Appendix

Table of Contents

A Additional Related Work

A.1 Programming with Natural Language

Due to the low learning requirements, programming with natural language has been seen an attractive programming mode since 1960s [2, 5, 6, 8, 11, 13, 15]. The relevant domains encompass semantic parsing [9, 23], language grounding [20, 21], and so on. Among these works, Wang et al. [22] is the most similar one which let users build complex voxel structures by defining alternative, more natural syntax, and increasingly complex natural concepts, starting from a core programming language. However, they are restricted by the pre-defined natural language specifications and domain-specific languages while this paper focuses on unconstrained natural language and general-purpose languages like Python. Furthermore, it should be noted that the interactive processes involved in these studies rely on manually annotated data, whereas our system operates in a truly debugging fashion for programs composed of natural language.

A.2 LLMs as an Interpreter

LLMs have been embedded in programs as an interpreter, due to their capabilities of common sense question answering and simple natural language reasoning [16, 17, 25]. For example, Cheng et al. [4]

leverages LLMs to generate programs for questions like "Is Mexico North America?" with Codex API calls like $f(\text{`}NorthAmerica\text{?`}, Mexico)$ and then answer it by executing and prompting. Dohan et al. [7] composes LLMs into a probabilistic programming framework, which allows control flow and dynamic structure. Different from these works, we focus on leveraging LLMs to implement natural language modules into executable programs. We take the utilization of taking LLMs as parts of our generated programs as future work.

B Detailed Limitations and Future Work

Though our system has shown excellent performance revealed by the large-scale human study, there are still three main limitations to our current system. The first limitation is the response of LLMs. ANPL gives users a detailed implementation of each hole in Python and asks them to further debug. This requires users to read Python code when editing ANPL programs. LLMs may employ certain APIs that users may not be familiar with, resulting in the implementation of a function that deviates from the users' intended approach. Consequently, comprehending Python code becomes more challenging. Additionally, when LLMs automatically break down a function into sub-functions, identifying the specific sub-function containing a bug becomes difficult for users. To enhance the user-friendliness of ANPL and alleviate the burden on users in terms of their code capabilities, **how can ANPL provides concise summaries of each function in easily understandable natural language** should be studied. These summaries would enable users to identify any misinterpretations or incorrect implementations based on the corresponding descriptions, allowing them to modify their natural language descriptions accordingly without the need for complete redrafting.

Another limitation is the absence of comprehensive natural language libraries. In the context of code generation using LLMs, the quality of the prompt and the accompanying description assumes paramount importance. However, the creation of effective natural language descriptions necessitates considerable expertise in prompt engineering. In order to mitigate this issue, **a natural language library should be established**. The natural language content within this library is derived from two primary sources: library learning methods [24] and user contributions, and users can share and reuse natural language and corresponding implementations made by each other.

So far, ANPL has been limited to generating Python code for solving ARC problems through communication with users. However, as mentioned earlier, some ARC tasks cannot be fully addressed with Python programs alone and require the use of neural modules like object detection. Therefore, we need to **integrate ANPL with neural modules**, similar to works including Cheng et al. [4], Shen et al. [18], Surís et al. [19]. This integration would further enhance the capabilities of the ANPL compiler and expand the range of tasks that ANPL can handle. Additionally, the application scope should not be confined solely to the ARC dataset but should extend to multiple domains, such as chip design, program writing, and robot control. However, due to the unique characteristics of ARC itself and the limited availability of human resources, it is well-suited for conducting user-programming experiments and human study reports. Thus, we have chosen ARC as the platform for conducting our experiments and presenting our findings.

C Broader Impacts

On one hand, ANPL sheds light on the human-computer interaction paradigm by making the interaction more stable and reliable through low-cost predefined programming conventions. On the other hand, ANPL has the potential to promote the development of the programming field and broaden the scope of programming applications. For example, ANPL enables users to program and debug with natural language which can significantly lower the programming barrier. Furthermore, the proposed DARC dataset reveals how humans systematically decompose complex problems into simpler ones when faced with logical problems similar to ARC. This could provide further insights to cognitive science and foster advancements in related fields.

However, ANPL also raises safety challenges by producing code that is unaligned with user intent and can be misused. We refer readers to the broader impacts and hazard analysis discussed comprehensively by Chen et al. [3] as the basic component of the ANPL compiler is the LLM. Note that, compared to Chen et al. [3], a more significant concern with the usage of ANPL is its lower barrier to entry, allowing individuals with limited programming experience to generate code. This may result in

code generated by ANPL lacking the scrutiny and maintenance of experienced personnel, making it more susceptible to misuse and thereby leading to more severe issues related to code security. Besides, since these users may have limited exposure to open-source communities such as GitHub, models trained on corresponding data may face greater challenges in user alignment.

D ANPL Details

D.1 An ANPL Code Example

Figure 2 is an ANPL code example for the task shown in Figure 1. In this example, users first decompose the task into `"Change the input into four new arrays based on the central dividing line in the x and y directions"` and `"Find an array that doesn't have just one color"`. Then, users can either define a *hole* by its name like `seperate_input` with corresponding parameters, or they can just code a piece of natural language description like `"Find an array that doesn't have just one color"` and set the *sketch* by specifying its input-output variables.

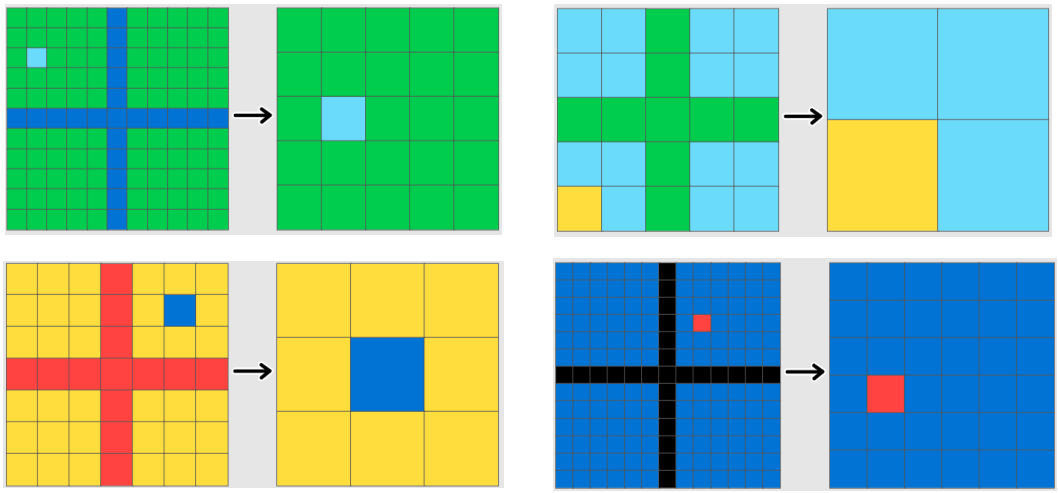

Figure 1: Task 64 of ARC.

```anpl
def seperate_input(input):
    "Change the input into four new arrays based on the central dividing line in the x and y directions"

def main(input):
    inputs = seperate_input(input)
    output = "Find an array that doesn't have just one color"(inputs)
    return output
```

Figure 2: An ANPL code example for task 64.

