Programming without a Programming Language: Challenges and Opportunities for Designing Developer Tools for Prompt Programming

Alexander J. Fiannaca
afiannaca@google.com
Google Research
Seattle, Washington, USA

Chinmay Kulkarni
ckulkarni@google.com
Google Research
Atlanta, Georgia, USA

Carrie Cai
cjcai@google.com
Google Research
Mountain View, California, USA

Michael Terry
michaelterry@google.com
Google Research
Cambridge, Massachusetts, USA

ABSTRACT

Existing tools for writing prompts for language models (known as "prompt programming") provide little support to prompt programmers. Consequently, as prompts become more complex with the addition of multiple input/output examples ("few-shot" prompts), they can be hard to read, understand, and edit. In this work, we observe that prompts are often used to solve complex problems, but lack the strict grammar of a traditional programming language. We describe methods for extracting the semantically meaningful structure of natural language prompts (e.g., regions of the prompt representing a preamble or input/output examples) in the absence of a rigid formal grammar, and demonstrate a range of editor features that can leverage this information to assist prompt programmers. Finally, we relate initial feedback from design probe explorations with a set of domain experts and provide insights to help guide the development of future prompt editors.

CCS CONCEPTS

• **Human-centered computing** → **User interface programming**.

KEYWORDS

prompt programming, language models

ACM Reference Format:
Alexander J. Fiannaca, Chinmay Kulkarni, Carrie Cai, and Michael Terry. 2023. Programming without a Programming Language: Challenges and Opportunities for Designing Developer Tools for Prompt Programming. In *Extended Abstracts of the 2023 CHI Conference on Human Factors in Computing Systems (CHI EA '23), April 23–28, 2023, Hamburg, Germany*. ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3544549.3585737

CHI EA '23, April 23–28, 2023, Hamburg, Germany

ACM ISBN 978-1-4503-9422-2/23/04.
https://doi.org/10.1145/3544549.3585737

1 INTRODUCTION

Large language models (LLMs) (e.g. [1, 3, 11]) can often follow natural-language-like instructions [2], and are increasingly designed for this purpose [12]. These capabilities allow users to rapidly accomplish many tasks through "prompt programming" [2, 13] (i.e., the process of engineering a natural language prompt [8]).

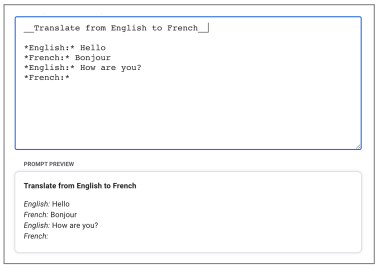
Prompt programming allows users to express their programming intent in plain language, rather than a specially designed programming language. For example, to create a basic English-French translation prompt, it is sufficient to provide a few examples to the model, followed by the word or phrase to translate:

```
English: Hello. French: Bonjour
English: Let's go to the market. French:
```

More recent instruction-tuned models [7, 12] are able to perform that translation with an even simpler expression of intent, such as: `Translate the following phrase into French: Let's go to the market.`

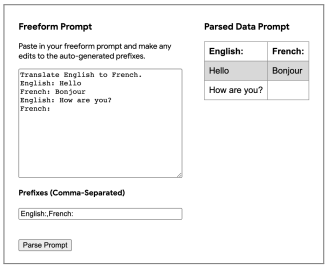
While early work has explored methods for aiding prompt engineers to find effective prompts [9, 10], existing prompt editing *interfaces* (e.g., [4, 6]) only provide basic text editing interactions. Just as programmers often need to perform editing operations over syntactically or semantically meaningful components in their code (e.g., refactoring, inserting code completions, etc.), prompt programmers often need to perform editing operations over semantically meaningful regions of their prompt (e.g., inserting a few-shot example, renaming keywords, refactoring the template for all examples, etc.). However, the lack of a predefined grammar for prompts makes it difficult to support these interactions. Therefore, prompt programming presents a fundamentally new challenge for programming tools: how do you support programming without a well-defined programming language?

