TracrBench: Generating Interpretability Testbeds with Large Language Models

Hannes Thurnherr [1] **Jérémy Scheurer** [2]

Abstract

Achieving a mechanistic understanding of transformer-based language models is an open challenge, especially due to their large number of parameters. Moreover, the lack of ground truth mappings between model weights and their functional roles hinders the effective evaluation of interpretability methods, impeding overall progress. Tracr, a method for generating compiled transformers with inherent ground truth mappings in RASP, has been proposed to address this issue. However, manually creating a large number of models needed for verifying interpretability methods is labour-intensive and time-consuming. In this work, we present a novel approach for generating interpretability test beds using large language models (LLMs) and introduce TracrBench, a novel dataset consisting of 121 manually written and LLM-generated, human-validated RASP programs and their corresponding transformer weights. During this process, we evaluate the ability of frontier LLMs to autonomously generate RASP programs and find that this task poses significant challenges. GPT-4-turbo, with a 20-shot prompt and best-of-5 sampling, correctly implements only 57 out of 101 test programs, necessitating the manual implementation of the remaining programs. With its 121 samples, TracrBench aims to serve as a valuable testbed for evaluating and comparing interpretability methods.

1. Introduction

Recent advancements in transformer-based language models have led to progress in various natural language processing tasks (Achiam et al., 2023; Anthropic, 2024). However, understanding the internal workings of these models remains challenging (Olah et al., 2018; Nanda et al., 2023; Black et al., 2022), which is problematic since models may generate harmful outputs (Shevlane et al., 2023; Perez et al., 2022a; Brundage et al., 2018) or harbor other unacceptable failure modes only revealed after deployment (Ngo et al., 2022; Scheurer et al., 2023; Hubinger et al., 2024). Despite various successes in interpretability (Bricken et al., 2023; Conmy et al., 2023; Nanda et al., 2023; Cunningham et al., 2023; Templeton et al., 2024), developing new interpretability methods remains difficult, partly due to the lack of models with fully understood internals (Casper et al., 2023; Casper, 2020), i.e. with ground truth mapping between weights and their functional form. Existing benchmarks for evaluating interpretability methods focus on input-output behavior (Casper et al., 2024; 2023; Mazeika et al., 2022), human evaluations (Templeton et al., 2024), or disentangling attributions of different entities (Huang et al., 2024), rather than the full mechanistic circuits, which hinders the rigorous and fast validation of novel interpretability methods.

Restricted Access Sequence Processing Language (RASP) (Weiss et al., 2021) maps the core components of a transformer-encoder, i.e., attention and feed-forward computation, into simple primitives, forming a programming language to model and analyze transformer behavior. Tracr (Lindner et al., 2024), compiles RASP programs into functional transformer weights with a known mapping from weights to their functional form, enabling, the evaluation of interpretability methods (Conmy et al., 2023). However, its adoption is limited due to the difficulty of writing RASP programs and the large number of models required to effectively evaluate interpretability methods.

In this work, we introduce and evaluate a method to automatically generate RASP programs using LLMs and present TracrBench, a novel dataset with 121 LLM generated and, where necessary, manually written RASP programs and their compiled transformers. We assess the ability of frontier LLMs to generate RASP programs and find that this is a challenging task. With best-of-5 sampling and a 20-shot prompt, gpt-4-turbo-2024-04-09 correctly generates only 57 out of the 101 RASP programs in the test set. After adjusting for the difficulty of the programs, using the number of RASP operations as a proxy, the model achieves a normalized, weighted difficulty score of 0.29 (the maximum score is 1.0). TracrBench aims to be a rich testbed for evaluating interpretability methods and accelerating their development.

[1]Independent, Zurich, CH [2]Apollo Research, London, UK. Correspondence to: Hannes Thurnherr <hannes.thurnherr@gmail.com>, Jérémy Scheurer <jeremy@apolloresearch.ai>.

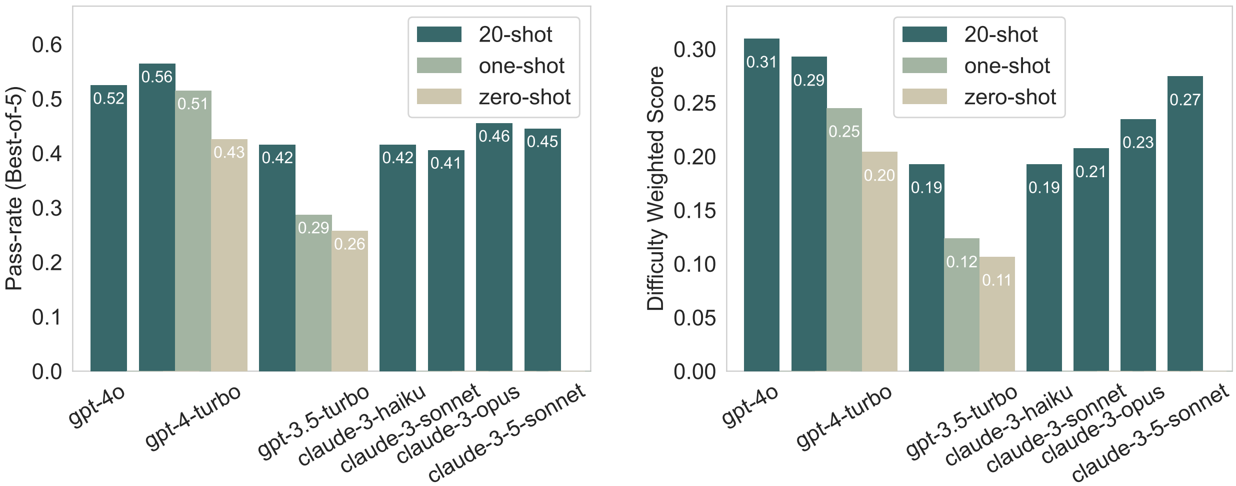

Figure 1. Results on the test set with 101 Tracr programs with pass-rate on the left and a normalized, difficulty-weighted score on the right (maximum score on both metrics is 1.0). The 20-shot prompt with best-of-5-sampling achieves the best performance to other prompts. gpt-4-turbo-2024-04-09 and gpt-4o models achieve the best performance overall. However, the task is challenging for all models.

2. Method

Current interpretability research faces challenges in rigorously evaluating novel methods due to the lack of models with fully understood internals. While Tracr compiles RASP programs into transformers with known mappings from weights to their functional form, writing programs in Tracr is time-consuming and difficult. This is partly because RASP is an unconventional, non-Turing-complete programming language that requires algorithms to be implemented differently than in standard Turing-complete languages like Python (see Appendix B for an example).