D.2 Syntactic Sugar

Recursion. ANPL employs a hole within the function's own body to implement recursion, following the thought of Y Combinator. A function can indirectly invoke itself by passing its own reference to a hole within its own body.

Figure 3 is an example demonstrating the use of hole-driven recursion with the Flood Fill algorithm. In this example, the floodfill function is passed as an argument to the hole `"apply floodfill to adjacent pixels: above, below, right, and left"`. The semantics of the hole suggests that floodfill will be applied to the adjacent pixels, establishing an indirect recursion.

```
def floodfill (grid, i, j):
    if "is outside the valid grid area"(grid, i, j) and grid[i, j] != black:
        return grid
    else:
        return "apply floodfill to adjacent pixels: above, below, right, and left"(grid, i, j, floodfill)
```

Figure 3: Recursion in ANPL.

D.3 Implementation Details

ANPL interacts with the LLM via the prompt shown in Figure 4. In the prompt, the **code** section will be substituted with all the executed codes up to that point, and the **hole** section will be replaced with the designated function name of the hole along with the natural language description given by the user. During the initial user input and function editing, it goes through a sequence of five attempts, starting with a temperature parameter of 0 and incrementing it by 0.1 with each try until it succeeds. In the resynthesis stage, ANPL requests the underlying LLM to produce 10 potential completions for each prompt. The text that ChatGPT generates will be subject to a maximum token constraint of 1024.

````
# system prompt
As a pythonGPT, your task is to complete the unimplemented functions in the given python code,
which are referred to as "holes" and are labeled as _hole0, _hole1, _hole2, and so on.
Your implementation should align with the code and documentation using Python.

# user prompt
```python
{code}
```
The function needs to be given a new name. Markdown format should be used to return it.
```python
{hole}
```
````

Figure 4: Prompts used in ANPL.

E Human Studies

E.1 Questionnaire

We conducted a survey to investigate users' programming abilities, LLM usage experiences, evaluations of system A and system B, as well as the perceived importance of various functionalities within system A. The detailed questions are as follows:

1. How would you rate your programming skills?
   1: Non-programmer
   2: Beginner, struggles with solving LeetCode medium-level problems
   3: Familiar with a programming language, understands basic data structures and algorithms, able to solve some LeetCode medium-level problems
   4: Proficient in common data structures and algorithms, capable of solving many LeetCode medium-level problems
   5: Skilled in data structures and algorithms, capable of solving LeetCode hard-level problems
2. How familiar are you with the Python language?
   1: No exposure to Python.
   2: Have used Python, familiar with basic syntax, but rarely used in daily activities.
   3: Occasionally use Python to write simple scripts, not familiar with Python libraries such

as NumPy and PyTorch.
4: Proficient in Python features, frequently use Python, and familiar with some libraries.
5: Mastery in Python and proficiency in using common libraries.

3. Have you used language models to generate code before?
4. How do you perceive the difficulty of ARC questions? (1: Very easy - 5: Very hard)
5. Do you find System A (ANPL) useful? (1: Very dissatisfied - 5: Very satisfied)
6. Do you find System B (natural language) useful? (1: Very dissatisfied - 5: Very satisfied)
7. How important do you consider the Trace feature of System A? (1: Very unimportant - 5: Very important)
8. How important do you consider the Edit feature of System A? (1: Very unimportant - 5: Very important)
9. How important do you consider the Resynthesis feature of System A? (1: Very unimportant - 5: Very important)

The results are shown in Figure 5. Participants in our human study are primary Python programmers and half of them are not familiar with code generation with LLMs. The average score of system A is 4.05, significantly greater than system B which scores 2.58. System A achieves not only a higher solving rate but also a better user experience than System B. Besides, most users find tracing and editing useful while resynthesizing less important, which shows the limitation of LLM code generation and indicates the importance of introducing user interaction in solving complex tasks like ARC.

E.2 Tutorials

The task requires you to solve ARC (Abstraction and Reasoning Corpus) tasks, each composed of several input-output pairs. These pairs maintain a uniform pattern and are structured as color grids with ten distinct colors: black, blue, red, green, yellow, grey, pink, orange, teal, and maroon.

The goal is to deduce patterns from the input-output pairs and communicate your solution to the system. In this experiment, you will interact with two systems: System A and System B. System A accepts ANPL inputs, whereas System B functions on full natural language. Both systems aim to generate Python code that transforms the input into the expected output.

Each task requires working with both systems in a specific order provided in the task assignment. To begin, find the ARC task solution independently (solve the task in your mind, i.e. not with Python). With a solution in hand, activate the system, which will initiate a timer.

Your target is to instruct the systems to generate accurate Python code based on your solution. We'll be evaluating the program strictly on the test input and output, yet it's essential for you to confirm that your program is capable of successfully handling all the training input and output. Once the correct code is produced, the system will automatically deactivate. Perseverance is crucial, but if the system fails to generate the accurate code within 30 minutes, you're permitted to terminate the process. If a task proves overly challenging at any point, it's acceptable to stop prematurely.

System A is comprised of three primary operations. The *trace* operation allows for a function name to be entered, which triggers the program to run on a test input and displays all the input and output data of the selected function. The *edit* operation allows for direct changes to a function's body, including alterations to the sketch and hole. This operation has four sub-operations:

- Splitting the original function into multiple holes linked by the sketch.
- Turning the original code into a hole and attempting code generation.
- Changing the natural language description associated with the hole.
- Modifying the sketch while maintaining the generated hole.

The *resynthesis* operation requires the user to provide correct input and output examples. The system then generates numerous functions and tests in which one meets these examples. The provided examples are kept for future use, and multiple sets of examples can be provided by the user.