In this paper, we consider the opportunities and challenges of creating a dedicated LLM prompt editor. Taking existing feature-rich integrated development environments (IDEs) as points of inspiration, we consider how we can support common IDE features in the context of prompt writing. Though language models do not require rigid programming semantics, prompt programmers nonetheless write prompts containing semantically meaningful components (e.g., regions of the prompt that set context, provide input/output examples, etc.). We leverage this observation to infer



(a) TagStruct: Manual metadata markup language proxy

(b) SelectStruct: Rich text editor for manual metadata labeling



(c) AutoStruct: Heuristic-Based Just-In-Time Grammar Detection

(d) AutoStruct+: Revised Heuristic-Based JIT Grammar Detection

Figure 1: Semantic Structure Prototypes: Manual Labeling and Automated JIT Grammar Detection

an approximate "grammar" for the prompt program that describes the semantic structure of the prompt. This "grammar" makes it straightforward to provide common IDE capabilities such as syntax highlighting and refactoring.

We demonstrate this approach with a set of functional prototypes that act as design probes (fig. 1). With these prototypes, we explore both manual annotation of prompt metadata representing the prompt's approximate grammar, and the benefits and challenges of inferring this metadata automatically as the prompt is being created (we call this "just-in-time/JIT grammar inference"). We then explore how this semantic structure for a prompt can be operationalized to support prompt programmers in dedicated prompt editing environments.

Finally, we gathered preliminary feedback on these design probes from a set of domain experts. We present insights from these pilot tests of the various prototypes, and describe open questions, design challenges, and opportunities for supporting prompt programmers in the future.

Taken altogether, we make the following contributions with this paper:

- We identify a new research challenge for the development of programming language tool support: supporting the practice of prompt programming when there is no rigid grammar or syntax for the "language" of prompt programs.
- We introduce techniques for understanding and automatically inferring the semantic structure of few-shot prompts as one response to this challenge, and instantiate these techniques through a set of prompt editor prototypes (fig. 1) inspired by IDEs for traditional programming.
- Finally, we provide insights and feedback from domain experts in initial pilot testing of these prototypes, suggesting implications for future work.

2 ANATOMY OF A PROMPT

Unlike traditional programming languages, LLM prompts do not have a predefined grammar. However, being composed of natural language, prompts have inherent semantic structure that reflects the programmer's intent. As we will demonstrate through our design probes below (Section 3), if we can identify the structural components of a prompt (what we call the prompt's metadata), we can derive an approximate "grammar" for each prompt. In this section, we define common structural components of prompts.

Zero-shot prompts are the simplest form of LLM prompts. These prompts simply state the problem to solve or the text to complete without providing repeated input and output examples. In the case of the English-to-French translation example, Fig. 2a demonstrates a zero-shot formulation of the prompt. **Few-shot prompts** (Fig. 2b) build on zero-shot prompts by including multiple examples of inputs and outputs. In practice, prompts can include many examples (the number of examples is often constrained by the maximum size of input, i.e., the input window of the model). A **preamble** is any text provided before the repeated examples in a few-shot prompt. Preambles are often used to describe the task the LLM is expected to perform or to provide additional contextual information to the model (Fig. 2c).

In few-shot prompts, examples often share a common structure consisting of **template components** (static text that is the same

```
The translation of the English phrase "Let's
go to the market" in French is
```

(a) Zero-Shot

```
English: Hello
French: Bonjour
English: How are you?
French: Comment ça va?
...
English: Let's go to the market.
French:
```

(b) Few-Shot

```
Translate the following English phrase to
French.
English: Hello
French: Bonjour
...
English: Let's go to the market.
French:
```

(c) Few-Shot with a Preamble

Figure 2: Example LLM prompts with varying semantic structures

in each repeated example) and **data** (inputs and outputs that are unique for each example). For example, in Fig. 2b, `English:` and `French:` are template components and `Hello`, `Bonjour`, `How are you?`, `Comment ça va?`, and `Let's go to the market` are data. Importantly, template components aren't necessarily required and when examples contain template components, they are not limited to a single input or a single output.