To address this issue, we propose to generate interpretability testbeds using LLMs, leveraging their ability to write code (Achiam et al., 2023; Li & Murr, 2024). We prompt LLMs to generate RASP programs that implement specified algorithms. We create TracrBench, a dataset of 121 RASP programs, by leveraging LLMs and manual annotation when they fail. These programs are then compiled into functional transformer weights using Tracr, resulting in transformer models with a known mapping between weights and their functional form. This allows researchers to validate the outputs of their novel interpretability methods against the ground truth. Our dataset of compiled models thus serves as an interpretability testbed.

To generate a program, we condition a language model $\mathcal{M}$ on a prompt $\mathcal{P}$ that includes a description of the specific algorithm to be implemented and at least one example input-output pair (see Fig. 2). To optimize LLM performance, $\mathcal{P}$ includes a detailed description of the RASP language and its five main components (SELECT, AGGREGATE, SELECTWIDTH, MAP, and SEQUENCEMAP), along with relevant Tracr source code defining these components and up to 20 RASP programs with their descriptions. We use Chain-of-thought prompting (Wei et al., 2022) to encourage reasoning and planning before generating code (see Appendix C for the prompt). We create three variations of this prompt: Zero-Shot, One-Shot (extending Zero-Shot with an RASP program and its description), and 20-Shot (extending Zero-Shot with 20 RASP programs and their descriptions).

Let $\mathcal{M}(\mathcal{P})$ represent the extracted program from the output of model $\mathcal{M}$ when conditioned on the prompt. We define a five-step verification pipeline to assess the correctness of the generated program $\mathcal{M}(\mathcal{P})$. Each step performs a specific verification relevant to the overall correctness of the program. Here are the five stages of the pipeline:

1. **Compilation and execution**: Test whether the program compiles without errors and runs error-free.
2. **Output correctness**: Test whether the function actually performs the correct computation and implements the specified program using 1,000 input-output pairs[1] generated by a manually written Python function equivalent to the desired RASP program.
3. **Tracr validation**: Run the program through the inbuilt Tracr (Lindner et al., 2024) validator[2] to filter out certain programs that aren't converted to equivalent transformer weights.
4. **Transformer weights compilation**: Run the actual RASP-to-transformer compilation process to expose runtime errors like a division by zero.

[1]The inputs are lists of random length between 1 and the maximum length selected for our compilation (which is 10).

[2]github.com

5. **Compiled transformer correctness**: Empirically test whether the resulting transformer actually performs the same computation as the RASP code using the same 1,000 test input-output pairs from step 2.

A program $\mathcal{M}(\mathcal{P})$ is considered correct if it passes all five steps; failure at any step counts as incorrect. This five-step verification pipeline helps identify and filter out programs with errors or inconsistencies, ensuring that the resulting dataset consists of high-quality, functionally equivalent RASP programs and transformer models. We employ best-of-5 sampling, allowing the model to attempt each task up to five times (from scratch) before moving on to the next. By evaluating the performance of LLMs with this process, we aim to assess the feasibility of using LLMs to create interpretability testbeds on demand.

3. Dataset

Writing RASP code to generate Tracr interpretability test beds is labor-intensive and has a steep learning curve (see Appendix B for an example). This has impeded the adoption of Tracr as a method to evaluate novel interpretability methods. To address this issue, we present TracrBench, a novel dataset of Tracr models that enables interpretability researchers to quickly test methods on transformers with known mappings from weights to their functional form. The dataset is generated as follows. First, we select 121 simple, sequence-to-sequence algorithms that cover a diverse range of tasks and difficulty levels (see the full list in Appendix A). We come up with these by sampling concrete algorithms from LLMs and manually selecting suitable ones. Some algorithms are also taken from Michaud et al. (2024). We then prompt gpt-4-0125-preview (which was the most competitive model at the time) to generate a RASP program for each program description. We test all outputs with our verification pipeline and verify them manually, finding that 49 of the generated RASP programs are correct. We then manually write the remaining RASP programs, ensuring that all programs in the dataset are correct and of high quality. Finally, we take 20 samples to use as examples in the prompt and use the remaining 101 samples as our test set.

The resulting dataset contains RASP programs of various complexity, from simple elementwise operations to more complex programs that lead to transformers with 2.7 million parameters. We use the number of RASP functions (such as `Select` and `Aggregate`, but also `rasp.indices` and `rasp.tokens`) as a proxy for the difficulty of the algorithm. This approach is more accurate than counting lines of code because some programs may have many lines that don't involve RASP (see Appendix B for an example). The distribution of task difficulties is depicted in Fig. 3 and Fig. 4. The first figure shows that most programs are quite easy, containing 3 to 10 RASP functions, but there is a long tail of more complex programs with up to 43 RASP function calls. The second figure empirically depicts the success and failure of gpt-4-turbo on various programs, showing that the number of RASP functions is a better indicator of task complexity than the number of lines of code.

```
# Your Task
Make a RASP program that replaces each element with
    the parity (0 for even, 1 for odd) of its index.
Example: [5, 5, 5, 5] --> [0, 1, 0, 1]
```

Figure 2. The description of the target algorithm to implement that is part of the prompt for the LLM.

To facilitate the use of our dataset, we provide both the RASP programs and their corresponding compiled transformers as PyTorch (Imambi et al., 2021) models in Transformerlens (Nanda & Bloom, 2022).

4. Experiments

In this section, we evaluate the capability of LLMs to generate correct RASP programs. As described in Section 2, we condition an LLM on a prompt that includes a program specification, a detailed description of the RASP language, and important parts of the RASP source code. We use three variations of the prompt: a zero-shot, a one-shot prompt, and a 20-shot prompt. These different prompt variations are used to assess how including examples affects the LLM's performance in generating RASP programs.

We evaluate the generated RASP programs using the verification pipeline described in Section 2. A program is considered correctly implemented if it passes all five pipeline steps. To account for the variance in program difficulty, we introduce a second metric called the difficulty-weighted score, which weights each success by the number of RASP functions in the program. Summing these weighted scores across tasks provides us with a composite score that more effectively represents the model's proficiency.

