Three Lenses on Improving Programmer Productivity: From Anecdote to Evidence

by

Madeline Endres

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in the University of Michigan
2024

Doctoral Committee:

Professor Westley Weimer, Chair
Professor Ranjit Jhala, University of California, San Diego
Lecturer Amir Kamil
Professor Ioulia Kovelman

Madeline Endres

endremad@umich.edu

ORCID iD: 0000-0002-4618-4939

DEDICATION

To my mom, Laura Koenig, without whom this would have been finished much later :)

ACKNOWLEDGEMENTS

I would like to acknowledge and express my deepest thanks to all those who have supported me throughout this PhD.

First and foremost, I would like to thank my advisor, Westley Weimer. Wes, your unwavering support, technical guidance, and broad advice on becoming a better researcher and mentor have been invaluable. Your flexibility in allowing me to pursue diverse and sometimes unconventional ideas, such as the intersection of cannabis and programming, has made this work possible.

I am also incredibly grateful to my committee members for your insightful feedback and encouragement. Over the course of this dissertation, you have all been fantastic mentors and collaborators, and your expertise and advice have greatly contributed to the quality of this dissertation. In addition, Amir, I'd like to thank you for mentoring me throughout my undergraduate, and encouraging me to pursue a PhD in the first place!

I also acknowledge the tremendous support from my family; Mom, Dad, and Naomi. In particular, I'd like to thank my parents, whose unwavering support and advice have been the bedrock of my academic pursuits—this dissertation would not have been possible without you.

To my partner, Dayton, thank you for your incredible help and encouragement throughout this journey. From helping me get through the imposter syndrome that was overwhelming the first year to suggesting grammar and spelling edits of my dissertation drafts, your support has been indispensable. I also want to acknowledge the companionship of our cat, Cleo, whose presence provided comfort and a sense of calm during the many long hours of work.

My heartfelt thanks to my friends who have made the doctorate process more manageable. In particular, I want to thank the many friends who have co-worked with me at various coffee shops over the past few years, including most recently Kaia, Grey, Sarah, and Maria. While I can't mention all of you by name, please know that your presence made a significant difference.

To my fellow WRG lab-mates throughout the PhD: Colton Holoday, Hammad Ahmad, Kevin Angstadt, Kevin Leach, Priscila Santiesteban, Yu Huang, and Zohreh Sharafi—thank you for your camaraderie, technical help, and insightful discussions.

I also want to express my gratitude to the amazing undergraduates and master's students that I've had the pleasure to work with, including Annie, Yiannos, Kaia, Wenxin, Manasvi, Emma, and

Danni. Your enthusiasm and dedication have been inspiring, and this dissertation would not have been possible without you.

Finally, I would like to acknowledge the interdisciplinary collaborators who have contributed to this work in various ways. Your perspectives and insights have enriched this dissertation, making it a truly collaborative effort.

Thank you all for your support, encouragement, and belief in me. This dissertation is as much a testament to your contributions as it is to my own efforts.

TABLE OF CONTENTS

LIST OF FIGURES

FIGURE

LIST OF TABLES

TABLE

LIST OF ALGORITHMS

ALGORITHM

ABSTRACT

In this dissertation, we present a series of algorithms and theoretically-grounded interventions that enhance programmer productivity. By combining large-scale exploratory empirical investigations with controlled human-focused experimental design, we both build mathematical models of the impact of understudied features on programmer productivity and also provide actionable, evidence-backed interventions that improve productivity in practice for targeted diverse programmer groups. We present findings from three primary lenses: *developing efficient and usable bug-fixing tools* for non-traditional novices, *designing effective programming training* informed by objective measures of programming cognition, and *understanding the impact of external factors*, such as psychoactive substance use. We briefly discuss the work conducted in each lens:

1. **Developing Efficient and Usable Programming Tools:** We propose and evaluate two novel methods of bug-fixing support targeting parse-errors and input-related bugs. Both are error types that we identify as commonly-encountered by *non-traditional novice programmers* (*e.g.*, those learning without the support of the traditional classroom) but are overlooked by existing program-repair tools.

2. **Designing Effective Developer Training:** To help novice programmers become more like experts faster, we develop a model of *novice programming expertise* using neuroimaging. We leverage our cognitive findings to design and evaluate a novel supplemental reading training that improves programming outcomes.

3. **Understanding External Productivity Barriers:** We argue that external factors also impact software productivity, including those anecdotally-reported but understudied by the scientific literature. In this dissertation, we study the impact of one such factor: *psychoactive substance use*. We both conduct the first survey of the prevalence of such substances in software and also develop a mathematical model of the true impact of one such substance, cannabis, on programming ability.

In this dissertation, we not only argue that varied external support can improve developer productivity, but we also specify which support can best do so. We contend that understudied factors and potential interventions can be identified through large-scale exploratory analyses. In addition,

we show how the impact of targeted interventions can be measured via causal experimental designs and large-scale human evaluations, even for factors impacting diverse populations that have previously only been considered anecdotally.

CHAPTER 1

Introduction

Software development is a driving force in the global economy; global employment in the technological sector grew drastically from 8 million employees in 2000 to 32 million in 2022, and is currently the "the most vibrant and fastest-growing segment of the global economy" [125]. Even as programming becomes increasingly integral to modern society, building software remains a challenging endeavor for novice and expert programmers alike [158, 202, 239]. Understanding and supporting programmer productivity is key to the efficacy and efficiency of software development [226, 303]. Factors such as a lack of usable programming support [223, 319], inadequate training [2, 319], a negative non-technical environment [226], or even social biases in the workplace [97] can decrease developer productivity.

Many ways to improve developer productivity have been proposed and implemented (see Wagner and Ruhe for a review [334]). However, companies can struggle to identify what productivity factors and interventions best support their developers [226, 319]. For example, most proposed approaches focus on improving productivity through additional technical support, but the top productivity factors are often non-technical [226]. In addition, large productivity differences between software developers have long been observed [4, 274], even when controlling for common quantitative [274] or environmental [105] factors, suggesting that some developer populations are currently not supported well by existing techniques. Finally, companies may not be willing to take the risk of implementing suggested productivity support (technical or not) until they know that it will work in practice: adoption of new tools is low unless an explicit business case is presented [20] and trust is established with both managers and engineers [236].

This reluctance may partially result from a history of suggested productivity support being overturned by later human-focused empirical evaluation. For example, in the automated debugging literature, a series of approaches aimed to help programmers fix broken code by providing them with a ranked list of the possible code locations most likely to contain the error (see [160] as an indicative example). The implicit assumption was that if the buggy code location was in the list, the developer could use it to identify and fix the issue immediately. The hope was that this would then lead to increased productivity via faster bug finding and fixing. In practice, however, this was not the case: a human evaluation found that providing this location support did not allow developers to gain perfect bug understanding. Instead, developers would skip around the provided ranked list, wasting 61% of their time inspecting irrelevant code statements [244]. Although this is only one example, we believe it is indicative of a more general trend: without considering the nuanced ways developers interact with their environment, even well-intentioned productivity interventions may fail to deliver their intended benefits in practice.

1.1 Approach and Thesis

In this dissertation, we present a set of systematic studies that *first*, identify previously understudied but important factors that influence software productivity, and *second*, provide rigorous human-focused evidence about the magnitude of their impacts. We use this approach to gain actionable productivity insights that generalize across programming populations. We argue that effective human-focused research on improving software productivity requires the following properties:

- **Provides Theoretically Grounded and Actionable Insights:** We desire grounded mathematical models that capture practical aspects of developer productivity. Based on those models, we want to propose and evaluate specific interventions that can help programmers in practice.

- **Includes Empirical or Objective Measures:** Many existing approaches for studying developer productivity and well-being rely primarily on programmer self-reporting. While valuable, prior work has found that subjective self-reporting can be unreliable in a software context [82, 110]. When possible, we prefer augmenting self-reported data with more objective measures (*e.g.*, recorded keystrokes, time on task, biometrics, *etc.*).

- **Minimizes Scientific Bias to Support Generalizability:** Studying humans can be challenging due to individual differences and sample biases. When possible, we favor leveraging techniques such as formal reasoning regarding theoretical outcomes, pre-registering hypotheses, and rigorous experimental control structures. We do so to minimize bias and promote generalizability.

- **Supports Diverse Developers:** Incoming preparation varies widely for software developers, leading to large productivity variance even between new developers with similar programming experience on paper [274]. We want solutions that consider programmers from diverse backgrounds, such as non-traditional novices trying to learn programming outside of the classroom or new programmers with different incoming cognitive abilities.

While previous research has considered factors influencing developer productivity and proposed developer support, it often lacks one or more of the elements above. For example, one prominent line of work identifies and ranks the importance of various productivity factors, frequently using a combination of surveys and interviews [226, 303]. While helpful for prioritizing various factors for future study, such approaches rely on subjective self-reporting [82] and typically do not test the relations between such factors and productivity in a causal manner. Thus, this work does not itself propose specific testable interventions. We prefer approaches that use more objective human-focused measures, such as biometric data or behavioral programming logs. Another line of work seeks to improve productivity by proposing better developer tools, including those that support debugging [189, 221]. While this work is important for advancing the state-of-the-art, most tool evaluations are entirely empirical and do not explicitly consider the user (*cf.* [169]). Those studies that do evaluate their tool from a user perspective often assume unrealistic user properties or focus participant recruitment on a different population than the one the tool is designed for (*e.g.*, students vs. professional developers), an oversight that can lead to tools not being useful in practice [100, 244]. We prefer solutions that explicitly consider human preferences via human studies with the target populations to increase the likelihood of supporting diverse groups of programmers in practice.

In this dissertation, we leverage three primary insights to conduct a set of studies that meet the aforementioned criteria of a good solution. **First**, we observe that exploratory empirical evaluations, especially those with hundreds of developers or millions of programming interactions, can identify previously understudied productivity barriers in software for diverse sets of programmers. This approach allows us to understand and leverage the anecdotal wisdom from sources such as developer-focused forums where programmers discuss concerns without oversight. In such situations, programmers may be more likely to contribute negative experiences as well as positive ones. **Second**, we note that rigorous and controlled experimental design can be used to discover evidence-backed conclusions to complex human-focused productivity questions. Real-world measurements of humans are inherently noisy. By carefully specifying the inputs and outputs in our experimental design, we are better positioned to capture any true signals through the noise. **Third**, we argue that ameliorations for productivity barriers that developers actually encounter in practice are more likely to be deployed or adopted. By providing generalized empirical evidence of the relevance of various factors and systematically validating the effectiveness of proposed interventions in diverse development contexts, we can offer solutions that are not only evidence-based but also tailored to the nuanced needs of specific programming communities.

We combine these insights into a set of systematic studies on understanding and improving programmer productivity. We consider three primary lenses: *designing efficient bug fixing algorithms* to help programmers quickly write correct code, *developing effective developer training* to help novices become more like experts faster, and *understanding external productivity factors* to help all developers regardless of background succeed in software and be happy while doing so. We present research on specific instances of each lens (*e.g.*, targeting a particular population or productivity factor). For each lens, we propose an initial exploratory phase and a subsequent rigorous human evaluation and/or actionable solution. Each lens also combines interdisciplinary methodologies (including those from programming languages, psychology, and medicine) with core software engineering techniques. The specific research components of this dissertation are as follows:

(1) Efficient Bug-fixing Support: Can we use programming languages techniques to support non-traditional novices in writing more correct code faster? Finding or fixing software bugs is one way to improve software productivity [319]. Novice programmers can find debugging particularly challenging, especially those who do not have the support of the traditional classroom [42, 87]. We hypothesize that such non-traditional novices may not only generally struggle more with program errors but also struggle with different errors than those faced by experts, error types not supported by existing tools. However, we also argue that programming language techniques useful for supporting more expert programmers can be expanded to support non-traditional novices. In this research lens, we first identify unsupported error types that non-traditional novices face via an empirical investigation. We then develop and evaluate support for two types of errors that we identify as barriers for novices: input-related bugs and parse errors.

(2) Effective Developer Training: Can we use medical imaging to inform supplementary cognitive training and improve programming outcomes? There is a growing body of work that uses neuroimaging to understand the cognitive processes behind programming (see [93] for a recent seminar). Such work, combined with a cognitive understanding of programming expertise, has the potential to lead to improved productivity through training targeted at identified relevant skills. However, this potential has yet to be explicitly tested in practice. In this dissertation, we pursue a two-phase approach: first, we use neuroimaging to construct a mathematical model of novice programmer cognition. Second, we combine insights from our model with those from other recent neuroimaging studies of programming to develop a cognitive training curriculum likely to transfer to programming. We evaluate the efficacy of our intervention using a controlled longitudinal study with programming students, finding a positive impact on programming outcomes. By integrating cognitive insights with software engineering education, we aim to develop a novel approach to programmer training that is empirically grounded and directly applicable to the challenges faced by novices, especially those with less incoming preparation in programming-related cognitive skills.

(3) Understanding External Productivity Factors: How does cannabis use impact software developer productivity? External factors beyond programming, such as cultural or environmental considerations, can also influence software developer productivity [105, 226]. Anecdotes connecting one such factor—cannabis use—to programming abound. There is a multitude of conflicting opinions on the cultural and productivity impacts of cannabis use (and psychoactive substance use in general) on software development (*e.g.*, [32, 64, 209]). However, little scientific research has been conducted to assess these claims empirically. In this dissertation, we demonstrate one instance of how external factors can impact software productivity through a series of studies that fill this gap. We first conduct an exploratory survey on the relationship between cannabis use and software development. We then carry out an observational study of the actual impact of cannabis on programming productivity. We hope this work helps replace anecdotes with evidence, enabling companies and individual developers to make more informed, evidence-based drug-related decisions.

Overall, the thesis of this dissertation is:

> *We can combine empirical evidence, theoretical modeling, and human-centered evaluation to develop and assess actionable interventions that improve programmer productivity in practice.*

Informally, we believe that more controlled methods from fields outside of software engineering can improve programming productivity for diverse groups. In this dissertation, we leverage techniques and insights from programming languages, machine learning, psychology, and medicine.

1.2 Summary of Contributions

This dissertation's contributions include:

1. *Lens 1:* The first identification of, and support for, input-related bugs as a productivity barrier for novice programmers, along with a novel neurosymbolic approach for efficiently fixing novice parse errors.

2. *Lens 2:* A model of novice programmer cognition, developed using neuroimaging, along with the first controlled evaluation of the practical use of neuroimaging findings on software productivity training.

3. *Lens 3:* A case study on how external factors can influence programming productivity via the first systematic survey of cannabis and programming, along with the first controlled study of the impact of cannabis on programming productivity.

We also make relevant source code and data publicly available (when ethically permitted).

1.3 Organization

In this dissertation, we conduct and report on a series of experiments that both identify and try to ameliorate (either through a proposed intervention or evidence-backed mathematical model) factors impacting developer productivity. As described in Section 1.1, these experiments are grouped into three different lenses on programmer productivity, each targeting a different sub-population or interdisciplinary methodology. The organization of this dissertation is as follows:

- *Chapter 2—Background and Related Work:* We cover background and related work relevant to each of the three lenses.
  - For the first lens on building better bug-fixing support for non-traditional novices, we first introduce Intelligent Tutoring Systems (subsection 2.1.1) and Automatic Program Repair (subsection 2.1.2). We then cover work related to our proposed techniques for fixing novice input-related errors (subsection 2.1.3) and parse errors (subsection 2.1.4 and subsection 2.1.5).
  - For the second lens on leveraging cognitive insights to design effective developer training, we provide relevant background on neuroimaging methodology (subsection 2.2.1), notation (subsection 2.2.2), and prior use in a programming context (subsection 2.2.3). We then provide additional context for two cognitive skills commonly connected to programming: technical reading (subsection 2.2.4) and spatial visualization (subsection 2.2.6). We conclude with a summary of how training such cognitive skills can transfer to improvement in another domain (subsection 2.2.8).
  - For the third lens, we first provide a brief overview of how external factors can influence programming productivity (subsection 2.3.1). We then provide background on general psychoactive substance use (subsection 2.3.2) and on cannabis in particular (subsection 2.3.2). We conclude with a discussion of related work on the intersection of psychoactive substances and programming (subsection 2.3.3 and subsection 2.3.4).
- *Chapter 3 (Lens 1)—Supporting Non-Traditional Novice Programmers' Productivity Through Targeted Bug-Fixing Support:* We focus on providing support for two error types that we identify as particularly challenging for novices: input-related errors, and parse errors.
  - *Lens 1 Overview (Section 3.1):* We overview Lens 1, detailing our motivation and summarizing our approaches for fixing input-related errors (INFIX) and parse errors (SEQ2PARSE).
  - *Input-Related Errors (Section 3.2):* We first identify input-related errors as a challenge for non-traditional novice programmers (subsection 3.2.1), and characterize common novice

  mistakes (subsection 3.2.3). We then present our novel input repair approach, InFix (subsection 3.2.2), which leverages insights from search-based program repair. We evaluate a Python implementation of InFix (subsection 3.2.4), finding that it is both effective (empirical evaluation, subsection 3.2.6) and helpful for novices (human evaluation, subsection 3.2.9).

  - *Parse Errors (Section 3.3):* We first motivate parse error repair for novice programmers (subsection 3.3.1). We then overview our approach, Seq2Parse (subsection 3.3.2), which combines insights from classical parsing algorithms and neural methods. As with InFix, we find that Seq2Parse is both effective (empirical evaluation, subsection 3.3.6) and of high quality (human study, subsection 3.3.10).
  - *Lens 1 Discussion and Conclusion (Section 3.4):* We summarize our first productivity lens and discuss the implications from our evaluations of InFix and Seq2Parse.

- *Chapter 4 (Lens 2)—Using Cognitive Insights to Design Effective Programmer Training:* To see how we can leverage cognitive insights to improve novice programmer productivity, we both use neuroimaging to build a cognitive model of programming and also conduct a controlled longitudinal study of the impact of training two cognitive skills that are relevant for programming: technical reading and spatial visualization.

  - *Lens 2 Overview (Section 4.1):* We overview Lens 2, detailing our motivation and overall experimental design for both building a cognitive model of new programmers and comparing two skills-based cognitive interventions.
  - *Neuroimaging Study (Section 4.2):* We use neuroimaging (fNIRS) with 31 participants to develop a mathematical model of new programmer cognition. We focus on understanding interactions with two cognitive skills identified as relevant for more expert programmers: reading (subsection 2.2.5), and spatial visualization (subsection 2.2.7). We find that both skills are distinct and relevant for novice programmers at the cognitive level (subsubsection 4.2.4.2).
  - *Transfer Training Study (Section 4.3):* We conduct a controlled longitudinal study with 57 participants, investigating if cognitive skills training can improve programmer productivity in practice. We find that our novel technical reading training (subsection 4.3.1) is effective, leading to improved programming outcomes compared to a standardized spatial assessment (subsection 4.3.5).
  - *Lens 2 Discussion and Conclusion (section 4.4):* We summarize our second productivity lens and discuss the implications from the finding that our reading training can transfer to improved programming outcomes.

