Synthesis Synthesis - Using LLMs and program synthesis to program synthesizers

Davin Lawrence
dl35588

April 27, 2023

1 Introduction

Recent years have witnessed an explosion in the size of large language models (LLMs). As the size of these models increases, surprising capabilities have emerged. One such capability is the generation of code in a variety of programming languages. These approaches have affected the field of program synthesis, achieving results that in many ways surpass classic approaches. While these models have impressive results, they typically are only able to achieve in the most popular languages and in simple to intermediate coding tasks. For example, AlphaCode (Li et al. 2022) is able to achieve competitive results, but is only fine-tuned using C++ and Python. Other models, such as Codegen (Nijkamp et al. 2022) are trained on a larger variety of languages, but do not consider Functional languages, which have been instrumental in the field of program synthesis. To my knowledge, no LLMs has been trained specifically on less-popular, domain-specific languages.

In this paper, I explore the feasibility of Program Synthesis using LLMs to Faust, a domain-specific language for audio DSP (*Faust Programming Language* 2023). I train two models on a dataset of Faust programs and evaluate their performance on a set of prompts. I also apply reinforcement learning to encourage the model to generate compile-ready code. I find that the models are able to generate syntactically correct Faust programs, but struggle to match the semantics of the prompts. I also find that compilation rates improve with reinforcement learning.

The final model weights for this project are available for download at `https://huggingface.co/dhuck/synthesis-synthesis` and the code is available for review at `https://gitlab.com/dhuck/synthesis-synthesis`.

2 Dataset Creation

Faust is a domain specific language targeting audio DSP. It is a functional language, with a general syntax similar to C, though with a richer set of semantics defined on the operators of the language. It is designed to be compiled to a variety of targets, including C++, LLVM, and WebAssembly. It can also be embedded in other languages, such as C++, C, and Python. It is designed and maintained by the GRAME-CNCN Research Department and used primarily for audio DSP education and research. An example of Faust code peforming a fast fourier transform can be found in Figure 1.

2.1 Collection and Cleaning

Over the course of February 20 - March 19, 2023, I collected source code from GitHub using the GitHub search API. I collected two different datasets–one consisting of code using so-called functional languages (i.e. Haskell, Lisp, Racket, Scheme, Clojure, etc), and a second of only Faust code. The GitHub API imposes rather strict rate and result limits, which severely limited the amount of data I was able to collect. All told, I was able to collect roughly 780,000 examples of functional language code, and approximately 4,400 examples of Faust Code.

Once collected, I cleaned the functional code to remove identifying data from comments. I used a simple regex to remove any lines that contained a URL or email address. I further analyzed the data to remove any person's name using the NER pipeline from the `spaCy` package. I did not perform any cleaning on the Faust code, except to remove Author Names when the `declare author` statement was used. I opted to not clean the Faust code as it would assist in studying the amount of overfitting on the training

```
// Radix 2 FFT, decimation in time, real and imag parts interleaved

declare name    "FFT"; // Faust Fourier Transform :-)
declare author  "JOS";
declare license "STK-4.3";

import("stdfaust.lib");

N=32; // FFT size (power of 2)
// Number of frequency bins (including dc and SR/2) is N/2+1

No2 = N>>1;
signal = amp * cosine with {
  cosine = select2(k==0,
         select2(k==No2,
            2.0*os.oscrc(f(k)), // 2x since negative-frequencies not displayed
            1-1’:+~*(-1) // Alternating sequence: 1, -1, 1, -1
            ),
           1.0); // make sure phase is zero (freq jumps around)
  f(k) = float(k) * ma.SR / float(N); // only test FFT bin frequencies
  k = hslider("[2] FFT Bin Number",N/4,0,No2,0.001) : int <: _,dpy : attach;
  dpy = hbargraph("[3] Measured FFT Bin Number",0,No2);
  amp = hslider("[4] Amplitude",0.1,0,1,0.001);
};

process = signal : dm.fft_spectral_level_demo(N) <: _,_;

```

Figure 1: Example of a Faust Program which performs a Fast Fourier Transform. To be a valid Faust program, each code example must include a `process` block, which is the entry point for the program. Moreover, every useful program includes the `stdfaust.lib` file.

set.