We first evaluate the performance of different prompts using gpt-3.5-turbo-0125 and gpt-4-turbo-2024-04-09. To minimize compute costs, we evaluate the performance of additional models using only the full prompt. These models include gpt-4o-2024-05-13, claude-3-haiku-20240307, claude-3-sonnet-20240229, claude-3-opus-20240229 and claude-3-5-sonnet-20240620. All models are evaluated on the test set with 101 samples and sampled at temperature 0.9, with top-p=0.95. To distinguish between an LLM's general programming ability and its RASP-specific capabilities, we establish a baseline where the LLM writes a Python program for the same target algorithms.

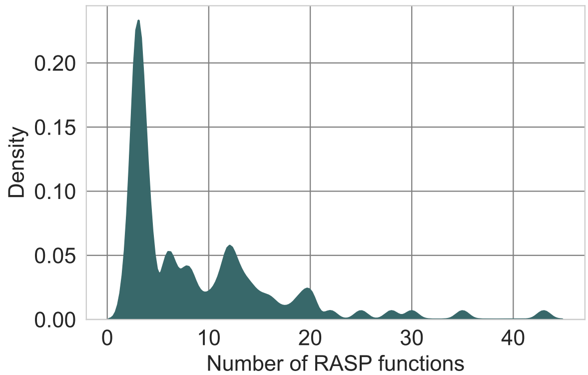

Figure 3. We show the distribution of RASP function calls within TracrBench using Kernel Density Estimation. The plot shows that most programs have around 6 RASP function calls, while a smaller number of more complex programs form a long tail.

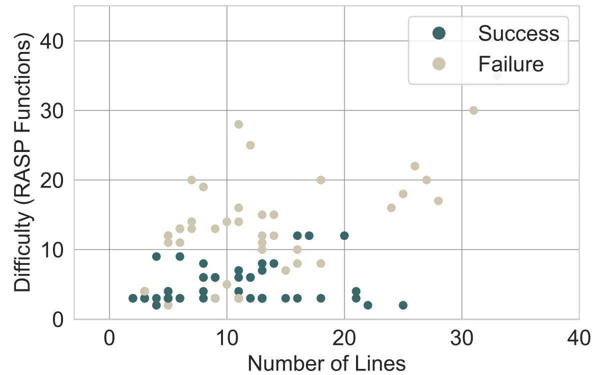

Figure 4. We compare the number of RASP functions and program lines as proxies for task difficulty. When plotting the pass-rate of gpt-4o-turbo on all programs, we can see that the number of RASP functions is a better indicator of task complexity than the total lines of code.

4.1. Results

The results of our experiment, visualized in Fig. 1, show that state-of-the-art LLMs are able to understand the RASP language and, to some extent, generate correct RASP programs. Adding examples to the prompt clearly improves the performance, as shown with gpt-4-turbo and gpt-3.5-turbo. Overall, gpt-4-turbo achieves the highest pass rate of 56%, outperforming claude-3-opus with a pass rate of 46%. In comparison, when generating Python programs for the target algorithms, gpt-4-turbo achieves a pass rate of 96%. When taking the difficulty of the target algorithms into account, i.e., when using the difficulty-weighted score as a metric, we observe that the successes are strongly concentrated among the easy, low-difficulty programs (see Fig. 4) with gpt-4-turbo achieving a score of 0.29 (out of 1.0) and gpt-4o performing best with a score of 0.31. Claude-3-5-sonnet has a similar pass rate (0.45) to claude-3-opus (0.46), however, it achieves a higher difficulty-weighted score (0.27), than claude-3-opus (0.23).

These results suggest that frontier LLMs cannot yet competently generate correct RASP programs. The relatively poor performance of generating RASP programs compared to conventional programming languages like Python may be attributed to RASP's limited representation in LLM training data. This finding highlights that the ability of frontier LLMs to extend their reasoning and programming capabilities to low-resource programming languages is limited, which may stand in contrast with their generalization in natural low-resource languages (Reid et al., 2024).

5. Related Work

Evaluating novel interpretability methods is challenging (Casper, 2020). While previous work has addressed this issue, it mainly focused on input-output level interpretability (Casper et al., 2024; 2023; Mazeika et al., 2022), human evaluations (Templeton et al., 2024), or disentangling attributes of different entities (Huang et al., 2024). RASP (Weiss et al., 2021) a programming language computationally equivalent to transformer, and Tracr (Lindner et al., 2024), which compiles RASP programs into corresponding transformers, have been used to create interpretable models for validating interpretability methods (Conmy et al., 2023). However, writing RASP programs in sufficient quantity is very time-consuming, which hinders the broad adoption of Tracr to evaluate interpretability methods. Notably, Tracr weights are more sparse and simple than any set of weights likely to result from gradient descent. Therefore, a method capable of interpreting Tracr weights may not necessarily be able to interpret trained transformers. However, interpretability methods that are capable of interpreting trained transformers should also be capable of interpreting Tracr transformers. Thus the latter still serve as a valid method to test (but not to develop) useful interpretability methods. Finally, both Thurnherr & Riesen (2024) and Langosco et al. (2024) programmatically generate large quantities of RASP programs with their corresponding weights to train decompiler models that generate RASP programs for a given set of transformer weights. Their RASP programs are, however, randomly generated by re-combining a few elemental operations, which leads to models that are often hard to decipher and do not correspond to realistic algorithms.

LLMs have been explored for generating datasets for model evaluations (Perez et al., 2022b) and automating part of the interpretability workflow (Bills et al., 2023). We extend this by using LLMs to scalably generate realistic and interpretable RASP programs. The generated programs serve as a test bed for evaluating interpretability methods.

6. Conclusion

We demonstrate that LLMs can be used to generate interpretability test beds. However, their performance rapidly deteriorates with the increasing difficulty of RASP programs, indicating that frontier LLMs struggle to generate interpretability test beds at scale. We expect that these current limitations, likely due to Tracr's low-resource nature, will diminish as LLM capabilities continue to advance. Finally, we introduce TracrBench, a novel dataset comprising 121 transformers with known mappings from weights to functional form. Its intended use is the testing of interpretability methods. It is unsuitable as a target for interpretability method development due to its small size and the fact that Tracr weights are very dissimilar to those of trained transformers in terms of sparsity and matrix-rank. TracrBench serves as a valuable resource for evaluating and comparing interpretability methods, facilitating the development of more effective techniques for understanding the inner workings of transformer-based models.

7. Author Contributions

Hannes Thurnherr executed the whole project, developed the prompts, created the dataset (i.e., the Tracr programs) with the help of LLMs, and manually, where necessary, ran all experiments and wrote the paper. **Jérémy Scheurer** developed the idea and ran exploratory experiments, oversaw the project, including detailed guidance on directions, experiments, presentation, and the final paper.

