Understanding Computer Programs: Computational and Cognitive Perspectives

by

Shashank Srikant

B. Tech., National Institute of Technology Kurukshetra (2011)
S.M., Massachusetts Institute of Technology (2020)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2023

Authored by:	Shashank Srikant Department of Electrical Engineering and Computer Science May 15, 2023
Certified by:	Una-May O'Reilly Principal Research Scientist of the Computer Science and Artificial Intelligence Laboratory Thesis Supervisor
Accepted by:	Leslie Kolodziejski Professor of Electrical Engineering and Computer Science Chair, Department Committee on Graduate Students

Understanding Computer Programs:
Computational and Cognitive Perspectives

by

Shashank Srikant

Submitted to the Department of Electrical Engineering and Computer Science
on May 15, 2023, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

In this thesis, I study the understanding of computer programs (code) from two perspectives: *computational* and *cognitive*. I ask what the human bases of understanding code are, and attempt to determine whether computational models trained on code corpora (also known as code models) share similar bases.

From the computational perspective, I start by proposing a framework to test the robustness of the information learned by code models (chapter 2). This establishes a baseline measure for how well models comprehend code. I then describe techniques for improving the robustness of these models while retaining their accuracy (chapter 3). I then propose a way forward for code models to learn and reason about concurrent programs from their execution traces (chapter 4). In doing so, I also demonstrate the limitations of heuristics developed over the past four decades for detecting data races in concurrent programs, highlighting the need for evaluating these heuristics further.

In the cognitive aspect, I study how our brains comprehend code using fMRI to analyze programmers' brains (chapter 5). I show that our brains encode information about comprehended code similar to how code models encode that information (chapter 6). I show how the framework I develop in chapter 2 can be used to automatically generate stimuli for experiments in psycholinguistics and cognitive neuroscience (chapter 7), which can improve our understanding of how our minds and brains comprehend programs. Finally, I propose a probabilistic framework which models the mechanism of finding *important* parts of a program when comprehending it (chapter 8).

Thesis Supervisor: Una-May O'Reilly
Title: Principal Research Scientist of the Computer Science and Artificial Intelligence Laboratory

Acknowledgments

The research presented in this thesis was done under the mentorship of Dr. Una-May O'Reilly. In Una-May, I found an advisor who cared deeply about, and shared a fascination for, programs and understanding them. She showed great confidence in me when I wanted to address the idea of program understanding from diverse perspectives–ML, cognitive neuroscience, and program analysis. Her enthusiasm in understanding and refining with me the questions I investigate in this thesis has been infectious and inspiring. Thank you Una-May for being the inspiring mentor and human being you are. All of the research presented in this thesis was conducted in collaboration with her.

A significant portion of my work in cognitive neuroscience and code model representations that I present in this thesis was done in collaboration with the labs of professors Ev Fedorenko and Sijia Liu. A cold-email to Ev to study the brain bases of code understanding in 2018 and a joint-proposal to explore the robustness of code models with Sijia in 2019 started my collaboration. Their relentless pursuit of their topics of expertise, their availability to engage with me on various (often half-baked) ideas I presented over the years, and their warmth and kindness in readily accepting me as a collaborator is something I am grateful for, and that I hopefully can emulate.

I thank Ev, Sijia, and Prof. Armando Solar-Lezama for the helpful feedback they provided as my thesis readers. Armando's course on program analysis was not only my first course at MIT but also the most influential, as it helped me appreciate the program analysis perspective to some of the solutions presented in this thesis. Thank you Amanda Abrams for patiently working with me on all my thesis-related administrative work, and thanks Nicole Hoffman, Janet Fischer, Alicia Duarte, and Prof. Leslie Kolodziejski, Prof. Berthold Horn for your diligence and care in making the administrative processes a breeze to navigate through.

I had the pleasure of working with, and I am thankful to, the following excellent collaborators who have contributed to the work I present in this thesis. I also mention in brief the *origin stories* for each of these collaborations.

• **Chapter 2** was in collaboration with Sijia Liu, Tamara Mitrovska, Shiyu Chang, Quanfu Fan, and Gaoyuan Zhang. Thank you David Cox for the helpful discussions. The chapter, in full, is a reprint of the material as it appears in Srikant et al. [2021], published at ICLR 2021.
The work was a result of a joint-proposal in 2020 between Sijia and ALFA, and which began my collaboration with Sijia's group.

• **Chapter 3** was in collaboration with Jinghan Jia, Sijia Liu, Tamara Mitrovska, and Chuang Gan. Jinghan contributed equally with me as a primary author, who also diligently handled the fairly involved implementation details. The chapter, in full, is a reprint of the material as it appears in Jia et al. [2022], published at SANER 2023.