- *Chapter 5 (Lens 3)—Understanding The Impact of External Productivity Factors; A Case Study on Cannabis Use and Programming:* To see how an understudied external factor can influence programming productivity, we both survey developers about their cannabis usage and also conduct a controlled observational study of how cannabis intoxication impacts programming productivity in practice.
  - *Lens 3 Overview (Section 5.1):* We overview Lens 3, detailing our motivation and summarizing our approach for determining if, when, and how cannabis use impacts software development.
  - *Cannabis Use Survey (Section 5.2):* We conduct and report on a survey of over 800 programmers, including over 450 professional developers. We focus on usage prevalence (subsubsection 5.2.4.1), context (subsubsection 5.2.4.2), motivations (subsubsection 5.2.4.3), and perceptions (subsubsection 5.2.4.4).
  - *Cannabis Observational Study (Section 5.3):* We develop a mathematical model of the impacts of cannabis intoxication on programming productivity via a controlled observational study. We focus on how cannabis impacts code correctness and programming speed, finding evidence for significant impairment for both (subsection 5.3.5).
  - *Lens 3 Discussion and Conclusion (Section 5.4):* We summarize our third productivity lens and discuss the implications of our results on software drug policies and in a broader software context.
- *Chapter 6—Summary and Conclusion:* We summarize the contributions of this dissertation, and discuss potential directions for future work.

Overall, the goal of this dissertation is to show that we can combine empirical evidence, theoretical modeling, and human-centered evaluation to develop and assess actionable interventions that improve programmer productivity in practice. By addressing the three aforementioned lenses that target both technical and non-technical factors, we hope to contribute to a more productive and inclusive software development environment. In the following chapters, we will delve into the specific studies, methodologies, and findings that support this goal.

CHAPTER 2

Background and Related Work

Before diving into our analyses and actionable interventions that improve programmer productivity in practice, we introduce concepts and vocabulary used throughout this dissertation. We provide background on our three productivity lenses: using programming language techniques to provide *efficient bug-fixing support* for non-traditional novices (section 2.1), using cognitive insights to improve programming outcomes via *effective developer training* (section 2.2), and understanding *how external productivity factors can impact software productivity* through a case study on cannabis use in software (section 2.3). Beyond general background, we further contextualize the contributions of this dissertation through an overview of the related literature to specific techniques in each lens, up to 2023.

Each primary related work section corresponds to one dissertation chapter. Specifically, Chapter 3 assumes the background in section 2.1 (Lens 1), Chapter 4 assumes the background in section 2.2 (Lens 2), and Chapter 5 assumes the background in section 2.3 (Lens 3). Readers interested in a specific chapter may benefit from the provided overview. Table 2.1 summarizes what work is relevant for each chapter or primary section.

2.1 Lens 1: Efficient Bug-Fixing Support

We present work relevant to our first lens into improving novice programmer productivity through automatically fixing programming errors. In this lens, we propose and evaluate two tools for helping novice programmers fix frustrating bugs via automatically generated patches: InFix, which repairs buggy program inputs, and Seq2Parse, which fixes parse errors in a *neurosymbolic* fashion (see subsection 2.1.2). We first give an overview of *Intelligent Tutoring Systems* (subsection 2.1.1) and *Automatic Program Repair* (subsection 2.1.2). We then cover work related to fixing input-related errors (subsection 2.1.3) and parse errors (subsection 2.1.4 and subsection 2.1.5), the two error types targeted in this dissertation.

Related Work Section	Relevant Chapter
Lens 1: Providing efficient bug-fixing support for non-traditional novices	
§ 2.1.1, Intelligent Tutoring Systems	Chapter 3, all
§ 2.1.2, Automatic Program Repair	Chapter 3, all
§ 2.1.3, Input Repair	Chapter 3, Section 3.2
§ 2.1.4, Error Correcting Earley Parsers	Chapter 3, Section 3.3
§ 2.1.5, Sequence Models for Software Engineering	Chapter 3, Section 3.3
Lens 2: Using cognitive insights to design effective training for new programmers	
§ 2.2.1, Neuroimaging and fNIRS	Chapter 4, all
§ 2.2.2, Neuroimaging Vocabulary and Notation	Chapter 4, Section 4.2
§ 2.2.3, Neuroimaging and Software Engineering	Chapter 4, Section 4.2
§ 2.2.4, Reading and Cognition	Chapter 4, all
§ 2.2.5, Reading and Programming	Chapter 4, all
§ 2.2.6, Spatial Reasoning and Cognition	Chapter 4, all
§ 2.2.7, Spatial Reasoning and Programming	Chapter 4, all
§ 2.2.8, Transfer Training	Chapter 4, Section 4.3
Lens 3: The impact of external productivity factors—a case study on cannabis use in software	
§ 2.3.1, External Factors and Programming Productivity	Chapter 5, all
§ 2.3.2, Cannabis and General Psychoactive Substance Use	Chapter 5, all
§ 2.3.3, Psychoactive Substances and Programming	Chapter 5, all
§ 2.3.4, Cannabis and Programming	Chapter 5, all

Table 2.1: Overview of Related Work: Summary of related work descriptions. Each related work topic is linked to its most relevant chapter or section. This related work is assumed in the linked chapters.

2.1.1 Intelligent Tutoring Systems and Computer Aided Instruction

Intelligent tutoring systems are learning platforms that integrate computational models from various disciplines, such as cognitive science, learning science, programming languages, or artificial intelligence [119]. There exists a large body of work investigating and evaluating intelligent tutoring systems for learning programming [23, 54, 140, 149, 188, 231, 265, 298]. These systems target a wide range of programming languages and experience levels, and have been found to improve learning outcomes effectively and efficiently [231].

Many intelligent tutoring approaches provide data-driven source code fixes to serve as general hints for learning [140, 188, 265]. For example, Hartmann *et al.* use crowdsourcing to provide selected solutions to error messages [140]. Others, such as Singh *et al.*, take advantage of a reference implementation to provide more specific feedback [298]. Data-driven approaches are also popular; for example, Lazar *et al.* use "textual edits commonly used by students on program code" to "provide hints with varying levels of detail" [188].

Many tutoring systems also use static analysis or constraint solving to provide state-based hints [130, 149]. One such approach is Python Tutor [130], a popular free online tutoring environment that is often used by novices who are learning programming without traditional classroom support [129, 130]. PYTHON TUTOR allows learners to write, run, step through, and visualize arbitrary Python code. Should the program contain an error either in the code's syntax or runtime behavior, PYTHON TUTOR will provide the user with Python's native *error-message* (a short textual description that provides additional context that might help the programmer understand and fix the bug).

In this dissertation, we follow the tradition of automatically generating program fixes that could be provided to students as debugging hints. Specifically, we generate fixes for two types of errors that we find are common for novices through an analysis of PYTHON TUTOR data (see subsection 3.2.3 and subsection 3.3.1): *input-related errors* and *parse errors*. Our approach combines insights from more formal program modification techniques (*e.g.*, automatic program repair or error-correcting parses) with data-driven insights from a large corpus of arbitrary novice-written code submitted to PYTHON TUTOR. We now discuss additional background relevant to this approach.

2.1.2 Automatic Program Repair

Automatic program repair (APR) is a set of techniques for automatically generating fixes for buggy or incorrect programs. A repair, also called a *patch*, is typically a set of changes or updates to a computer program's source code designed to fix bugs, improve functionality, or address security vulnerabilities. There is a vast literature on automatically repairing programs: proposed in 2008, APR has developed into a vibrant research area with over 500 publications (*cf.* the review [221]). Overall, traditional techniques take as input the source code for the buggy program and output a synthesized patch.

Beyond program source code, APR techniques typically require as input the location of the bug (or some method for finding it), and some method to determine that a candidate patch correctly fixes the bug (*e.g.*, a test suite or formal specification). APR has seen some industry adoption, including use at Facebook (Meta) where it is used to repair bugs in multiple real-world systems at scale (*e.g.*, each with millions of lines of code) [206]. Various methods are used to generate candidate patches, including search-based (*e.g.*, GENPROG [191], CAPGEN [341]), synthesis-based (*e.g.*, SEMFIX [233], ANGELIX [217]), data-driven (*e.g.*, DEEPFIX [132]), and template-based (*e.g.*, PAR [169], TBAR [195]) approaches.

Pedagogy-Focused APR. Most APR approaches target expert programmers or professional software developers, and such expert-focused tools can be confusing for novices in practice [354]. For example, Yi *et al.* study the feasibility of state-of-the-art program repair tools for helping students repair source-level errors [354]. They find expert-focused tools and their derivatives unhelpful for students, though they are potentially useful for course graders.

Previous work on pedagogically motivated automatic program repair and fault localization focuses on large course assignments [6, 190, 240, 354] rather than tutoring support for non-traditional students. For example, Ahmed *et al.* build statistical models to help repair submissions for 14 different problem sets, using between 400 and 9,000 submissions per problem for training [6]. However, as we focus on fixing errors in generic student programs, our approach must operate without a large corpus of fixes for the same program.

In this dissertation, we contribute two novel techniques for efficiently and effectively automatically fixing errors commonly encountered by novices: InFix and Seq2Parse. InFix leverages insights from search-based and template-based APR to automatically fix buggy program inputs (rather than modifying the program's source code). Seq2Parse repairs parse errors by combining the theoretical guarantees of *Error Correcting Parsers (EC-Parsers)* [8] with the efficiency of Sequence to Sequence Neural Transformers. This is a *neurosymbolic* approach because it enhances a traditional symbolic technique (EC-Parser) with a neural module. We now discuss relevant background and related work for each approach.

2.1.3 Input Repair: Automatic Input Rectification, Sanitization or Fuzzing

We now discuss existing work related to InFix, our approach for repairing buggy program inputs. Overall, limited work has been done on automatically repairing input data [11, 198, 221]. Most existing research focuses on improving security for industrial programs. For example, Long *et al.* use provided tests to learn what non-malicious inputs look like for a program [198]. They then automatically correct "atypical" inputs to fit the learned pattern. For arbitrary programs created by novice programmers, however, the input format is rarely specified and there are rarely test cases. To the best of our knowledge, there is no prior work on automatically repairing novice input errors or investigating their repair quality.

While not designed to fix buggy program inputs, *Fuzz testing* (also called *fuzzing*) is a popular software testing technique that involves generating and inputting random or invalid data into a program to identify bugs, crashes, or security vulnerabilities (*e.g.*, see [112, 113, 287]). Given a model for programming functions relevant for input gathering (*e.g.*, INPUT, SPLIT and INT in Python), fuzzing could be applied to the task of generating non-erroneous inputs. However, test input generators often struggle with real-time answers to semantic or dependent input paths (which we find often appear in novice's program inputs, see the scenario in Figure 3.2). While a few input generation algorithms have efficient web deployments, such as Pex [314], approaches that handle complex input constraints generally require minutes [57] rather than seconds. In addition, while there are some evaluations of fuzzing in pedagogical contexts (*e.g.*, as a game [315]), we are unaware of any work evaluating fuzzing quality for novice input repairs. We view a more thorough

```
S        → Stmts end_marker
Stmts    → Stmt \n | Stmt \n Stmts
Stmt     → FuncDef | ExprStmt
          | RetStmt | PassStmt | ...
FuncDef  → def name Params : Block
Block    → \n indent Stmts dedent
RetStmt  → return | return Args
Args     → ExprStmt | ExprStmt , Args
ExprStmt → ArExpr | ...
ArExpr   → Literal
          | ArExpr BinOp Literal
Literal  → name | number | ...
```

Figure 2.1: Simplified Python Grammar Production Rules.

```python
def foo(a):
  return a + 42

def bar(a):
  b = foo(a) + 17
  return b +
```

Figure 2.2: Ill-parsed Python Program: A Python program with two functions that manipulate an integer. The second one has a parse error.

evaluation of fuzz testing as an approach for fixing novice program inputs as future work.

In our approach, we use randomization to generate a single input repair (see subsection 3.2.2). We hypothesize that automatically synthesizing input grammars for student programs could result in richer information for providing hints. Synthesizing generic input grammars remains a challenging task. While there has been some recent work in this area [25, 117, 150, 182], some of this work fails to generalize or replicate [29]. For example, Kulkarni *et al.* develop an approach for learning and synthesizing context-free input grammars [182]. Unfortunately, our characterization of novice input structures in subsection 3.2.3 found that many are context-sensitive. While there are some first steps towards generating context-sensitive grammars (*e.g.*, generating predicates for select command-line invocations, see Gupta *et al.* [131]), generating arbitrary context-sensitive input grammars remains an open problem.

2.1.4 Error Correcting Earley Parsers

We now introduce *Error Correcting Earley Parsers* (ECE) Parsers [8]. We leverage ECE parsers in our approach to fixing novices' parse errors, SEQ2PARSE. While we briefly define all of the relevant terms, an interested reader can find more detailed information on parsing in Aho, Sethi, and Ullman [9]. Throughout this section, we include a running example using the Python programming language to help clarify relevant concepts.

Earley Parsers. An Earley parser accepts programs that belong to a language that is defined by a given *grammar* G by using dynamic programming, to store top-down partial parses in a data structure called a *chart* [88]. The grammar G has a starting symbol S and a set of *production rules*. As an example, Figure 2.1 presents some simplified production rules for the Python programming language that will help parse the program in Figure 2.2. *Terminal* symbols (or *tokens*) are syntactical symbols and are here presented in lowercase letters. Uppercase letters denote *non-terminal* symbols, which are rewritten using production rules during a parse. For example, the non-terminal `Stmt` defines all possible Python statements, including expressions (`ExprStmt`), return statements (`RetStmt`), *etc.*

Error-Correcting Earley Parsers. An *Error-Correcting Earley* (ECE) Parser extends the original algorithm's operations, to find a *minimum-edit* parse for a program with parse errors [8]. An ECE-Parser extends the original grammar G with a set of *error production rules* to create a new *error grammar* G' which has rules to handle *insertion*, *deletion*, and *replacement* errors. As an example, we demonstrate how to adapt Python's production rules for an ECE-Parser. First, the ECE-Parser adds to G' a new start symbol `New_S`, the helper symbol `Replace` that is used for replacement errors and the symbols `Insert` and `Token` that introduce insertion errors. Additionally, for each terminal `t` in G it adds the new non-terminal `E_t` that introduces errors relevant to the `t` symbol.

Next, in addition to the existing production rules, the error grammar G' has the following error rules. The new start symbol uses the old one with the option of an insertion error at the end:

- `New_S → S | S Insert`

Additionally, for each production rule of a non-terminal `T` in G, another non-terminal error rule is added that introduces the terminal symbols `E_t`, for each original terminal `t` it has. For example, the `Stmts`, `Block` and `RetStmt` rules are updated as:

- `Stmts  → ... | Stmt E_\n | Stmt E_\n Stmts`
- `Block  → ... | E_\n E_indent Stmts E_dedent`
- `RetStmt → ... | E_return | E_return Args`

Next, for each terminal `t` in G, we add four error rules of the type:

- `E_t → t | ϵ | Replace | Insert t`

These four new error rules have the following usage for each terminal `t`:

1. The `E_t → t` rule will match the original terminal `t` without any errors. This error rule is used in cases where the *non-error* version of the rule is needed. For example, in `Block → E_\n E_indent Stmts E_dedent` it can be the case that only `E_dedent` is needed to match the error and `E_\n` and `E_indent` can match their respective symbols.

2. Using `E_t → ϵ` a *deletion* error is considered. The error rule will match *nothing*, or the *empty token* ϵ, in the program, meaning the terminal is missing.

3. Using `E_t → Replace` a *replacement* error is considered. `Replace` will match any terminal token that is *different* than `t`, making a replacement possible.

4. The rules `E_t → Insert t` will introduce an *insertion* error, *i.e.* `Insert` will match any *sequence* of `Token`s that are not supposed to precede `t` to make the program parse.

For example, for the terminal tokens `return`, `number` and `\n` (a new line) the relevant error production rules are:

- `E_return → return | ϵ | Replace | Insert return`
- `E_number → number | ϵ | Replace | Insert number`
- `E_\n    → \n | ϵ | Replace | Insert \n`

Finally, the `Replace` non-terminal can match any possible terminal in G to introduce replacement errors, the `Insert` non-terminal will introduce a sequence of insertion errors by using `Token` which also matches every terminal and we just differentiate the name to be able to distinguish the different types of errors.

- `Replace → return | pass | \n | + | ... [all terminals]`
- `Insert → Token | Insert Token`
- `Token  → return | pass | \n | + | ... [all terminals]`

This eliminates backtracking and prevents a combinatorial explosion. Worst case, it has a time complexity of $O(n^3G^2)$ for generic context-free grammars, where n is the number of *tokens* of the input program and G is the grammar size, *i.e.* the number of *production rules* it includes.

The new and larger error-correcting grammar G' is at least 3 times larger than G. As a result, ECE-Parsers are not scalable for real-time repairs for large modern-day programming languages which often have grammars with hundreds of production rules. In this dissertation, we overcome this limitation of ECE parsers via leveraging sequence-to-sequence transformers (see subsection 2.1.5) to predict a subset of production rules most likely to repair a given program (see subsubsection 3.3.2.1 for details).

Other Error-Correcting Parsers. While we use Error-Correcting Earley Parsers in this dissertation, researchers have proposed other Error-Correcting parsing approaches. For example, Burke and Fisher [50] describes another EC-Parser that uses three phases: first, it attempts to repair the parse error by symbol insertions, deletions, or substitutions. If that fails, it tries to close one or more open code blocks, and if that fails, it removes code surrounding the erroneous symbol. Finally, it uses *deferred parsing* that may be viewed as double parsing, where one main parser moves forward as much as possible, whereas a second parser is k steps behind, so that it can backtrack to a state k steps before efficiently if a phase fails. Van der Spek *et al.* [321] have shown that the previous approach is not applicable in real-world languages for some specific cases (*e.g.* multiple function definitions) and has suggested an improvement that works with the JavaCC parser generator and a form of *follow-set error recovery*. Corchuelo *et al.* [72] have suggested an error-correcting version of the popular *LR parser* (for an overview of LR parsing, see Aho, Sethi, and Ullman [9]). Rather than focusing on error production rules, this method adds *error-repair transitions* along with the regular shift/reduce operations. It employs a simple cost model and heuristics to limit the explosion of the repair search space. Finally, Thompson [311] has suggested using *probabilistic parsing* to overcome the drawback of selecting the minimal-edit repair by using a PCFG to select the most *probable* repair parse. However, these approaches are impractical and inefficient for real-world applications, as they can only successfully parse small examples or use tiny grammars. In contrast, our approach to repairing parse errors (outlined in section 3.3) relies on *pre-trained sequence models* (see subsection 2.1.5) to efficiently explore the repair search space for a minimal overhead in real-time parsing.

2.1.5 Sequence Models for Software Engineering

We now provide relevant context on the use of sequence models in Software Engineering, with a focus on their use for fixing parse errors. A *sequence model* is a type of machine learning model designed to process and make predictions based on sequential data, where the order of the data points matters (*e.g.*, time series data or natural language text). *Sequence to sequence transformers* (seq2seq) are a type of sequence model where the output is also an ordered sequence. For more information on transformer models, see Jurafsky and Martin [162].

Sequence transforms have been used to augment approaches targeting a multitude of software engineering problems, including adding code comments, test-case generation, and code-clone detection (see Yang *et al.* for a review [353]). One task relevant to software engineering is *program synthesis*, or the automatic generation of executable code from a high-level specification of its intended behavior. While program synthesis traditionally leverages techniques from formal methods to reason about and generate code, Rahmani *et al.* [256] and Verbruggen *et al.* [328] suggested using pre-trained auto-regressive transformer models, such as GPT-3 [46], to augment pre-existing program synthesis techniques. They use pre-trained models to acquire semantic power over smaller subproblems that can't be solved with the syntactic power of classic program synthesis. Similar to our tool Seq2Parse (introduced in section 3.3), their work uses established pre-existing algorithms from natural languages processing and programming languages research areas. However, Seq2Parse trains its own Transformer-based model to augment an error-correcting parsing algorithm, providing more focused prior knowledge than a pre-trained sequence model, thus making our model highly accurate.