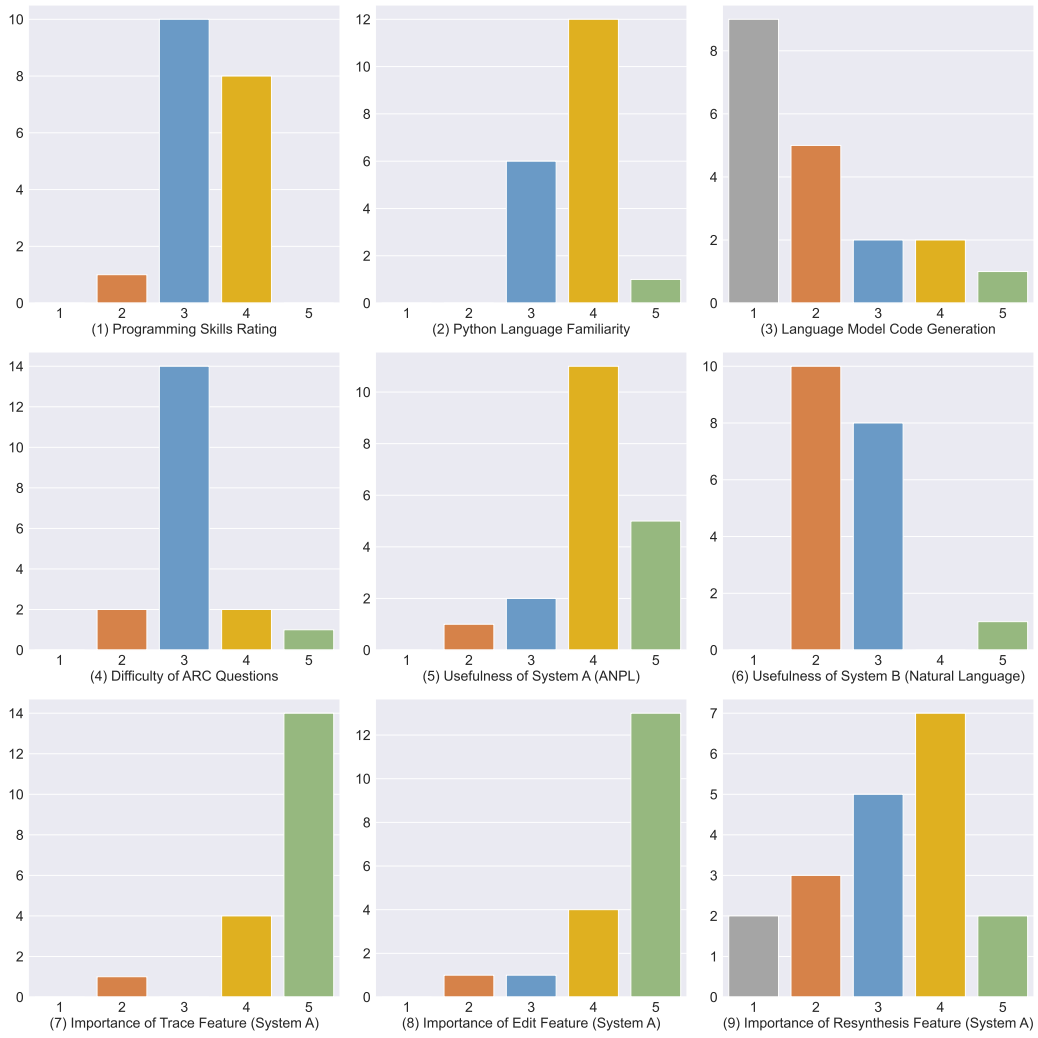

Figure 5: Questionnaire Analysis. The average score of System A is 4.05, and System B is 2.58.

System B incorporates two operations. The *chat* operation allows the user to interact with the system using natural language, leading to code generation or modification based on these descriptions. The *remove history* operation allows the user to select and delete some historical conversation data.

E.3 User Interface

The user interface consists of 4 components: tracing operation, editing operation, resynthesizing operation, and a grid editor.

Tracing. The tracing operation has three panels, namely function selection, visual IO, and textual IO, see Figure 6. The function selection section allows users to choose from a list of available functions eligible for tracing. After selecting a function, IOs are shown to users within the visual IO and the textual IO panels, where the visual IO panel visualizes the IO into grids and the textual IO panel prints the IO as NumPy arrays.

Editing. The editing operation has three panels: function selection, function editing, and code synthesis, see Figure 7. The function selection panel serves the same purpose as the one in the tracing operation. After selection, users can modify ANPL code in the function editing panel and submit it to

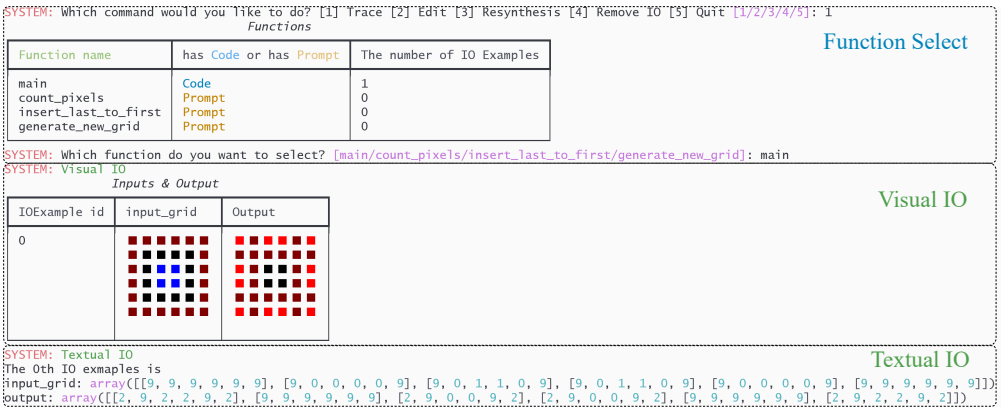

Figure 6: The tracing operation. Users can check the execution trace between high-level holes.

the LLM for code generation. They can then view the code generation progress in the code synthesis panel.

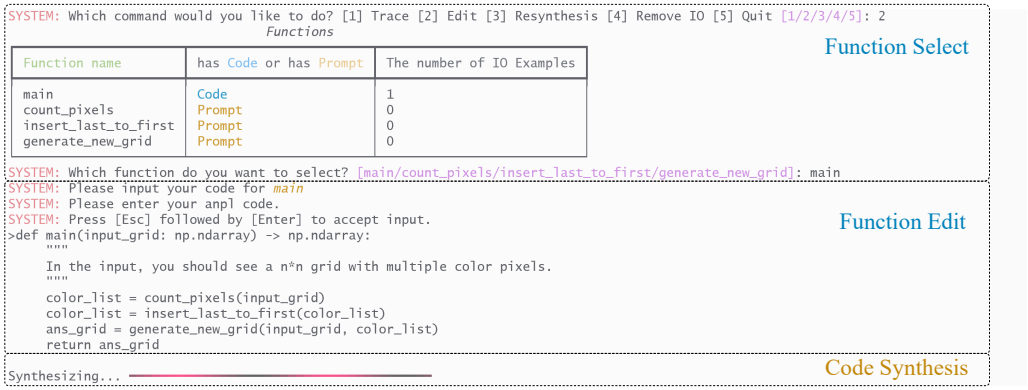

Figure 7: The editing operation. Users can edit the existing ANPL program through further decomposition or just modifying the code or natural language.