• **Chapter 4.** was in collaboration with Teodor Rares Begu, Malavika Samak, and Michael Wang. Specifically, Section 4.3 in the chapter refers to a thesis I mentored, authored by Teodor Rares Begu [Rares Begu, 2020]. Section 4.4, in full, is a re-print of the material as it appears in Wang et al. [2023] published at SOAP 2023 at PLDI. I was inspired to study the problem of concurrency bug detection after having unsuccessfully addressed it in my S.M. thesis [Srikant, 2020]. As a side project, I proposed using a toy language to simulate concurrent threads and assess the limitations of various ML models. Teodor led the project, refining and implementing it in 2020. Soon after, my work on min-max optimization informed the theoretical formulation I propose in this chapter. The problem surfaced again when I reached out to Malavika Samak in December 2021, during a talk by Prof. Michael Pradel, to learn more about her work. She too had coincidentally considered using a data-driven approach to reason about concurrent programs. We then scoped the problem, and realized the lack of any labeled datasets. Michael Wang, who joined the ALFA group as a graduate student then, joined this project, took charge, and developed the solution which came to be *RaceInjector*.

• **Chapter 5** was in collaboration with Anna Ivanova, Yotaro Sueoka, Hope Kean, Riva Dhamala, Marina Bers, and Ev Fedorenko. Steve Shannon and Atsushi Takahashi provided valuable support at the Martinos imaging center. This chapter is, in full, a reprint of the arXiv report Srikant et al. [2023a], which in turn is a rewrite of Ivanova

et al. [2020], published at eLife 2020. The chapter informs the results of the eLife work to a computer science audience; the original eLife work was written to primarily inform a cognitive neuroscience audience.

A cold-email to Ev in September 2018 began this collaboration. Anna from Ev lab had also serendipitously begun working on understanding the brain bases of code comprehension in the week I emailed Ev. The neuroimaging perspective of program understanding intrigued me from my days at Aspiring Minds. I even had conversations about fMRI recordings with faculty in India before joining graduate school (see Chapter 1 for an excerpt). Serendipity ensured I went through with my desire to study this problem.

- **Chapter 6** was in collaboration with Benjamin Lipkin, Anna Ivanova and Ev Fedorenko. Ben Lipkin contributed equally with me as a primary author in this work; he played a central role in carefully analyzing all the data, which eventually ensured the success of this project. The chapter, in full, is a reprint of the material as it appears in Srikant et al. [2022], published at NeurIPS 2022.
- **Chapter 7** was done with Greta Tuckute from Ev lab. It is, in full, a reprint of the material as it appears in Srikant et al. [2023b], in submission at the time of writing this thesis.

During a casual dinner conversation in December 2021, Greta introduced me to the problem of stimuli generation, for which she was exploring alternate solutions at the time. I repurposed the solution I present in chapter 2 to address the problem.

- **Chapter 8.** The push to explore behavioral responses to code comprehension came from a growing sense of frustration. Around 2020, I grew jaded by the constant stream of code models applied to software tasks willy-nilly. It seemed then that models were being trained for the sake of it, without taking into account if programmers cared for those tasks, and worse, likely ignoring more fundamental tasks that programmers needed help with, such as reading others' code. I decided to seek out startups and other software product research groups (as opposed to the typical large research groups) in this field which could offer me exposure and access to programmer behavior in the software development process. I cold-emailed Stephen Magill in 2021, co-founder of

Muse Dev Inc (now acquired by Sonatype Inc.), in whose group I spent a summer and tested some ideas on code comprehensibility prediction. I later developed other ideas on this theme which came to be the work I present in this chapter. Ev later introduced me to Prof. Yevgeni Berzak who was working on similar ideas. Thanks to Prof. Daniel Jackson for sharing with me his thoughts on program understandability, and David Darais for informing me of Stephen Magill's group.

- **Unpublished work.** The following works were also related to the theme explored in this thesis. With Tamara Mitrovska, then an undergraduate researcher, I pursued some more challenging directions on probing code models for their understanding. With Erik Hemberg, I proposed a way to use satisfiability modulo theory (SMT) to detect loopholes that aid tax avoidance in contract law. We demonstrated success on a small class of contract laws. This work demonstrates how symbolic approaches can enhance machine comprehension of domain-specific languages, such as contract law, which are generally considered inscrutable. With Stephen Magill at Sonatype Inc., I analyzed $\sim$ 100K Java programs in production to test what makes a snippet of code harder to understand. I had access to before-after snapshots of code reviews and fixes for these programs, from which I learned patterns.

All of the research presented in this thesis was funded by a grant from the MIT-IBM Watson AI lab. For this, I am very grateful to David Cox, Aude Oliva, and the lab. Research I did during the summers of 2020 and 2021 was supported by the MIT-IBM Watson AI lab and Stephen Magill's team at Sonatype Inc. (previously Muse Dev Inc.) respectively. The first year of my PhD was partially funded by the FinTech initiative at CSAIL. MIT's generous undergraduate research programs (Quest for Intelligence and the MIT SuperUROP program) supported multiple superb undergraduate researchers I worked with, including my collaborators Tamara (chapters 2, 3) and Teodor (chapter 4). My gratitude to all these sources for funding me and my collaborators.

I thank the following for their outstanding support with the computational resources I needed for my experiments: John Cohn, Jessie Rosenberg, Christopher Laibinis, and Luke Inglis from MIT-IBM AI lab and IBM Research; the *ninjas* at TIG, CSAIL - Jonathan Proulx, Alex Closs, Garrett Wollman, and Shaohao Chen from BCS.