Sequence Models for Parsing. In the rest of this section, we consider work that is similar to our approach, SEQ2PARSE, which uses sequence models to fix program bugs including parsing errors. As part of our evaluation in subsection 3.3.11, we compare the performance of SEQ2PARSE to the performance of the most closely related approaches in this section.

SYNFIX [33] and *sk_p* [254] are two systems that use seq2seq models. They mostly focus on educational programming tasks, learning task-specific patterns for fixing erroneous task solutions. SYNFIX uses a model per task and takes as input the program prefix until the parser's error location. *sk_p* (while it does not solely focus on syntax errors) makes sequence predictions per program line, by considering only the abstracted context lines (previous and next lines). The model is applied to every program line and the predictions with the highest probabilities are selected. While both SYNFIX and *sk_p* learn task-specific patterns, our approach SEQ2PARSE can instead parse and repair a large number of programs regardless of the task they are trying to solve by encoding the full erroneous programs with a Transformer model and using an EC-Parser to parse them accordingly, thus achieving a much higher accuracy. Additionally, we use a real-world dataset of millions of Python programs to learn to effectively parse programs, while SYNFIX and *sk_p* are trained on smaller datasets of correct programs that have errors manually introduced on training, possibly skewing the predictions away from real-world fixes.

DEEPFIX [132] is another seq2seq approach for repairing syntactical errors in C programs. It relies on stacked *gated recurrent units* (GRUs) with attention and applies some simple abstraction over the terminal tokens. The input programs are broken into subsequences for each line and the model gets as input all the line subsequences with their associated line numbers. DEEPFIX only predicts single-line fixes and its predictions are applied iteratively multiple times, if multiple parse errors exist or until the parse error is fixed. DEEPFIX struggles with the same problems as previous work, as it solely relies on the sequence models' capability to learn the full grammar and repair programs with minimal abstraction and prior knowledge over the language.

Lenient parsing [5] presents another sequence model approach. It uses *two seq2seq Transformer models* and trains them with a large corpus of code. One model is trained to repair and create properly nested blocks of code, called BLOCKFIX, and the second one, called FRAGFIX, repairs and parses fragments of code (*e.g.* program statements) within a repaired block. While mostly automatic, it relies on the manual corruption of a dataset to generate erroneous programs that may not correlate to the errors actual developers make. It also relies on the seq2seq models to learn the underlying language model and make repairs. In contrast, our SEQ2PARSE approach mitigates this problem by learning how programmers fixed programs from a large corpus Additionally, our use of EC-Parsers and the language grammar significantly improves program repairs.

Finally, Graph-based Grammar Fix (GGF) [349] uses a *Gated Graph Neural Network* encoder for the partial program parse trees and a separate *GRU* encoder for the program part that is not parsed. Its models then predict an error location in the program sequence and a token suggestion for the repair. This single-token repair approach is applied iteratively until the program parses. While much more accurate than the other previous work, GGF still lacks the advantage we leverage in SEQ2PARSE of using a parser with the actual grammar as the final step of the repair process.

2.2 Lens 2: Cognitive Insights and Effective Developer Training

In our second lens (see Chapter 4), we investigate how to improve programmer productivity via leveraging cognitive insights to design better training. Software engineering researchers have long been interested in *program comprehension*, or understanding the cognitive processes that drive reading and writing source code [73]. By understanding what cognitive skills are relevant, researchers hope to gain insights that inform improved programming pedagogy and developer retraining, as well as to better understand productivity gaps between novices and experts [153]. Many cognitive skills have been associated with programming. Two of the more prominent ones are reading comprehension [53, 200, 224, 249] and spatial reasoning ability (a person's capacity to understand and reason about spatial relationships among objects, including activities such as mentally rotating 3D shapes) [207, 241, 243].

In this dissertation, we focus on 1) understanding the role these two cognitive skills play in the program comprehension of novice programmers, and 2) determining if cognitive training on these skills can *transfer* to improved programming outcomes. For the first phase, we conduct a neuroimaging study using *functional Near Infrared Spectroscopy* (fNIRS). In this background section, we give an overview of fNIRS in subsection 2.2.1, cover relevant neuroimaging vocabulary and notation in subsection 2.2.2, and discuss the existing work using neuroimaging in a programming context in subsection 2.2.3. For the second phase, we define technical reading and discuss reading training and transfer in subsection 2.2.4. We then outline connections between software engineering, program comprehension, and reading ability in subsection 2.2.5. Finally, we define spatial ability and describe its connections with computer science in subsection 2.2.6 and subsection 2.2.7.

2.2.1 Neuroimaging and fNIRS

One way to study program comprehension is using *functional neuroimaging* to capture relevant brain activity. It can provide a physically-grounded insight into cognition without relying on potentially-unreliable self-reporting [153, 293].

We focus on one technique: *functional Near Infrared Spectroscopy* (fNIRS). It is non-invasive, avoids the ionizing radiation present in other methods (*e.g.*, PET, CT), and can measure activity in brain regions not accessible to some invasive techniques (*e.g.*, electrocorticography). Importantly, fNIRS can be used in more natural and ecologically valid environments (*e.g.*, standard desktop computer use, *etc.*) compared to alternatives like *functional Magnetic Resonance Imaging* (fMRI), which requires participants to lie still in a small tube and also complicates the use of keyboards [181]. In addition, fNIRS offers higher spatial resolution than EEG, and higher temporal resolution than fMRI, which is important for studies relating a brain region's contribution to a specific task (such as program comprehension). These properties motivate our decision to use fNIRS in our study of novices' program comprehension.

fNIRS uses the *hemodynamic response*, or change in neuronal blood flow to active brain regions, to measure brain activity [55]. fNIRS measures this via the use of near-infrared light: transmitters and receivers are placed on a "cap" worn by participants. Oxygen-rich and oxygen-poor blood have different light absorption properties, so the hemodynamic responses in a given brain region between a transmitter and receiver pair (referred to as a *channel*) can be measured over time. fNIRS measures concentration changes in such oxygenated and deoxygenated blood. The number of fNIRS publications has doubled every 3.5 years since 1992 [36], and it has been used to study human development, injury, and psychiatric conditions [36, 90, 196, 238].

For our purposes, the use of fNIRS imposes two key experimental constraints: contrast-based design and task duration. First, fNIRS experiments typically involve participants completing tasks (*e.g.*, mentally rotating objects or solving programming problems) while time-series data is recorded. Carefully controlled experiments are necessary, in which the activity observed during one task is contrasted against the activity observed during another. This allows confounding brain activations (*e.g.*, motor cortex activity from moving the lungs to breathe) to be eliminated from consideration.

Second, because fNIRS is based on the hemodynamic response, care must be taken to model [30, 283] the onset of neuronal blood flow (which peaks slightly after stimuli are presented [1, 146]) and the design must avoid saturation and weaker signals for tasks involving long activity [194]. As a result, the hemodynamic response can typically be studied only in experiments with brief stimuli (*e.g.*, under 30 seconds per question). Furthermore, fNIRS is only able to penetrate a few centimeters down into the brain. More specifically, the near-infrared light can reach a depth that is roughly half the distance between a transmitter-receiver pair, depending on the wavelengths and light intensities used [102].

Despite these limitations, fNIRS specifically, and medical imaging in general, are growing in popularity for use in software engineering studies (*e.g.*, [61, 86, 99, 104, 153, 155, 181, 229, 247, 293, 294]). They provide a physically grounded insight into cognition without relying on potentially unreliable self-reporting [153]. We discuss the intersection of neuroimaging and software engineering in greater depth in subsection 2.2.3.

2.2.2 Neuroscience Vocabulary and Notation

We provide an overview of neuroscience vocabulary and notation used in this dissertation. This vocabulary is intended to help the reader understand our study of novice program comprehension using fNIRS (see Chapter 4). While we cover relevant neuroimaging terminology at a high level, the interested reader can learn more about neuroimaging fundamentals in Baars, Bernard, and Gage [16]. We note that this neuroimaging background is not necessary for readers primarily interested in the bug-fixing tools proposed in Chapter 3 or the impact of external factors discussed in Chapter 5.

Vocabulary. The cerebrum of the human brain is composed of two (largely symmetric) hemispheres, left and right, and four primary lobes: frontal, temporal, parietal, and occipital. Loosely, the *frontal lobe* is at the front of each hemisphere, the *temporal lobe* is on the side of each hemisphere, the *parietal lobe* is at the top of each hemisphere, and the *occipital lobe* is at the back of each hemisphere. Activation is called *bilateral* if both hemispheres are activated and *lateralized* if one hemisphere activates disproportionately. Throughout this dissertation, we will use various schema to refer to locations on the cerebrum's *cortex*, the brain's outer layer of neural tissue. One such schema is *Brodmann Areas*, an anatomical classification system for the cortex [44]. Brodmann areas (BAs) divide the cortex into 52 bilateral regions based on architectural neurological features. Many BAs also have associated neurological functions. For instance, BAs 41 and 42 are associated with auditory processing. We will also sometimes refer to regions by their common names or location on a lobe. Three important regions mentioned in this dissertation are Wernicke's area (left hemisphere, back of the parietal lobe), Broca's area (left hemisphere, lower frontal lobe), and the dorsolateral prefrontal cortex or DLPFC (bilateral in the frontal lobe). Wernicke's and Broca's areas are strongly associated with language functions while the DLPFC is associated with working memory.

Notation. In this dissertation, we will use the neuroimaging notation *Task A* $>$ *Task B* to indicate the contrast between brain activation patterns for two experimental tasks. The results of these contrasts are reported as statistical t-values corresponding to each fNIRS channel. These t-values range from -8 to $+8$; a positive t-value indicates that the specified brain area was *more* active during Task A than Task B, while a negative t-value indicates that the specified brain area was *less* active during Task A than Task B. Values closer to -8 and $+8$ represent stronger activation contrasts between the two tasks, and only areas with significant contrast ($p < 0.01$) after correction for multiple comparisons ($q < 0.05$) are reported. Finally, we note that contrast tests are directional: a significant contrast in *Task A* $>$ *Task B* does not imply that the inverse contrast *Task B* $>$ *Task A* will also produce significant results. This is because the differences in the inverse contrast may be too small to be statistically significant.

2.2.3 Neuroimaging and Software Engineering

Following the pioneering work of Siegmund *et al.* [293], a number of papers have used medical imaging techniques to investigate software engineering activities (*e.g.*, [61, 86, 99, 104, 155, 181, 229, 293, 294]). Studied activities include program comprehension [293], debugging [61], and code writing [181]. Related to our work in particular, Huang *et al.* used fNIRS to compare mental rotation tasks to data structure manipulation [153].

A key distinction of our work is that those studies focus on programmers with years of experience. Explicit investigations of programming expertise using neuroimaging are relatively rare (*cf.* [246]), and tend to involve either proxies such as undergraduate grades [104] or comparisons between graduates or professionals and undergraduates (*e.g.*, Siegmund *et al.* measure 8 students and 3 professionals [294, Sec. 3.3]). Floyd *et al.* found that coding and prose tasks are more similar in terms of neural activity for senior undergraduates than for mid-level undergraduates [104] (*i.e.*, as programmers become more experienced). In section 4.2 of this dissertation, we investigate if *the pattern continues*: that is, for programmers with even less experience (e.g., true novices or first-year students), are programming and reading even less cognitively similar? We discuss neuroimaging work connecting reading ability and spatial ability with programming in more depth in section subsection 2.2.5 and subsection 2.2.7.

2.2.4 Reading and Cognition

Next, we briefly summarize the neurological processes associated with reading. Neuroimaging has revealed that language is supported by a complex network of cognitive areas that are generally lateralized to the left hemisphere (see Price [251] and Vigneau *et al.* [329] for surveys). Some language processes are localized to specific structures in the brain, while other language processes arise from a distributed network of areas with multiple functions [251].

Two left-hemisphere regions associated with reading are Broca's area and Wernicke's area. In the frontal lobe, Broca's area (BAs 44 and 45) plays an important role in language production and, to a lesser extent, language comprehension [329]. Wernicke's area is in the posterior temporal lobe and is associated primarily with comprehension of both spoken and written language [329].

Closely related to general reading ability, *Technical Reading Ability* is a person's ability to read and understand technical or scientific texts [271]. Both reading and technical reading can be predictive of success in various fields. For instance, when investigating elementary school students' facility with math word problems, Vilenius-Tuohimaa *et al.* found that both technical and general reading ability were strongly related to student success [330]. Similarly, reading was found to be the second most important academic area for predicting success in nursing school behind science and ahead of mathematics [346]. Reading is also an essential skill for program comprehension [104, 293]. In the next section (subsection 2.2.5), we discuss connections between reading and programming in more detail.

2.2.5 Technical Reading and Programming

From documentation to code review to code summarization, many software engineering activities involve a significant reading component [134, 213, 350]. Experimentally, several studies report a correlation between programming and reading abilities [53, 200, 224, 249, 292]. For example, Shute found that scores on algebra word problems predicted programming skill acquisition [292]. Others connect programming to reading or tracing through a program and describing its function in natural language [200, 212, 224]. That is, there is a positive correlation between verbally contextualizing programs "in plain English" and programming ability.

In an educational context, Gunawardena *et al.* observed "there may be a stable correlation between students' ability to successfully comprehend and detect errors in a document and their course performance" [128]. Similarly, Hebig *et al.* conducted focus groups with masters' students discussing their perceptions of various software comprehension techniques, finding generally positive views of many reading-related comprehension aids such as systematic code reading, documentation, and analysis dashboards [143]. These lines of preliminary results invite a more controlled study.

Beyond direct connections between programming ability and reading ability, many essential software engineering tasks, such as code review, code summarization, or managing documentation, require technical reading ability. For example, professional software developers use a significant portion of their time on the web reading documentation and debugging forums [350]. Furthermore, software developers often use prose summaries of code, either in the form of documentation or inline comments, to facilitate program comprehension and communication with other programmers [106]. Techniques for automatically generating prose summaries of code reduce time spent writing documentation, keep up with rapidly changing code bases, and aid developer comprehension [134, 213].

Programming and reading also relate on a cognitive level. Siegmund *et al.* identified five brain areas that activate when reading programs, most of which are also associated with language [293]. Similarly, Floyd *et al.* found that while code review and prose review were distinct neurological processes, they became less distinct with more programming experience [104] (see subsection 2.2.3 for more detail on the work by Floyd *et al.*).

Researchers have also explored reading and program comprehension using eye-tracking. For example, Busjahn *et al.* used eye-tracking results to argue for using natural language reading as a basis for understanding code comprehension [53]. Rodeghero *et al.* found that, when possible, programmers use keywords to capture high-level program information rather than reading code details [268]. Similarly, Rodeghero and McMillan found that developers prefer to skim code and follow common reading patterns [267]. Furthermore, Busjahn *et al.* found that developers generally read code non-linearly, a pattern that increases with expertise [52]. These results inform the design of our reading training (see subsubsection 4.3.1.1).

2.2.6 Spatial Reasoning and Cognition

We now turn to a discussion of Spatial Reasoning, its correlates, and its neurological representations. *Spatial reasoning* refers to a person's capacity to understand and reason about spatial relationships among objects, and is a blanket term for skills such as mental rotation, mental folding, spatial perception, and spatial pattern recognition [207]. There is a large body of research on spatial ability and its correlates. Generally, spatial ability correlates with gender and socioeconomic status [60, 332]; on average, men have higher spatial ability than women, and spatial ability and affluence are positively correlated. Spatial ability positively correlates with performance in a variety of fields including mathematics [144, 335], natural sciences [35, 318, 352], and engineering [10]. Spatial ability is malleable and *can be improved through training* [320]. For example, spatial training may reduce, or even eliminate, observed gender gaps in spatial ability [163].

Mental Rotation and Cognition. Part of spatial reasoning, mental rotation involves the imagined rotation of a two- or three-dimensional object around an axis in three-dimensional space [356]. Figure 4.3a, Figure 4.9, and Figure 4.10 contain examples of the various mental rotation stimuli we use in this dissertation. The difficulty of a mental rotation problem is determined by the size of the angle of rotation; Shepard and Metzler found that the time a participant took to solve a given problem increased linearly with respect to the angle of rotation between the two objects [290]. We use mental rotation ability as a validated proxy for more general spatial reasoning ability.

Neuroimaging has found that mental rotation activates the posterior parietal and occipital cortices (BAs, 7, 17, 19, 39, and 40—see Zacks [356] for a survey). While bilateral, the parietal and occipital activation tends to be slightly stronger in the right hemisphere; the right parietal lobe is believed to be important for spatial ability and spatial attention tasks [76]. Many mental rotation neuroimaging studies have also revealed bilateral activation in the supplementary motor cortex, an area associated with motor control and planning (BA 6) [356]. This frontal activation is most common for mental rotation tasks that allow motor stimulation strategies. We use these known connections between spatial reasoning and brain region activation in our analysis of our fNIRS study in Chapter 4.

2.2.7 Spatial Reasoning and Programming

Several studies have found evidence of a medium to strong positive correlation between programming and spatial reasoning ability [103, 161, 241, 243]. For instance, Parkinson and Cutts found that "spatial skills typically increase as the level of academic achievement in computer science increases" [243]. Furthermore, Parker *et al.* found that spatial reasoning is a better mediating variable for affluence discrepancies in computer science than computing access [241].

Researchers have used various methods to connect spatial ability with programming. For example, Huang *et al.* used medical imaging to find that similar parts of the brain are recruited to solve mental rotation problems and tree-based data structure problems, indicating that core software engineering tasks use visuospatial cognitive processes [153]. Additionally, Margulieux proposed "spatial encoding strategy" (SpES), a theory about the cognitive processes behind the transferability between spatial ability and programming [207]. SpES posits that improving spatial ability helps develop general strategies for encoding oriented "mental representations of non-verbal information," strategies that can be applied to programming problems.

There have also recently been studies establishing a causal transfer between spatial reasoning training and computer science performance. We discuss this more in our related work section on *transfer training* in subsection 2.2.8.

2.2.8 Transfer Training

One potential intervention for supporting struggling novices is facilitating *learning transfer* to programming from a concurrently-taught supplemental training course [37], where learning transfer refers to the use of skills in a separate context from where they were learned.

One of the most common cognitive training interventions proposed and validated for various fields (including improving programming) is *spatial ability training* (often in a one-credit class with one hour of additional instruction per week) [37, 71, 302, 320]. In a meta-analysis [320], Uttal *et al.* found that spatial training can transfer to increased performance on a variety of tasks. For example, Pribyl *et al.* found that spatial training improved organic chemistry students' ability to "mentally manipulate two-dimensional representations of a molecule" [250].

In addition, supplemental spatial ability training can increase engineering degree retention [300] and directly improve programming outcomes [37, 71]. For example, Cooper *et al.* ran an exploratory study with 38 high school programmers, finding that those who participated in spatial training performed better on a final programming assessment [71]. Significantly, Bockmon *et al.* replicated the Cooper study at four different universities with a much larger sample size ($n = 345$) [37]. They found that on a final programming assessment, introductory students who participated in a spatial training intervention significantly outperformed students who did not participate. All participation in the spatial intervention was voluntary, leaving open the possibly of self-selection biases affecting their results. Even so, the study provides compelling evidence for the feasibility of spatial training transferring to CS1, and thus software engineering, performance. Thus, we use spatial training as one of our two skills-based transfer training interventions. In our experiment, spatial training functions as a baseline for analyzing the efficacy of our reading treatment. Both Cooper *et al.* and Bockmon *et al.* used spatial training materials based on interventions created by Sorby and Baartmans [301], materials that have also been found to improve performance in general engineering classes [300, 302]. We also use Sorby and Baartmans's materials.