Resynthesizing. The resynthesis operation has three panels: function selection, IO entering, and code synthesis, see Figure 8. The function selection panel and the code synthesis panel serve the same purpose as the ones mentioned above. The IO entering panel enables users to constrain the programs generated by the LLM by providing IOs. Specifically, LLM generates a set of 10 candidate Python programs, and subsequently selects the program(s) that satisfy the given IO constraints as the compiled program.

Grid editor. In order to facilitate the transition for users between visual IO (*i.e.* colored grids) and textual IO (*i.e.* NumPy array), we have implemented a grid editor (Figure 9). This editor consists of the following elements:

1. Resize: Allows users to specify the dimensions of the grid.
2. Generate: Generate the corresponding grids by inputting a Numpy array.
3. Reset: Reinitializes the current grid to its initial state.
4. Copy: Converts the current grid into a Numpy array and copies it to the clipboard.

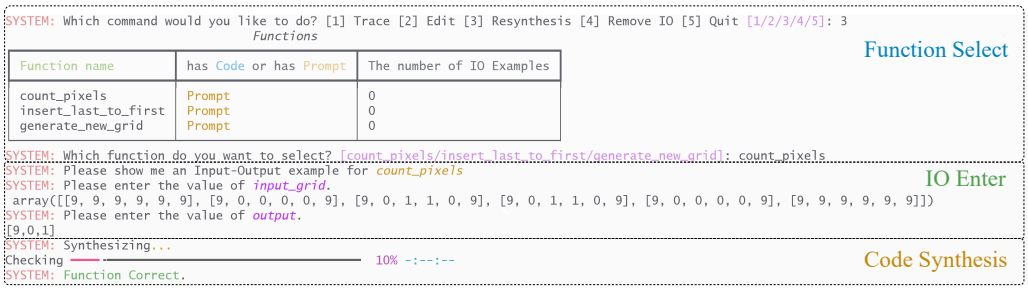

Figure 8: The resynthesis operation. Users can provide IO constraints and ask the compiler to resynthesize the program.

5. Graphical editing area: Provides three operations, including edit, select, and flood fill, along with 10 different colors, for direct editing of the colored grids.

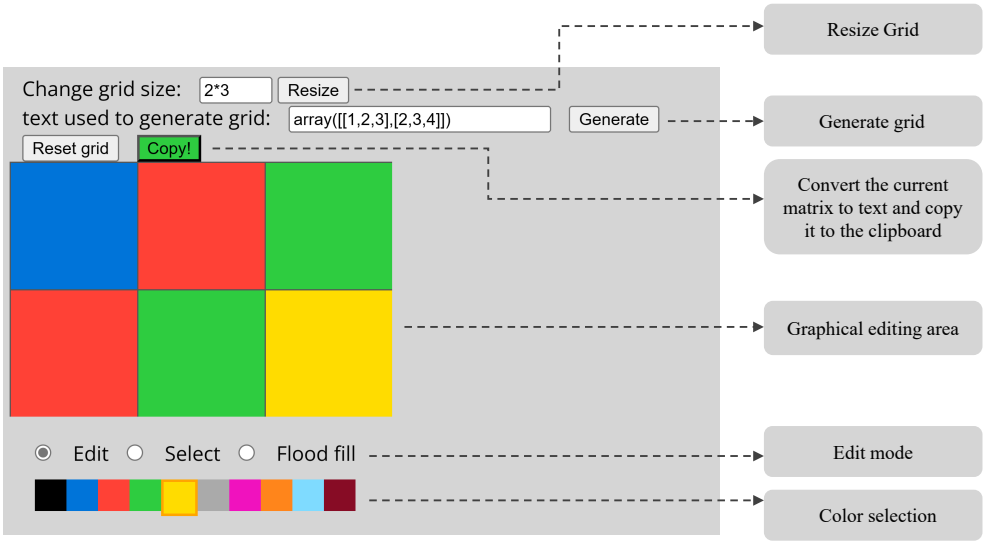

Figure 9: The grid editor.

E.4 Suggested Prompts for System B

Users can input natural language without restrictions in system B. To enhance the user experience, we offer them a suggested prompt template shown in Figure 10.

E.5 Task Assignment

Table 1 provides an assignment for ARC tasks. Each number corresponds to a question; a number highlighted in blue instructs use of System A before System B, whereas an orange number, conversely, denotes System B to be operated before System A.

```
You are a skilled Python programmer.
Your task is to write Python code to transform the input grid into the output grid.
In the input grid, you should see ...
To make the output grid, you should ...
Return your Python code in Markdown format.

```python
import numpy as np
black, blue, red, green, yellow, grey, pink, orange, teal, maroon = range(10)
def main(input_grid: np.ndarray) -> np.ndarray:
```
```

Figure 10: prompt for System B.

F Additional Experiments

F.1 Datasets

Why not evaluate ANPL on common code generation benchmarks. We evaluate ANPL on ARC instead of common code generation benchmarks [1, 3, 12, 14] according to the following two reasons: (1) Most code generation benchmarks are too easy for users. Though competitive programming benchmarks exist [10], they demand high data structure and algorithm skills. Instead, ARC is a general artificial intelligence benchmark. Solving tasks directly in ARC is easy for humans, but expressing the solution using code is the main challenge. The results we obtained on this benchmark better reflect the impact of our system on programming, i.e., taking an algorithm and expressing it in Python. (2) Solutions for common code generation benchmarks can be found on the Internet, which runs the risk of data contamination. On the other hand, it is unlikely programmatic (python) solutions for ARC exist in any online corpus, making tasks in ARC unique enough for LLMs and participants of human study.

F.2 Results

The importance of the *control structure*s and *hole*s. In order to analyze the significance of the programming model, we conduct an analysis on the proportion of *control structure*s (*e.g.*, for, while, if) and *hole*s in programs that were correct in both system A and system B, and the proportion of programs that were correct in system A but incorrect in system B. In the programs that were correct in both systems, 47.8% utilized control flow, maintaining an average of 2.47 *hole*s. On the other hand, a remarkable 97.4% of programs that were correctly functioning in system A but failed in system B incorporated control flow, and had an increased average of 3.37 *hole*s. Results show that the introduction of *control structure* and *hole* has a significant positive impact on the user's ability to accomplish highly complex tasks.