A significant portion of the work in this thesis is built on the effort of several researchers—those who prepared and publicly released datasets, codebases and ML models, those who put out helpful blogs and videos explaining their work, those who thanklessly answered queries on public forums, and those who have diligently worked on several important open source efforts such as Pytorch and Hugging Face. I thank them all; I stand on the shoulders of such giants.

Kim Martineau, Rachel Gordon, Steven Nadis, Matt Busekroos, Jake Lambert, Phil Arsenault, and Erin Underwood from CSAIL, MIT News and EmTech MIT helped in communicating much of the work presented in this thesis to the public via portals like MIT News, CSAIL spotlights, talks at EmTech, and posts on social media.

Several courses I took in computer science and cognitive science have shaped my views on the topics I address in this thesis. Thanks especially to professors Armando Solar-Lezama, Regina Barzilay, Nickolai Zeldovich, Frans Kaashoek, Nancy Kanwisher, Pawan Sinha, Josh Tenenbaum, Ted Gibson, and Athulya Aravind for their inspiring courses. They posed big-picture questions and constantly encouraged us to think beyond.

It takes a village to raise a child. Some instilled a sense of rigor and curiosity while others mentored me while I was finding my way into academia. Thanks to the many inspiring teachers from my school; Prof. Jitender Chhabra – my undergraduate advisor; Varun Aggarwal – my manager at Aspiring Minds; Sumit Gulwani, professors Rupesh Nasre, Jitender Chhabra and Lav Varshney – my letter writers to graduate school; Sumit for recommending a visit to the Microsoft Research India lab in 2017, and to Bill Thies, Sriram Rajamani and Swami Manohar for being my hosts there and talking to me about life after graduate school; professors Bogdan Vasilescu and Jonathan Aldrich for being welcoming hosts at CMU in 2017. A special note of thanks to Varun who patiently taught me to do rigorous research and eventually introduced me to Una-May, and for setting up Aspiring Minds where I met excellent colleagues to do fun research with.

Lastly, thanks to the many new friends I made during my stint here, and older ones who kept in touch, many through long phone calls, all of whom ensured I remained in

high spirits. Cybersecurity, privacy, and GPT-4 can operate with reduced internet data, so I will avoid mentioning them all. Michael Collins, previous flatmates, and friends at Chateau ensured I had a comfortable stay. Multiple instructors ensured I stayed physically healthy. Members of ALFA shared my enthusiasm for the outdoors, finding good food spots, and having fun at work. Thank you all for the warm memories. I am grateful to Una-May and Blake for their benevolent gesture of offering their summer home to a few of us labmates when COVID was at its worst in the USA; they set a very high bar for care and empathy. Thanks to my aunts and their families, who made me a part of theirs during my stay here. Thanks to my parents for making me who I am, and all my family for their support through the years and for visiting me here. Thanks to my partner for always making me smile. Thanks also to the many canine friends who kept me company through my stint here; they are too important to remain anonymous: Kencha, Julie, Todd, Maisy, Talula, Zuko, Bucky, Luna, and Alpha. Olivia, Lily, and Cascade make the list despite their feline forms.

This perhaps will be the most read page of my thesis. If I've missed mentioning you, know that I'll always appreciate all that you do.

Contents

Chapter 1

Introduction

The central theme of this thesis is how we humans understand computer programs, and how we can teach machines to understand programs the way we do. It stems from two puzzling observations I made before starting graduate school, which were the following.

1.1 Puzzle 1 - Human intelligence tasks

Before graduate school, I worked for a research group where we assessed and quantified skills which signal employability in a labor market. Skill assessments had until then largely been confined to objective tests (multiple-choice questions) because subjective assessments (free-form responses) were harder to assess. Our group was one of the first to view the problem of subjective assessments of skills as problems in computer science. We demonstrated how many free-form response assessments can be cast as problems in machine learning (ML) [Srikant et al., 2019]. Test-takers' responses were treated as data-points in a high dimensional space, from which we predicted their true, latent, underlying score. We developed novel assessments for skills like spoken English [Shashidhar et al., 2015a], written English [Shashidhar et al., 2015b, Unnam et al., 2019], fine motor skills [Singh and Aggarwal, 2016] and situational judgement [Stemler et al., 2016], in addition to domain-general skills like logic and quantitative reasoning [Aggarwal et al., 2016].

One area that I was closely involved in was the assessment of computer programming skills [Srikant and Aggarwal, 2014a, Singh et al., 2016, Takhar and Aggarwal, 2019]. We trained predictive models to look beyond the functional correctness of programs, and assessed their partial correctness based on their semantic content. This helped a common issue arising in program assessments: zero credit for a failed test-suite despite having written a program that was *almost* what was expected. We successfully demonstrated how to utilize corpus-level statistics to assess programs that match those written by expert programmers.

As background, the notion of *code models*—language models or statistical models trained on code corpora, was beginning to be put to test around the same time we were developing our predictive models for assessing programs (*circa* 2013). Works by Allamanis and Sutton [2013] and Raychev et al. [2015] demonstrated practical applications like code summarization and variable renaming in Java and Javascript respectively. The idea of using a corpus of programs to train language models and other statistical models to ease developer workload was becoming mainstream. Allamanis et al. [2018b] surveys subsequent works in this space.