References

Achiam, J., Adler, S., Agarwal, S., Ahmad, L., Akkaya, I., Aleman, F. L., Almeida, D., Altenschmidt, J., Altman, S., Anadkat, S., et al. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.

Anthropic. The claude 3 model family: Opus, sonnet, haiku. *Preprint*, 2024. URL https://www-cdn.anthropic.com/de8ba9b01c9ab7cbabf5c33b80b7bbc618857627/Model_Card_Claude_3.pdf.

Bills, S., Cammarata, N., Mossing, D., Tillman, H., Gao, L., Goh, G., Sutskever, I., Leike, J., Wu, J., and Saunders, W. Language models can explain neurons in language models. https://openaipublic.blob.core.windows.net/neuron-explainer/paper/index.html, 2023.

Black, S., Sharkey, L., Grinsztajn, L., Winsor, E., Braun, D., Merizian, J., Parker, K., Guevara, C. R., Millidge, B., Alfour, G., et al. Interpreting neural networks through the polytope lens. *arXiv preprint arXiv:2211.12312*, 2022.

Bricken, T., Templeton, A., Batson, J., Chen, B., Jermyn, A., Conerly, T., Turner, N., Anil, C., Denison, C., Askell, A., et al. Towards monosemanticity: Decomposing language models with dictionary learning. *Transformer Circuits Thread*, pp. 2, 2023.

Brundage, M., Avin, S., Clark, J., Toner, H., Eckersley, P., Garfinkel, B., Dafoe, A., Scharre, P., Zeitzoff, T., Filar, B., et al. The malicious use of artificial intelligence: Forecasting, prevention, and mitigation. *arXiv preprint arXiv:1802.07228*, 2018.

Casper, S. Eis vii: A challenge for mechanists, 2020. URL https://www.lesswrong.com/posts/KSHqLzQscwJnv44T8/eis-vii-a-challenge-for-mechanists. Accessed: 2024-05-23.

Casper, S., Bu, T., Li, Y., Li, J., Zhang, K., Hariharan, K., and Hadfield-Menell, D. Red teaming deep neural networks with feature synthesis tools. *Advances in Neural Information Processing Systems*, 36:80470–80516, 2023.

Casper, S., Yun, J., Baek, J., Jung, Y., Kim, M., Kwon, K., Park, S., Moore, H., Shriver, D., Connor, M., et al. The satml'24 cnn interpretability competition: New innovations for concept-level interpretability. *arXiv preprint arXiv:2404.02949*, 2024.

Conmy, A., Mavor-Parker, A., Lynch, A., Heimersheim, S., and Garriga-Alonso, A. Towards automated circuit discovery for mechanistic interpretability. *Advances in Neu-*

ral Information Processing Systems, 36:16318–16352, 2023.

Cunningham, H., Ewart, A., Riggs, L., Huben, R., and Sharkey, L. Sparse autoencoders find highly interpretable features in language models. *arXiv preprint arXiv:2309.08600*, 2023.

Huang, J., Wu, Z., Potts, C., Geva, M., and Geiger, A. Ravel: Evaluating interpretability methods on disentangling language model representations. *arXiv preprint arXiv:2402.17700*, 2024.

Hubinger, E., Denison, C., Mu, J., Lambert, M., Tong, M., MacDiarmid, M., Lanham, T., Ziegler, D. M., Maxwell, T., Cheng, N., et al. Sleeper agents: Training deceptive llms that persist through safety training. *arXiv preprint arXiv:2401.05566*, 2024.

Imambi, S., Prakash, K. B., and Kanagachidambaresan, G. Pytorch. *Programming with TensorFlow: Solution for Edge Computing Applications*, pp. 87–104, 2021.

Langosco, L., Baker, W., Alex, N., Quarel, D. J., Bradley, H., and Krueger, D. Towards meta-models for automated interpretability. Manuscript in preparation, 2024.

Li, D. and Murr, L. Humaneval on latest gpt models–2024. *arXiv preprint arXiv:2402.14852*, 2024.

Lindner, D., Kramár, J., Farquhar, S., Rahtz, M., McGrath, T., and Mikulik, V. Tracr: Compiled transformers as a laboratory for interpretability. *Advances in Neural Information Processing Systems*, 36, 2024.

Mazeika, M., Hendrycks, D., Li, H., Xu, X., Hough, S., Zou, A., Rajabi, A., Yao, Q., Wang, Z., Tian, J., et al. The trojan detection challenge. In *NeurIPS 2022 Competition Track*, pp. 279–291. PMLR, 2022.

Michaud, E. J., Liao, I., Lad, V., Liu, Z., Mudide, A., Loughridge, C., Guo, Z. C., Kheirkhah, T. R., Vukelić, M., and Tegmark, M. Opening the ai black box: program synthesis via mechanistic interpretability. *arXiv preprint arXiv:2402.05110*, 2024.

Nanda, N. and Bloom, J. Transformerlens. `https://github.com/TransformerLensOrg/TransformerLens`, 2022.

Nanda, N., Chan, L., Lieberum, T., Smith, J., and Steinhardt, J. Progress measures for grokking via mechanistic interpretability. *arXiv preprint arXiv:2301.05217*, 2023.

Ngo, R., Chan, L., and Mindermann, S. The alignment problem from a deep learning perspective. *arXiv preprint arXiv:2209.00626*, 2022.

Olah, C., Satyanarayan, A., Johnson, I., Carter, S., Schubert, L., Ye, K., and Mordvintsev, A. The building blocks of interpretability. *Distill*, 3(3):e10, 2018.

Perez, E., Huang, S., Song, F., Cai, T., Ring, R., Aslanides, J., Glaese, A., McAleese, N., and Irving, G. Red teaming language models with language models. *arXiv preprint arXiv:2202.03286*, 2022a.

Perez, E., Ringer, S., Lukošiūtė, K., Nguyen, K., Chen, E., Heiner, S., Pettit, C., Olsson, C., Kundu, S., Kadavath, S., et al. Discovering language model behaviors with model-written evaluations. *arXiv preprint arXiv:2212.09251*, 2022b.

Reid, M., Savinov, N., Teplyashin, D., Lepikhin, D., Lillicrap, T., Alayrac, J.-b., Soricut, R., Lazaridou, A., Firat, O., Schrittwieser, J., et al. Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context. *arXiv preprint arXiv:2403.05530*, 2024.

Scheurer, J., Balesni, M., and Hobbhahn, M. Technical report: Large language models can strategically deceive their users when put under pressure. *arXiv preprint arXiv:2311.07590*, 2023.