The impact on programming of training other relevant cognitive skills (such as reading) are less studied. However, there is evidence that technical reading training may contribute to improved performance in various fields outside of computer science. For instance, a writing-intensive technical reading course improved biology students' perceived understanding of primary scientific papers and their ability to communicate science [47].

While limited, there are some more direct hints that reading training could transfer to programming. For example, Mayer found that "asking learners to put technical information in their own words (through making comparisons) results in broader learning" that supports transfer to computer science [212]. Furthermore, Fedorenko *et al.* argue that learning to program is akin to learning a language and thus teaching programming can be "informed by pedagogies for developing linguistic fluency" [101]. This hypothesis was supported by a recent study by Prat *et al.* which found that natural language aptitude was a significant factor in predicting programming success [249].

We are not aware of any previous studies directly exploring transfer training between technical reading and programming ability. A main contribution of this dissertation, therefore, is the development and evaluation of the first (to our knowledge) CS-focused technical reading software engineering intervention. A large body of work explores how to teach reading (see Grabe for a summary [118]). Grabe proposes key guidelines for developing reading curricula, including "emphasizing vocabulary learning," "promoting strategic reading," "activating background knowledge in appropriate ways," and developing "intrinsic motivation for reading" [118]. To both increase student motivation and also to deliberately promote transfer, we use these guidelines when developing our own CS-focused technical reading training (see subsubsection 4.3.1.1).

2.3 Lens 3: Impact of External Factors—A Cannabis Case Study

Finally, we present work relevant to our third lens into improving programmer productivity via understanding the impact of external non-technical factors (see Chapter 5). In this lens, we study this impact via a case study on one external factor understudied in the existing literature, psychoactive substance use, with a focus on cannabis in particular. To explore this relationship, we 1) conduct a qualitative interview study and large scale survey to understand the interaction space in more detail and 2) carry out a controlled observational study of one psychoactive substance that we identify as particularly relevant for software engineering productivity, cannabis. We first provide a brief overview of previous efforts to identify how external factors can influence programming productivity in 2.3.1. We then provide background regarding general psychoactive substance use, with a focus on cannabis, in subsection 2.3.2. Finally, we overview existing related work at the intersection of psychoactive substances and programming in subsection 2.3.3.

2.3.1 External Factors and Programming Productivity

Various methods to enhance developer productivity have been proposed and implemented (for more information, see the review by Wagner and Ruhe's systematic overview [334]). However, identifying the most effective productivity factors and interventions for developers remains a challenge in practice [226, 319]. For instance, while many approaches emphasize improved technical support to boost productivity, non-technical factors can sometimes have an equal or greater impact on developer productivity and wellbeing [226]. Furthermore, significant productivity variations among developers have been noted for decades [4, 274], even after accounting for common quantitative and environmental factors [105]. This further indicates that existing techniques may not fully support certain developer groups.

External factors connected to programming productivity include office environment (*e.g.*, distractions and interruptions [203]), world events (*e.g.*, the COVID-19 pandemic [105]), and mental health (*e.g.*, impostor syndrome [126]). For example, the happiness of software developers has been correlated positively with their productivity and quality of their work [120, 164], supporting what is commonly referred to as the "happy-productive" thesis [307, 357]. Beyond this, the *un*happiness of software developers has been identified to have dozens of potential negative outcomes [121]. It remains in a company's best interest to prioritize the happiness of its employees for the best results. Unfortunately, a considerable stigma remains around discussing mental health (*e.g.*, [45, 147]) or neurodivergence issues [222], hindering constructive reform.

While previous studies have established that external or environmental factors such as mental health can impact developer productivity, many features with anecdotal connections remain understudied (such as psychoactive substance use, see subsection 2.3.3). In addition, while many studies identify or correlate potential factors [105, 226], controlled experimentation regarding the magnitude of the impact of non-technical factors is less common (*cf.* Ma *et al.*'s recent investigation of how environmental distractions impact programming productivity [203]). In this dissertation we show how external factors beyond those previously considered can significantly impact programming productivity through a survey and controlled study of one factor currently understudied in the software literature, psychoactive substance use.

2.3.2 Cannabis and General Psychoactive Substance Use

We give an overview of general psychoactive substance use, with a focus on cannabis. This background is assumed for Lens 3 in Chapter 5. Readers not interested in Lens 3 or already familiar with psychoactive substances may skip this section.

A *psychoactive* (or psychotropic) substance influences the brain or nervous system and thus behavior, mood, perception, and thought [344]. Alcohol, caffeine, cannabis, LSD, and nicotine are examples of such substances. Additionally, many medications prescribed for mood disorders (such as depression or anxiety) are psychoactive. Different substances have different cognitive impacts: for example, alcohol suppresses nervous system activity while stimulants increase alertness and focus via dopamine in the brain [234]. Psychoactive drugs have a long history and have impacted multiple aspects of human culture, from recreation to war [333]. Prevalence of use and legality vary by substance and area [260, 304]. In this dissertation, we highlight those substances that are most often associated with programming: cannabis, alcohol, prescription stimulants (*e.g.*, Adderall, Ritalin), mood disorder medications (*e.g.*, SSRIs, Wellbutrin), and psychedelics (*e.g.*, LSD, microdosing). Of these, we focus on cannabis in particular due to its general prevalence and its use in online programming discourse (subsection 2.3.4). Notably, although officially psychoactive, we exclude caffeine due to its near-universal prevalence in software.

Cannabis. *Cannabis sativa* is the world's most common illicit substance [308], used for both medical and recreational purposes [24, 38, 192]. In the US, cannabis is criminalized, hampering research on its effects [230]. However, in the past century, cannabis legality has shifted dramatically in the US. While many cannabis preparations were listed in the US pharmacopeia in the early 20th century, cannabis was officially criminalized under the Controlled Substances Act in 1970. Since 1996, however, 38 states have legalized cannabis for medical purposes, and 23 for adult recreational use. Even so, cannabis remains illegal at the federal level, where as of 2023 it is classified as a Schedule I drug. The policy whiplash has been largely driven by politics and popular opinion rather than science, with the re-emergence of cannabis occurring partially due to: 1) acknowledgment of the cruel excesses and ineffectiveness of the War on Drugs [89, 347]; 2) increasing acknowledgement of cannabis's therapeutic benefits [230]; and 3) the ongoing opioid crisis, which has driven home the need for alternative, less harmful pain medicines [65]. In conjunction with liberalizing policies, past-year cannabis use has increased among Americans 12 or older: 11% in 2002 to 17.5% in 2019 [135]. In addition, its prohibition is contrary to popular opinion [168, 322].

Cannabis is used for many reasons, including medical (*e.g.*, pain, nausea) and recreational (*e.g.*, social or perceptual enhancement, altered consciousness) purposes [24, 40, 192]. This wide variety of uses is enabled by the pharmacopeia of compounds in cannabis, which include more than 100 active cannabis-derived compounds (cannabinoids) such as cannabidiol (CBD) and Δ-9-tetrahydrocannabinol (THC) as well as terpenes and flavonoids, which are responsible for taste and odor [230]. The most common medical reason for cannabis use is chronic pain, demonstrated by patient surveys [24, 263] and registry data for medical cannabis patients [38]. Within broad categories of medical vs. recreational use, however, there is a subtle shading of motivations. Lee *et al.* developed the comprehensive marijuana motives questionnaire, in which they identified factors associated with cannabis use, including enjoyment, coping, experimentation, altered perception, sleep/rest, and availability [192]. These factors are broad and may encompass recreational, medical, or other use patterns such as for creative work or performance enhancement.

Cannabis is well-known for its mind-altering effects. In particular, acute cannabis use (especially for non-heavy users) can impair various cognitive processes such as memory and learning, attention control, fine motor control, and emotion processing [180]. The use of cannabis in medical contexts (*e.g.*, to treat chronic pain) has been examined in the literature, including via observational studies [41]; in this dissertation, we focus on cognitive aspects such as memory, fine motor control, decision-making, and creativity [108] that are related to programming [293] (see subsection 2.3.4).

2.3.3 Psychoactive Substances and Programming

We now discuss related work at the intersection of psychoactive substance use and programming. Anecdotes and accounts connecting psychoactive substances and programming abound [167, 209, 337]. Psychedelics, such as LSD, have been associated with early software development [209], with folk wisdom suggesting positive creativity benefits [337]. Specifically, LSD was commonly reported by many early personal computer (PC) and internet developers as a source of innovation and creativity [337], a phenomenon likely connected to 1960s counterculture [209]. This connection continues to this day: popular media have reported that Silicon Valley culture includes using stimulants or other "smart drugs" (*e.g.*, nootropics) to increase productivity [167]. Similar creativity benefits have been suggested for alcohol; Jarosz *et al.* found that intoxicated subjects completed more creative problem-solving tasks than sober counterparts [156]. However, despite preliminary evidence that micro-dosing on psychedelics may increase creativity [253], connections between creativity and programming generally remain cultural and anecdotal rather than causal.

As for explicit research on the intersection of psychoactive substances and software, the limited prior work mostly focuses on alcohol [43, 80]. In a study of IT professionals in India, Darshan *et al.* linked high levels of work-related stress to increased rates of alcohol abuse and depression [80]. In addition, Brabrand found that alcohol impairs programming performance for novices at a level that is comparable to heat [43].

To the best of our knowledge, as of 2023, only one formal work has studied the intersection of software and general psychoactive substance use. In 2023, Newman *et al.* conducted a qualitative study, where they interviewed 25 professional programmers who used psychoactive substances, finding connections between substance use and software from the individual developer to organizational policy [232].[1] This work highlights the importance of future scientific study at the intersection of psychoactive substances and programming.

2.3.4 Cannabis and Programming

As with studies on psychoactive substance use and programming as a whole, we are not aware of previous studies focusing on programming and cannabis use specifically. Reports either combine cannabis with other illegal substances or include programmers only as a sub-population [159, 218]. For instance, one study has a table of the percentage of workers in various fields reporting illegal drug use [159, Tab 3.5, p. 26]. Programmers appear in the table, with 10.4% reporting usage in the last year. However, this is the only mention of programmers, cannabis is not separated from other drugs, and the data is from the early 1990s, well before cannabis was legalized in numerous states. Thus, questions of cannabis prevalence, usage motivations, and culture in software engineering remain unanswered.

Despite the paucity of empirical research, anecdotes of cannabis use while programming abound. Questions inquiring about cannabis's effects on programming are common on online forums,[23] often inspiring numerous conflicting answers. In their interview study with 25 professional programmers, Newman *et al* reported positive views on the impact of cannabis on brainstorming, neutral views on coding and testing, and negative views on debugging, design, and documentation [232, Tab. III]. Popular tech-related media sites cover the topic, positing that cannabis may help with programming-related chronic pain [174] or enhance programming through increasing focus [32] and creativity [338]. Cannabis use in Silicon Valley appears to be high, with area dispensaries reporting that around 40% of their clientele are tech workers [326]. Similarly, a qualitative study of coding bootcamps identified "lots and lots of [weed]" as one key element of support [56]. There is even evidence that cannabis use can impact software hiring and retention [64, 166, 193].

Physiologically, cannabis has a range of cognitive effects that could impact software development. Specifically, acute cannabis use impairs memory and learning as well as inhibits motor responses

[1] We note that the author of this dissertation was a co-author on the interview study by Newman *et al.*

[2] https://www.reddit.com/r/computerscience/comments/dbzp5v/how_many_of_you_found_that_smoking_weed_gets_you/

[3] https://news.ycombinator.com/item?id=509614

and reaction times [49, 180]. These cognitive processes are used while programming (*e.g.*, during code comprehension tasks [293] or while typing code [181]). The impact of cannabis on other programming-related cognitive processes is less understood. Recent reviews report insufficient evidence of how cannabis impairs working memory or decision making [49, 180], both essential to software [74, 245, 293]. Similarly, cannabis's impact on creativity is the subject of contradictory claims: for example, LaFrance *et al.* found that cannabis users exhibit higher creativity due to increased openness to experiences [184] while Kowal *et al.* found that cannabis impairs aspects of creativity at high dosages [178]. These conflicting claims preclude using a "bottom-up" approach to infer the impact of cannabis on programming by considering its impact on relevant cognitive processes. In this dissertation, we instead use a "top-down" approach, studying the relationship between cannabis use and programming performance directly in a real-world context.

Cannabis use has also been studied in the context of creative problem solving, a key component of many software engineering tasks [123, 124, 204]. Generally, cannabis is seen as a creativity enhancer: in one study, 54% of participants believed it increased their creativity [122]. Similarly, Steve Jobs, who was regularly using cannabis when he founded Apple, stated that "the best way [he] would describe the effect of the marijuana . . . is that it would make [him] relaxed and creative" [359]. From a cognitive perspective, the link between creativity and cannabis use is more tenuous. LaFrance *et al.* found that cannabis users were more creative than non-cannabis users, however, this difference may be explained by underlying personality [184]. On the other hand, Kowal *et al.* found that highly potent cannabis can actually *impair* divergent thought, a type of creative thinking [178]. These conflicting results, along with the potential cognitive and physiological impairment of cannabis on programming motivate our observational study in Chapter 5.

CHAPTER 3

Supporting Non-Traditional Novices Through Bug-Fixing Tools

In our first lens into improving programmer productivity, we focus on designing efficient bug fixing algorithms to help non-traditional novice programmers write more correct code faster. A key part of the software development process, debugging can be a time-consuming and challenging task [28]. As a result, effective support that helps developers debug faster has the potential to greatly increase programming productivity [189].

Building and fixing software are highly trained activities: as challenging as debugging is for experts, for novices it can be even more so [225]. With the increasing number of students pursuing computer science degrees [360] along with the increasing number of non-traditional novices turning to online resources beyond the traditional classroom to learn computing [214, 288], there is a significant need for effective educational support. But even as demand soars for such resources, the educational support provided by online tools leaves much room for improvement [42], especially for those students who need the most help [87]. Free tutoring environments seek to close this gap by providing educational support beyond structured course assignments. However, such sites can still suffer from low retention, reducing their ability to help students in practice. We hypothesize that one reason for this low retention is the frustration novices face without instructional support; the time spent debugging a single error has been shown to correlate with student frustration [269].

Automatic bug-fixing techniques such as Automatic Program Repair (see subsection 2.1.2) and Error-Correcting Parsers (see subsection 2.1.4) have the potential to help non-traditional novices effectively and efficiently fix frustrating program errors. However, there is relatively limited sustained research into using source-level automatic program repair and fault localization techniques for pedagogical purposes as of 2023 (see subsection 2.1.2). At the same time, heavyweight expert-focused automatic program repair tools and their derivatives, such as GenProg [189] and Angelix [217], can be confusing for novices [354]. In addition, such techniques may not target errors commonly encountered by non-traditional novices, as evidenced by the separate benchmarks for and performance of APR tools for each context [190, 289].

3.1 Lens 1: Overview

In this dissertation, we identify and develop automated support for two error types that we find are commonly encountered by non-traditional novices: input-related bugs and parse errors. We identify these error types through an empirical evaluation of PYTHON TUTOR programming interactions; PYTHON TUTOR is a free online programming tutoring environment that is often used by novices learning programming without traditional classroom support [129, 130] (see subsection 2.1.1 for more detail). Through this evaluation, we find that input-related errors and parse errors are both common barriers for non-traditional novices. For example, we find that 35% of PYTHON TUTOR interactions involve user input, with 6.6% of interpreter errors fixed by only modifying input data (see subsection 3.2.3). Similarly, we find that 77.4% of all faulty PYTHON TUTOR programs failed with a parse error, accounting for the vast majority of the errors that novices face (see subsection 3.3.1). Both of these error types are not well supported by extant Automatic Program Repair tools, which typically operate only on program source code (rather than inputs) and require the code to parse (see subsection 2.1.2 for more detail).

While identifying factors non-traditional novices struggle with is important, we also wish to help improve novice productivity in practice. In pursuit of this goal, we present two automatic bug-fixing tools, one that targets program inputs rather than standard code and another that targets parse errors.

For input-related bugs, we present INFIX, a randomized search optimization algorithm for automatically repairing program inputs for generic novice programs. Our key insight is that input repairs are often composed of a small number of common mutations and that are also often heavily correlated to specific error messages (such as those visible on PYTHON TUTOR, see subsection 2.1.1). As a result, we use an iterative approach where we repeatedly modify the buggy input according to error-message-specific mutation templates until it either finds a repaired input or times out. INFIX does not require test cases or training data to assess correctness. Instead, we use the implicit specification of eliminating interpreter errors (as tested by running the program after each input mutation). This, along with permitting a pleasingly parallel implementation, enables INFIX to provide efficient real-time programming support. To the best of our knowledge, we are the first to extend APR techniques to support input-related bugs (see Monperrus for a review [221]).

For parse errors, we propose SEQ2PARSE, a language-agnostic approach that combines the theoretical accuracy of *Error Correcting Earley Parsers* (ECE-Parsers, see subsection 2.1.4) with the efficiency of *Sequence to Sequence Transformers* (seq2seq, see subsection 2.1.5) to quickly generate high-quality fixes. In theory, ECE-Parsers can generate minimal-edit parse error repairs using special error production rules to handle programs with syntax errors. Sadly, ECE-Parsers (along with more modern equivalents such as [50, 72, 311, 321]) remain inefficient in practice, as their running time is cubic in the program size and quadratic in the size of the language's grammar [216, 257]. Our key insight is that novice edits are highly predictable, and thus Neural approaches can pinpoint the relevant rules. Thus, we propose addressing parse error repair via a neurosymbolic approach where we first train sequence classifiers to predict the relevant EC-rules for a given program, and then use those predicted rules to synthesize repairs via EC-Parsing. To the best of our knowledge, we are the first to propose augmenting ECE-Parsers with Neural methods.

For both tools, we evaluate a Python-supporting implementation on data from PYTHON TUTOR. As both tools are aimed at helping novices debug, we argue that it is essential that repairs are helpful and of high quality. Previous work has shown that student-generated source repairs and expert human tutor input hints can be helpful for students by decreasing debugging time and increasing learning [265]. Therefore, we augment our automatic evaluation with human studies comparing tool-made repairs to those developed by the PYTHON TUTOR users themselves.

In the rest of this chapter, we first characterize novices' input-related errors and present our repair approach, INFIX, in section 3.2. We then characterize novices' parse errors and introduce our repair approach, SEQ2PARSE, in section 3.3. Finally, we summarize Lens 1 in section 3.4.

3.2 INFIX: Automatically Repairing Novice Program Inputs

To help support non-traditional novice programmer productivity, we first focus on providing support for *input-related errors*. We define an input-related error as a mistake in the input to a program, rather than in the program source code (*e.g.*, ill-formatted data, string misspelling, *etc.*).

From reviewing over six million programs submitted to PYTHON TUTOR [130] (see subsection 2.1.1 for more about PYTHON TUTOR), we identify input-related errors as common and time-consuming to fix for non-traditional novices. As described in more detail in subsection 3.2.3, we observe that 35% of novice interactions involve both user input and source code. We also find that PYTHON TUTOR users fixed 6.6% of interpreter errors by only modifying input data. Although student errors extend from simple syntactic mistakes to more involved semantic errors, we observe that a surprisingly large portion are input-related (*e.g.*, entering `1,2` instead of `1.2`). Therefore, we make providing automatic support for input-related errors one of this dissertation's two focus areas for helping novices with rapid debugging hints.