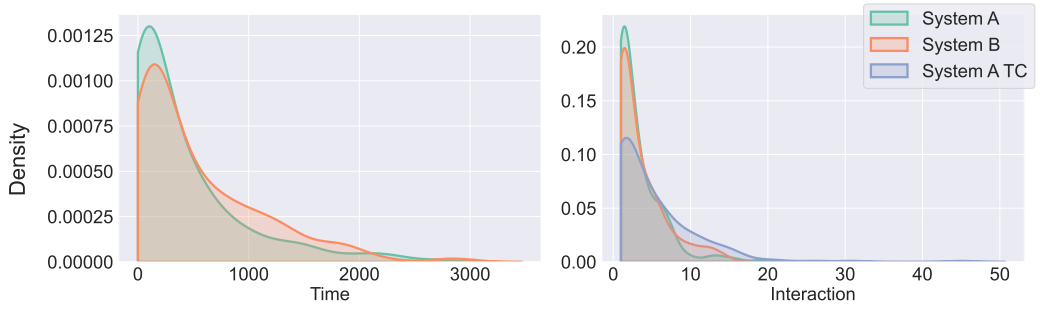

Figure 11: The distribution of time and the number of interactions. Trace Calculated (TC) means the trace mode is considered into interactions.

User id	Tasks
0	4 34 36 48 50 116 128 222 230 309 384
1	6 74 101 102 104 139 141 150 152 218 235 247 257 279 289 312 332 340 343 356 368 370 374 382
2	21 55 61 65 69 86 92 164 165 182 183 186 227 234 258 261 262 271 307 310 344 378 379 386
3	0 8 22 75 95 108 125 140 157 192 198 202 205 228 237 245 255 265 280 316 339 346 363 365 371 381
4	24 28 37 68 71 80 87 98 114 124 166 170 200 206 208 241 272 297 298 301 302 303 315 317 319 327 336 360 361 389 398
5	16 25 30 56 62 70 111 120 130 142 151 173 180 185 212 215 226 275 295 347 364 388 393 397
6	53 63 105 135 137 159 176 224 244 260 278 300 320 341 359
7	10 12 14 35 43 44 90 97 144 161 162 175 199 219 231 236 263 287 299 323 331 345
8	91 94 112 126 129 168 181 190 191 193 248 264 285 306 387 391
9	18 26 41 77 79 122 127 131 160 223 252 277 292 321 338
10	1 2 9 38 42 54 81 88 96 99 115 136 145 147 153 246 283 308 322 355 380 383 385
11	13 23 31 32 49 57 118 132 138 149 211 225 240 251 267 270 286 304 313 314 318 353 396
12	15 58 64 83 117 119 155 156 167 189 210 214 221 233 242 254 256 276 281 293 296 305 325 354 367 369 376 392 395
13	5 27 33 51 82 123 134 172 177 178 201 216 229 232 243 282 288 311 330 358 377 399
14	20 40 47 73 171 187 195 217 220 328 337 351
15	11 29 46 60 93 106 107 110 113 163 184 209 213 239 274 326 333 342 372
16	39 59 67 76 78 103 158 179 188 204 207 253 266 294 334 335 348 350 357 362 366 390 394
17	3 17 109 133 197 259
18	7 19 45 52 66 72 84 85 89 100 121 143 146 148 154 169 174 194 196 203 238 249 250 268 269 273 284 290 291 324 329 349 352 373 375

Table 1: Task assignment. Blue: use System A and then System B. Orange: use System B and then System A

The distribution of time and the number of interactions. Furthermore, we conduct an analysis of the time consumption and the number of interactions involved in solved problems. That is, we filter the problems that can be solved by both systems A and B, collect the time and number of interactions spent by users, and then construct the two distributions shown in Figure 11. Results show that, for these tasks, system A exhibits faster completion times and fewer interaction counts (without TC) compared to system B, with slight advantages in terms of time and interaction. Nevertheless, this advantage is not highly pronounced, which could be attributed to the relatively low difficulty level of the problems that both system A and system B are capable of solving.

A \ B	✓✓	✓✗	✗✗	Total
✓✓	177	5	45	227
✓✗	10	32	31	73
✗✗	7	3	90	100
Total	194	40	166	400

Table 2: Generalization on ARC. ✓✓means the compiled Python program passes both train cases and test cases. ✓✗means the Python program passes the test cases but fails train cases. ✗✗means the program fails in all the cases.

Generalization ability. We further examine the accuracy of programs from System A and System B using all IO cases, see Table 2. System A and system B have similar generalization abilities based on the observation that for tasks passed on test cases by both two systems (224), the number of tasks that system A failed to generalize on all IO cases is 10/224 and the number of system B is 5/224, and these two numbers have no significant differences. Thus, we conclude that some programs failed to generalize to all IO cases because of the difficulty of the corresponding tasks and the designing of the underlying algorithm, which is independent of the usage of systems.

G DARC Details

time	role	action	content
1684141011.7825	user	enter code	"def main(input): centers = `traverse the input which is a 2-dim numpy array, return positions which satisfies that there is no grey in its 3*3 neighbor`(input) scores = `for each position in the centers, count the yellow position in its 3*3 neighbor`(input, centers) max_score = np.max(scores) center_yellow = centers[scores==max_score] center_black = centers[scores!=max_score] output = `for each position in the position list, make its 3*3 neighbor yellow`(input, center_yellow) output = `for each position in the position list, make its 3*3 neighbor black`(output, center_black) return output"
1684141011.7839	system	parser	"user enter correct code"

Figure 12: The header of CSV files.

```python
def main(input):
    centers = "traverse the input which is a 2-dim numpy array, return positions which satisfies that
    ↪  there is no grey in its 3*3 neighbor"(input)
    scores = "for each position in the centers, count the yellow position in its 3*3 neighbor"(input,
    ↪  centers)
    center_yellow, center_black = "return the center with the max scores and other centers"(centers,
    ↪  scores)
    output = "for each position in the position list, make its 3*3 neighbor yellow"(input, center_yellow)
    output = "for each position in the position list, make its 3*3 neighbor black"(output, center_black)
    return output
```

Figure 13: An ANPL program in DARC.