Shevlane, T., Farquhar, S., Garfinkel, B., Phuong, M., Whittlestone, J., Leung, J., Kokotajlo, D., Marchal, N., Anderljung, M., Kolt, N., et al. Model evaluation for extreme risks. *arXiv preprint arXiv:2305.15324*, 2023.

Templeton, A., Conerly, T., Marcus, J., Lindsey, J., Bricken, T., Chen, B., Pearce, A., Citro, C., Ameisen, E., Jones, A., Cunningham, H., Turner, N. L., McDougall, C., MacDiarmid, M., Freeman, C. D., Sumers, T. R., Rees, E., Batson, J., Jermyn, A., Carter, S., Olah, C., and Henighan, T. Scaling monosemanticity: Extracting interpretable features from claude 3 sonnet. *Transformer Circuits Thread*, 2024. URL `https://transformer-circuits.pub/2024/scaling-monosemanticity/index.html`.

Thurnherr, H. and Riesen, K. Neural decompiling of tracr transformers. Manuscript in preparation, 2024.

Wei, J., Wang, X., Schuurmans, D., Bosma, M., Xia, F., Chi, E., Le, Q. V., Zhou, D., et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837, 2022.

Weiss, G., Goldberg, Y., and Yahav, E. Thinking like transformers. In *International Conference on Machine Learning*, pp. 11080–11090. PMLR, 2021.

A. Complete list of Algorithms

Program Name	Program Description
make_sum_digits	replaces each element with the sum of its digits.
make_absolute	takes the absolute value of each element in the sequence.
make_first_element	returns the first element of the sequence.
make_nth_fibonacci	replaces each element with the nth Fibonacci number.
make_count_greater_than	replaces each element with the number of elements greater than it in the sequence.
make_double_first_half	doubles the first half of the sequence. For uneven number of entries, round up to half.
make_decrement	decrements each element in the sequence by 1.
make_count_frequency	counts the frequency of each unique element.
make_increment_by_index	increments each element by its index.
make_decrement_to_multiple_of_three	decrements each element until it becomes a multiple of 3.
make_hyperbolic_cosine	applies the hyperbolic cosine to each element.
make_check_fibonacci	checks if each element is a Fibonacci number.
make_square_root	takes the square root of each element.
make_increment_odd_indices	increments elements at odd indices.
make_hyperbolic_tangent	applies the hyperbolic tangent to each element.
make_hyperbolic_sine	applies the hyperbolic sine to each element.
make_zero_every_third	sets every third element to zero.
make_element_second	replaces each element with the second element of the sequence. If the sequence has fewer than two elements you should return [None].
make_mirror_first_half	mirrors the first half of the sequence to the second half.
make_sorting	sorts the sequence.
make_increment	increments each element in the sequence by 1.
make_rank	ranks each element according to its size.
make_factorial	replaces each element with its factorial.
make_count_less_than	replaces each element with the number of elements less than it in the sequence.
make_cube_each_element	cubes each element in the sequence.
make_cube_root	takes the cube root of each element.
make_round	rounds each element to the nearest integer.
make_multiply_by_length	multiplies each element by the number of elements in the sequence.
make_increment_to_multiple_of_three	increments each element until it becomes a multiple of 3.
make_sign	determines the sign of each element (positive, negative, or zero).
make_cosine	applies the cosine function to each element.
make_divide_by_length	divides each element by the number of elements in the sequence.
make_negation	negates each element in the sequence.
make_sine	applies the sine function to each element.
make_histogram	creates a histogram of elements.
make_element_double	doubles each element in the sequence.
make_zero_even_indices	sets all even indices to zero.
make_tangent	applies the tangent function to each element.
make_count_occurrences	replaces each element with the number of times it appears in the sequence.
make_compute_median	computes the median of the sequence.
make_halve_second_half	halves the second half of the sequence. Note that you should divide sequences with odd number of elements into [first half of size n, second half of size n+1].
make_triple	triples each element in the sequence.
make_arctangent	applies the arctangent function to each element.
make_square_each_element	squares each element in the sequence.
make_check_power_of_n	checks if each element is a power of n (make the default for n 2). 1 and n itself, also count as power of n since they correspond to n^0 and n^1.

make_binarize	binarizes elements based on a threshold (make the default threshold 3).
make_average_first_last	sets each element to the average of the first and last elements.
make_check_increasing	checks if every element is greater than or equal to the previous one. The output should only contain all ones if every entry, that has a previous entry, meets this condition. Otherwise the output should be all 0s.
make_identity	returns the same sequence.
make_apply_threshold	applies a threshold, setting elements below it to zero (make the default threshold 3).
make_replace_small_tokens	replaces tokens smaller than a threshold with zero (make the default threshold 2).
make_swap_odd_index	swaps the nth with the $n+1$th element if $n\%2 == 1$. Note that this means that the first element will remain unchanged. The second will be swapped with the third and so on.
make_check_descending	checks if the sequence is in descending order.
make_rotate_left	rotates elements to the left by 1 position.
make_remove_duplicates	removes (replaces with 0) duplicates from the sequence. The first occurrences of the duplicated numbers also have to be removed.
make_scale_by_max	scales each element by the maximum value in the sequence.
make_sum_with_next	replaces each element with the sum of it and the next element. For the last element you can sum it with itself.
make_swap_elements	swaps two elements at specified indices (make the default indices 0 and 1). If an input sequence only has 1 element return [None].
make_one_if_equal_to_next	sets elements to one if they are equal to the next element. The last element should be compared with the first.
make_swap_consecutive	swaps every two consecutive elements. If the number of entries is odd, the last entry should stay in place.
make_check_palindrome	checks if the sequence is a palindrome.
make_next_prime	replaces each element with the next larger prime number. If the element is already prime, it should stay the same.
make_mask_sequence	masks a sequence, replacing every element with 0 except the one at a specified index (make the default index 1).
make_wrap	wraps each element within a range (make the default range [2, 7]). Wrapping here means that the values are projected into the range starting from the lower bound, once they grow larger than the upper bound, they start again at the lower.
make_alternate_elements	alternates elements with their indices.
make_check_last_two_equal	checks whether the last two entries of a sequence are equal. If the sequence only has one entrance, return [0].
make_insert_zeros	inserts zeros between each element. This means that the latter half of the sequence will be cut off (no 4 and 5 in the following example).
make_last_element	returns the last element of the sequence and pads the rest with zeros.
make_difference_to_next	replaces each element with the difference to the next element.
make_invert_if_sorted	inverts the sequence if it is sorted in ascending order, otherwise leaves it unchanged.
make_logarithm	applies logarithm base 10 to each element.
make_product_with_next	replaces each element with the product of it and the next element. The last element should be multiplied with itself.
make_check_multiple_of_first	checks if each element is a multiple of the first element.
make_sum_of_last_two	returns the sum of the last two elements in the sequence. If the sequence only has one entry, return [None].
make_pairwise_sum	replaces each element with the sum of it and the previous element. The first element can be left as it is.