To train our machine learning (ML) models on code, we were routinely annotating *ground truth labels* for the programs in our corpus. This required experienced programmers evaluating a subset of programs in our corpus with a carefully constructed rubric. In watching expert programmers perform this task, I made a puzzling observation: experts were needed to annotate even the simplest of programming tasks, and irrespective of their expertise, they found annotation challenging and time consuming. This was in contrast to domains like speech, images, and text understanding, in which most tasks which are hard to describe using algorithms can be quite easily done by non-experts, essentially invoking their innate *human intelligence*. Examples of such tasks include identifying objects in an image or identifying inarticulate speech or text samples. Amazon's Mechanical Turk [Paolacci et al., 2010], a popular crowdsourcing platform, uses the term human intelligence tasks (HIT) to describe such tasks. Strangely, understanding and annotating programs were never HITs, even for the best, expert programmers. Any one of the following explanations possibly justifies this

observation.

- The time the human race has been using and inventing programming languages is much less than the time the human race has learned and acquired skills like natural language, vision, and speech. It is hence possible that there exist dedicated brain regions for processing language, vision, and speech while none exist for programming languages, thus requiring us more time to process programs.
- Experience could be another factor. A typical adult is exposed to many more years of language, vision, and speech than a programming language in their lifetime. Perhaps a child who is exposed to a programming language as its first language will process programs as easily as adults process natural languages.
- It is possible that the nature of programming models and environments offered by different languages contribute to the ease with which we understand programs. For instance, *visual learners* may find web mark-up languages simpler and more *natural* to reason about. Similarly, some find it easier to visualize and mentally manipulate rows and columns of data, thus finding languages like R, Matlab, and libraries like Numpy easier to understand. Some anecdotally find functional languages easier to comprehend than others.

While the explanations for the underlying processes were not clear, it was clear that in order to solve this puzzle, it was necessary to address the behavioral factors that underlie comprehension. Further, it suggested room for rigorously defining ideas like *visual learners* and *easier to reason about* in this context.

This observation also raised questions about the different code models proposed in the literature. The accuracy of code models at tasks such as summarization and predicting tokens were not as high as at tasks in language processing. Was this because training code models on tasks that were not HITs inherently harder? Understanding the brain and behavioral bases of comprehension would hopefully inform how we could improve training computational models to perform code reasoning tasks.

Around the same time, Siegmund et al. [2014] published their influential study on using fMRI to study programmers' brains. This provided additional support for the potential to study and establish the behavioral foundation for comprehending