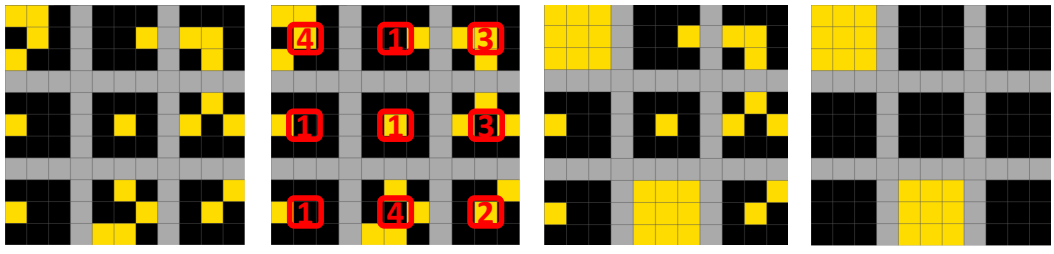

Figure 14: The trace of the given ANPL program. (2) Centers are framed by red rectangles and we mark scores inside each rectangle.

```python
import numpy as np
from typing import *
(black, blue, red, green, yellow, grey, pink, orange, teal, maroon) = range(10)

def get_max_score_center(centers: List[Tuple[int, int]], scores: np.ndarray) -> Tuple[List[Tuple[int,
↪  int]], List[Tuple[int, int]]]:
    max_score = np.max(scores)
    max_centers = [centers[i] for i in range(len(centers)) if scores[i] == max_score]
    other_centers = [centers[i] for i in range(len(centers)) if scores[i] < max_score]
    return (max_centers, other_centers)


def find_positions_without_grey_neighbors(input: np.ndarray) -> List[Tuple[int, int]]:
    positions = []
    for i in range(1, input.shape[0]-1):
        for j in range(1, input.shape[1]-1):
            if np.all(input[i-1:i+2, j-1:j+2] != grey):
                positions.append((i, j))
    return positions


def make_neighbors_yellow(input: np.ndarray, positions: List[Tuple[int, int]]) -> np.ndarray:
    for position in positions:
        input[position[0]-1:position[0]+2, position[1]-1:position[1]+2] = yellow
    return input


def make_neighbors_black(input: np.ndarray, positions: List[Tuple[int, int]]) -> np.ndarray:
    for position in positions:
        input[position[0] - 1:position[0] + 2, position[1] - 1:position[1] + 2] = black
    return input


def count_yellow_neighbors(input: np.ndarray, centers: List[Tuple[int, int]]) -> np.ndarray:
    scores = np.zeros(len(centers))
    for i, position in enumerate(centers):
        scores[i] = np.sum(input[position[0] - 1:position[0] + 2, position[1] - 1:position[1] + 2] ==
        ↪  yellow)
    return scores


def main(input):
    centers = find_positions_without_grey_neighbors(input)
    scores = count_yellow_neighbors(input, centers)
    center_yellow, center_black = get_max_score_center(centers, scores)
    output = make_neighbors_yellow(input, center_yellow)
    output = make_neighbors_black(output, center_black)
    return output
```

Figure 15: The compiled Python program in DARC.

The Recursive Decomposition Dataset of ARC Tasks (DARC) is an assemblage of interaction records associated with 400 ARC tasks. These records involve the intercommunication between users, the system, and GPT-3.5-turbo. Figure 12 presents the header of each CSV file. For each interaction, data concerning the role, action, content, and timestamp are completely collected and stored. The result is the entire user interaction history with our systems can be perfectly *replayed* at a later time. However, the response from LLMs will be different for the following reasons: (1) we used temperature = 1.0, which will cause different tokens to be sampled (2) the GPT-3.5-turbo implementation might be changed, which is outside of our control.

300 of the 400 ARC tasks were effectively decomposed and converted into Python using the ANPL, averaging 2.7 *hole*s per task – these tasks solves the test-input, but some may fail to generalize to other input-outputs, see Appendix F.2. The DARC dataset not only houses the final solutions to the ARC tasks but also encapsulates the diverse problem-solving approaches employed by different users. Crucially, it is a dataset detailing how humans decompose abstract procedural tasks into simpler sub-tasks, and ground each task into a program (e.g. Python) in collaboration with an LLM. We give an example of an ANPL program, the compiled Python program and its corresponding trace in Figure 13, Figure 15 and Figure 14.

The DARC dataset provides a valuable window into the system's task completion processes. By documenting ANPL decompositions, Python code, and detailed interaction histories, it permits us to gain insights into the practical application of Language Learning Models (LLMs) for programming. We hope that this dataset will be useful for others seeking to understand and refine similar systems.

H Computational Resources

For our human study, LLM APIs were called 4304 times for System A (10.76 per task), and 1923 times for System B (4.81 per task). The distributions of the number of LLM API calls are presented in Figure 16. Since System A generates 10 candidates when resynthesizing, the number of API calls on several tasks exceeds forty times. For most tasks, the number of interactions in the two systems is less than twenty.

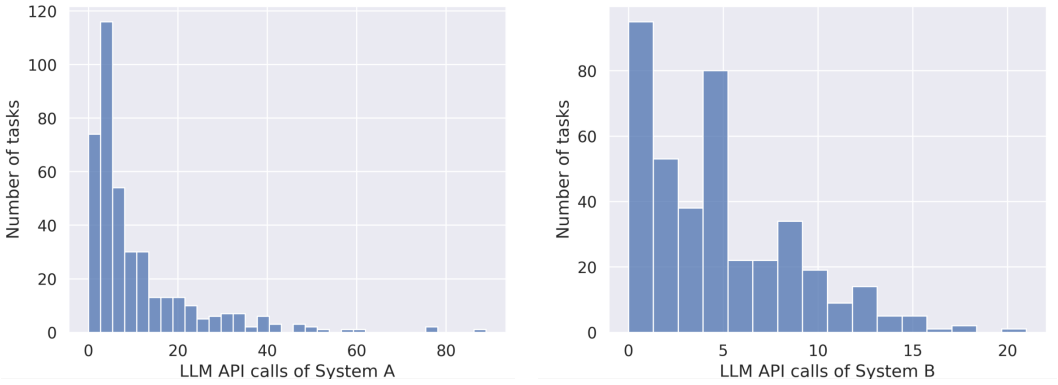

Figure 16: LLM API calls of Systems A and B.

I Case Study

A case of user interaction. Figure 17 serves as an illustrative instance. Initially, the user enters the ANPL code into the system, which consequently produces a function for every hole and automatically verifies the code's validity. Upon encountering a programming error, the system issues a warning to the user. In response, the user examines the input and output of the *find_smallest_unit* function and discovers it doesn't align with expectations. The user then adjusts the natural language description, leading to ANPL successfully meeting the test input and output samples following a code regeneration. Eventually, the system presents the complete code to the user, as shown in the Figure 18.

Difficult Tasks. From the overall 400 tasks under consideration, several tasks pose a significant challenge to users because it is hard or impossible to find the solving logic. We present one case in Figure 19.