3 PROTOTYPES: EXTRACTING THE SEMANTICALLY MEANINGFUL STRUCTURE OF A PROMPT

Just as IDEs require knowledge of code syntax and semantics to support editing features, prompt editors need to have an understanding of the semantic structure of natural language prompts. For few-shot prompts, it is sufficient to know 1) the set of shared template components and 2) the locations they occur at in the prompt (Fig. 3). From this information (which we refer to as **prompt metadata**), we can find the locations of the preamble, each example, and the input/output data within each example. We can consider the list of template components repeated for each example as an **example template** defining the shared structure for all examples in the prompt (Fig. 3).

We consider this metadata a **just-in-time (JIT) grammar**, approximating the semantic structure of the prompt. This grammar and semantic structure is computed as-needed (hence, "just-in-time"), and represents the system's current estimate of the conceptual intent (e.g., preamble, examples, etc.) rather than its exact underlying syntax. This distinction parallels the fundamental difference between programming language code (where parsing focuses on abstract syntactic representations) and natural language prompts (where the focus is on conceptual intent), and highlights the core challenge in developing prompt editors: *defining a robust process of acquiring prompt metadata.*

3.1 Semantic Structure Prototypes

We developed a series of prototypes (Fig. 1) to explore two potential avenues for acquiring the metadata that describes the semantically meaningful structure of a prompt: manual structure labeling, and automated structure inference.

For manual structure labeling, we explored two prototypes. **TagStruct** (Fig. 1a) is a Markdown editor with a live-rendered HTML view we appropriated as a proxy for an editor supporting a hypothetical custom prompt markup language. **SelectStruct** (Fig. 1b) is a custom rich text editor with support for labeling semantically meaningful regions by selecting text and applying a semantic style (akin to italicizing text in a standard rich text editor).

For automated structure detection, we developed **AutoStruct** (Fig. 1c). This prototype consists of an HTML `textarea` component for composing a prompt and a heuristic-based algorithm for automatically detecting the semantically-meaningful components of the prompt. The output from this algorithm is the *prompt metadata* (Section 3.2). Following initial testing with this prototype, we developed a revision, **AutoStruct+** (Fig. 1d), that improved upon the AutoStruct heuristics for prompt metadata detection and implemented several prompt editor features (Section 4).

3.2 Inferring Prompt Metadata Automatically

Implementing AutoStruct and AutoStruct+ required developing a method to automatically infer prompt metadata (the JIT grammar). Without the rules of a programming language's predefined grammar to rely upon, we must rely instead on heuristics to infer the JIT grammar. We implemented these heuristics as regular expressions applied to the prompt to extract template component strings. The heuristics include:

- Template components must be at least 3 characters in length and occur multiple times in the prompt.
- The first occurring template component must start at the beginning of a line.
- No template component may be a prefix of another template component.
- All template components must occur in every example, except the final example.

These heuristics seek to strike a balance that allows for correct inference of template components for most prompts while not being overly prescriptive with respect to the expected structure of natural language prompts. While this implementation allows for rapid prototyping, heuristics inevitably result in edge cases where the JIT grammar cannot be accurately inferred. We discuss ways to overcome this limitation in the Discussion section.

4 APPLYING METADATA TO BUILD A PROMPT EDITOR

Our four prototypes differ in how the JIT grammar is detected, whether by explicit user actions (TagStruct, SelectStruct) or automatically (AutoStruct, AutoStruct+). These prototypes then allow us to implement a range of IDE-inspired prompt editor features to improve the experience of prompt programming.

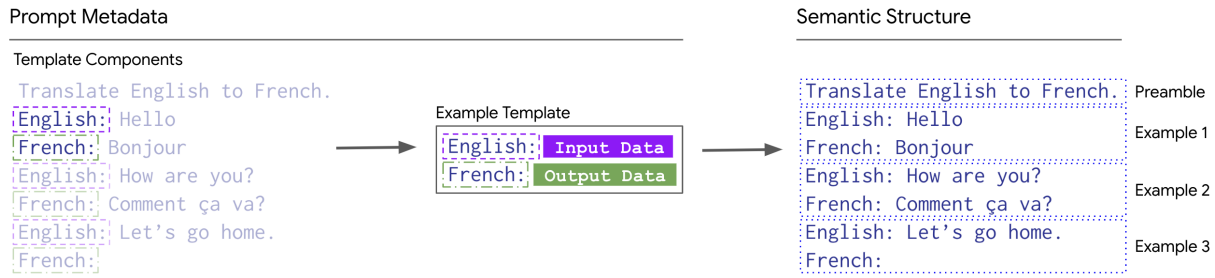


Figure 3: Prompt Metadata and Semantic Structure Overview. Template components are identified with manual labeling or heuristic-based detection. Given the template components, the prompt can be parsed into the preamble (text before the first template component) and examples (each containing the set of template components and input and output data).