We argue that a technological solution to effective novice input repair should find repairs quickly to fit into the student's workflow (*cf.* [336]) and actually be helpful for novices. In our characterization of novice inputs (see subsection 3.2.3), we observe that novice input-repairs are generally short, and thus propose to use algorithms that explore the search space of nearby edits. We also note that learners' errors are not uniformly distributed, and many student programs that show defects with the same error message can be fixed using the same mutation. Therefore, a small number of indicative templates can increase search speed. Finally, we note that the structure of student inputs can be unexpectedly complex; programs often contain interdependent input values, making a randomized approach surprisingly effective.

Based on these insights, this dissertation presents InFix, a randomized search algorithm for automatically repairing program inputs for generic novice programs. InFix iteratively searches for input repairs using language-specific error message templates combined with additional randomized mutations. InFix does not require test cases or large amounts of training data, instead using the implicit specification of eliminating interpreter errors [140, 286, 340]. InFix also admits a pleasingly parallelizable implementation, improving efficiency.

We evaluate a Python implementation[1] of InFix. Our error message and mutation templates are developed from our observational study characterizing novice Python input patterns and errors. In total, we abstract five error message templates and five additional simple mutations. Each error message template corresponds to a single interpreter message. We evaluate our implementation on 25,995 input-related scenarios arising from 22,282 unique programs from four years of Python Tutor data (subsection 3.2.5). Each scenario contains a program and input that, when run, generates an input-related error. Overall, we find that with just five error message templates and five mutations, InFix repairs 94.5% of input-related scenarios.

As InFix is designed to help novices debug, the repairs must be helpful and of high quality. Previous work has shown that student-generated source repairs and expert human tutor input hints can help students, decreasing debugging time and increasing learning [140, 219]. Therefore, we compare InFix's repairs to those of the learners themselves. From a human study involving 97 participants, we find that InFix produces high-quality repairs. Specifically, participants find InFix's repairs as helpful as human repairs and within 4% of their quality in a statistically significant manner. We conduct this study with both undergraduates and crowdsourced Amazon Mechanical Turk workers.[2] We also find that 80% of our participants often experience input-related errors in their programming. In summary, the contributions of our research into input-related errors include:

- InFix, a novel template-based search algorithm for repairing buggy inputs to novice programs
- A characterization of common novice input patterns and input repairs for Python programs
- A Python implementation of InFix based on our characterization that fixes 94.5% of 25,995 input-related errors, in one second each on average
- The results of an IRB-approved human study with 97 participants indicating that InFix's repairs are within 4% the quality of, and equally helpful as, student repairs

[1] Source code can be found at `https://github.com/CelloCorgi/InputFixer`

[2] Crowdsourced participants from sites such as Amazon Mechanical Turk or Prolific are common in programming studies [306], though we note that it is essential to use additional checks to ensure data quality and that participants have sufficient programming background.

Program Code:

```
x = float(input())
print(x * math.e / 2)
```

Erroneous Input	Student Repair	InFix Repair
`26,2`	`29.2`	`4.5`

Python Error Message:

```
ValueError: could not convert string to float: '26,2'
```

Figure 3.1: Syntactic Input-related Error: Example of a syntactic input-related error.

3.2.1 Input-related Errors: Motivating Examples

To motivate automatic input repair, we present two Python scenarios with input-related errors. We adopt both examples from actual student programs submitted to Python Tutor. These examples demonstrate the difference between syntactic and semantic input-related errors, motivating InFix's hierarchical use of error message templates and random mutations.

The scenario in Figure 3.1 exemplifies a syntactic input-related error explainable as a misunderstanding of Python language behavior. The program in this example accepts a float from the user and carries out a calculation based on this input value. Python's `input()` call accepts floats using period-based decimal notation. However, the student entered `26,2` into the interpreter using a comma instead of a period. Because of this, the Python interpreter is unable to cast the input to a float and throws a `ValueError`. Notice that in this simple case, the input that needs to be modified to fix this error is included in the error message itself. Specifically, the error message points out that the program accepts a float, but that `26,2` is not a float. While this error may appear trivial to fix for expert programmers, novices can have surprising difficulty debugging even simple errors [83]. In fact, novices may not even read error messages in some cases [205, Sec. 3], using their existence as a boolean indicator of success or failure.

When designing InFix, we therefore use error-message templates to take advantage of the copious information some messages include. In particular, we observe that error messages associated with syntactic mistakes contain the richest information for algorithmic repairs. Our implementation includes a template that specifically addresses `ValueErrors` like those in Figure 3.1. For this scenario, our template is effective: participants find InFix's repair equal in quality and helpfulness to the human-generated repair.

Despite the applicability of error message templates to simple errors, many input-related errors are semantic and program-specific rather than syntactic. Figure 3.2 exemplifies one such semantic error. The program uses the first two inputs to build a dictionary that is then accessed using

Program Code:

```
input_a = input()
input_b = input()
input_c = input()
c_array = []
dictionary = {}
for i in range(len(input_a)):
    dictionary[input_a[i]] = input_b[i]
for j in range(len(input_c)):
    c_array += dictionary[input_c[j]]
print(c_array)
```

Erroneous Input	Student Repair	InFix Repair
abcd	abcd	-Et
*d%#	badc	abcd
#*%*d*%	abcd	-Et

Python Error Message:

```
KeyError: '#'
```

Figure 3.2: Semantic Input-related Error: Example of a semantic input-related error.

the characters in the third input. Unfortunately, the novice includes incorrect values in the third input line, resulting in a KeyError. Specifically, the student flips the lines corresponding to the keys and the values, perhaps indicating a misunderstanding of Python dictionaries. This error is time-consuming, taking the student almost three minutes to fix.

This error is significantly more complex than the error from the first example: any repaired input for Figure 3.2 must satisfy several program-specific constraints. First, input_b must have at least as many characters as input_a. Second, all characters in input_c must also be in input_a.

Furthermore, notice that unlike the example in Figure 3.1, the error message in Figure 3.2 is not a rich source of debugging hints. However, we observe that repairs similar to the student generated-repair could be created by simple mutations of the original error-generating input. As a result of these observations, we propose using simple mutations to repair erroneous input data when there is not enough information in the error message alone. For these mutations, we consider the input as a whitespace-separated list of tokens. In our evaluation, InFix finds a solution to the scenario in Figure 3.2 in under 2 seconds, and in our human study, we find InFix's repair to be equivalent in quality and helpfulness to the student's repair.

From our observations, these two examples are indicative of the majority of input-related errors encountered by novices in online tutoring environments. For syntactic errors, such as the error in Figure 3.1, we observe that novices tend to repair the same error in similar ways and that error messages are rich sources of information. For semantic errors, like the one in Figure 3.2, fixes are more varied and the error messages are more opaque. These observations motivate our decision to include both error message templates and mutations in INFIX; we use error message templates to quickly repair the most common errors and random mutations to address more complex semantic errors. Our observations also inspire our decision to prioritize one category above the other: we only use mutations when there is no applicable error message template.

3.2.2 INFIX Algorithm

INFIX is a randomized search optimization algorithm that iteratively modifies the original erroneous input until either a correct input is found or the maximum number of probes has been exhausted. The key insights behind InFix are that input repairs are often composed from a small number of common mutations and that these mutations are often heavily correlated to specific error messages. Furthermore, for some simple specific errors, student edits are highly predictable.

Since we target generic novice programs, we define an erroneous input as an input that causes a program to raise an interpreter error. In contrast, we define a correct input as one that causes the program to terminate without error. A user-interaction scenario has an input-related error if the programmer runs both erroneous and correct inputs with the same program (not every error-avoiding input needs to be helpful to novices; we formally investigate repair quality via a human study in subsection 3.2.9). Given two inputs for one program such that the first is erroneous and the second is correct, we refer to the latter as a fix or repair. We now describe the INFIX algorithm, present information on template selection, and describe a parallel version of INFIX.

INFIX *Algorithm Architecture.* At a high level, INFIX takes six arguments: the program P, the erroneous input I, the original error message M, a template function T, a set of mutations R, and a maximum number of probes N. The template function T's domain consists of a finite set of error messages that uniquely determine a corresponding input mutation. R is a set of additional input mutations. INFIX iteratively mutates I until it either finds a repaired input or has tried N mutations. When successful, INFIX returns a correct input I', such that $P(I')$ terminates normally without raising any exceptions. The pseudocode for INFIX can be found in Algorithm 3.1.

During each iteration, INFIX mutates I using either the error message template function T or a random mutation from R. These two sources of modification are hierarchical: INFIX always applies an error message template if possible (line 15). However, if there is no transformer associated with the error message (*i.e.*, M is not in the domain of T: line 14) or if the resulting mutation has already been considered, a random mutation is applied instead. This mutation is chosen unweighted from a set of mutation templates (line 17).

The program P is then invoked on the modified input (line 20). If the run is error-free, then the input is minimized and the process terminates. Otherwise, INFIX continues iterating until

Algorithm 3.1 Main InFix Algorithm

```
Type Definitions:
Type TokSeq : Sequence of Tokens
Type Mutation : TokSeq $\rightarrow$ TokSeq
Type Message : Error Message
Require:
Program $P$ : Prog
Original erroneous input $I$ : TokSeq
Error message $M$ : Message
Error message template function $T$ : Message $\rightarrow$ Mutation
Set of mutations $R$ : Mutation Set
Maximum number of probes $N$ : $\mathbb{N}$

procedure InFix ($P$, $I$, $M$, $T$, $R$, $N$)
    $V \leftarrow \emptyset$                                  ▷ $V$ is a set of visited inputs
    for $n$ in [1...$N$] do
        if $(M \in domain(T)) \wedge (I \notin V)$ then
            $mut \leftarrow T(M)$
        else
            $mut \leftarrow choose(R)$
        $V \leftarrow V \cup \{I\}$
        $I \leftarrow mut(I)$
        $M \leftarrow run(P, I)$
        if $M$ is GOOD then break
return $minimize(I)$ if $M$ is GOOD else TIMEOUT
```

the probe budget has been exhausted. Any minimization approach that finds a correct fix is acceptable. Previous program repair algorithms have used various minimization methods, such as Delta Debugging [189, 358]. In subsection 3.2.6, we discuss the minimization method in our Python instantiation.

InFix *Template Selection.* InFix relies on the selection of templates T and mutations R. In practice, the domain of T is a small set of the most common error messages. Note that in the Algorithm 3.1 there is a single transformation associated with each error message. InFix thus works best when it is possible to identify a highly effective transformation for the most frequent messages.

However, there are instances where input-related errors that yield the same error message cannot be fixed with the same template. Therefore, R should contain error-independent transformations associated with common student mutations. For our Python implementation of InFix, the transformations in T and R were developed by characterizing novice Python input-related errors, the results of which are discussed in subsection 3.2.3. The specific templates and mutations of our implementation are described in subsection 3.2.4.

***Parallelizing* InFix.** InFix's structure is pleasingly parallel. Running multiple searches in parallel can both decrease the time to the first repair and also increase the likelihood of finding a repair. In InFix's parallel form, each thread runs the main InFix loop. However, as with similar parallel repair algorithms, there is a tension between possibly repeating work on independent parallel threads and incurring overhead caused by coordination [282].

We propose an approach without online coordination: instead, each thread is seeded with a random mutation of the original erroneous input. This increases the likelihood, but does not guarantee, that threads explore different areas of the search space. Our low-overhead approach is critical to finding repairs online in the timescales associated with novice interactions; we empirically evaluate InFix's sensitivity to the number of threads in subsection 3.2.8.

If multiple threads find repairs in the same iteration, InFix selects a repair with the highest statement coverage. We note that parallelization is especially helpful for finding repairs that rely on random mutations rather than on error templates.

3.2.3 Characterization of Novice Input Errors in Python

In this section, we present qualitative and quantitative analyses of real input-related errors and their associated novice interactions. We consider 6,949 input-related scenarios, each consisting of a program, an erroneous input, and a student-generated input repair as defined in section 3.2. These errors make up approximately 6.6% of all erroneous interactions on Python Tutor [130] from Jan-1-2017 to Dec-31-2017. The results of this study inform our adaptation of InFix for Python (see subsection 3.2.4). We restrict our analysis to just one year, 2017, to mitigate overfitting: specifically, we will show that the insights, and ultimately, templates derived from 2017 yield an input repair strategy that generalizes across all the years from 2015 to 2018.

We first present a general analysis of all 6,949 scenarios to understand better the size of the programs, the inputs, and the messages most commonly associated with novice input-related errors. We then present observations from a more in-depth manual examination of 100 randomly selected scenarios, permitting a rich understanding of both the structure of erroneous inputs and also the repairs students made to fix them. Overall, we find that input-related errors are varied and that novice input patterns can be complex. We also demonstrate that some error messages are significantly more common than others and that similar errors are often fixed in similar ways.

Erroneous Input-Related Scenarios: Quantitative Analysis. To characterize the 6,949 scenarios in our data set, we consider the average input and program size, the number of `input` calls per program, the time it takes users to generate repairs, and the prevalence of specific error messages. In these scenarios, there are 6,017 unique programs.

Generally, Python Tutor programs with input-related errors are small; the average program length (excluding blank lines and comments) is 17.1 lines. There is a large range, however: some instances have up to 151 lines. The average input size also varies, ranging from 1 to 5,238 characters with an average of 14.2. These inputs can typically be interpreted as white-space separated token lists, though some use custom delimiters. On average, erroneous inputs have 3.1 tokens, although we observe examples with up to 500.

On average, there are only 1.8 textual `input` calls per program (although we observe programs with up to 20 calls). This small number of calls, however, does not necessarily lead to programs with simple input structures. Loops and type constraints often complicate `input` calls. Befitting their varying complexities, input-related errors take a wide range of time for PYTHON TUTOR users to resolve. The median time to solve input-related errors is 49 seconds. However, 34.6% take users over two minutes to solve and 5.5% take over seven minutes.

The most common error message associated with input-related errors is `ValueError`. Out of the 6,949 scenarios, `ValueErrors` are generated in 3,785 (or 54.5%). The next most common error messages are `IndexError` (1290 scenarios) and `NameError` (778 scenarios). We also note that `ValueError` has several variations, differentiated by the trailing error message text. The most common of these are `invalid literal for int() with base 10` (2,392 scenarios), `could not convert string to float` (599 scenarios), and `not enough/too many values to unpack` (632 scenarios). Together, these `ValueError` variations account for 3,623 scenarios (52.1%).

Erroneous Input-Related Scenarios: Qualitative Analysis. We now present a qualitative analysis of 100 randomly sampled erroneous input-related scenarios. Generally, we find that inputs consist almost exclusively of string, integer, and float literals. We also note that integers and floats in erroneous inputs are typically small and positive; of the 77 inputs in this sample with numerical literals, only 12 (15%) of them involve a number above 15 and only one contains a negative. This pattern is similar in the student-generated fixes: only 25/97 number-containing fixes have a number over 15. We also note that in our sample, more fixes (97/100) contain numerical values than erroneous inputs do (77/100). Therefore, we conclude that many errors are caused by omitting a numerical literal required by the program's input structure. We further observe that such errors are typically resolved through the insertion of a number into the erroneous input.

Subjectively, we note that many erroneous inputs likely imply a misunderstanding of Python data formatting. These syntactic errors fall into two main groups, erroneous string-to-type conversions and mistakes involving library functions. Altogether, these account for 33/100 errors in our sample.

The first group involves misunderstanding how literals are represented as strings. For example, we observe novices who use commas instead of periods for decimal notation or who include quotes around numbers. Our data set contains one extreme example where the student entered `math.pi/6` to a program expecting an integer. These errors typically result in a `ValueError`, and students typically fix them by correcting the format. While students often preserve the original numerical value in the repair, this is not always the case (see Figure 3.1). This indicates that the fix's exact numerical value is not always as important as correcting the formatting.

The second group of syntactic errors involves misunderstanding Python's `input` and `split` behavior. By default, `input` reads until the next newline, and `split` breaks strings on whitespace. Some students, however, may be unclear on both default behaviors. For example, we observe student attempts to use comma-separated lists or to include the data for multiple `input` calls on the same line.

The remaining errors in our sample are semantic. These errors are diverse, ranging from simple swapped input orderings to complicated indirections. For example, one program that accepts a list of integers raises an error if there is a duplicate in the list. Student input fixes are similarly diverse. However, we note that elements in the erroneous input are often permuted in the fixed input. We also notice that other elements are often slightly modified in the fix (for example, `abcb` to `abdb`). We further observe student fixes that insert strings occurring as literals in the program's source code. For example, one program contains a dictionary with the hard-coded key `"pollution"`. However, the student entered the misspelled `polution`, inducing a `KeyError`.

Intriguingly, we find that, despite a small number of static `input` calls in a given program, exhaustively specifying the set of valid inputs can be challenging. While 42/100 scenario programs only have one static `input` call, in 17 (40.5%), that call is embedded in either a loop or a `split`, resulting in a complex dynamic input structure.

We also observe that the length of one input portion often depends on the value of a different input portion. We consider two indicative cases, dependent list lengths and sentinel values. Figure 3.3a shows a Python program where the number of times the second `input` is triggered depends on the value of the first `input`. We find a version of this value dependence in 21/100 scenarios. Figure 3.3b depicts a sentinel loop pattern where exiting the loop requires a specific input value. We observe instantiations of pattern b in 10% of sampled programs. We further observe that errors involving these patterns usually relate to value interdependencies. For example, many `IndexErrors` were caused when one input was used as an index into a structure created by another input. Students typically resolve these errors by inserting or deleting tokens.

Finally, we observe that student fixes for input-related Python errors are generative as well as corrective. As Python is interpreted, no more input is accepted once an uncaught error is thrown. Therefore, when a program with multiple `input` calls aborts after the first call, a fix must both correct this first error and also generate additional input data for any remaining calls. Of the 100 scenarios we examine, 31 fixes generate additional input. We sometimes see novices submit multiple erroneous inputs before achieving an error-free program run, indicating that debugging for novices is an iterative process. An example of this pattern is shown in Figure 3.3c.

```
num = int(input())
val = [int(x) for x
    in
    input().split()]
for y in len(num):
    print(val[y])
```

(a) Example where the length of the second input depends on the value of the first: list `val` must be at least as long as `num`. *e.g.*, this program accepts `2\n1 2` and not `2\n1`.

```
while True:
  value = int(
      input("Num?"))
  if value == 10:
    print("Found!")
    break
```

(b) Example of a sentinel input pattern. To terminate, the input must be exactly equal to 10.

```
name = input ("Name?")
age =
    float(input("Age?"))
num = int(input("Fave
    num4?"))
```

Try	Input	Error
1	`Bob 1,21`	`ValueError`
2	`Bob 9.5 3.4`	`ValueError`
3	`Bob 21 345`	No Error

(c) Example of an iterative debugging process. Note: for space, `input` shown is whitespace-separated.

Figure 3.3: Novice Programmer Input Patterns: Common input and debugging patterns for novices.

Overall, novice inputs and input-related errors are varied and complex. Input-related errors range from simple syntactic mistakes to more insidious semantic errors that can expose tricky interdependencies between input values and calls. Despite this variety, we note that fixes often contain similar mutations of the original erroneous input. We also find that, for simple syntactic errors that result in a `ValueError`, the error message is highly predictive of the eventual fix.

3.2.4 InFix: Python Implementation

To create Python-specific error message templates and mutations for InFix, we exploit patterns discovered in our qualitative study. We design error message templates to quickly target syntactic errors and select mutations inspired by common debugging patterns for semantic errors. In total, our InFix implementation contains five error message-based templates and five mutation operators.