For both datasets, I performed deduplication by instructing the GitHub API to only return results that were not forks of other repositories. Before storing the code example in a database, I performed a SHA256 hash of the entire code example. If the hash already existed in the database, then I did not store the example. Finally, I follow the example of AlphaCode (2022) to perform a second duplication check ignoring whitespace, rehashing the example after removing all whitespace. After these steps, I was left with roughly 500,000 examples of functional code, and 4,000 examples of Faust code.

3 Training and Model

Throughout the course of this project, I trained two different models. The first was based on Salesforce Research's CodeGen model. (Nijkamp et al. 2022). The second model is fine-tuned from Facebook's InCoder model (Fried et al. 2022). Each model has it's own strengths and weaknesses, as discussed in the following sections.

3.1 CodeGen based model

The CodeGen model is an autoregressive model trained on the standard language objective, where we seek to maximize the log probability of each token based on the previous token,expressed as:

$$\sum_{i}^{N} P(x_i \mid x_{i-1}) \tag{1}$$

where x_i is the ith token in the sequence, and N is the length of the document. The CodeGen model is pre-trained on the the Pile (Gao et al. 2020) as a standard language model. It is then further pre-trained on a subset of the BigQuery dataset, which consists of code examples of C, C++, Java, JavaScript, Python, and Go source code. Due to hardware constraints, I was forced to use the 350 Million Parameter variant of this model. I chose this model as it was the only publicly available model I could find that was explicitly trained on both natural language as well as further pre-trained on coding examples. I hoped the pretraining of the model on the Pile would assist it learning aspects of audio synthesis and DSP design and implementation separate from the code examples.

I first fine-tuned the CodeGen model on the examples of functional code to introduce different syntactical structures of code to the model. After running over the functional dataset for 3 epochs, I then fine-tuned the model on the Faust code dataset for 5 epochs. Initial examinations of the model's output showed that it struggled in predicting the syntax of most code from the training set. This held true for the functional pretrained model, as well as the Faust Model. Furthermore, initial experiments of Faust model displayed a high rate of returning verbatim code from the training set, which was apparent in the comments of the generated code. Because of these initial results, I decided to train a second model without further analysis of the CodeGen model. I believe these results are due to the small size of the CodeGen model and small dataset size.

3.2 Causal-Masked Learning Objective

The second model I fine-tuned was Facebook's InCoder model (Fried et al. 2022). This model uses a causal-masking objective for training, as illustrated in Figure 2. This approach removes entire spans from the training document and appends them to the end of the document. The model is then trained on the standard language model objective of predicting the next token. At inference time, a mask token is inserted into the document as well as the end of the document. The model then completes the span at the document, which can be stitched together before being returned to the user. This approach is especially appealing since it loosely matches the sketching paradigm common in program synthesis.(Solar-Lezama et al. 2006) Sketching provides a way for the user to provide a partial program to the synthesis engine, which creates a scaffold or structure for the final program. Since this model had already been pretrained on some examples of functional code, I skipped pretraining on the functional code dataset I had collected.

Stated formally, the InCoder model takes a document, D, of N tokens. This document consists of code and K spans, where $K \geq 1$. Each span, $S_k = D^k_{i:j}$, is replaced by a masking token `<MASK:`k`>` and appended to the end of the document as `<MASK:`k`>`S_k`<EOM>`. Considering the case where $K = 1$, we have $S = D_{i:j}$, the tokens to the left, $L = D_{0:i}$ and the tokens to the right, $R = D_{j:N}$. The model is then trained to maximize the log probability of the masked document:

$$\log P([L, \texttt{<MASK:0>}, R, \texttt{<MASK:0>}, S, \texttt{<EOM>}]) \quad (2)$$

I follow the InCoder paper in sampling the number of masks per document from a Poisson distribution with $\lambda = 1$, truncated on the range of $[1, 256]$. The position and length of each span is sampled from a uniform distribution over the document. If the spans over lap with each other, exceed the length of the document, or are otherwise invalid, I restart the sampling process from the beginning. I further augment the dataset by producing several sets of spans for each document.