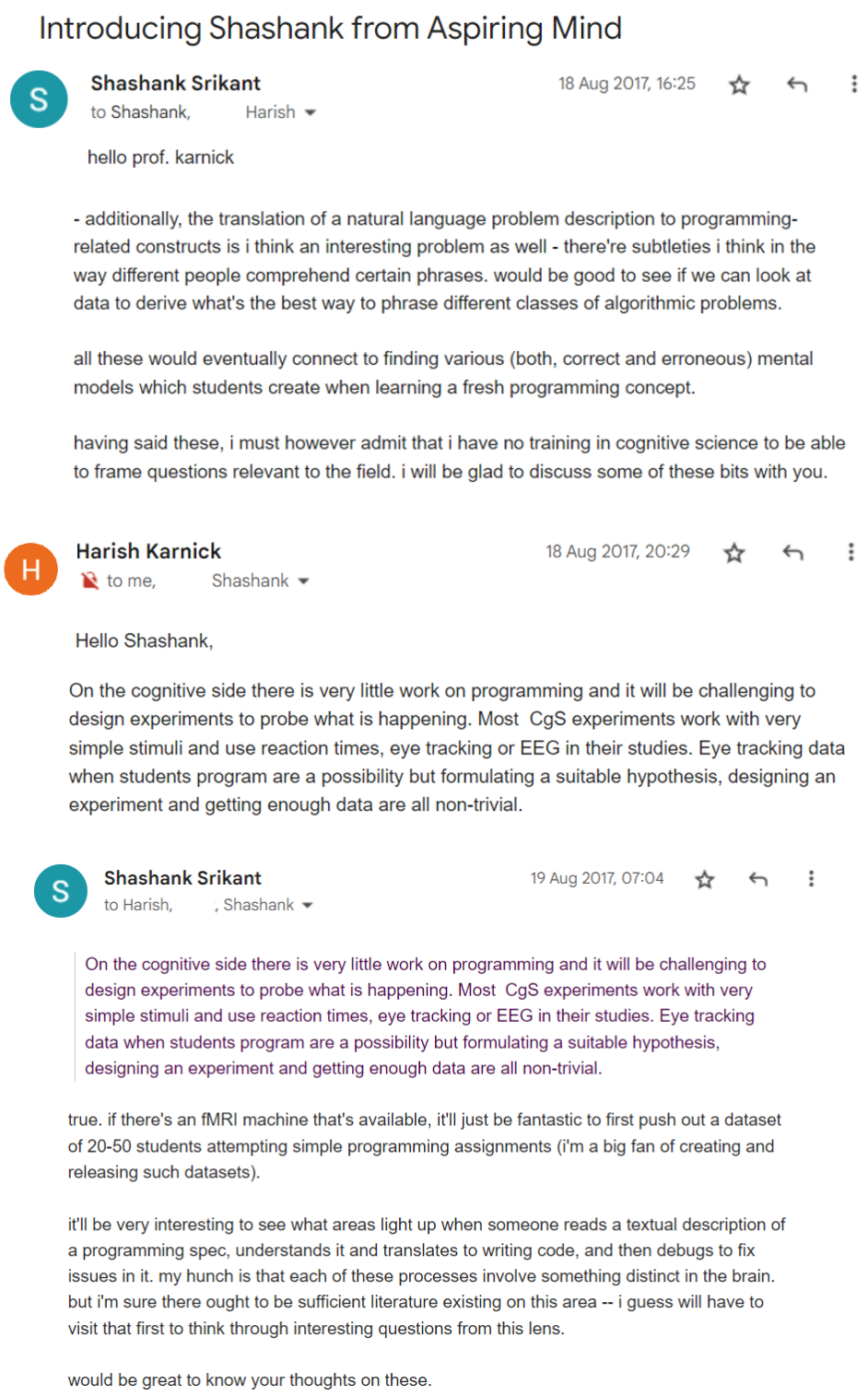

Figure 1-1: Excerpts from an email conversation with Prof. Harish Karnick, Emeritus Fellow, IIT Kanpur, on the possibility of studying the brain bases of programming, *circa* August 2017.

code. Shown in Figure 1-1 is my conversation on this topic with Prof. Harish Karnick, Emeritus Fellow at Indian Institute of Technology Kanpur (IITK)[1], just before I started graduate school.

1.2 Puzzle 2 - Programs and patterns

Around the same time as I was asking these questions, I came across a now prominent work on the naturalness of software by Hindle et al. [2016]. The authors analyzed code *in the wild*—available in public software projects, open-source repositories *etc.*, and showed the frequency distribution of the tokens and phrases (collection of tokens) appearing in code corpora followed a Zipf-like distribution. This frequency distribution is obtained by first computing the frequency of all the unique tokens across all programs appearing in a corpus, and then plotting these frequencies in a ranked (usually descending) order. Figure 1-2 shows an example of a Zipf-like distribution.

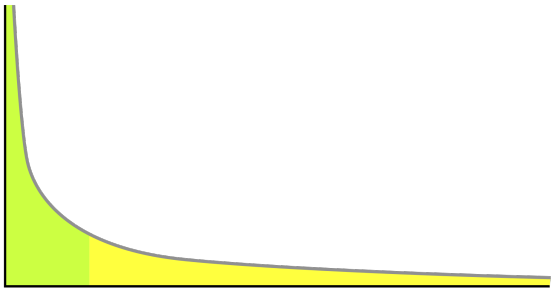

Figure 1-2: A Zipf-like distribution. For language, the X-axis represents unique words or phrases appearing in corpora of text, and the Y-axis the frequencies corresponding to each of those words/phrases occurring in the corpora. Image source: Wikimedia Commons

The authors found this distribution to hold irrespective of the programming language itself: the distribution of tokens in corpora across languages like C, Java, Python each showed a similar distribution. Word frequencies from corpora of text have long been shown to follow a Zipf-like distribution [Piantadosi, 2014]. Such a statistical regularity of words and tokens has been used in applications such as data compression [Schwartz, 1963], and cryptography [Boztas, 1999]. Hindle et al. [2016]

[1] https://iitk.ac.in/new/dr-harish-karnick

propose similar applications for code that can make use of such regularity, such as code auto-completion, code summarization, and more.

While the reason for the occurrence of this particular distribution is currently not well established, one popular account attributes it to a communicative optimization principle [Piantadosi, 2014]. According to this principle, the task of speaking and writing text is considered to have been evolved to optimally facilitate communication. The principle suggests any communication language will exhibit a statistical regularity that helps maximize its successful reception. In light of this principle, the results from Hindle et al. [2016], which had also been reported in previous studies [Shooman and Laemmel, 1977, Clark and Green, 1977, Chen, 1991], remain particularly puzzling. For one, the distributions are similar despite the many differences in the grammar and the execution semantics of programming languages and natural languages. Further, the communication recipient in the case of code is an assembly-level, register-based execution model. Why do humans then produce code with a Zipf-like token distribution when communicating with a machine that has been invented by humans?

Could it follow that our *minds*—our consciously aware perceptions and thoughts [Shiffrin et al., 2020]—use some mechanism which naturally constrains the way we express communicative thought? What is an analytic description of this mechanism? As a corollary, when understanding code, do our minds inherently expect this distribution? How we might is unclear. Bicknell and Levy [2012], Malmaud et al. [2020] propose a probabilistic framework to explain text understanding. Does a similar probabilistic account explain code understanding? Importantly, are code models encoding this probabilistic mechanism, thus giving them the ability to reason about the communicated intent in programs? These were the open questions motivating me, which I attempt to investigate in this thesis.

1.3 Reconciling these puzzles - Questions that arise

These two puzzles inform the theme of the questions I explore in this work. Reconciling the questions raised by the puzzles, I ask the following questions, which lie at the

intersection of computational models of code understanding and human behavioral bases of code understanding.