4.1 Semantic Highlighting

As prompt programmers iterate on a few-shot prompt, they often add examples (e.g., to increase example diversity). As the number of examples grows, prompts can became unwieldy, and difficult to read and understand. However, with even a few examples, the prompt metadata described in section 3.2 allows us to apply styles that visually distinguish the semantically meaningful components of the prompt (similar to syntax highlighting for code in IDEs).

In SelectStruct and AutoStruct+ (Fig. 1), we implemented three types of semantic highlighting. First, we styled the font color of template components and example input/output data to help programmers perceive/comprehend structure within examples. Second, we visually grouped examples by adding additional visual padding above and below each example to help programmers perceive/comprehend the overarching structure of the prompt. Finally, we explored displaying contextual information above examples to assist prompt programmers in more quickly understanding the context of a given example.

4.2 Refactoring

Another key pain point for prompt programmers using existing prompting tools is when the prompt programmer wants to change the template that is shared by all examples in the prompt (i.e., edit a template component, such as the "English" and "French" terms in the example in figure 3). Changing the shared template for a single example requires that change be propagated to every example in the prompt. In Prototypes B and D, we implemented a "mass editing" feature which ensures that changes to example template components are automatically reflected in the corresponding template components for all examples simultaneously.

4.3 Autosuggest and Prompt Templates

Code completion for contextually appropriate symbols (e.g., variable names) and valid code templates (e.g., the scaffolding for a loop) are important features in traditional IDEs. We drew upon these ideas to implement similar features to support prompt programmers in writing their prompts. As observed in section 3, template components from a prompt's metadata can be viewed as an example template with placeholders for input and output data. In SelectStruct, we suggest these templates as code completions, allowing the user to focus on adding new content rather than typing repeated template text. Furthermore, knowledge of the example template and the data in each input/output field for each instantiation of the template allows us to suggest appropriate new values for these fields as new examples are inserted. In SelectStruct, when the user inserts a suggested prompt template, or types a know template component, the prototype sends a request to the language model itself containing a list of corresponding example inputs or outputs from the preceding examples in order to generate an appropriate suggestion for data to insert. This feature can help the user to find creative new example data for their prompt.

4.4 Structured Editing

While LLM prompts are simply a string of natural language text, knowledge of the prompt's metadata allows for prompts to also be viewed as data objects consisting of an optional preamble and a table of example data, with each row corresponding to an example and each column corresponding to an input or output field in the example template. Fig. 1d shows an example of how this type of data could be rendered as a table of values. We refer to the plain-text view of a prompt as the **freeform** view, and the structured view of a prompt as the **data** view.

Importantly, the freeform and data views of a prompt are equivalent, though the data view may provide additional affordances which become useful as the scale of the example dataset increases. For instance, the data view of a prompt may support integration with large datasets stored in different formats (e.g., connecting a prompt to large external dataset and automatically sampling a selection of examples to include in the final prompt submitted to the model). Additionally, displaying a prompt in the data view allows for separately editing the prompt template and the data. Finally, the data view may also be a helpful form factor when working to prepare a prompt for prompt tuning [5].

5 PILOT EXPLORATORY STUDIES

We piloted a series of semi-structured interviews and design ideation sessions with domain experts using the probes described above (see fig. 1). These sessions were not designed to be rigorous evaluations of the prototypes, but rather to further develop our initial design ideas and discover limitations and challenges. The mixed levels of fidelity of the various design probes presented in each

session supported the goal of encouraging ideation and creative exploration.