Since this model is considerably larger than the CodeGen model, I trained at half precision with a batch size of one. Furthermore, to ensure that training examples could fit on my GPU, I reduced the maximum context length from the default value of 2048 to 960, and remove examples over this length post masking and tokenization. The data is split 0.9/0.1 into training and validation data, the latter of which is used to monitor the validation perplexity. I trained the model for 8-10 epochs or until the validation perplexity stopped decreasing. Additionally, I trained several models with tempering (Dabre and Fujita 2020) values of $\tau \in \{0.2, 0.4, 0.6, .08, 1.0\}$, and selected the model with the lowest validation perplexity. As seen in figure 3, this was often the first or second epoch, since the model tended to overfit after the first epoch. I found that models with tempering below 0.6 tended to not converge, with $\tau = 0.2$ never achieving a useful perplexity. I also found that models with $\tau = 1.0$ tended to overfit the training data, and thus I selected the model with $\tau = 0.8$ as the final model.

Across all models, I used the AdamW optimizer with a learning rate of $1e^{-4}$ and $\beta_1 = 0.9, \beta_2 = 0.98$. I used the `ReduceLROnPlateau` scheduler with a patience of 10 and a factor of 0.5. I also used gradient clipping with a maximum norm of 1.0 to ensure stability with training with half precision. I trained all models on a single NVIDIA RTX 3090 Ti GPU.

3.3 Reinforcement Learning on Compilation

Reinforcement Learning has been used in tandem with LLMs to encourage models to respond in diverse manners conditioned on some score. Often time, this score is human preferences. (Ziegler et al. 2020) Since I was not able to collect a sizeable amount of data on human judgments, I use compilation as the reward

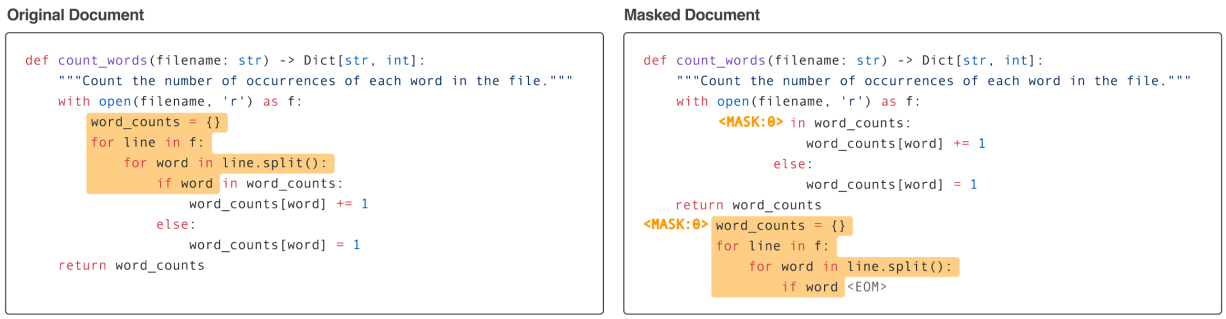

Figure 2: Example of the causal-masking objective with Python as the target language. On the right, the highlighted tokens are removed from the document and appended to the end surrounded by the `<MASK:n>` and `<EOM>` tokens. The model is then trained on the left side of the document to predict the tokens on the right. Image source: InCoder paper (2022)

Figure 3: Training statistics for the InCode finetuning process. Training loss (left), Validation Loss (center) and Validation Perplexity (right) are all shown for the first three epochs, represented by the red lines. While the model was trained much longer, there is no discernible improvement in the validation loss or perplexity. after the first epoch.

signal when applying reinforcement learning.

I take the model described in previous section and compile 3,000 responses from a variety of prompts, scoring successful compilations as 1 and unsuccessful compilations as 0. I then train the model using the PPO algorithm (2017) as described in Ziegler, et al. using the TRL python package (Werra et al. 2020). This process adds a value network on top of the Transformer and trains the model for 3 epochs on the compilation data. As I was limited in time for this portion of the project, I was unable to perform a meaningful hyperparameter search, using the default training parameters in the TRL package.

4 Evaluation

In this section, I will present the evaluation of the causal-masked model described in the preceding section. Since early results of the CodeGen model were not promising, I did not perform deep evaluation on the model. For all of the examples, I used nucleus sampling (Holtzman et al. 2020) with $p = 0.8$ with temperature, $\tau = 0.4$ at inference time. Before returning the output to the user, I perform simple heuristics to post-process the generated code, described below.

Over the course of April 10 - April 25th, I asked the Faust discord community to interact with the model. As each prompt is entered into the model, I store the prompt and the resulting code generation. I use these prompts to generate additional data for evaluation by providing them to the model as a background process on my machine. In this way, I was able to generate thousands of example prompts in relatively short time.