make_polynomial	evaluates a polynomial with sequence elements as parameters. The x is represented by the first entry, the rest are parameters for example $$[3, 4, 2, 1]$$ is equal to $4x^2 + 2x + 1$ for $x = 3so4 * 3^2 + 2 * 3 + 1 = 36 + 6 + 1 = 43$ represented as $$[43, 43, 43, 43]$$.
make_flip_halves	flips the order of the first and second half of the sequence. Note that you should divide sequences with odd number of elements into [first half of size n, second half of size n+1].
make_arcsine	applies the arcsine function to each element.
make_check_divisibility	checks if the sequence consists of numbers divisible by some parameter (make the default 3).
make_arccosine	applies the arccosine function to each element.
make_check_all_equal	checks whether all elements are equal.
make_position	replaces each element with its position in the sequence.
make_set_to_median	replaces each element with the median of all elements.
make_swap_min_max	swaps the largest and smallest elements in the sequence. If the maximum or minimum appears more than once, both occurrences must be replaced.
make_clip	clips each element to be within a range (make the default range [2, 7]). ”Clipping” means that values outside of the range, are turned into the lower or upper bound, whichever is closer.
make_pairwise_max	makes each element the maximum of it and the previous element, leaving the first element as it is.
make_check_alternating	checks if the sequence consists of alternating odd and even numbers. If this is not true, all the entries in the output sequence should be zero.
make_exponential	exponentiates each element.
make_interleave_reverse	interleaves elements with their reverse order Numbers at the odd indices should be in reverse order.
make_element_divide	divides each element by the division of the first two elements. If either the first or second element are zero, or if the sequence has fewer than two entries, you should just return the original sequence.
make_set_to_index	sets elements to their index value.
make_check_multiple_of_n	checks if all elements are a multiple of n (set the default at 2). The output should be all 1s if this is true for all elements, otherwise all 0s.
make_swap_first_last	swaps the first and last elements of the sequence. If the sequence only has one entry, just return the original sequence.
make_test_at_least_two_equal	checks whether at least two elements are equal.
make_reflect	reflects each element within a range (make the default range [2, 7]). Reflect means that the values will be projected into the range, ”bouncing” from the borders, until they have travelled as far in the range as they travelled outside of it.
make_check_square	checks for every entry of the sequence whether it is a square number or not.
make_count_prime_factors	replaces each element with the number of prime factors it has.
make_zero_if_less_than_previous	sets elements to zero if they are less than the previous element.
make_element_subtract_constant	subtracts a constant from each element (make the default constant 2).
make_check_prime	checks if each element is a prime number.
make_index_parity	replaces each element with the parity (0 for even, 1 for odd) of its index.

B. Example Program

RASP, a programming language designed to be computationally equivalent to transformers, requires a conceptually different approach to implementing algorithms compared to conventional programming languages. For instance, sorting algorithms in RASP must be implemented unconventionally due to the language's unique constraints. Unlike traditional programming languages that allow iteration over a sequence, RASP processes all elements in a sequence in parallel, mimicking the behavior of transformers. Consequently, a sorting algorithm in RASP would count, for each entry, the number of other entries smaller than itself and then use these counts to rearrange the original elements. While this approach would be considered inefficient in conventional programming languages, it is a straightforward implementation under the constraints of RASP. This example highlights the need for a different mindset when writing algorithms in RASP, as the language's parallel processing nature requires unconventional solutions to common problems.

```python
def make_sort_unique(vals: rasp.SOp, keys: rasp.SOp) -> rasp.SOp:
    smaller = rasp.Select(keys, keys, rasp.Comparison.LT) # find the smaller elements for each entry
    target_pos = rasp.SelectorWidth(smaller) # count the number of smaller elements for each entry
    sel_new = rasp.Select(target_pos, rasp.indices, rasp.Comparison.EQ) # create the rearrangement selector according to target_pos
    return rasp.Aggregate(sel_new, vals) # apply the rearrangement selector to the original sequence

def make_sort(vals: rasp.SOp, keys: rasp.SOp, *, max_seq_len: int, min_key: float) -> rasp.SOp:
    keys = rasp.SequenceMap(lambda x, i: x + min_key * i / max_seq_len, keys, rasp.indices) # turn all the elements unique by adding a small fraction of their index
    return make_sort_unique(vals, keys) # apply sort_unique to the sequence using the now unique elements as keys
```

RASP programs written for the Tracr compiler are written in Python using the tracr.rasp module. Sometimes they consist of a number of lines where the tracr.rasp module is not used. These parts of the RASP program can be written independently of one's understanding of the RASP language. The following is an example of a program where most lines don't involve RASP. This illustrates why the number of rasp functions in a program is a better approximation of difficulty than the number of total lines when it comes to evaluating a model's ability to write RASP code.

```python
def primecheck(n):
    for i in range(2,int(n/2)):
        if n%i==0:
            return 0
    return 1

def make_check_prime() -> rasp.SOp:
    return rasp.Map(lambda x: primecheck(x), rasp.tokens)
```