- **Thesis Question 1: Computational perspective.** To start with, what is a good framework to evaluate code models' understanding of programs. Code models are either *supervised*—trained to infer a specific task like code summarization, type inference, *etc.*; or *unsupervised*—language models trained on code corpora. Further, these models are typically trained either on *source code* or, as I propose, can be trained on *execution traces of programs*.
- **Thesis Question 2: Cognitive neuroscience perspective.** Similarly, what is a good framework to understand how code comprehension happens in our brains and minds. *Brain* refers to the neurons, cells, and chemicals that govern activities of an organism. *Mind* is often considered consciously aware perceptions and thoughts [Shiffrin et al., 2020].
- **Thesis Question 3: Bridging the two perspectives.** Is there any correspondence between the information encoded by code models and human brains when comprehending programs? Can computational models help in learning how our brains and minds comprehend programs? Can our brain and minds inform the better design of computational models?

In the following section, I describe how the chapters in this thesis correspond to these key research questions.

1.4 Thesis map

I develop multiple ideas to address each of the three questions introduced in Section 1.3. Each idea has been described as a separate chapter in this thesis. I motivate the relevance of each idea and summarize key results here. I also describe how these ideas contribute to addressing the three broader thesis questions.

Figure 1-3 summarizes the key themes of the different chapters in this thesis, and shows how they relate ontologically.

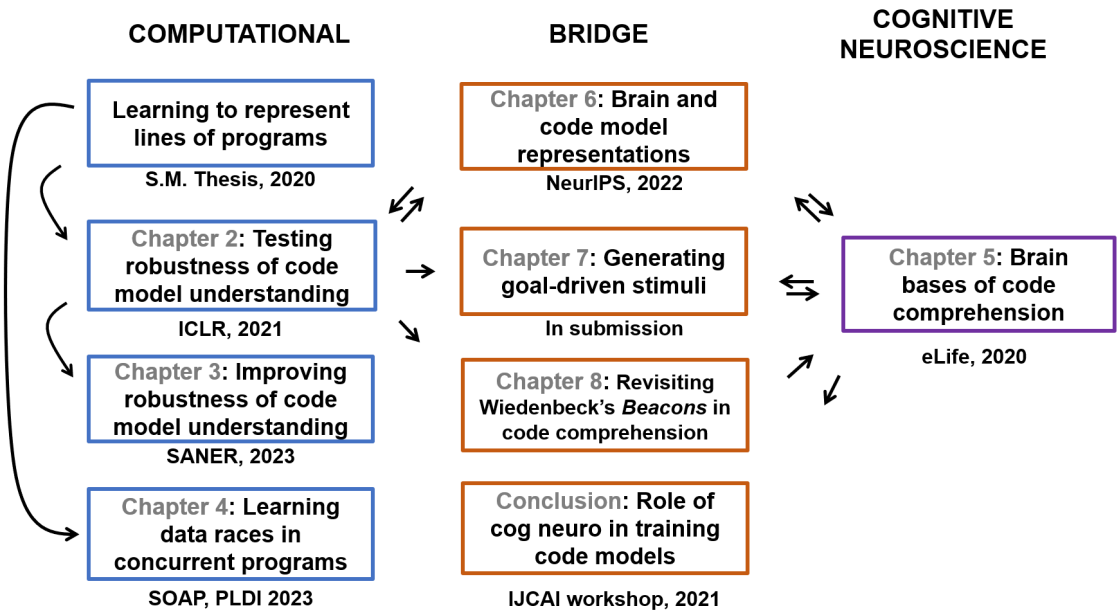

Figure 1-3: Thesis map. The three verticals correspond to the three broad thesis questions I introduce in Section 1.3. Each box describes the theme of one chapter in this thesis. The *bridge* chapters inform and are informed by ideas from chapters in both - the *computational* and *cognitive neuroscience* verticals. The arrows indicate these relationships.

Thesis Question 1. Computational perspective

Chapter 2. Testing the robustness of code model understanding using source code modifications.

I propose a principled method to test how well models trained on source code understand programs. The key idea is that humans' understanding of code is not affected by minor changes made to the code. We hold code models to the same test. For example, a model's output should not be affected by a variable being consistently renamed from `x` to `y`. The proposed method attempts to find such small changes which (a) do not alter the semantics of the original program, but (b) change the model's output. If such minor modifications are easy to find, it demonstrates the brittle *understanding* these models have of programs. I formulate finding such minor modifications as a first-order optimization problem. The optimization solves for two key components: which parts of the program to transform, and what transformations to use. I show that it is important to optimize both these aspects to generate the best candidate changes which are minimal and can flip a model's decision. Although I evaluate this

method on Python and Java programs, the proposed method is independent of the model (supervised or unsupervised), or the languages the models are trained on.

The details of this method are presented in chapter 2. It is, in full, a reprint of *Generating adversarial computer programs using optimized obfuscations.* Srikant, S., Liu, S., Mitrovska, T., Chang, S., Fan, Q., Zhang, G., and O'Reilly, U.M. (2021). ICLR 2021. [Srikant et al., 2021]

Chapter 3. Improving the robustness of code model understanding while retaining model accuracy

In this work, I propose improving the baseline understanding of code models that I test and measure in chapter 2.