Topics explored in these sessions varied (and evolved across sessions), including: JIT grammar inference, structured editing, semantic highlighting, refactoring, autosuggest, and prompt templates. Altogether, the sessions included 8 participants, 7 of whom are domain experts with extensive experience writing LLM prompts, and 1 of whom is more novice (only having written several prompts). Sessions lasted between 30 minutes and one hour and were conducted via an online video conferencing tool. We describe the initial findings from these sessions below, but we leave rigorous evaluation and analysis of these prototypes to future work.

5.1 Participants did not want to manually tag prompt semantics

We found that because manually tagging or selecting prompt semantics required some effort (even if minimal), participants wondered if there was sufficient pay-off, especially when the number of few-shot examples is small. This suggests that prompt editors may need to automatically infer prompt grammar, and deal with grammar ambiguities automatically, with a way to either "fail softly" or correct and recover when the parsing fails.

5.2 Users needed clear separation between visual markups to the prompt and the prompt itself

When asked to mark up semantic units of the prompt, users were concerned about which parts of the prompt were being sent to the model. This was especially true with TagStruct, where users wanted to ensure that the text used to mark up the prompt was not also being sent to the model. For these reasons, semantic highlighting also confused users without programming experience with IDEs. For example, one non-programming user wondered whether the bolding or text colors would also be sent to the model. For this reason, we updated the AutoStruct+ prototype to show two views, with an explicit toggle to switch between viewing the plain text prompt and the prompt with semantic highlighting.

5.3 Semantic Highlighting may aid reading prompts and collaboration

In general, participants appreciated semantic highlighting, especially because they saw it as a way to reduce the effort of reading and understanding prompts. In our design sessions, several participants spontaneously brainstormed the ways in which semantic highlighting could be implemented. Using colors to differentiate between template components and data was particularly seen as helpful. Some users felt that the improved legibility would be particularly useful when sharing their prompt with collaborators.

5.4 Participants found structured editing useful and intuitive, but differed on utility of structured views

Participants were divided on preferences between the freeform view and the structured data view. Participants who preferred the freeform view indicated that this was because it allowed iteration, integrated with other tools (e.g. python notebooks), and was subjectively faster to use (keyboard-only editing is an experience that is familiar to people who write code with IDEs). Multiple participants cited the ability to quickly copy-paste as a benefit of the freeform view over the structured data view. In the SelectStruct and AutoStruct+ prototypes, these participants found it useful to quickly edit all instances of a template component simultaneously, and to be able to quickly insert suggested example templates.

Participants who preferred a structured view indicated that the structured data table may better facilitate collaboration with multiple collaborators working together as it allowed participants to visually separate out rows each collaborator was working on. Additionally, participants who preferred the structured data view found it useful to have the ability to operate over "rows" of data and edit template components in a table. Participants also proposed that the structured form factor could support data sheet functionality (e.g., randomizing the data order, or drag-and-drop to reorder rows).

Taken together, these findings suggest an additional benefit of inferring JIT grammar: it allows for multiple views of the same underlying data to suit different participants. For instance, future work could consider views that prioritize real-time collaboration, or data auditing.

5.5 Current tools do not scaffold formation of useful mental models

In our sessions, we found that participants had several questions about whether they were formulating effective prompts, and even what constitutes an effective prompt. For instance, participants wondered if delimiters were necessary to hint the model (e.g., the "`:`" after `English` in fig. 2b), and if so, was there a "best" delimiter to use? Participants also wondered if line breaks, formatting, and white space affected their prompt performance, and if so, how they could improve it.

While participants' mental models were influenced by the tools they currently used and their preferences for prompt views (e.g., those that preferred structured views were less concerned with formatting issues), overall, we found that current tools (whether model "Playgrounds", or Python notebooks) did little to build effective mental models.

6 DISCUSSION AND IMPLICATIONS FOR FUTURE WORK

This paper identifies a new challenge to programming tool design – supporting prompt programming, where there is no well-defined programming language. In our prototype-driven explorations, we found that users expected programming support to be automatic, and found such automatic support useful. However, challenges remain in scaffolding useful mental models and in making programming tools accessible to non-traditional programmers who use prompt programming. This work also opens up new research questions, some of which we discuss below.