InFix *for Python: Error Message Templates.* In our study of common input patterns (see subsection 3.2.3), we discover that just four subtypes of ValueError account for 52.1% of all input-related errors encountered by novices on Python Tutor. We further find that many of these ValueErrors correspond to syntactic errors such as incorrect Python type formatting or misunderstood input and split behavior, noting that students fix these errors in predictable ways. Additionally, we realize that novice fixes for input-related errors are often necessarily generative. Based on these observations, we implement five error message templates: four that address the most common ValueError subtypes and one that addresses EOFErrors generated when the program runs out of input. Table 3.1 shows our error message templates.

InFix *for Python: Mutation Templates.* While our error message templates directly address a significant portion of syntactic errors, they do not cover all errors. In our observational study, we

Error Message	Template Fix
VALUEERROR: INVALID LITERAL FOR INT() WITH BASE 10: 'X'	Replace last instance of X with random integer between -1 and 10. Note: novices mostly use small numbers in their fixes.
VALUEERROR: COULD NOT CONVERT STRING TO FLOAT: 'X'	Replace last instance of X with random float between -1.00 and 10.00. Note: novices mostly use small numbers in their fixes.
VALUEERROR: NOT ENOUGH VALUES TO UNPACK	Append duplicate of last token. Deliminator is either whitespace or extracted from code.
VALUEERROR: TOO MANY VALUES TO UNPACK	Remove last token from input. Deliminator is either whitespace or extracted from code.
EOFERROR: EOF WHEN READING A LINE	Append a duplicate of a random token from the original input or append a new random three-character string.

Mutation	Description
Insert a token	Insert token at random location. The new token is a short random string, a token from the original input, or a string literal from the source code.
Split whitespace list	Transform a line separated by spaces (or other SPLIT deliminator from source code) into content separated by newlines.
Swap a token	Swap a random token with either a short random string, a token from the original input, or a string literal from the source code.
Remove a token	Delete a random token. A token may be an entire line, but we consider whitespace separation when applicable.
Empty the input	Replace the entire input with an empty sequence. Useful for unhelpful student-provided initial inputs.

Table 3.1: Python Error Message Templates and Input Mutations: Descriptions of error message templates and mutations in INFIX's Python implementation.

found that the student fixes could often be generated by applying a small set of mutations. From this finding, we propose four mutations: inserting a new token, transforming a space-separated list into one separated by newlines, reordering tokens, and deleting tokens. While many fixes are similar to the original input, we also observed instances where they were seemingly unrelated. As a result, we create a fifth mutation that clears the given input. Table 3.1 summarizes the mutations in our INFIX implementation.

3.2.5 INFIX Evaluation Setup

This section contains our experimental setup, including our research questions, an overview of the PYTHON TUTOR Dataset, and our human study methodology.

Research Questions. We focus on five research questions:

- *RQ1*: How effective is INFIX at repairing Python input-related errors?

- *RQ2*: Are the assumptions behind InFix, such as the benefits of the hierarchical ordering between error messages and templates, valid?

- *RQ3*: How sensitive is InFix to available resources?

- *RQ4*: What are the quality (*e.g.*, correctness) and helpfulness (*e.g.*, usefulness in debugging) of the repairs produced by InFix, as subjectively judged by humans?

- *RQ5*: How do the perceived quality and helpfulness of InFix's repairs vary with programmer expertise?

Benchmark 1:* Python Tutor *Data Set. Our first evaluation benchmark consists of 25,995 erroneous input-related scenarios collected from four years of Python Tutor data (Jan-1-2015 to Dec-31-201). Each scenario consists of a Python program, an error-causing input, and a student-generated repair (see subsection 2.1.1). By year, there are 1,640 scenarios from 2015, 4,683 from 2016, 6,949 from 2017, and 12,723 from 2018. This data is an expansion of the data used in our qualitative study described in subsection 3.2.3. To mitigate overfitting, we only used a subset of the data earlier. Across these scenarios, there are 22,282 submissions from 13,968 unique IP addresses. We find that 69.8% of users are single-time users, only ever submitting one input-related error to Python Tutor.

Benchmark 2: Repair Quality Human Study. Our second data source is an IRB-approved human study with 97 participants. This data includes quality and helpfulness ratings for InFix-generated and student-generated repairs for 60 scenarios randomly selected from the Python Tutor data without modification. Of the 97 participants, 24 are undergraduate or graduate students at the University of Michigan. The remaining 73 are workers recruited from Amazon Mechanical Turk (MTurk). These participants have varying levels of self-reported Python programming experience.

Each participant was shown an online series of 16 novice Python programs randomly selected from the 60 stimuli corpus.[3] Each stimulus consists of a Python program, erroneous input, error message, repaired input, and repaired output. There are two versions of each stimulus, one with the historical student-generated repair and one where it is generated by InFix. To avoid training effects, a single participant is never shown both the machine and human repair for the same error. For all 16 stimuli, participants are asked to provide a textual description of the cause of the error and to assess the quality and helpfulness of the repair on a Likert scale of 1 to 7. We also gathered self-reported estimates of both programming experience and Python-specific experience as well as qualitative data pertaining to what factors influence a subjective judgment of repair quality. Study stimuli are very similar to Figure 3.5, except that participants were only shown one repair per scenario.

[3] Copies of all stimuli can be found at `https://github.com/CelloCorgi/InputFixer`.

Table 3.2: InFix Overall Evaluation Results: All reported results are run with a 60 probe budget and 5 threads. Median and average probe and time costs (wall clock) are shown.

	Input-Error Scenarios			**Probes**		**Time (sec)**	
Year	*Total*	*Repaired*	*%*	*Med.*	*Avg.*	*Med.*	*Avg.*
2015	1,640	1,582	96.5%	1	2.98	0.87	1.12
2016	4,683	4,440	94.8%	2	3.23	0.88	1.16
2017	6,949	6,590	94.8%	2	3.47	0.90	1.23
2018	12,723	11,947	93.9%	2	3.70	0.88	1.28
Total	25,995	24,559	**94.5%**	2	3.50	**0.88**	**1.23**

The study takes around 45 minutes to complete. A participant's response was only considered valid if it correctly identified the cause of 6 / 16 errors. This high threshold is relevant for trusting MTurk worker ratings. Previous work shows that much of the data submitted on MTurk is of low quality; some users even "collude" to take advantage of the system [173]. MTurk workers were compensated with $4.50 upon successful completion while students could opt to be entered to win one of two $50 Amazon gift cards.

3.2.6 RQ1: How Effective is InFix?

We evaluate InFix on 25,995 Python Tutor scenarios with input-related errors. For our initial effectiveness assessment, we set the maximum number of probes-per-thread N to 60 and the number of threads to five (subsection 3.2.2). We perform a sensitivity analysis on these input parameters in subsection 3.2.8. We use a simple brute-force minimization technique: due to the typically short input length, we find more heavy-weight approaches unnecessary. All experiments were conducted on an Ubuntu 18.04.2 LTS server with a 4.3 GHz Intel i7-7740X quad-core CPU with 32 GB of RAM.

Our evaluation demonstrates that InFix is highly effective; it can repair 94.5% of input-related errors. We also find that InFix's repair rate is consistent, achieving similar accuracy for each separate year of data. As InFix's templates were developed from observations of the 2017 data, we believe this consistency indicates our templates generalize. A detailed breakdown of this analysis can be found in Table 3.2.

We also find that InFix is efficient, able to repair the majority of errors in under one second of wall clock time. This is important because InFix is intended to provide real-time debugging hints and repairs to novices.

> InFix is both highly effective and efficient, repairing 94.5% of 25,995 input-related scenarios in a median of 0.88 wall clock seconds (vs. 49 median seconds for novices).

Table 3.3: InFix Validation Results: Experimental results for validation of InFix's design assumptions. Tested on the 1640 scenarios from 2015.

Algorithm Variation	**Number of Inputs Fixed**	**Percent Fixed**	**Average Probes To Solve**
Error Messages Only	741	45.2%	2.70
Mutations Only	1048	64.5%	10.50
Non-Hierarchical	1561	95.2%	4.10
InFix (complete)	1582	**96.5%**	**2.98**

3.2.7 RQ2: Validating InFix's Design Assumptions

Beyond assessing the overall effectiveness of InFix, we also perform an experiment to validate our design assumptions that both error message templates and randomized mutations are helpful and that error message templates should take precedence. To do so, we implement three variants of InFix: one with only error message templates, one with only mutation templates, and one with both that uses random selection instead of a hierarchical prioritization (see subsection 3.2.2).

We compare the performance of these variations on the 1,640 2015 Python Tutor scenarios using five parallel threads and 60 maximum probes per thread. We find that InFix outperforms all three variations in repair rate, average number of probes, or both, indicating that the error message templates, mutations, and their associated hierarchy all contribute to InFix's high performance. In particular, the error-message-only and mutation-only implementations have markedly lower repair rates than InFix. Interestingly, we observe that the 45.2% of scenarios that the error-message-only version repairs is similar to the 52.1% of errors we templated (see subsection 3.2.4), indicating that our abstracted error message templates are highly effective. We also note that while the non-hierarchical version's performance is only slightly lower than InFix's, the average number of probes is 27% greater. This validates our assumption that the hierarchy between error message templates and mutations leads to increased efficiency. A detailed breakdown of our results is given in Table 3.3.

> Error message templates, mutations, and the hierarchical mutation structure are critical for InFix's high performance. Therefore, the assumptions made in our algorithm design (subsection 3.2.2) and the abstractions from our observational study (subsection 3.2.3) are validated.

3.2.8 RQ3: InFix Parameter Sensitivity

To understand InFix's parameter sensitivity, we evaluate InFix with different probe budgets (1–500) and parallel threads (1–5). We choose to focus on sensitivity with respect to more constrained resources because, unlike traditional automatic program repair, we target real-time repairs for low-

Table 3.4: InFix Parameter Sensitivity: InFix's sensitivity to the maximum number of probes and the number of threads. Each box contains the percentage of the programs solved when InFix is run with the specified parameters.

Maximum Number of Probes	Number of Threads 1	2	3	4	5
1	30.8%	36.4%	39.9%	42.6%	44.6%
5	64.1%	72.7%	77.3%	80.3%	82.6%
10	73.6%	81.0%	84.5%	86.7%	88.4%
20	80.5%	86.1%	88.8%	90.6%	91.7%
30	83.1%	88.2%	90.5%	92.0%	93.0%
60	86.7%	91.0%	92.7%	93.8%	94.5%
500	92.5%	94.5%	95.3%	95.8%	96.1%

budget tutoring sites. We do, however, include a larger probe budget to compare against previous work. For probe budgets $N \leq 60$, we evaluate all 25,995 scenarios. For 500 probes, we evaluate only 4,683 scenarios from the 2016 Python Tutor data. We find that InFix's repair rate is influenced by the values of these two input parameters; as the probe budget and threads increase, InFix's repair rate also increases. Numerical results from our sensitivity analysis are shown in Table 3.4.

We emphasize two of our findings. First, note that even with a single probe, InFix repairs a large number of input scenarios. We observe that most of these correspond to instances where the initial error message is templated. This demonstrates that InFix can still be effective even with highly constrained resources. Second, even when the probe budget doubles from 30 to 60, the repair rate with a consistent number of threads increases by at most 4.1%. Between 60 and 500 probes, an 8x resource increase, this pattern is even more pronounced, with a maximum repair rate increase of 6.3%. This indicates that InFix is largely insensitive to resource constraints on the order of those bounds established by previous work (*e.g.*, up to 307 probes reported for three algorithms on the similarly sized IntroClass student program repair benchmark [190, Fig. 5]).

InFix is insensitive to expected resource parameters and is usable even for non-parallel architectures and tight resource budgets. InFix also repairs a non-trivial amount of input-related errors in a single iteration.

3.2.9 RQ4: What is the Quality of InFix's Repairs?

As human-generated inputs and code repairs can be useful hints for novices [140, 219], we objectively and subjectively investigate how InFix's repairs compare to historical repairs made by the

Table 3.5: INFIX Human Study Results: Quality and Helpfulness from 1,544 human study ratings (1–7 Likert scale). INFIX's patches are 4% lower quality than human-written patches in a statistically significant manner ($p = 0.047$); helpfulness is not statistically distinguishable.

	Rated Patch Quality			**Rated Patch Helpfulness**		
Raters	*Human*	INFIX	*p-value*	*Human*	INFIX	*p-value*
MTurk	4.7	4.5	0.042	4.7	4.6	0.086
University	4.6	4.5	0.36	4.5	4.8	0.11
All Raters	4.7	4.5	**0.047**	4.7	4.6	0.27

PYTHON TUTOR users themselves.

Our objective evaluation compares the statement coverage of INFIX's repairs to the coverage of student repairs. We choose coverage because it is a well-understood and commonly used metric for software engineering quality assurance [14, 208].

From an analysis of the 2018 PYTHON TUTOR data, we find that the median coverage of INFIX's repairs (83.3%) is lower than the median coverage of student repairs (90.3%). Even so, INFIX's coverage is high, comparable to that achieved by PEX [313], an automated test generation tool evaluated in an educational setting [314], and greater than that of tools such as KATCH that focus on coverage for expert-written patches [208, Tab. 1].

In our second evaluation, we asked humans for subjective assessments of repair quality (see subsection 3.2.5). We collected 1,544 helpfulness and quality scores for machine and student input repairs on a Likert scale between 1 (low) and 7 (high). Details of our results are in Table 3.5, including subgroup breakdowns for university students and MTurk workers. Overall, INFIX's repairs were as helpful as student-generated repairs: we found no statistically significant difference between the helpfulness of student- and machine-generated repairs using the two-tailed Mann-Whitney U test. We did, however, find a statistically significant difference for repair quality ($p = 0.047$): the quality of human repairs is 4% higher than INFIX repairs. This subjective 96% quality assessment is very high compared to previous investigations of automated repairs.[4]

To tease apart this nuanced notion of repair quality, we consider two case studies with statistically significant differences between human and machine repairs. Figure 3.4 shows an example where the machine repair is rated better than the human repair ($p = 0.028$). This program asks the user to enter a monetary amount. The erroneous input contains a simple syntactic error: the novice includes a dollar sign with the float. The human repair simply removes the dollar sign. INFIX, however, suggests a different float. Perhaps unexpectedly, participants find the machine repair of

[4] For example, while not directly comparable, humans found PAR's patches 75.5% as acceptable as human patches and GenProg's 51.4% as acceptable [169, Tab. VII]. Similarly, Long and Rinard report 18 of Prophet's 39 patches to be correct [199, Fig. 10] in a manual human assessment, with other algorithms such as GenProg and Kali performing worse.

Program Code:

```
ticket_num = int(input("How many?: "))
cost = float(input("How much each?: "))
total = ticket_num * cost
print("Your cost is", total, ".")
```

	Erroneous Input	Student Repair	InFix Repair

Input	`3`
	`$1.50`
		`3`
		`1.50`
			`3`
			`8.10`

Output	`Error`	`Your cost is`
		`4.5 .`
			`Your cost is`
			`24.2999997 .`

Python Error Message:

```
ValueError: could not convert string to float: '$1.50'
```

Figure 3.4: Higher Quality InFix Repair Example: Example where the *machine* repair is of higher quality than the student repair by 1.4 ($p = 0.028$).

higher quality than the human repair. We hypothesize this is because only the machine repair's output reveals floating point precision formatting behavior undesirable for monetary notation.

Contrastingly, Figure 3.5 depicts an example where participants thought the human repair was better than the machine repair ($p = 0.019$). In this program, the input-related error is primarily caused by a defect in the code: on line 9, the programmer incorrectly calls `leap()` without an argument. The student input fix includes a different year, avoiding the defect due to short-circuit evaluation. InFix, however, generates `2` for the year. We believe the fact that `2` makes no contextual sense as a modern year to be the reason for its lower perceived quality. In fact, one participant singled out this repair as particularly poor, noting that "a valid date… would give a better example than a wrong (logically) year value of 2".

These two examples demonstrate that the stochastic elements of InFix can benefit repair quality, revealing edge cases that would otherwise be missed. However, they also show a limitation of template-based repair: we deliberately used small numerical values in our repairs based on our observations in subsection 3.2.3. However, this is unhelpful when the student program involves numerical inputs with other contextual constraints (Figure 3.5).

> InFix's repairs are of high quality, attaining 90.2% the statement coverage of student repairs. More importantly, 97 study participants found InFix's repairs to be equally helpful as, and to have 96% of the quality of, human repairs.

Program Code:

```
day = int(input("enter a day :"))
month = int(input("enter a month :"))
year = int(input("enter a year :"))

def leap(year):
   pass # Removed for space considerations

def checkDay(day,month,year):
   if day == 29 and month == 2 and leap():
      return day
   return False

print("Date: ",day,"/",month,"/",year)
```

Erroneous Input	Student Repair	InFix Repair
29	12	6
2	2	10
2016	2000	2

Python Error Message:

```
TypeError: leap() missing 1 required positional argument: 'year'
```

Figure 3.5: Lower Quality InFix Repair Example: Example where the *human* repair is of higher quality than the student repair ($p = 0.019$).

3.2.10 RQ5: InFix and Programmer Expertise

As InFix is designed to help novices, we are interested in the effect of programmer expertise on repair helpfulness. We analyzed the helpfulness scores of experience-based sub-populations. In our human study (see subsection 3.2.5), participants were asked to self-report their Python experience as either minimal (less than a semester), moderate (1–2 semesters), or expert (3+ semesters). Of the 96 responses, 31 are minimal, 49 are moderate, and 16 are experts.

Initially, we observed that experts rated InFix repairs more helpful than novices (4.0 vs. 5.2 out of 7). However, since the stimuli for our study were randomly sampled from the Python Tutor data set, they vary in difficulty: some programs contain Python language features that participants with minimal experience may not have encountered. We hypothesize that these hard programs are confusing for our most novice participants, leading to lower helpfulness ratings.

We thus analyzed the helpfulness of InFix's repairs for the easiest 14 and hardest 11 programs as determined by four expert annotators (Fleiss $\kappa = 0.71$). We find that participants with the least experience give repairs for easy programs higher helpfulness ratings than they give repairs for the hardest programs (4.7 vs. 3.6). Participants with the most experience, however, rate machine repairs for both easy and hard programs as equally helpful. For the easiest programs, there is no statistically significant difference in the helpfulness scores between novice and expert participants, indicating that InFix is helpful for programmers with varying experience levels. Our results are detailed in Table 3.6.

		Participant Experience Level		
	#	Minimal	Moderate	Expert
Easiest Stimuli	14	4.7	4.9	5.1
Hardest Stimuli	11	3.6	4.8	5.1
All Stimuli	60	4.0	4.8	5.2

Table 3.6: Programming Experience and InFix Helpfulness: Helpfulness ratings of InFix's repairs depending on experience. Scores are on a scale from 1 to 7.

After controlling for program difficulty, InFix's repairs are rated equally helpful by developers with varying Python experience, including novices ("less than a semester").

3.2.11 InFix: Evaluation Summary

InFix is highly effective and efficient enough to help students debug in real-time. InFix is relatively insensitive to resource-based parameters, indicating that input-related repair can be cost-effective to deploy under constraint. Beyond its high 94.5% repair rate and sub-second efficiency, we find that InFix produces very high-quality repairs (96%) that are helpful for novices and experts alike.

3.2.12 InFix: Threats to Validity

Although our experiments indicate that InFix is effective and efficient, our results and subjective quality data may not generalize. We also consider that novices may feel they rarely encounter input-related errors, making repairs unnecessary.

We first recognize that while InFix is highly effective for Python, our results may not generalize to other languages. We find it likely that InFix's success depends of the expressiveness of the language's error messages. We deliberately implement InFix for a language widely used by novices. However, investigating cross-language effectiveness remains for future work.