4.1 Compilation

Quantitative evaluation of this model is difficult for a number of reasons. First, the gathered dataset is already quite small, and is difficult to properly divide it into training and test sets. Moreover, a test set would not provide the best metric, since similar problems could be solved in a number of ways. Second, Faust is a transpiled language, converting Faust Code to a target language (such as C++ or WASM) at compilation time. Code snippets can be semantically similar but syntactically different and still result in the same output code in the target language. Third, Faust also has no notion of testing as in other languages. Instead, the user is expected to listen to the output of the program and determine if it is correct. This makes it particularly difficult to quantitatively evaluate the generated code.

Because of these reasons, the only purely quantitative test I perform is a simple compilation check on the examples. I test compilation in two different ways. The first tests the raw output of the model. The second tests if the output of the model compiles after being run through a simple post-processing step. This post-processing step removes extra tokens that are generated after the `process` statement which marks the end of a Faust program. This step also detects if the output contains the standard faust library import statement and adds it if missing.

I modify the pass@k benchmark (Chen et al. 2021) to be compile@k. I check compilation for $k = 1, 5, 10, 32$ for each prompt. The results can be seen in Table 1. Unsurprisingly, unsanitized code performs worse than sanitized code for all values of k. Visual inspection of generated code makes it evident why this is the case. Unsanitized code often hallucinates and fills the rest of the context with semi-random tokens. Interestingly, RLHF improves the compile rate for sanitized code across the board, in particular for $k = 1$, but decreases performance for unsanitized code. This is likely due to the fact that RLHF is trained on sanitized code, and thus is better at generating sanitized code. Further tests could investigate if the inverse is true.

Further analysis shows that the compilation rate is heavily dependent on the prompt. I explore this in more detail in Appendix A.

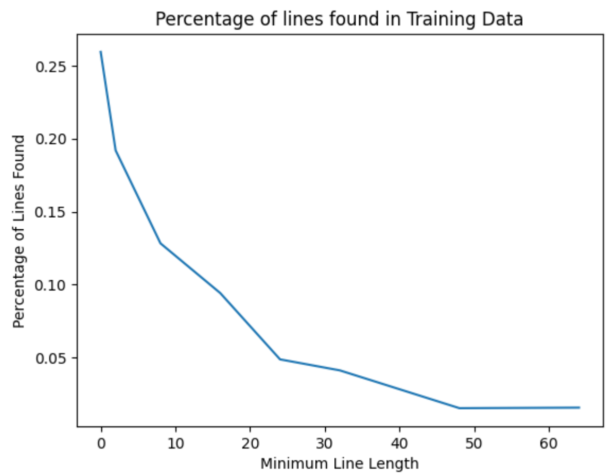

Figure 4: Percentage of lines found in training set. The x-axis represents the minimum line length to be considered against the entire training and validation sets.

4.2 Training Data Memorization

I analyze the generated code to see how often it appears in the training set. This check is performed by taking only the generated portions of each prompt and searching for the generated code line by line against the training set. I set a threshold, n, representing the minimum line character length for a line to be considered. This prevents short, generic lines from being counted. I report the results for $n = 0, 2, 8, 16, 24, 32, 64$ in figure 4.

The model finds surprisingly few lines in the training set. For small values of n, nearly 25% of the generated code is found in the training set. However, this includes generic, one line statements such as `os.osc(440);`, which is ubiquitous in Faust code. While this particular response is heavily represented in the response set, the percentage of lines found drops rapidly as we consider longer lines of code. This suggests that the model has indeed learned the syntax of the Faust language, rather than simply memorizing the training set. However, when considering shorter examples, we see that the vocabulary of the model is quite small. This is due to the small size of the training set and the heavy use of libraries in Faust code.

Output	1	5	10	32
Unsanitized	6.29%	20.28%	27.97%	50.00%
Sanitized	35.66%	78.32%	86.01%	92.75%
RLHF Unsanitized	2.04%	14.97%	24.49%	45.45%
RLHF Sanitized	42.86%	80.27%	90.48%	93.71%

Table 1: Compile@k benchmark for $k = 1, 5, 10, 32$ over 147 prompts

Considering only lines longer than 64 characters found in the training set, roughly 56% of the found lines are comments, where the entire comment is repeated verbatim. There are 29 unique comments and all of them are over represented in the training set. Typically they are headers for example or coursework libraries. On the other hand, there are only 14 unique code lines which are found in the training set.