6.1 Mitigating and handling errors in JIT grammar parsing

Our prototypes use heuristics for detecting the grammar of the prompt. By definition, heuristics are not always reliable. For example, in our testing, we found a prompt that summarized SQL queries in natural language for which our heuristic-based algorithm incorrectly detected `SELECT` and `WHERE` as template components because they occurred more frequently that the actual template components in the prompt (as a result of SQL subqueries in the example data).

Our heuristic-based approach may be improved by employing models trained on labeled semantics from prompt, or program synthesis techniques [14]. But our experience suggests that graceful degradation of tool support or recovery from errors may be equally important. This problem is made more complex because users may be unable to distinguish between grammar parsing errors and the possibility they wrote a "bad prompt" (Section 5.5).

6.2 Leveraging users' implicit knowledge about writing prompts

Our participants repeatedly shared that they took a long time to learn how to write "effective" prompts (e.g., accurate classifiers, generation free of hallucinations, etc), and that they felt that current tools required them to learn primarily through experimentation and intuition.

In general, we found that experienced prompt programmers possessed a lot of implicit knowledge about how to write prompts, such as how to scope down a problem to a scope that the LLM can solve. Tool support could accelerate such learning, for instance, through nudging users towards creating more well-scoped prompts.

6.3 Limitations

Models are changing rapidly, and consequently, the way prompt programmers design prompts will also change to accommodate the abilities of the models they are target. With the rise of instruction-tuned models [7, 12], techniques for supporting prompt programmers that are designed for few-shot prompts, such as those described in sections 3 and 3.2 may become less relevant.

7 CONCLUSION

In this work, we explored challenges and opportunities for supporting prompt programmers through the development of prompt editor features that operate over the semantic structure of LLM prompts. We presented methods for automatically inferring a prompt's semantic structure, and showed how this structure can be leveraged to implement editor features like semantic highlighting, autosuggest, and structured data views. We conducted initial pilot testing of these prototypes and presented key insights from this early testing.

REFERENCES

[1] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. In *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin (Eds.), Vol. 33. Curran Associates, Inc., 1877–1901. https://proceedings.neurips.cc/paper/2020/file/1457c0d6bfcb4967418bfb8ac142f64a-Paper.pdf

[2] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. In *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin (Eds.), Vol. 33. Curran Associates, Inc., 1877–1901. https://proceedings.neurips.cc/paper/2020/file/1457c0d6bfcb4967418bfb8ac142f64a-Paper.pdf

[3] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, Parker Schuh, Kensen Shi, Sasha Tsvyashchenko, Joshua Maynez, Abhishek Rao, Parker Barnes, Yi Tay, Noam Shazeer, Vinodkumar Prabhakaran, Emily Reif, Nan Du, Ben Hutchinson, Reiner Pope, James Bradbury, Jacob Austin, Michael Isard, Guy Gur-Ari, Pengcheng Yin, Toju Duke, Anselm Levskaya, Sanjay Ghemawat, Sunipa Dev, Henryk Michalewski, Xavier Garcia, Vedant Misra, Kevin Robinson, Liam Fedus, Denny Zhou, Daphne Ippolito, David Luan, Hyeontaek Lim, Barret Zoph, Alexander Spiridonov, Ryan Sepassi, David Dohan, Shivani Agrawal, Mark Omernick, Andrew M. Dai, Thanumalayan Sankaranarayana Pillai, Marie Pellat, Aitor Lewkowycz, Erica Moreira, Rewon Child, Oleksandr Polozov, Katherine Lee, Zongwei Zhou, Xuezhi Wang, Brennan Saeta, Mark Diaz, Orhan Firat, Michele Catasta, Jason Wei, Kathy Meier-Hellstern, Douglas Eck, Jeff Dean, Slav Petrov, and Noah Fiedel. 2022. PaLM: Scaling Language Modeling with Pathways. https://doi.org/10.48550/ARXIV.2204.02311

[4] Ellen Jiang, Kristen Olson, Edwin Toh, Alejandra Molina, Aaron Donsbach, Michael Terry, and Carrie J Cai. 2022. PromptMaker: Prompt-based Prototyping with Large Language Models. In *CHI Conference on Human Factors in Computing Systems Extended Abstracts*. 1–8.