I separately address two types of models - unsupervised code models (language models) and supervised, fine-tuned models. For unsupervised models, I provide a constrastive learning setup to learn those properties which humans can naturally reason about when comprehending code, such as invariance to variable names. For supervised models, I identify the existence of a *sweet-spot* in the frequency of updates made to the model parameters when being updated to learn the different human-like properties.

I show that these two solutions bring models closer to what humans can reason about when comprehending code.

I describe this work in Chapter 3. It is, in full, a reprint of *CLAWSAT: Towards Both Robust and Accurate Code Models.* Jia*, J., Srikant*, S., Mitrovska, T., Chang, S., Gan, C., Liu, S., and O'Reilly, U.M. (2023). SANER 2023 [Jia et al., 2022]. Jinghan Jia contributed equally with me as a primary author in this work.

Chapter 4. Learning code models to understand concurrent programs using program execution traces.

Of the several applications and developer tasks which code models can learn and assist, tasks that reason about concurrent programs have been studied the least [Allamanis et al., 2018a]. In this work, I describe how code models can learn to understand

concurrent programs, and specifically reason about data races.

I first propose a theoretical formulation for an ML model to learn data races. I discuss how operationalizing this idea is challenging. I then study the limits of neural networks architectures in learning and detecting data races from execution traces. I model events appearing in a program thread as a string of characters in a toy language that I design. Using such a language to denote threads, I study how well different ML models can be trained to detect the presence of specific substrings in the toy language that represent data races.

I then attempt to learn ML models on the execution traces of real concurrent programs. In my attempt, I learned the following severe limitations in prior work:

- No comprehensive dataset of concurrent programs exists in which data race conditions have been clearly labeled. Such a dataset is essential to get started with any ML-based approach.
- Data race detection algorithms proposed over the last four decades have not been evaluated on such comprehensive, labeled datasets. Instead, they have typically compared their performance to other prior algorithms and have reported only relative improvement. It is thus unclear how accurate these different algorithms are.

We develop *RaceInjector* to address this issue of a lack of a comprehensive dataset, which uses a Satisfiability Modulo Theories (SMT)-based solver to generate multiple possible traces which contain an injected data race. This generates a dataset of traces, wherein each trace is guaranteed to contain a data race. Such a dataset is suitable as a benchmark to rigorously evaluate other data race detection algorithms, and train ML models to detect data races.

I describe this work in Chapter 4. Section 4.2 describes the theoretical formulation. Section 4.3 refers to a thesis I mentored, authored by Teodor Rares Begu: *Modeling concurrency bugs using machine learning.* Rares Begu, T., Srikant, S., and O'Reilly, UM (2020). MIT SuperUROP Thesis [Rares Begu, 2020]. Section 4.4, in full, is a reprint of *RaceInjector: Injecting Races To Evaluate And Learn Dynamic Race Detection Algorithms.* Wang, M., Srikant, S., Samak, M., and O'Reilly, U.M. (2023) Wang et al. [2023].

How the chapters contribute to the computational perspective.

- **Chapter 2.** Humans can understand code despite simple changes made to it. Can models do the same? The method proposed in this work uses this idea, and serves as a practical baseline test of how well code models understand code.
Given how general our formulation is, we also show its application in generating English sentences that can elicit specific neural responses in the brain (details in Chapter 7).
- **Chapter 3.** This work identifies ways to fix the brittleness in code model understanding which the method from Chapter 2 identifies.
- **Chapter 4.** This work takes the first step towards training code models to comprehend and reason about concurrent programs. It specifically develops a way forward for designing data-driven data race detectors, which can potentially improve upon the heuristics that have been proposed over the last four decades. To the best of my knowledge, no previous attempts have been made in this regard.

Thesis Question 2. Cognitive neuroscience perspective

Chapter 5. In this chapter, I identify the regions of our brains involved in code comprehension. We consider two candidate brain systems—the Multiple Demand (MD) system and the Language system (LS). While the MD system is known to respond to stimuli involving general problem solving, math operations, and logic operations, the LS is known to be sensitive to language inputs alone. We establish whether reading and comprehending programs activates the LS or the MD system by using fMRI to study brain activity in participants reading code. We find that the LS does not consistently respond when comprehending programs, while the MD strongly does.

This chapter, in full, is a reprint of the arXiv report Srikant et al. [2023a]. The report is a rewrite of *Comprehension of computer code relies primarily on domain-general executive brain regions.* Ivanova, A. A., Srikant, S., Sueoka, Y., Kean, H. H., Dhamala, R., O'Reilly, U.M., Bers, M. U., and Fedorenko, E. (2020). Elife, 9:e58906

[Ivanova et al., 2020]. The chapter informs the results of the eLife work to a computer science audience; the original eLife work was written to primarily inform a cognitive neuroscience audience.

How the chapter contributes to the cognitive neuroscience perspective.

This work establishes the regions of the brain most responsible for code comprehension. Knowledge of the functional regions of the brain involved in code comprehension allows us to probe more into the nature of information represented (stored) in these brain regions.