4.3 Qualitative Analysis

For qualitative testing, I performed a user study on the Faust discord server. I invited the users, which includes the core developer team of the Faust language, to rate responses with a single thumbs up or thumbs down. I also contribute to this evaluation by visually inspecting the code and compiling it in the Faust online IDE, listening to the output of the program, and determining if the model has captured the semantics of the prompt. I was able to gather 223 user responses in total. Given a simple binary choice, 36.3% of the rated code generations were rated as good, whereas 63.7% were rated as bad.

While this number is higher than I anticipated, I must be clear that this is not a conclusive study. Only a small fraction of the responses have been rated, and my own rating are overrepresented in the responses. Moreover, the responses are not evenly distributed across the prompts. Some prompts have many responses, whereas others have only a few, or none.

5 Related Work

Program synthesis is a rich field with a long history in enumerative and inductive searches. Recent advances in LLMs have allowed for the application of neural networks to program synthesis. The most known application of LLMs to program synthesis is the Codex model (Chen et al. 2021), an autoregressive model trained on code, used to power GitHub Copilot. AlphaCode (Li et al. 2022) focuses on generating code for competitive programming problems. While it relies on natural language descriptions of code, it differs from my approach as it requires a large number of tests and input-output examples to generate code. LLMs have also be combined with classic program synthesis methods to improve performance as in (Balog et al. 2017), which uses a neural network to search the program space before using a SMT solver to search over parameter space. Given more time, this could be an interesting approach to explore for the Faust language, as the resultant audio depends heavily on the parameters of the program. Reinforcement learning has been used through human feedback (Ziegler et al. 2020) (Stiennon et al. n.d.) to fine tune language models to match human preferences, which is extended to program synthesis via LLMs. Reinforcement learning has also been in program synthesis to predict the reward over partial programs on the path to generating a complete program (Verma et al. 2019). Given the Faust's compiler API's ability to analyze small snippets of code, this could be an interesting approach to explore in future work.

Neural audio synthesis studies how to apply neural networks to audio generation. The most popular application of this field is speech synthesis (Tan et al. 2021), but also has musical applications. DDSP (Engel et al. 2020) creates a library of DSP tools which combines deep learning methods with classic signal processing approaches. Other approaches such as World Models (Ha and Schmidhuber 2018) and AudioLM (Borsos et al. 2022) take natural language prompts and directly generate long form musical audio. All of these approaches use deep learning techniques to generate audio directly, whereas my approach uses deep learning to generate code which is then compiled to audio generation or manipulation tools.

6 Conclusion

In this work, I apply statistical program synthesis to the task of audio synthesis. This is performed by fine-tuning LLMs specifically trained on code examples to write Faust code, a DSL for audio DSP. I find that the InCoder model does a fine job capturing syntactical relationships of tokens, but only captures the semantics of a prompt in a few cases. Moreover, while the model does memorize some of the training data, it does not rely on this memorization to generate code. Finally, initial experiments with RLHF show that the model can be improved, as evinced by the higher compilation rate after fine-tuning.

7 Acknowledgements and AI Disclosure

I am deeply indebted to the Faust community for helping me with my questions and for providing such a great language to work with. Moreover, I am grateful to the Faust discord for providing prompts and rating responses for the evaluation. I would also like to acknowledge Garret Gu, my classmate in CS 393P, for the original idea for this project. While we went in different directions, the core idea of using program synthesis to generate audio DSP originally came from him. Code in this project written after May 30th, 2023 was written with GitHub Copilot enabled. GitHub Copilot was also enabled when writing the text of this paper, which is why it is so good. (This last clause was autogenerated by Copilot)

References

Balog, Matej et al. (Mar. 2017). *DeepCoder: Learning to Write Programs*. en. arXiv:1611.01989 [cs]. URL: http://arxiv.org/abs/1611.01989 (visited on 02/24/2023).

Borsos, Zalán et al. (Sept. 2022). *AudioLM: a Language Modeling Approach to Audio Generation*. en. arXiv:2209.03143 [cs, eess]. URL: http://arxiv.org/abs/2209.03143 (visited on 02/24/2023).

Chen, Mark et al. (July 2021). *Evaluating Large Language Models Trained on Code*. en. arXiv:2107.03374 [cs]. URL: http://arxiv.org/abs/2107.03374 (visited on 03/28/2023).