[5] Brian Lester, Rami Al-Rfou, and Noah Constant. 2021. The Power of Scale for Parameter-Efficient Prompt Tuning. https://doi.org/10.48550/ARXIV.2104.08691

[6] OpenAI. [n. d.]. Playground - OpenAI API. https://platform.openai.com/playground. (Accessed on 03/02/2023).

[7] Long Ouyang, Jeff Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul Christiano, Jan Leike, and Ryan Lowe. 2022. Training language models to follow instructions with human feedback. https://doi.org/10.48550/ARXIV.2203.02155

[8] Laria Reynolds and Kyle McDonell. 2021. Prompt programming for large language models: Beyond the few-shot paradigm. In *Extended Abstracts of the 2021 CHI Conference on Human Factors in Computing Systems*. 1–7.

[9] Hendrik Strobelt, Albert Webson, Victor Sanh, Benjamin Hoover, Johanna Beyer, Hanspeter Pfister, and Alexander M Rush. 2022. Interactive and Visual Prompt Engineering for Ad-hoc Task Adaptation with Large Language Models. *IEEE transactions on visualization and computer graphics* 29, 1 (2022), 1146–1156.

[10] Ben Swanson, Kory Mathewson, Ben Pietrzak, Sherol Chen, and Monica Dinalescu. 2021. Story centaur: Large language model few shot learning as a creative writing tool. In *Proceedings of the 16th Conference of the European Chapter of the Association for Computational Linguistics: System Demonstrations*. 244–256.

[11] Romal Thoppilan, Daniel De Freitas, Jamie Hall, Noam Shazeer, Apoorv Kulshreshtha, Heng-Tze Cheng, Alicia Jin, Taylor Bos, Leslie Baker, Yu Du, YaGuang Li, Hongrae Lee, Huaixiu Steven Zheng, Amin Ghafouri, Marcelo Menegali, Yanping Huang, Maxim Krikun, Dmitry Lepikhin, James Qin, Dehao Chen, Yuanzhong Xu, Zhifeng Chen, Adam Roberts, Maarten Bosma, Vincent Zhao, Yanqi Zhou, Chung-Ching Chang, Igor Krivokon, Will Rusch, Marc Pickett, Pranesh Srinivasan, Laichee Man, Kathleen Meier-Hellstern, Meredith Ringel Morris, Tulsee Doshi, Renelito Delos Santos, Toju Duke, Johnny Soraker, Ben Zevenbergen, Vinodkumar Prabhakaran, Mark Diaz, Ben Hutchinson, Kristen Olson, Alejandra Molina, Erin Hoffman-John, Josh Lee, Lora Aroyo, Ravi Rajakumar, Alena Butryna, Matthew Lamm, Viktoriya Kuzmina, Joe Fenton, Aaron Cohen, Rachel Bernstein, Ray Kurzweil, Blaise Aguera-Arcas, Claire Cui, Marian Croak, Ed Chi, and Quoc Le. 2022. LaMDA: Language Models for Dialog Applications. https://doi.org/10.48550/ARXIV.2201.08239

[12] Jason Wei, Maarten Bosma, Vincent Y. Zhao, Kelvin Guu, Adams Wei Yu, Brian Lester, Nan Du, Andrew M. Dai, and Quoc V. Le. 2021. Finetuned Language Models Are Zero-Shot Learners. https://doi.org/10.48550/ARXIV.2109.01652

[13] Tongshuang Wu, Michael Terry, and Carrie Jun Cai. 2022. AI Chains: Transparent and Controllable Human-AI Interaction by Chaining Large Language Model Prompts. In *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems* (New Orleans, LA, USA) *(CHI '22)*. Association for Computing Machinery, New York, NY, USA, Article 385, 22 pages. https://doi.org/10.1145/3491102.3517582

[14] Tianyi Zhang, London Lowmanstone, Xinyu Wang, and Elena L Glassman. 2020. Interactive program synthesis by augmented examples. In *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology*. 627–648.

Received 19 January 2023