Thesis Question 3. Bridging the two perspectives

Chapter 6. Mapping brain and model representations

In this work, I attempt to describe the nature of information encoded in the different brain regions identified in Section 1.4. This gives us an insight into the division of labor during code comprehension. For instance, token-related information can be encoded more prominently in Language system (LS), but, say, loops can be encoded in the Multiple Demand (MD) system. Similarly, numbers-related processing can happen in the MD while strings-related processing happen in the LS.

In addition to understanding this division of labor, our approach of decoding program-related information the first step at establishing the bases of the MD system. The MD system has typically been associated with *fluid intelligence*, but no description exists of *fluid intelligence*. Thus, the nature of operations performed in the MD system have not been rigorously described. Programs are a natural way to describe tasks that resemble *problem-solving* and *general intelligence* [Newell et al., 1958]. Newell et al. [1958] were the first to propose how tasks requiring some form of problem-solving can be described using computer programs. Thus, if information from programs are well encoded in the MD system (as opposed to the LS), it provides initial evidence to the bases of the MD system.

Additionally, I test whether information encoded in the brain can predict repre-

sentations (embeddings) learned by code models. A strong correspondence between the two representations—of brain regions and code models—would suggest that the learning objective used by code models to learn the parameters of the models resembles that employed by our brains, which thus result in the emergence of similar representations.

We show that the program-related information is encoded both in the MD and Language systems. Execution-related properties are more strongly encoded in the MD system. We find that representations from more complex models tend to align best with the MD system than the LS.

I describe this work in Chapter 6. It is, in full, a reprint of **Convergent representations of computer programs in human and artificial neural networks.** Srikant*, S., Lipkin*, B., Ivanova, A. A., Fedorenko, E., and O'Reilly, U.M. (2022). NeurIPS 2022 [Srikant et al., 2022]. Ben Lipkin contributed equally with me as the primary author of this work.

Chapter 7. Generating stimuli for cognitive neuroscience and psycholinguistics

Experiments in psycholinguistics and the cognitive neuroscience of language rely on linguistic stimuli (sentences) which either possess specific linguistic properties or which target specific cognitive processes. Such stimuli are generally assembled using manual or semi-manual methods, limiting their quality, quantity, and diversity.

I show how the method I propose in Chapter 2 can be reformulated to automate the generation of stimuli which target specific cognitive processes or possess desired linguistic properties while not being subject to experimenter biases which may arise from manual methods.

I describe this work in Chapter 7. It is, in full, a reprint of *GOLI: Goal-Optimized Linguistic Stimuli for Psycholinguistics and Cognitive Neuroscience.* Srikant, S., Tuckute, G., Liu, S., and O'Reilly, U.M. (2023) [Srikant et al., 2023b].

Chapter 8. What is *important* to programmers when comprehending code?

Soloway and Ehrlich [1984] and Wiedenbeck [1986] proposed the presence of *beacons* in programs: substrings in a program which programmers deem important to their understanding of the program. In this work, I verify whether common factors known to affect the comprehension of text such as surprisal and word length, and whether code model's representations can predict the behavioral finding by Wiedenbeck [1986]. The motivation is to determine the factors influencing programmer notions like "*important part of the code*", "*confusing part of the code*", and other such vaguely defined terms typically used by programmers.

I conduct a behavioral experiment in which I find the model's representations to be good predictors of the *importance* of a token in a program's overall comprehension, while the surprisal of a token as a signal is a weak predictor. I describe this work in Chapter 8.

How the chapters contribute to bridging the two perspective.

- **Chapter 6.**

– Our work is the first to describe the nature of program-related information encoded in the two brain regions most closely associated with code comprehension—the Multiple Demand system and the Language system.

– We take the first steps in describing the foundations of the MD system. Programs are a natural way to describe problem-solving tasks, which the MD system is believed to specialize in.

– We show a weak correspondence between the representations of a program in the brain and in code models. Future work may try to improve the architecture of current code models to improve this correspondence [Srikant and O'Reilly, 2021].

- **Chapter 7.** This work shows how experiment stimuli can be generated that satisfy diverse goals. This utility of this method was recently demonstrated in Tuckute et al. [2023], which establishes the ability to noninvasively control neural activity in higher-level cortical areas, like the language network.

While we do not demonstrate the generation of code stimuli in this work, the method can be used to similarly learn more about the sensitivity of the MD and Language system to the presence of specific code patterns.

- **Chapter 8.** I show how language models of code, when used as proxies of expert programmer knowledge, can help study different behavioral responses seen when understanding code.

1.5 Software

Software repositories relevant to the chapters presented in this thesis:

- **Chapter 2.** `https://github.com/ALFA-group/adversarial-code-generation`
- **Chapter 3.** `https://github.com/ALFA-group/CLAW-SAT`
- **Chapter 4.** `https://github.com/ALFA-group/RaceInjector-counterexamples`
- **Chapter 5.** `https://github.com/ALFA-group/neural-program-comprehension`
- **Chapter 6.** `https://github.com/ALFA-group/code-representations-ml-brain`
- **Chapter 7.** `https://github.com/alfa-group/goli`
- **Chapter 8.** `https://github.com/ALFA-group/beacons-in-code-comprehension`