To mitigate the possibility of fraudulent MTurk data, we require workers to correctly identify the cause of at least 6 / 16 stimuli errors for payment. We only considered responses meeting that threshold, as assessed through manual analysis, in our evaluation. Filtering for data set inclusion based on response quality is a best practice for studies involving crowdsourced participants [85]. While 186 workers requested payment on MTurk, only 73 (39%) met the validity threshold for inclusion. This MTurk retention rate is similar to that reported by other crowdsourced studies involving debugging [111].

Finally, to mitigate threats involving problem significance, we asked our study participants how strongly they agree with the phrase: "I often encounter bugs where the input data is part of the problem". Participants report commonly encountering input-related errors: 75 / 94 responses (80%) agree or strongly agree, while only three participants strongly disagree. This result indicates that input-related errors are a common and memorable challenge faced by novices.

3.3 SEQ2PARSE: Neurosymbolic Parse Error Repair

To help support non-traditional novice programmer productivity, we next focus on providing support for *parse errors*. As with input-related errors, parse errors can vex non-traditional novices who are learning to program. Traditional error messages only indicate the first error or produce messages that are either incomprehensibly verbose or not descriptive enough to help swiftly remedy the error [255, 321]). When they occur in larger code bases, parse errors may even trouble experts, and can require a great deal of effort to fix [3, 84, 183].

Owing to their ubiquity and importance, there are *two* established lines of work on automatically suggesting *repairs* for parse errors. In the first line, Programming Languages researchers have investigated *symbolic* approaches starting with classical parsing algorithms (*e.g.*, LR [7] or Earley [88]). These algorithms can accurately locate syntax errors, but do not provide *actionable* feedback on how to fix the error. Aho and Peterson [8] extend these ideas to implement *error-correcting parsers* (EC-Parsers) that use special error production rules to handle programs with syntax errors and synthesize minimal-edit parse error repairs (see subsection 2.1.4 for an overview). Sadly, EC-parsers have remained mostly of theoretical interest, as their running time is cubic in the program size, and quadratic in the size of the language's grammar, which has rendered them impractical for real-world languages [216, 257].

In the second line, Machine Learning researchers have pursued *Deep Neural Network* (DNN) approaches using advanced sequence-to-sequence models [137, 305] that use a large corpus of code to predict the next token (*e.g.*, at a parse error location). Unfortunately, these methods ignore the high-level structure of the language (or must learn it from vast amounts of data) and hence, lack accuracy in real-world contexts. For example, state-of-the-art methods such as Ahmed *et al.* [5] parse and repair only 32% of real student code with up to 3 syntax errors while Wu *et al.* [349] repair only 58% of syntax errors in a real-world dataset.

In this dissertation, we present SEQ2PARSE, a new language-agnostic approach to automatically repairing parse errors based on the following key insight. Symbolic EC-Parsers [8] can, in principle, synthesize repairs, but, in practice, are overwhelmed by the many error-correction rules that are not *relevant* to the particular program that requires repair. In contrast, Neural approaches are fooled by the large space of possible sequence-level edits, but can precisely pinpoint the set of EC-rules that are *relevant* to a particular program. Thus, SEQ2PARSE addresses the problem of parse error repair by a neurosymbolic approach that combines the complementary strengths of the two lines of work via the following concrete contributions.

1. Motivation. Our first contribution is an empirical analysis of a real-world dataset of more than a million novice Python programs that shows that parse errors constitute the majority of novice errors, take a long time to fix, and that the fixes themselves can often require multiple edits. This analysis demonstrates that an automated tool that suggests parse error repairs in a few seconds could greatly benefit many novices (see subsection 3.3.1).

2. Implementation. Our second contribution is the design and implementation of SEQ2PARSE, which exploits the insight above to efficiently and accurately suggest repairs in a *neurosymbolic* fashion (see subsection 2.1.2): (1) train sequence classifiers to predict the *relevant* EC-rules for a given program (subsection 3.3.4), and then (2) use the predicted rules to synthesize repairs via EC-Parsing (subsection 3.3.5).

3. Abstraction. Predicting the rules is challenging. Standard token-sequence based methods from *Natural Language Processing* (NLP) are confused by long trailing contexts that are independent of the parse error. We find that this confusion can lead to inaccurate classifiers that predict irrelevant rules yielding woefully low repair rates. Our second key insight eliminates neural confusion via a symbolic intervention: we show how to use Probabilistic Context Free Grammars (PCFGs) to *abstract* long low-level token sequences so that the irrelevant trailing context is compressed into single non-terminals, yielding compressed abstract token sequences that can be accurately understood by DNNs (subsection 3.3.3).

4. Evaluation. Our final contribution is an evaluation of SEQ2PARSE using a dataset of more than 1,100,000 Python programs that demonstrates its benefits in three ways. First, we show its *accuracy*: SEQ2PARSE correctly predicts the right set of error rules 81% of the time when considering the top 20 rules and can parse 94% of our tests within 2.1 seconds with these predictions, a significant improvement over prior methods which were stuck below a 60% repair rate. Second, we demonstrate its *efficiency*: SEQ2PARSE parses and repairs erroneous programs within 20 seconds 83% of the time, while also generating *the user fix in almost 1 out of 3 cases*. Finally, we measure the *quality* of the generated repairs via a human study with 39 participants and show that humans perceive both SEQ2PARSE's edit locations and final repair quality to be useful and helpful, in a statistically significant manner, even when not equivalent to the user's fix (subsection 3.3.6).

3.3.1 A Case for Parse Error Repair

As with InFix and input repair in section 3.2, we motivate Seq2Parse by analyzing a dataset gathered from Python Tutor [130] (see subsection 2.1.1). This dataset was gathered from Python Tutor between the years 2017 and 2018, and contains *1,100,000 erroneous Python programs* along with their respective fixes. Each program that throws an uncaught Python exception is paired with the next program by the same user that does not crash, under the assumption that the latter is the fixed version of the former. We discard pairs that are too different between buggy and fixed versions, since these are usually unrelated submissions or complete refactorings. We also discard submissions that violate Python Tutor's policies (*e.g.*, those using forbidden libraries). The resulting dataset contains usable program pairs, representing students from dozens of universities (Python Tutor has been used in many introductory courses [130]) as well as non-traditional novices.

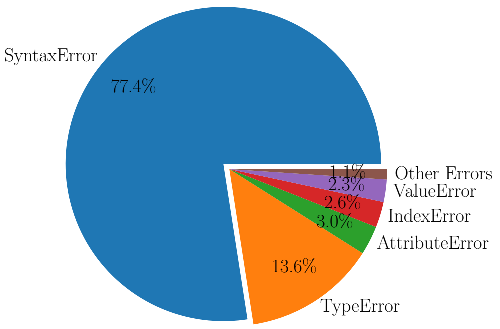


Figure 3.6: Python Error Type Distribution: The Python error type distribution for programs submitted to Python Tutor.

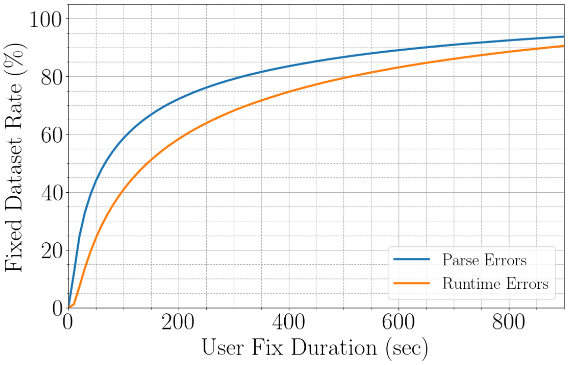


Figure 3.7: Python Tutor Users' Error Repair Rate: The historical programmer repair rates for parse errors and runtime errors vs. the time it took the programmer to repair the error.

One might imagine that parse (or *syntax*) errors are usually easier to locate and repair than other algorithmic or runtime errors [84]. For example, the Python parser will immediately inform the programmer about missing parentheses in function argument lists or improper indentation. However, as has also been shown in previous work [3, 183], our data confirm that programmers (especially novices) deal with these kinds of errors regularly and spend a considerable amount of time fixing them.

Observation 1: Parse errors are very common. Figure 3.6 presents the statistics of the different types of errors that users encountered in this dataset. We observe that 77.4% of all faulty programs failed with a syntax error, accounting for the vast majority of the errors that novice programmers

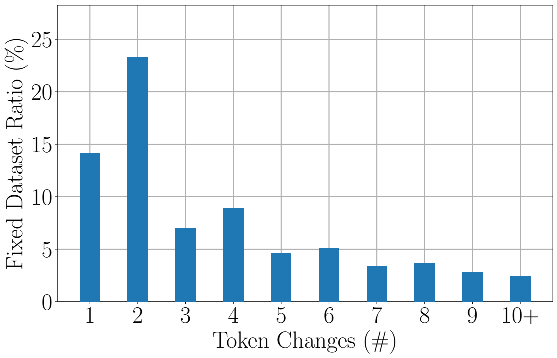


Figure 3.8: Length of PYTHON TUTOR Error Repairs: The PYTHON TUTOR dataset ratio that is fixed by modifying the given number of tokens.

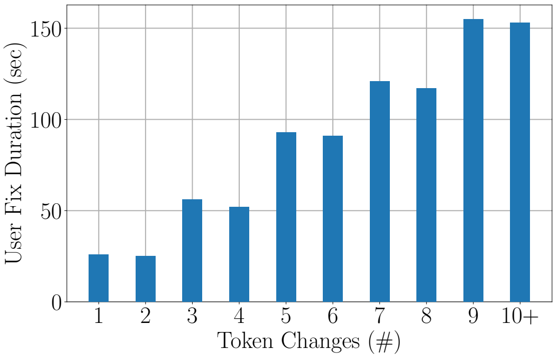


Figure 3.9: Time to Error Repair on PYTHON TUTOR: The average time the user needed to fix the erroneous PYTHON TUTOR program for the needed token changes.

face with their programs. The second category is only 13.6% of the dataset and represents Python `TypeErrors`. This is a strong indication that parse errors are a very common category of error.

Observation 2: Parse errors take time to fix. The web-based compiler used to obtain this dataset provides *server timestamps*. The timestamp is associated with each program attempt submission, erroneous or not. The *repair time* is the difference between the erroneous and fixed program timestamps. This timing can be imprecise, as there are various reasons these timings may be exaggerated, (*e.g.*, users stepping away from the computer, internet lag *etc.*). However, in aggregate, due to the large dataset size, these timings can still be viewed as an approximate metric of the time it took novice programmers to repair their errors.

Figure 3.7 shows the *programmer repair rate*, *i.e.* the dataset percentage that is repaired under a given amount of time. It presents the repair rate for parse errors and the rest of the error types, grouped together here as *runtime* errors. As expected, parse errors are fixed faster than the rest, but *not by a large difference*. For example, we observe that within 2 minutes, 46% of the runtime errors are repaired, while 63% of the syntax errors are. Although this is a non-trivial difference, we observe that there are still many "simpler" parse errors that required more than 2 minutes to fix.

Observation 3: Parse errors may need multiple edits to fix. The average *token-level changes* used to fix a program with syntax errors, *i.e.* the number of changes in the lexed program token sequence, is *10.7 token changes*, while the *median is 4*. (This does not count lexeme content changes, such as variable renamings, and thus underapproximates the work required.) As shown in Figure 3.8, 14.2% of errors need only 1 token change, 23.2% need 2 token changes, 7.0% need 3 and 9.0% need 4. Ultimately, 46.6% of these errors require more than 4 token changes.

```
1 def foo(a):
2   return a + 42
3
4 def bar(a):
5   b = foo(a) + 17
6   return b +
```

(a) A Python program with two functions that manipulate an integer. The second one has a parse error.

```
1 def foo(a):
2   return a + 42
3
4 def bar(a):
5   b = foo(a) + 17
6   return b
```

(b) A fixed version for the previous example that has no parse errors.

Figure 3.10: Python Program with a Parse Error: A Python program example with a syntax error (left) and its fix (right).

Observation 4: Parse errors with more edits take longer to fix. Figure 3.9 shows the average time for users to fix syntax errors as a function of the number of token changes needed. As expected, with an increasing number of token changes needed, programmers need more time to implement those changes. Most importantly, even for 1 or 2 token changes the average user spends 25 seconds, which is still a considerable amount of time for such simple and short fixes. The repair time jumps to 56 seconds for three token changes.

These four observations indicate that, while some errors can be easily and quickly fixed by programmers using existing error messages, there are many cases where novices struggle with fixing syntax errors. Therefore, we can conclude that an automated tool that parses and repairs such programs in only a few seconds could benefit many novices.

3.3.2 SEQ2PARSE: Overview

We begin with an overview of SEQ2PARSE's neurosymbolic approach to repairing parse errors, that uses two components. *(Neural)* Given a dataset of ill-parsed programs and their fixes, we partially parse the programs into *abstract sequences* of tokens (subsubsection 3.3.2.2), that can be used to train *sequence classifiers* (subsubsection 3.3.2.3), that predict program-relevant error rules for new erroneous programs (subsubsection 3.3.2.4). *(Symbolic)* Next, given an erroneous program, and a (small) set of predicted *program relevant* error rules, the Error Correcting Earley (ECE) parser (for a detailed description of ECE parsers, see the overview in subsection 2.1.4) can exploit the high-level grammatical structure of the language to make short work of synthesizing the best repair (subsubsection 3.3.2.1). We now give an overview of SEQ2PARSE, describing these elements in turn, using as a running example, the program in Figure 3.10a where the programmer has introduced an extra + operator after the `return b` on line 6. This extra + should be deleted, as

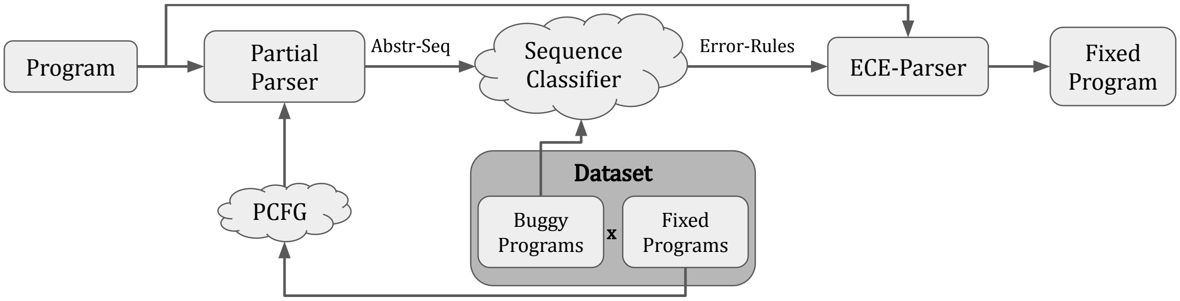


Figure 3.11: SEQ2PARSE's Overall Approach

```
S        → Stmts end_marker
Stmts    → Stmt \n | Stmt \n Stmts
Stmt     → FuncDef | ExprStmt
          | RetStmt | PassStmt | ...
FuncDef  → def name Params : Block
Block    → \n indent Stmts dedent
RetStmt  → return | return Args
Args     → ExprStmt | ExprStmt , Args
ExprStmt → ArExpr | ...
ArExpr   → Literal
          | ArExpr BinOp Literal
Literal  → name | number | ...
```

Figure 3.12: Simplified Python Production Rules.

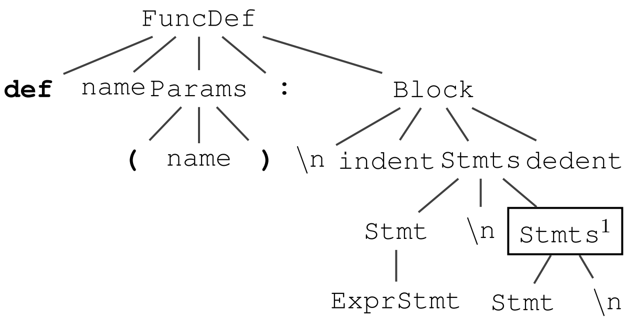


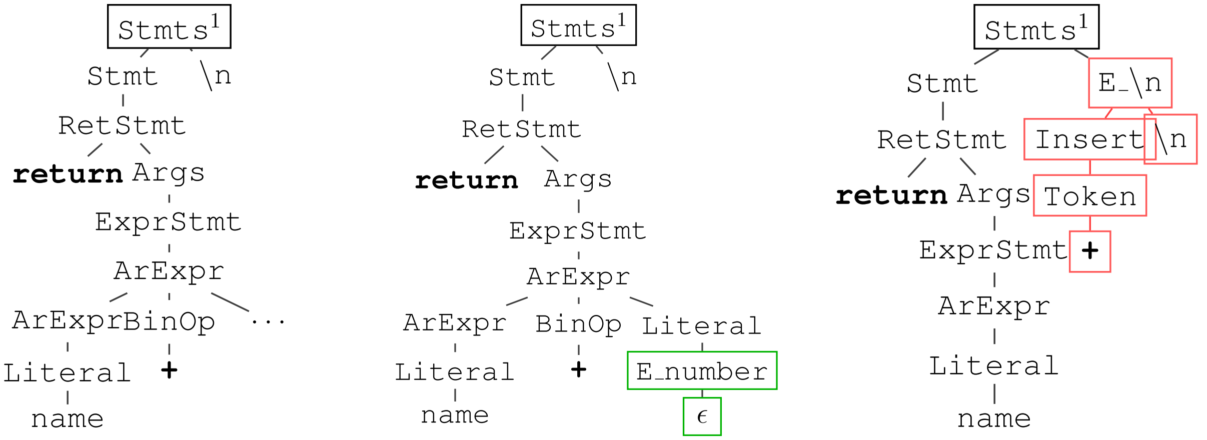
Figure 3.13: Partial Parse Tree for Example Python Program: The partial parse tree generated for `bar` in the example Python program in Figure 3.10a

shown in the developer-fixed program in Figure 3.10b.[5]

3.3.2.1 ECE Parsing Considerations with Real-World Languages

As described in subsection 2.1.4, ECE-Parsers use an extended grammar *G'* that encodes error production rules directly. This augmented grammar allows ECE-Parsers to find a minimum-edit parse for a program with parse errors [8]. Unfortunately, we run into various problems if we try to directly use an ECE-Parser for large, real-world languages like Python. Take the buggy program in Figure 3.10a which has the top-level parse tree in Figure 3.13 generated using the production rules in Figure 3.12. Figure 3.14a presents a *partial parse* of the problematic statements $\texttt{Stmts}^1$. Considering a deletion error (Figure 3.14b), the `E_number` → ϵ error rule is used to match the empty symbol and generate a parse that suggests that a *number* is missing after the + operator. On the other hand, the `E_\n` → `Insert \n` error rule can be used to consider an insertion error (Figure 3.14c) before the new line character, basically *deleting* the + operator. In this case, `ArExpr` → `Literal` is

[5] Note, this is the same running example as the one used when introducing ECE parsers in subsection 2.1.4. We reproduce it here for ease of reference.



(a) The partial parse tree for the example at Figure 3.10a.

(b) Adding a number with the green `E_number` error rule.

(c) Deleting the + with red `E_\n` error rule.

Figure 3.14: Example Error Correcting Parses for a Buggy Program: The rest of the problematic function in Figure 3.13 and two possible error-correcting parses

used to parse the program instead of `ArExpr` → `ArExpr BinOp Literal`.