User-Specific Solution. The potential solution of the ARC problem is heavily reliant on the user's algorithm design, which has an impact on the difficulty of programming. For instance, in the ARC task illustrated in Figure 20, some users might attempt to identify the color pattern across each row or column. Others might note the black square's location, then rotate the grid 90 degrees and select the

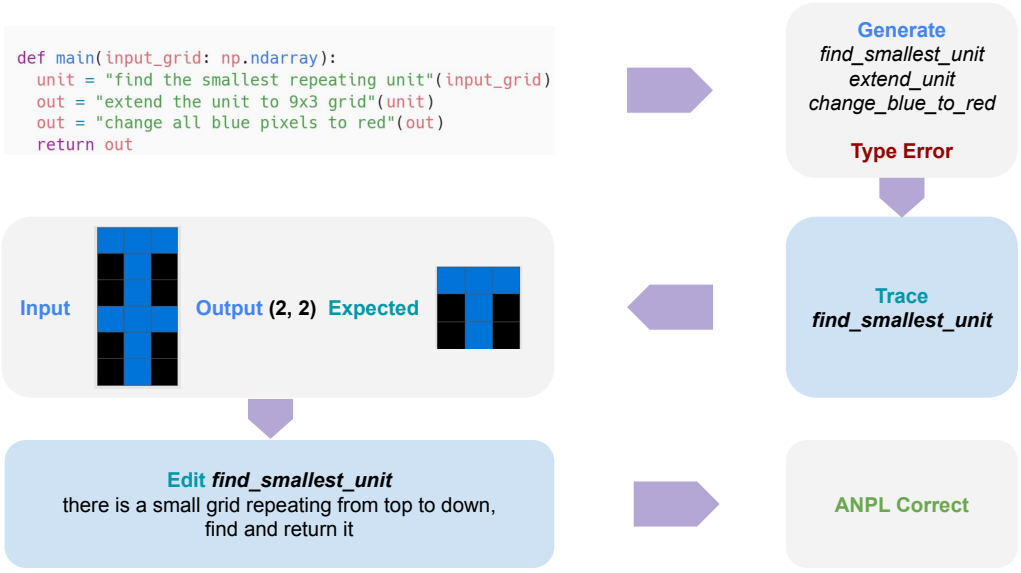

Figure 17: User Interaction with System A and Subsequent Code Generation Process

grid in the same position as the answer. The first approach is quite challenging, often too demanding to be completed within a time constraint. Conversely, the latter method leverages the rotational symmetry of the grid, enabling a direct translation into the correct Python code, eliminating the need for debugging.

```python
import numpy as np
from typing import *
(black, blue, red, green, yellow, grey, pink, orange, teal, maroon) = range(10)

def find_smallest_unit(input_grid: np.ndarray) -> np.ndarray:
    """
    Given a 2D numpy array, this function finds the smallest repeating unit in the grid and returns it.
    """
    (width, height) = input_grid.shape
    for i in range(1, width+1):
        if width % i == 0:
            for j in range(1, height+1):
                if height % j == 0:
                    unit = input_grid[:i, :j]
                    if np.all(np.tile(unit, (width//i, height//j))[:width, :height] == input_grid):
                        return unit
    return input_grid

def extend_unit(unit: np.ndarray) -> np.ndarray:
    """
    Given a 2D numpy array, this function extends the array to a 9x3 grid.
    """
    (width, height) = unit.shape
    if width >= 9 and height >= 3:
        return unit[:9, :3]
    else:
        extended_unit = np.zeros((9, 3), dtype=unit.dtype)
        for i in range(9):
            for j in range(3):
                extended_unit[i, j] = unit[i % width, j % height]
        return extended_unit

def change_blue_to_red(unit: np.ndarray) -> np.ndarray:
    """
    Given a 2D numpy array, this function changes all blue pixels to red.
    """
    unit[unit == blue] = red
    return unit


def main(input_grid: np.ndarray):
  unit = find_smallest_unit(input_grid)
  out = extend_unit(unit)
  out = change_blue_to_red(out)
  return out
```

Figure 18: The code synthesized by ANPL.

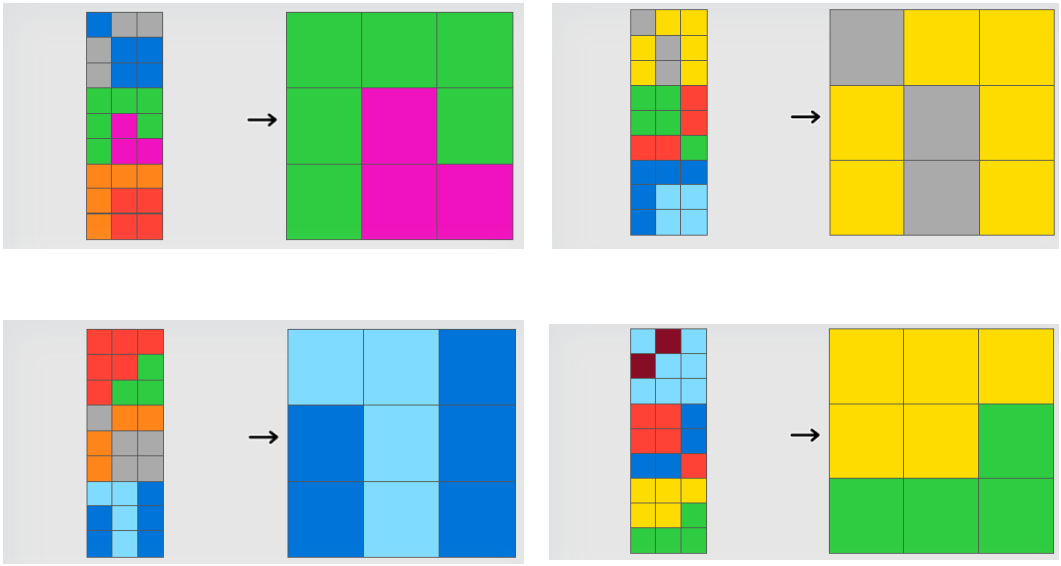

Figure 19: Unsolvable ARC task for the user.

