reason about preconditions, as items can only be crafted when the requisite ingredients have been collected.

The queries ask for a particular item to be crafted. For example the query can be "*craft behive*" along with crafting commands:

craft 4 oak planks using 1 oak log
craft 1 honeycomb block using 4 honeycomb

craft 1 beehive using 6 planks and 3 honeycombs
craft 1 white bed using 3 white wool, 3 planks,
etc.

The environment has three primitives: `inventory`, `craft`, and `get` which we convert into Python variants, described in Table 7.

Primitive	Description
`getObject(obj_name)`	get `obj_name` from the environment
`craftObject(obj_name, [ingredients])`	craft `obj_name` using the list of ingredients
`checkInventory()`	return the contents of the inventory

Table 7. TextCraft Primitives

B. Analysis

B.1. What kinds of programs are discovered

Table 8, Table 9, and Table 10 show examples of the discovered programs most commonly used by the agent.

```
def draw_small_5gon():
    for i in range(5):
        forward(2)
        left(72.0)
def draw_semicircle():
    for i in range(HALF_INF):
        forward(EPS_DIST * 2):
        left(EPS_ANGLE)
```

Table 8. Examples of discovered programs for LOGO

```
def get_date_today(date_obj):
    return date_obj
def get_date_one_week_from_today(
   date_obj):
    return date_obj + relativedelta(
        weeks=1)
def get_date_one_week_ago(date_obj):
    return date_obj - relativedelta(
        weeks=1)
def get_date_one_year_ago(date_today):
    return date_today - relativedetla(
        years=1)
```

Table 9. Examples of common discovered programs for Date

C. Hyperparameters

Table 11 lists the refactoring and testing hyperparameters used for each domain.

```
def craft_object_with_ingredients(
   target,
    ingredients):
    inventory = check_inventory()
    for ingredient in ingredients:
        if ingredient not in inventory:
            get_object_from_env(
                ingredient)
        craft_object(target,
            ingredients)
def check_and_get_object(target):
    inventory = check_inventory()
    if target not in inventory:
        get_object(target)
```

Table 10. Examples of common discovered programs for TextCraft

Setting	LOGO	Date	TextCraft
Rounds of refactoring	3	1	1
`refactorEvery`	5	5	5
`filterEvery`	5	5	5
Add comments	True	False	False
Batch size	5	3	4
Filtering threshold	0.0	0.0	0.0
Filter before testing	True	True	False
ICL budget split	0.5	0.5	0.5

Table 11. Hyperparameter settings for all experiments

D. Prompts

Below, we detail the prompts used in all sections.

Table 12. Batch refactoring prompt (`refactorBatch()`). Comments indicate where text is repeated. Brackets indicate variables filled in by the environment. <> strings are passed as-is to the LLM.

```
Please rewrite the following two programs to be more efficient.
{primitive description string}
The resulting programs MUST execute to the same result as the original programs.
Start by writing helper functions that can reduce the size of the code.
You can also choose from the following helper functions:
{codebank function definitions}

// repeated for all in batch
QUERY {i}: {query}
PROGRAM {i}: {program}

Please format your answer as:
// repeated for i
NEW PROGRAM {i}:
// once at end
NEW HELPERS:

Do not include any text that is not valid Python code.
Recall that no matter what, your program MUST be formatted in the following fashion:
// repeated for i
NEW PROGRAM {i}:
# Thoughts:
# 1. The query asks for: <query intention>
2. <query> can be solved by <components>.
# 3. I will use helper function <function> to <goal>.
<code for program {i}>

Try to make your new programs as short as possible by introducing shared helper functions.
     Helper function parameters should be as general as possible and helper functions
    should be informatively named.
{logo_special_instr}
```

```
logo_special_instructions =

If the original function uses `embed`, you will likely need to use `embed` in your version
    . All code to be repeated needs to be included within the triple quotes passed to
    embed.
```

Table 13. Query decomposition prompt. Output is used by the comment prompt in Table 14

```
You are an expert coder. For each query below, decompose it into its parts.
Example:
Query: Do some action 5 times and then do another action
Query (decomposed):
The query asks: Do some action and then do another action
This can be decomposed into:
1. repeat an action
2. some action
3. another action

Query: {query}
Query (decomposed):
```

Table 14. Prompt to add comments to primitive programs. Takes output of Table 13 as input.

```
Please add comments to the following program to explain what each chunk of code does with
    respect to the query.
First, decompose the query into parts. Then comment the code with the query parts.
Example:
Query: Do some action and then do another action
Code:
do_some_action()
do_another_action()

Query: Do some action 5 times and then do another action
Query (decomposed):
The query asks: Do some action and then do another action
This can be decomposed into:
1. repeat an action
2. some action
3. another action
Commented code:
# repeat an action
for i in range(5):
    # do some action
    do_some_action()
# do another action
do_another_action()

{primitive description}

Query: {query}
Code:
{program}

Query (decomposed):
{output from decompose prompt}
```

Table 15. Prompt for `editCodeBank`.

```
Refactor the following function to improve performance.
FUNCTION:
```
{func_str}
```

{library_str}

You may also use the following helper functions:
{codebank_str}

Try to increase the number of passing programs. Try to make programs general. For example,
     you can add parameters instead of hardcoded values or call other helper functions.
    First, for each failing query, explain why the programs do not accomplish the query's
    goal. Output this reasoning as:
Thoughts:
1. The function passes some tests and fails others because <reason>.
2. The failing queries <repeat queries here> asked for <intent>.
3. The program failed because <reason>.
4. This can be addressed by <change>.
Then output your program so that all test cases pass, using the following format: NEW
    PROGRAM: <program>
Currently, {func.__name} passes in {pass_perc * 100:.1f}\% of cases and fails in {fail_perc
    *100:.1f}\%.

SUCCEEDED:
{example of passing case}
FAILED:
{example of failing case}
Thoughts:
```

Table 16. Agent prompt for Python tasks like Date understanding. Note that the baseline and REGAL agent use the same prompt, but {`codebank_str`} is empty for the baseline agent, and the REGAL sees some demonstrations from the Demo Bank in {`icl_string`}.

```
Your task is to solve simple word problems by creating Python programs.
{codebank_str}

You will be given a query and have to produce a program. {thought_str}
Examples:
{icl_string}

Please generate ONLY the code to produce the answer and nothing else.
Query: {query}
{thought_and}Program:
```

Table 17. Prompt for the LOGO agent.

```
Your task is to draw simple figures using python Turtle graphics.
You will use a custom turtle library, similar to the built-in library, which is sufficient
     for all tasks.

Here's a description of the custom library:
- forward(x): move forward x pixels
- left(theta): rotate left by theta degrees
- right(theta): rotate right by theta degrees
- penup(): stop drawing
- pendown(): start drawing
- teleport(x, y, theta): move to position (x, y) with angle theta
- heading(): get the current angle of the turtle
- isdown(): check if the pen is down
- embed(program, local_vars): runs the code in program using the current context and
    teleports back to the original position. Allows you to nest programs.
    Implementationally, embed gets the turtle state (is_down, x, y, heading), executes
    program, then returns to the original state.
- save(path): save the picture to file
{codebank_str}

You will be given a query and have to produce a program. Begin your program with a comment
     that explains your reasoning. For example, you might write:\n# Thought: the query
    asks for a line, so I will use the forward() function.
Examples:
{icl_string}

Please generate ONLY the code to produce the answer and nothing else.
Query: {query}
Thought and Program:
```