C. Full Prompt

Prompt ”Paraphrased + Tip about different stock”

```

# Introduction to Task:
Your assignment is to generate RASP programs capable of implementing a variety of algorithms using sequence operations. "RASP" stands for "Restricted Access Sequence Processing Language". RASP allows you to articulate complex sequence to sequence in a format equivalent to what a neural network of the transformer architecture can do. RASP programs always output a sequence that has the same length as the input sequence.
```

```
# Your Task
Make a RASP program that replaces each element with the parity (0 for even, 1 for
    odd) of its index.
Example: [5, 5, 5, 5] --> [0, 1, 0, 1]
Name the function that you can call to make this program 'make_index_parity()'

Keep your task in mind while reading the following information.

# Understanding RASP:

RASP programs are unique because they always process sequences and output
    transformed sequences of equivalent length. While doing so they void
    conditional branches or loops if possible. Instead, they rely on a series of
    operations that interpret and manipulate the input data in a sequence-to-
    sequence fashion. The length of the sequence never changes during this process
    .

## Fundamental Principles:

- Input and Output: Each RASP program receives an input sequence and yields an
    output sequence of identical length.
- Structure: Loops and if statements cannot depend on attributes or individual
    elements of the input sequence. If you make loops, they should have a fixed
    length or depend on a "max_sequence_length" parameter.
- Operation Calls: Programs can only invoke core RASP functions or refer to other
    RASP programs. Never attempt to access the internals of the sequence.

## Technical operational Jargon:

Here are descriptions of various operations that are used in RASP.

- `rasp.Select`: Matches elements from two sequences based on a boolean comparison
     condition and returns a corresponding matrix of "True" and "False" values
    called a selector.
- `rasp.Aggregate`: takes as input a selector and an SOp (Sequence Operation,
    which is an operation that transforms a sequence), and produces an SOp that
    averages the value of the SOp weighted by the selection matrix.
- `rasp.Map`: Transforms a sequence by applying a function to each element
- `rasp.SequenceMap`: Produces a new sequence based on two previous sequences and
    a lambda function that gets applied to each pair of elements.
- `rasp.SelectorWidth`: returns the number of "True" values in each row of a
    selector

### Function overview:

#### Select:
Function: Creates a selector to define relationships between elements of sequences
    .
Syntax: `rasp.Select(keys: SOp, queries: SOp, predicate: Predicate)`
Example: `rasp.Select(rasp.indices, rasp.indices, rasp.Comparison.EQ)` selects
    elements where indices are equal.

#### Aggregate:
Function: Takes as input a selector and an SOp, and produces an SOp that averages
    the value of the SOp weighted by the selection matrix.
Syntax: `rasp.Aggregate(selector: Selector, sop: SOp, default: Optional[VT] = None
    )`
Example: `rasp.Aggregate(select_all, any_negative, default=0)` aggregates based on
     select_all.

#### Map:
Function: Applies a function element-wise on the input SOp.
```

```
Syntax: `(f: Callable[[Value], Value], inner: SOp)`
Example: `Map(lambda x: x + 1, tokens)` adds 1 to each element of tokens.

#### SequenceMap:
Function: Applies a function element-wise on two given SOps.
Syntax: `rasp.SequenceMap(f: Callable[[Value, Value], Value], fst: SOp, snd: SOp)`
Example: `rasp.SequenceMap(lambda x, y: x - y, rasp.indices, rasp.tokens)` subtracts tokens from indices.

#### SelectorWidth:
Function: Returns the "width" of a selector, which corresponds to the number of "True"-values in each row.
Syntax: `rasp.SelectorWidth(selector: Selector)`
Example: `rasp.SelectorWidth(selectAll)`

#### Tokens, Indices:
rasp.tokens: The original input sequence.
rasp.indices: Returns the position index at each token.

### Example use of above Functions:
This is an example use the rasp.Select function. Here, it produces a selector based on rasp.tokens applied to itself with the "Greater Than" or GT comparison operator:

```python
greater_than_selector = rasp.Select(rasp.tokens, rasp.tokens, rasp.Comparison.GT).named("greater_than_selector")
```
If the rasp.tokens-sequence is [1, 2, 3, 4] the selector will look like this:
[False, True, True, True]
[False, False, True, True]
[False, False, False, True]
[False, False, False, False]
If we now apply this to the original rasp.tokens again with:
```python
output = rasp.Aggregate(greater_than_selector, rasp.tokens)
```
We will get an average of all the values selected in each row. The output looks like this:
[3, 3.5, 4, None]
[
3, # as an average of the selected 2,3 and 4
3.5, # as an average of the selected 3 and 4
4, # as an average of the selected 4
None # because none of the values were selected as none of them are greater than 4 at this position. So, None, which is always the default value, takes this spot.
]
Note that, in the programs you create, you should avoid using rasp.Aggregate with selectors that have more than one true value in each row. In other words: you can use rasp.Aggregate to shift elements around, but avoid using it for averaging multiple elements. However, using rasp.SelectWidth with selectors that have more than one "True" value per row is completely fine.
If we now call:
```python
count_GT_selector = rasp.SelectorWidth(greater_than_selector)
```
We will get a sequence that contains the count of the truth values in each row:
[3,2,1,0]
If we call:
```python
map_count_GT = rasp.Map(lambda x: x*3+1, count_GT_selector)
```
```

````
We will get a sequence where this lambda function has been applied to all the values of count_GT_selector:
[10, 7, 4, 1]

But if we call:
```python
sequenceMap_combination = rasp.SequenceMap(lambda x, y: x*y+x, count_GT_selector, output)
```
We get an output where the sequences "count_GT_selector" and "output" are combined element-wise according to the lambda function.
At this point, "count_GT_selector" is [3,2,1,0] and output is [3, 3.5, 4, None], so sequenceMap_combination is [12, 9, 5, None]
[
12, #because 3 * 3 + 3 = 12
9, #because 2 * 3.5 + 2 = 9
5, #because 1 * 4 + 1 = 5
0 #because 0 * None + 0 = 0
]

# Rules and Constraints:
- Use provided operation types (Select, Aggregate, SelectorWidth Map, SequenceMap) as the building blocks of your program. Feel free to be creative in how to combine them but remember which kind of output (Selector or Sop) they produce.
- Each operation must be traceable and reproducible, implying a transparent translation from instructions to action.

# Source Code
To make you better understand the RASP language you can look at the following code. These are the most important parts of rasp.py, which defines the library of RASP. Use this as a reference to find out what kind of functions exist in RASP, which inputs they take, and what they do.



# Example use of Functions:
This is an example use the rasp.Select function. Here, it produces a selector based on rasp.tokens applied to itself with the "Greater Than" or GT comparison operator:

```python
greater_than_selector = rasp.Select(rasp.tokens, rasp.tokens, rasp.Comparison.GT).named("greater_than_selector")
```
If the rasp.tokens-sequence is [1, 2, 3, 4] the selector will look like this:
[False, True, True, True]
[False, False, True, True]
[False, False, False, True]
[False, False, False, False]
If we now apply this to the original rasp.tokens again with:
```python
output = rasp.Aggregate(greater_than_selector, rasp.tokens)
```
We will get an average of all the values selected in each row. The output looks like this:
[3, 3.5, 4, None]
[
3, # as an average of the selected 2,3 and 4
3.5, # as an average of the selected 3 and 4
4, # as an average of the selected 4
None # because none of the values were selected as none of them are greater than 4 at this position. So, None, which is always the default value, takes this spot.
````

````
]
Note that, in the programs you create, you should avoid using rasp.Aggregate with selectors that have more than one true value in each row. In other words: you can use rasp.Aggregate to shift elements around, but avoid using it for averaging multiple elements. However, using rasp.SelectWidth with selectors that have more than one "True" value per row is completely fine.
If we now call:
```python
count_GT_selector = rasp.SelectorWidth(greater_than_selector)
```
We will get a sequence that contains the count of the truth values in each row:
[3,2,1,0]
If we call:
```python
map_count_GT = rasp.Map(lambda x: x*3+1, count_GT_selector)
```
We will get a sequence where this lambda function has been applied to all the values of count_GT_selector:
[10, 7, 4, 1]

But if we call:
```python
sequenceMap_combination = rasp.SequenceMap(lambda x, y: x*y+x, count_GT_selector, output)
```
We get an output where the sequences "count_GT_selector" and "output" are combined element-wise according to the lambda function.
At this point, "count_GT_selector" is [3,2,1,0] and output is [3, 3.5, 4, None], so sequenceMap_combination is [12, 9, 5, None]
[
12, #because 3 * 3 + 3 = 12
9, #because 2 * 3.5 + 2 = 9
5, #because 1 * 4 + 1 = 5
0 #because 0 * None + 0 = 0
]


Start your process by looking at the examples and the RASP language basics, then write down a plan based on the information in the files and the examples above, and then write your program.
If your plan includes the usage of a certain function, look up all of the allowed parameters for this function, write them down before you start writing the program and make sure you do not make up any new parameters to any of the RASP functions.
Note that you are not allowed to directly call the above examples as functions in your code, without explicitly writing/copying them into your output yourself. This means if you want to call functions like `make_length()` or `shift_by()`, you have to rewrite them in your output code.

# Output Format
Use the following Format for your answer:

<Task>
[Reiterate your understanding of the task and add a new example of an input and the corresponding desired output.]
</Task>

<Plan>
[Your plan on how the program should broadly work.]
[Some details on which functions you'll have to use and what their inputs will be.]
</Plan>

````

````
<PlanVerification>
[Look back at your plan. Will it really work? Is this compatible with the
    functionality of the functions you're using? Are you using your functions
    correctly? (Look at the source code to verify this) Answer these questions
    here explicitly]
[List changes you have to make to the plan based on your verification]
</PlanVerification>

```python
[write out your RASP-python code in a code block here]
```

### Example Use of Format:

Here is an example of how you might use this output format:

<Task>
The task is to create a RASP program that takes a sequence and returns a new
    sequence of identical length where each element is the maximum value found in
    the original sequence.

For example:
max = make_max()
max([1,2,6,-2,1]) # returns [6,6,6,6,6]
</Task>

<Plan>
1. Create a selector that compares each element with every other element using a "
    Less Than or Equal" (LEQ) comparison.
2. Use SelectorWidth to count the number of elements that each element is less
    than or equal to.
3. The maximum element will have a count equal to the length of the sequence, so
    create a selector that selects the elements where the count from step 2 equals
     the length of the sequence.
4. Use Aggregate with the selector from step 3 to broadcast the maximum element
    across the entire sequence.

The functions we will use include:
- Select: for creating the comparison selector.
- SelectorWidth: for counting the number of comparisons that are true for each
    element.
- Map: for creating a sequence of the sequence length.
- Aggregate: for selecting and broadcasting the maximum element.
</Plan>

<PlanVerification>
The plan seems feasible and aligns with the capabilities of the RASP functions:
- The Select operation can create a comparison matrix that identifies where each
    element is less than or equal to every other element.
- SelectorWidth can count the number of True comparisons for each element.
- Map can create a sequence where each element is the length of the sequence.
- Aggregate can then broadcast the maximum element where the comparison count
    equals the sequence length.

There are no changes needed for the plan based on verification.
</PlanVerification>

```python
def make_max() -> rasp.SOp:
    # Selector that creates a comparison matrix where each element is compared to
        every other element.
    leq_selector = rasp.Select(rasp.tokens, rasp.tokens, rasp.Comparison.LEQ).
        named("leq_selector")
````
````

```

    # Count the number of comparisons where each element is less than or equal to other elements.
    leq_count = rasp.SelectorWidth(leq_selector).named("leq_count")

    # Create a Map to get the sequence length for each element.
    sequence_length = rasp.Map(lambda x: len(x), rasp.tokens).named("sequence_length")

    # Selector that selects the element where the leq_count equals the sequence_length.
    max_element_selector = rasp.Select(leq_count, sequence_length, rasp.Comparison.EQ).named("max_element_selector")

    # Use Aggregate to broadcast the maximum element across the entire sequence.
    max_sequence = rasp.Aggregate(max_element_selector, rasp.tokens).named("max_sequence")

    return max_sequence
'''


# Your Task
Make a RASP program that replaces each element with the parity (0 for even, 1 for odd) of its index.
Example: [5, 5, 5, 5] --> [0, 1, 0, 1]
Name the function that you can call to make this program 'make_index_parity()'


Examples provided are references; use them to grasp the syntax and structure required for RASP. From there, your original programs should follow these established patterns but are not limited to the examples' specific functions.

Keep in mind:
- Adhere strictly to RASP's core operations.
- Keep your programs simple, if possible. (E.g. For identity, just return rasp.Map(lambda x: x, rasp.tokens)
- Meticulously add comments to your code for clarity.
- Output functional, executable Python code utilizing RASP's parameters.
- Don't import any additional packages. Write pure RASP code.
- Provide functional, complete Python code, not pseudo-code or placeholders.

Also Note:
- Do not import rasp. It is already imported. You should also not try to import the rasp components individually.
- Aggregate functions should always have None as the default (meaning you should leave the default as is.) This is because we want to compile these functions later, which only works with a default of None.
- Again, do not use any functions from the example without defining them yourself. You cannot assume any function from the examples is already defined.
- If your 'make_x()' functions have additional parameters like 'make_x(n)' or 'make_x(threshold)', you should always have a default value like 'make_x(threshold = 2)'
- Avoid the 'rasp.Full()' functionality. It will prevent compiling. Instead of 'rasp.Full(n)'' use the following function: 'rasp.Map(lambda x: n, rasp.indices)'

Endeavour to follow these guidelines to construct accurate and efficient RASP programs. Your expertise in Python will be fundamental to this task, so make sure that your code is both clean and precise, adhering to the RASP principles.
```