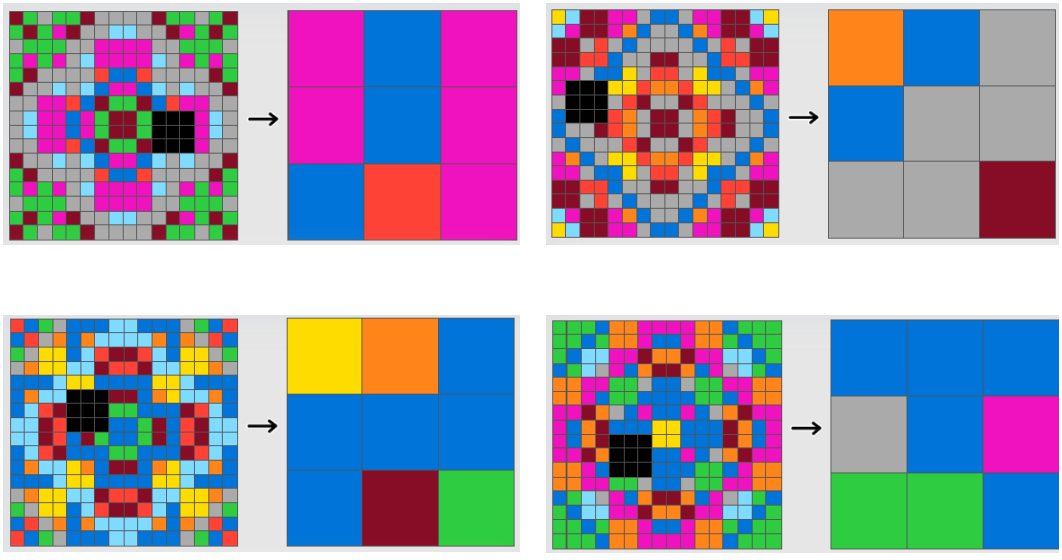

Figure 20: Example of an ARC task demonstrating differing user strategies.

References

[1] Jacob Austin, Augustus Odena, Maxwell I. Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie J. Cai, Michael Terry, Quoc V. Le, and Charles Sutton. Program synthesis with large language models. *CoRR*, abs/2108.07732, 2021.

[2] Alan W. Biermann, Bruce W. Ballard, and Anne H. Sigmon. An experimental study of natural language programming. *Int. J. Man Mach. Stud.*, 18(1):71–87, 1983.

[3] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé de Oliveira Pinto, Jared Kaplan, Harrison Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code. *CoRR*, abs/2107.03374, 2021.

[4] Zhoujun Cheng, Tianbao Xie, Peng Shi, Chengzu Li, Rahul Nadkarni, Yushi Hu, Caiming Xiong, Dragomir Radev, Mari Ostendorf, Luke Zettlemoyer, Noah A. Smith, and Tao Yu. Binding language models in symbolic languages. *CoRR*, abs/2210.02875, 2022.

[5] William R. Cook. Applescript. In *HOPL*, pages 1–21. ACM, 2007.

[6] Edsger W. Dijkstra. On the foolishness of "natural language programming". In *Program Construction*, volume 69 of *Lecture Notes in Computer Science*, pages 51–53. Springer, 1978.

[7] David Dohan, Winnie Xu, Aitor Lewkowycz, Jacob Austin, David Bieber, Raphael Gontijo Lopes, Yuhuai Wu, Henryk Michalewski, Rif A. Saurous, Jascha Sohl-Dickstein, Kevin Murphy, and Charles Sutton. Language model cascades. *CoRR*, abs/2207.10342, 2022.

[8] Mark Halpern. Foundations of the case for natural-language programming. In *AFIPS Fall Joint Computing Conference*, volume 29 of *AFIPS Conference Proceedings*, pages 639–649. AFIPS / ACM / Spartan Books, Washington D.C., 1966.

[9] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, Jayant Krishnamurthy, and Luke Zettlemoyer. Learning a neural semantic parser from user feedback. In *ACL (1)*, pages 963–973. Association for Computational Linguistics, 2017.

[10] Yujia Li, David H. Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. Competition-level code generation with alphacode. *CoRR*, abs/2203.07814, 2022.

[11] Hugo Liu and Henry Lieberman. Programmatic semantics for natural language interfaces. In *CHI Extended Abstracts*, pages 1597–1600. ACM, 2005.

[12] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin B. Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie Liu. Codexglue: A machine learning benchmark dataset for code understanding and generation. *CoRR*, abs/2102.04664, 2021.

[13] Lance A. Miller. Natural language programming: Styles, strategies, and contrasts. *IBM Syst. J.*, 20(2):184–215, 1981.

[14] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. Codegen: An open large language model for code with multi-turn program synthesis. *arXiv preprint arXiv:2203.13474*, 2022.

[15] Jean E. Sammet. The use of english as a programming language. *Commun. ACM*, 9(3):228–230, 1966.

[16] Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. *CoRR*, abs/2302.04761, 2023.

[17] Imanol Schlag, Sainbayar Sukhbaatar, Asli Celikyilmaz, Wen-tau Yih, Jason Weston, Jürgen Schmidhuber, and Xian Li. Large language model programs. *CoRR*, abs/2305.05364, 2023.

[18] Yongliang Shen, Kaitao Song, Xu Tan, Dongsheng Li, Weiming Lu, and Yueting Zhuang. Hugginggpt: Solving AI tasks with chatgpt and its friends in huggingface. *CoRR*, abs/2303.17580, 2023.

[19] Dídac Surís, Sachit Menon, and Carl Vondrick. Vipergpt: Visual inference via python execution for reasoning. *CoRR*, abs/2303.08128, 2023.

[20] Stefanie Tellex, Thomas Kollar, Steven Dickerson, Matthew R. Walter, Ashis Gopal Banerjee, Seth J. Teller, and Nicholas Roy. Understanding natural language commands for robotic navigation and mobile manipulation. In *AAAI*. AAAI Press, 2011.

[21] Stefanie Tellex, Ross A. Knepper, Adrian Li, Daniela Rus, and Nicholas Roy. Asking for help using inverse semantics. In *Robotics: Science and Systems*, 2014.

[22] Sida I. Wang, Samuel Ginn, Percy Liang, and Christopher D. Manning. Naturalizing a programming language via interactive learning. In *ACL (1)*, pages 929–938. Association for Computational Linguistics, 2017.

[23] Yushi Wang, Jonathan Berant, and Percy Liang. Building a semantic parser overnight. In *ACL (1)*, pages 1332–1342. The Association for Computer Linguistics, 2015.

[24] Catherine Wong, Kevin M Ellis, Joshua Tenenbaum, and Jacob Andreas. Leveraging language to learn program abstractions and search heuristics. In *International Conference on Machine Learning*, pages 11193–11204. PMLR, 2021.

[25] Tongshuang Wu, Ellen Jiang, Aaron Donsbach, Jeff Gray, Alejandra Molina, Michael Terry, and Carrie J. Cai. Promptchainer: Chaining large language model prompts through visual programming. *CoRR*, abs/2203.06566, 2022.