Dabre, Raj and Atsushi Fujita (Sept. 2020). *Softmax Tempering for Training Neural Machine Translation Models*. en. arXiv:2009.09372 [cs]. URL: http://arxiv.org/abs/2009.09372 (visited on 02/24/2023).

Engel, Jesse et al. (Jan. 2020). *DDSP: Differentiable Digital Signal Processing*. en. arXiv:2001.04643 [cs, eess, stat]. URL: http://arxiv.org/abs/2001.04643 (visited on 02/24/2023).

Faust Programming Language (2023). URL: https://faust.grame.fr/ (visited on 02/25/2023).

Fried, Daniel et al. (Apr. 2022). *InCoder: A Generative Model for Code Infilling and Synthesis*. en. arXiv:2204.05999 [cs]. URL: http://arxiv.org/abs/2204.05999 (visited on 04/09/2023).

Gao, Leo et al. (Dec. 2020). *The Pile: An 800GB Dataset of Diverse Text for Language Modeling*. en. arXiv:2101.00027 [cs]. URL: http://arxiv.org/abs/2101.00027 (visited on 02/25/2023).

Ha, David and Jürgen Schmidhuber (Mar. 2018). "World Models". en. In: arXiv:1803.10122 [cs, stat]. DOI: 10.5281/zenodo.1207631. URL: http://arxiv.org/abs/1803.10122 (visited on 02/24/2023).

Holtzman, Ari et al. (Feb. 2020). *The Curious Case of Neural Text Degeneration*. en. arXiv:1904.09751 [cs]. URL: http://arxiv.org/abs/1904.09751 (visited on 03/28/2023).

Li, Yujia et al. (Dec. 2022). "Competition-level code generation with AlphaCode". en. In: *Science* 378.6624, pp. 1092–1097. ISSN: 0036-8075, 1095-9203. DOI: 10.1126/science.abq1158. URL: https://www.science.org/doi/10.1126/science.abq1158 (visited on 02/24/2023).

Nijkamp, Erik et al. (Sept. 2022). *CodeGen: An Open Large Language Model for Code with Multi-Turn Program Synthesis*. en. arXiv:2203.13474 [cs]. URL: http://arxiv.org/abs/2203.13474 (visited on 02/25/2023).

Schulman, John et al. (Aug. 2017). *Proximal Policy Optimization Algorithms*. en. arXiv:1707.06347 [cs]. URL: http://arxiv.org/abs/1707.06347 (visited on 02/24/2023).

Solar-Lezama, Armando et al. (2006). "Combinatorial Sketching for Finite Programs". en. In.

Stiennon, Nisan et al. (n.d.). "Learning to summarize from human feedback". en. In: ().

Tan, Xu et al. (July 2021). *A Survey on Neural Speech Synthesis*. en. arXiv:2106.15561 [cs, eess]. URL: http://arxiv.org/abs/2106.15561 (visited on 04/27/2023).

Verma, Abhinav et al. (Apr. 2019). *Programmatically Interpretable Reinforcement Learning*. en. arXiv:1804.02477 [cs, stat]. URL: http://arxiv.org/abs/1804.02477 (visited on 02/24/2023).

Werra, Leandro von et al. (2020). *TRL: Transformer Reinforcement Learning*. https://github.com/lvwerra/trl.

Ziegler, Daniel M. et al. (Jan. 2020). *Fine-Tuning Language Models from Human Preferences*. en.

arXiv:1909.08593 [cs, stat]. URL: http://arxiv.org/abs/1909.08593 (visited on 03/03/2023).

Appendices

A Prompt Analysis

In this appendix, I perform a brief analysis of prompts and their success rates. This only considers compilation rates as it is the only metric which is available for all prompts. Furthermore, I only include prompt examples which have been compiled more than 32 times to ensure that the results are not skewed by a single response. The results can be found in table 2. Overall, simple and well structured prompts tend to perform better in terms of compilation. In other words, prompts which provide some structure to the model in the format of a sketched program are more likely to compile and have syntactically valid code. Most of these examples display an improvement in compilation rates after sanitization is performed on the model output.

Table 3 shows examples of code that never compiled without sanitization. Roughly 46% of the examples considered never had a single compilation in > 32 attempts. From the small sample in the table, these prompts tend to ask for complex DSP tasks of the model. A few cases not present in the table also provide typos in the sketch which prevent the code from compiling, even if they generated text was valid. This is a limitation of the current approach, as the model is not able to correct for typos in the sketch, though this could be improved by checking compilation of prompts and masking out lines that return errors from the compiler.

While this studies the compilation of code, this does not consider the semantic validity of the generated code. While the model may generate code which compiles, it was often the case that the code did not perform the task described in the prompt. Failure cases in varied, but typically followed a few patterns. First, the model would often just insert `process = os.osc(440);` as the answer when the prompts were vague. In other cases, the model completely misunderstands what part of the library does. For example, many prompts may ask for overdrive or distortion. In many cases, the model generates code which contains `fi.dcblocker()`, which has nothing to do with distortion. This is due to the noisy nature of the data and the small size of the dataset. DC Blocker is often used in distortion effects, but it is not a distortion effect itself. The third class of failure rates is long chaining of effects. This is something that happens in the dataset and in valid code, but the model is not able to properly discern how this should happen. For example, the model may generate code which links 12 filters, one after the other when attempting to create a comb filter, but does not vary the frequencies in the parameters, resulting in a collapsible single filter.

Further work would be needed to address these issues. A possible way forward would be to find more valid faust code, and weight the examples in the dataset by their quality. Furthermore, additional comments describing individual blocks of code could be helpful in guiding the model to generate more semantically valid code. Finally, pre training on natural language audio synthesis and dsp descriptions, as well as the Faust documentation could further guide the model. However, I do not anticipate that these approaches would lead to a large jump in code quality, outside of more examples of high quality Faust code.

Prompt	Compilation Rate	Sanitized Rate

`// synth with filter`
`import("stdfaust.lib");`
`freq = hslider("freq", 440, 20, 20000, 0.01);`
`gain = hslider("gain", 0.5, 0, 1, 0.01);`
`gate = button("gate");`
`cutoff = hslider("cutoff", 100, 50, 10000, 0.01);`
`envelope = gain*gate : si.smoo;`
`process = os.osc(freq)*envelope:` **??** `*gain;`
	75.00%	69.44%

`// oscillator in a stereo output` **??**	67.31%	88.46%
`// A simple oscillator with a lowpass filter` **??** `process =` **??**	66.67%	75.56%

`// synth with Moog Ladder filter`
`import("stdfaust.lib");`
`freq = hslider("freq", 440, 20, 20000, 0.01);`
`gain = hslider("gain", 0.5, 0, 1, 0.01);`
`gate = button("gate");`
`cutoff = hslider("cutoff", 100, 50, 10000, 0.01);`
`envelope = gain*gate : si.smoo;`
`process = os.osc(freq)*envelope:` **??** `*gain;`
	59.62%	53.85%

`// write an oscillator` **??**	55.00%	81.67%
`// write a panner with a sqrt law` **??**	55.00%	60.00%
`// write a sawtooth oscillator` **??**	46.67%	61.67%
`// write a filtered noise going in a stereo output` **??**	44.44%	50.00%
`// write a filtered noise going in a 4 channels output` **??**	38.89%	52.78%
`// write a noise going in a reverb` **??**	36.11%	61.11%

Table 2: Top 10 prompts with unsanitized output by compilation rate

Prompt	Compilation Rate	Sanitized Rate
`// a compressor with saturation controls` **??**	0.00%	11.54%

`// reverb effect to the input signal with`
`    // adjustable decay time and room size parameters`
??
	0.00%	5.00%

`// a phaser effect to the input signal with`
`    // adjustable frequency and depth parameters.`
??
	0.00%	13.89%

`// pitch shifter effect by changing the`
`    // playback rate of the input signal.`
??
	0.00%	22.22%

`// a tremolo effect using a volume`
`    //modulation source.`
??
	0.00%	15.38%

`// make a kick drum using an envelope and`
`    //highly resonant low pass filter`
??
	0.00%	11.11%

`// a program that creates a flanging effect`
`    // using a comb filter and a delay line.`
??
	0.00%	17.31%

`// This program implements a simple sawtooth`
`    // oscillator with adjustable frequency`
`    // and amplitude parameters.`
??
	0.00%	55.56%

`osc =` **??**	0.00%	88.89%

`// a stereo delay effect with adjustable`
`    // delay time and feedback parameters.`
??
	0.00%	17.31%

Table 3: Examples of code which never compiles. Note the prompts have been modified to fit the provided table. Any `<tab>\\` should be considered a continuation of the previous line.