The ECE-Parser is an effective approach for finding minimum distance parses for programs that do not belong in the given programming language, *i.e.* have parse errors. However, this parsing algorithm has limited use in large real-world programming languages, *e.g.* Python or Java, and more time- and memory-efficient parsing algorithms are often used, *e.g.* LR parsing [62, 175], *etc.* For example, Python has 91 *terminal symbols* (including the program's `end_marker`) which means that for all the cases of the error rules `E_t` (excluding the non-error base case `E_t` → `t`), `Replace` and `Token`, 455 *terminal error rules* have to be added to the grammar G'. The Python grammar that we used also has 283 *production rules*, from which 182 *rules* have terminals in them, meaning another 182 *error rules* need to be added. Including the four helper production rules, *e.g.* for the new start symbol, the new grammar G' has 641 *new error production rules*. This large amount of error rules renders the ECE-Parser not scalable for large programs or programs with a lot of parse errors when using real-world programming languages.

One of our insights, as seen in our running example in Figure 3.14, is that only a handful of error rules are relevant to each parse error. Therefore, we propose to improve ECE-Parsing's scalability by only adding *a small set* of error production rules, *i.e.* keeping the size of G' limited and only slightly larger than the original grammar G. We propose to do so by *training classifiers* to select a small set of error rules only relevant to the parse error. However, the program token sequences that we can use may have irrelevant information, *e.g.* the `foo` function in our example in Figure 3.10 that does not contribute to the parse error. To address this problem, we propose to further *abstract* our program token sequences.

```
def name(name): \n
indent return name + number \n
dedent \n

def name(name): \n
indent name = name(name) + number \n
return name + \n
dedent end_marker
```

(a) The program token sequence generated by the lexer.

```
Stmt \n

def name Params: \n
indent Stmt \n
return Expr BinOp \n
dedent end_marker
```

(b) The abstracted token sequence for the same program. Parts of the program that can't be abstracted (*e.g.* `def name`) remain the same.

Figure 3.15: Token Sequences for an Example Python Program: The token sequences for the Python program example in Figure 3.10.

3.3.2.2 Abstracting Program Token Sequences

As shown in Figure 3.11, our neural component has the task of training a classifier to predict the relevant error rules for a given ill-parsed program.

Problem: Representing Ill-parsed Programs. As the inputs are ill-parsed, the training and classification cannot use any form of analysis that requires a syntax tree as input [127, 211, 277, 339]. One option is to view the ill-parsed program as a plain *sequence of tokens* eliding variable names and such, as shown in Figure 3.15a. Unfortunately, we found such token sequences yielded inaccurate classifiers that were confused by irrelevant trailing context and predicted rules that were not relevant to repair the error at hand.

Solution: Abstract with Partial Parses. Seq2Parse solves the problem of irrelevant context by *abstracting* the token sequences using *partial parses* to abstract away the irrelevant context. That is, we can use partial parse trees to represent ill-parsed programs as an *abstracted token sequence* shown in Figure 3.15b, where any *completed* production rules can be used to abstract the relevant *token sub-sequences* with the high-level *non-terminal*.

Figure 3.13 shows how partial parses can be used to abstract long low-level sequences of tokens into short sequences of non-terminals. (1) The function `foo` is completely parsed, since it had no parse errors and the highest level rule that can be used to abstract it is `Stmt → FuncDef`. (2) Similarly, note that `Params → ( name )` is another completed production rule, therefore the low-level sequence of parameter tokens in the `bar` function can be abstracted to just the non-terminal `Params`. (3) However, the production rule for `FuncDef` is *incomplete* since the last statement `Stmt` (under `Stmts`[1]) has a parse error as shown in Figure 3.14a.

Problem: Ambiguity. The generation of this abstraction, however, poses another difficulty. Earley parsing collects a large amount of partial parses (via dynamic programming) until the program is

```
S        → Stmts end_marker (p = 100.0%)
Stmts    → Stmt \n (p = 38.77%) | Stmt \n Stmts (p = 61.23%)
Stmt     → ExprStmt (p = 62.64%) | RetStmt (p = 7.59%) | ...
RetStmt  → return (p = 1.61%) | return Args (p = 98.39%)
Args     → ExprStmt (p = 99.20%) | ...
ExprStmt → ArExpr (p = 29.40%) | ...
ArExpr   → Literal (p = 86.89%) | ArExpr BinOp Literal (p = 13.11%)
Literal  → name (p = 64.89%) | number (p = 20.17%) | ...
```

Figure 3.16: Simplified Probabilistic Context-Free Grammar: The production rules shown in Figure 3.12 with their learned probabilities.

fully parsed. That means at each program location, multiple partial parses can be chosen to abstract our programs. This *ambiguity* can be seen even in the two suggested repairs in Figure 3.14: if we delete the colored nodes in Figure 3.14b and Figure 3.14c we obtain two possible partial parses for our program, the first one matching Figure 3.14a and the second one not shown here.

Solution: Probabilistic Context-Free Grammars. SEQ2PARSE solves the ambiguity problem of choosing between multiple possible partial parses via a data-driven approach based on *Probabilistic Context-Free Grammars* which have been used in previous work to select *complete* parses for ambiguous grammars [70, 157]. A PCFG associates each of its production rules with a *weight* or *probability*. These weights can be learned [70] by using the data set to count the production rules used to parse a corpus of programs belonging to that language. SEQ2PARSE uses PCFGs to resolve the ambiguity of partial parses by associating each partial tree (in the Earley table) with a probability which is the *product* of the used rules' probabilities. The tree with the highest probability is selected. It is then used to generate an abstracted token sequence, as described above.

Figure 3.16 shows the learned probabilities for the example Python grammar in Figure 3.12. We observe, for example, that `ReturnStmt` has two possible production rules and almost 98.4% of the times a `return` is followed by an argument list. Additionally, 62.6% of the times a `Stmt` is an `ExprStmt` and only 7.6% of the times it is a `RetStmt`. Thus, in our example, the probability that would be assigned to the partial parse for `Stmts`^{1} in Figure 3.14b (only the sub-tree without the colored error nodes) is the product of the probabilities of the production rules `Stmts` → `Stmt \n`, `Stmt` → `RetStmt`, `RetStmt` → `return Args`, `Args` → `ExprStmt` *etc.* which is $38.77\% \cdot 7.59\% \cdot 98.39\% \cdot 99.20\% \cdot \cdots = 4.57‰$ (parts per thousand), while the partial parse for `Stmts`^{1} in Figure 3.14c would similarly be calculated as 47.61‰, making it the proper choice for the abstraction of the program.

3.3.2.3 Training Sequence Classifiers

The abstracted token sequences we extracted from the partial parses present us with short abstracted sequences that abstract irrelevant details of the context into non-terminals. Next, SEQ2PARSE uses the NLP approach of *sequence models* [137, 305] to use the abstract token sequences to train a classifier that can predict the relevant error rules.

Seq2Seq Architectures. Sequence-to-sequence (seq2seq) architectures transform an input sequence of tokens into a new sequence [305] and consist of an *encoder* and a *decoder*. The encoder transforms the input sequence into an *abstract vector* that captures all the essence and context of the input. This vector does not necessarily have a physical meaning and is instead an internal representation of the input sequence into a higher dimensional space. The abstract vector is given as an input to the decoder, which in turn transforms it into an output sequence.

SEQ2PARSE uses a *sequence classifier* that can correctly predict a small set of relevant error production rules for a given abstracted token sequence. We use a *transformer encoder* [325] to encode the input sequences into abstract vectors that we then feed into a DNN classifier to train and make accurate predictions [280].

Training From a Dataset. Given a dataset of fixed parse errors, such as Figure 3.10, we extract the small set of relevant error rules needed for each program to make it parse with an ECE-Parser. Running the ECE-Parser on every program in the dataset with the *full set* of error production rules is prohibitively slow. Therefore, we extract the erroneous and fixed program token-level differences or *token diffs* and map them to *terminal error production rules*. The non-terminal error rules can be inferred using the grammar and the terminal rules. Next, we run the ECE-Parser with the extracted error rules to confirm which ones would make the program parse and assign them as *labels*.

For example, the diff for the program pair in Figure 3.10 would show the deleted `+` operator, thus extracting the error rules `Token → +` and `E_\n → Insert \n`, since the extra `+` precedes a newline character `\n`. Similarly, if a token `t` is added in the fixed program, the error rule `E_t → ϵ` is added and if a token `t` replaces a token `a`, the error rules `E_t → Replace` and `Replace → a` are added.

3.3.2.4 Predicting Error Rules with Sequence Classifiers

The learned sequence classifier model, which has been trained on the updated error-rule-labeled data set, can now be used to predict the relevant rules for new erroneous programs. Additionally, neural networks have the advantage of associating each class with a *confidence score* that can be used to rank error rule predictions for new programs, letting us select the top-N ones that will yield accurate repairs when used with the ECE-Parser.

For our running example, in Figure 3.10a, we abstract the program as shown in Figure 3.15b and then we predict the error production rules for it with the trained sequence classifier. We rank the full set of *terminal* error rules based on their predicted confidence score from the classifier and return the *top 10* predictions for our example. Therefore, the predicted set of error rules is the following: `E_number` → ϵ, `E_number` → `Insert number`, `E_\n` → `Insert \n`, `E_(`→ ϵ, `E_return` → ϵ, `Token` → `)`, `Token` → `+`, `Token` → `:`, `Token` → `name`, `Token` → `number`.

The classifier predicts mostly relevant error rules such as the ones that use `E_number`, `E_\n` and `E_return` for example, as we showed previously. There are also rules that are not very relevant to this parse error but the classifier predicts probably due to them being common parse errors, *e.g.* `Token` → `)`, `Token` → `:`. Finally, we added the *non-terminal* error rules needed to introduce these errors, which can be inferred by them. For example, we can infer `Stmts` → `Stmt E_\n`, `Stmts` → `Stmt E_\n Stmts` and `Block` → `E_\n indent Stmts dedent` from `E_\n` (we don't need `E_indent` or `E_dedent` here since no such terminal error rules were predicted).

We then parse the program in Figure 3.10a with the ECE-Parser and these specific error rules to generate a valid parse. We observe that it takes our implementation (as we show later in depth) less than *2 seconds* to generate a valid parse, which is also the one that leads to the user repair in Figure 3.10b. On the other hand, when we use a baseline ECE-Parser with the full set of error rules it takes *2 minutes and 55 seconds* to generate a valid parse, which is, however, not the expected user parse but the one shown in Figure 3.14b. These examples demonstrate the effectiveness of accurately *predicting error rules using sequence classifiers, which are trained on abstracted token sequences*.

In the next three sections, we describe in depth the specifics of our approach by defining all the methods in Figure 3.17. We start by presenting the program abstraction (subsection 3.3.3) using partial parses and a learned PCFG, we then explain how we train sequence models for making error rule predictions (subsection 3.3.4) and, finally, we demonstrate our algorithms for building SEQ2PARSE (subsection 3.3.5), an approach for efficiently parsing erroneous programs.

3.3.3 Abstracting Programs with Parse Errors for SEQ2PARSE

SEQ2PARSE abstracts programs (with parse errors) into sequences of *abstract* tokens that are used to train sequence classifiers. Next, we explain how a traditional Earley parser can be used to extract *partial* parses using a Probabilistic Context-Free Grammar (PCFG), to get a higher level of abstraction that preserves more contextual information than the low-level sequence output by the lexer.

Lexical Analysis. *Lexical analysis* (or lexing or tokenization) converts a sequence of characters into a sequence of tokens comprising strings with an assigned and thus identified meaning (*e.g.* numbers,

G	$\doteq$	$(N,\ \Sigma,\ P,\ S)$
PCFG	$\doteq$	$(N,\ \Sigma,\ P,\ S,\ W)$
G'	$\doteq$	$(N',\ \Sigma,\ P',\ S')$
P'	$\doteq$	$P \cup$ ErrorRules
N'	$\doteq$	$N \cup \{S',\ H,\ I\}$ $\cup \{E_a \mid a \in \Sigma\}$
e	$\in$	$L(G)$
$e_\perp$	$\notin$	$L(G)$
Pair	$\doteq$	$e_\perp \times e$
DataSet	$\doteq$	[Pair]

§ 3.3.3	**learnPCFG**	:	$G \to [e] \to$ PCFG
	partialParse	:	PCFG $\to e_\perp \to t^a$
§ 3.3.4	**trainDL**	:	$[t^a \times$ ErrorRules$] \to$ DLModel
	predictDL	:	DLModel $\to t^a \to$ ErrorRules
§ 3.3.5	**diffRules**	:	Pair $\to$ ErrorRules
	ECEParse	:	ErrorRules $\to e_\perp \to e$
	train	:	DataSet $\to$ DLModel
	predict	:	DLModel $\to G \to e_\perp \to$ ErrorRules
	Seq2Parse	:	$G \to$ DataSet $\to (e_\perp \to e)$

Figure 3.17: High-Level SEQ2PARSE API: A high-level API of the SEQ2PARSE system that learns to repair syntax errors.

identifiers *etc.*). Lexing is usually combined with a parser, which together analyze the syntax of a programming language $L(G)$, defined by the grammar G. When a program has a syntax error, the output token sequence of the lexer is the highest available level of abstraction since the parser fails without producing a parse tree.

Token Sequences. Our goal is to parse a *program token sequence* t^i, which is a lexed program *with* parse errors (*i.e.* $t^i \notin L(G)$), and repair it into a *fixed token sequence* $t^o \in L(G)$ that can be used to return a repaired program without syntax errors. Let t^i be a sequence $t^i_1, t^i_2, \ldots, t^i_n$ and t^o be the updated sequence $t^o_1, t^o_2, \ldots, t^o_i, \ldots, t^o_j, \ldots, t^o_k$. The subsequence $t^o_i, \ldots, t^o_j$ can either *replace* a subsequence in t^i, it can *be inserted* in t^i or can be the empty subsequence ϵ and *delete* a subsequence in t^i to generate the t^o. It can be the whole program, part of it, or multiple parts of it. t^o will finally be a token sequence that can be parsed by the original language's $L(G)$ parser.

However, programs (and hence n) can be large which makes these token sequences unsuitable for training sequence models effectively. Therefore, our goal is to first generate an *abstracted token sequence* t^a that removes all irrelevant information from t^i and gives hints for the parse error fix by using the internal states of an *Earley* parser.

3.3.3.1 Earley Partial Parses

SEQ2PARSE uses an *Earley parser* [88] to generate the abstracted token sequence t^a for an input program sequence t^i. An Earley parser holds internally a *chart* data structure, *i.e. a list of partial parses*. Given a production rule $X \to \alpha\beta$, the notation $X \to \alpha \cdot \beta$ represents a condition in which α has already been parsed and β is expected and both are sequences of terminal and non-terminal symbols (tokens).

Each state is a tuple $(X \rightarrow \alpha \cdot \beta,\ j)$, consisting of

- the production rule currently being matched $(X \rightarrow \alpha\beta)$
- the current position in that production (represented by the dot $\cdot$)
- the origin position j in the input at which the matching of this production began

We denote the state set at an input position k as $S(k)$. The parser is seeded with $S(0)$ consisting of only the top-level rule $S \rightarrow \gamma$. It then repeatedly executes three operations: *prediction, scanning,* and *completion*. There exists a *complete parse* if the complete top-level rule $(S \rightarrow \gamma\cdot, 0)$ is found in $S(n)$, where n is the input length. We define a *partial parse* to be any partially completed rules, *i.e.* if there is $(X \rightarrow \alpha \cdot \beta, i)$ in some state $S(k)$, where $i < k \leq n$.

Let, $t^i_1, t^i_2, \ldots, t^i_j, \ldots, t^i_k, \ldots, t^i_n$ be the input token sequence t^i, where there is a parse error at location k and the parser has exhausted all possibilities and can not add any more rules in state $S(k+1)$, *i.e.* $S(k+1) = \emptyset$. We want to abstract program subsequences $t^i_j, \ldots, t^i_k$ by getting the longest possible parts of the program t^i that have a partial parse. For example, we start from the beginning of the program t^i_1 by finding the largest j for which there is a rule $(X \rightarrow \alpha \cdot \beta, 0) \in S(j)$. We use this rule for X to replace $t^i_1, t^i_2, \ldots, t^i_j$ in t^i with α, thus getting an abstracted sequence t^a. In the same manner, we use the longest possible partial parses that we can extract from the chart to abstract $t^i_{j+1}, \ldots, t^i_k$, iteratively, until we reach the parse error at location k.

Problem: Multiple Partial Parses. However, each of the states $S(j),\ 0 \leq j \leq k$, holds a large number of partial parses. Thus, our heuristic to choose the longest possible partial parse to abstract programs may not be able to abstract the token sequence fully until the error location k. It may even not be able to do so until the end location n of the program, which may not have any more parse errors. Additionally, there may be two or more partial parses in $S(k)$, with different lengths, *e.g.* $\{(X \rightarrow \alpha \cdot \beta,\ j),\ (X' \rightarrow \alpha' \cdot \beta',\ h)\} \in S(k),\ j \neq h$. We propose selecting the most *probable parse* with the aid of a PCFG.

3.3.3.2 Probabilistic Context-Free Grammars

We learn a PCFG from a large corpus of programs $[e], e \in L(G)$, that belong to a language $L(G)$ that a grammar G defines, with the **learnPCFG** procedure as shown in Figure 3.17. We use the learned PCFG with an augmented Earley parser in **partialParse** to abstract a program $e_\perp$ into a abstract token sequence t^a.

Probabilistic CFG. A PCFG can be defined similarly to a *context-free grammar* $G \doteq (N,\ \Sigma,\ P,\ S)$ as a quintuple $(N,\ \Sigma,\ P,\ S,\ W)$, where:

- N and Σ are finite disjoint alphabets of non-terminals and terminals, respectively.

- P is a finite set of production rules of the form $X \to \alpha$, where $X \in N$ and $\alpha \in (N \cup \Sigma)^*$.
- S is a distinguished start symbol in N.
- W is a finite set of probabilities $p(X \to \alpha)$ on production rules.

Given a dataset of programs $[e], e \in L(G)$ that can be parsed, let $count(X \to \alpha)$ be the number of times the production rule $X \to \alpha$ has been used to generate a final complete parse, while parsing $[e]$, and $count(X)$ be the number of times the non-terminal X has been seen in the left side of a used production rule. The probability for a production rule $X \to \alpha$ is then defined as:

$$p(X \to \alpha) = \frac{count(X \to \alpha)}{count(X)}$$

learnPCFG invokes an instrumented Earley parser to calculate all the values $count(X \to \alpha), \forall X \to \alpha : P$ and $count(X), \forall X : N$. The *instrumented parser* keeps a *global record* of these values, while parsing the dataset $[e]$ of programs. Finally, **learnPCFG** outputs a PCFG that is based on the original grammar G that was used to parse the dataset with the learned probabilities W.

3.3.3.3 Abstracted Token Sequences

Given a program $e_\perp$ with a parse error and a learned PCFG, **partialParse** will generate an abstracted token sequence t^a. The PCFG will be used with an *augmented Earley parser* to disambiguate partial parses and choose one, in order to produce an abstracted token sequence as described in § 3.3.3.1.

We augment Earley states $(X \to \alpha \cdot \beta,\ j)$ to $(X \to \alpha \cdot \beta,\ j,\ p)$, where p is the probability that $X \to \alpha \cdot \beta$ is a correct partial parse. When there are two (or more) conflicting partial parses $\{(X \to \alpha \cdot \beta,\ j,\ p),\ (X' \to \alpha' \cdot \beta',\ h,\ p')\} \in S(k)$, the augmented parser selects the partial parse with the highest probability $max(p,\ p')$. The augmented parser calculates the probability p for a partial parse $(X \to \alpha \cdot \beta,\ j,\ p)$ in the state $S(k)$, as the product $p_1 \cdot p_2 \cdot \dots \cdot p_{k-1}$ of the probabilities $p_1,\ p_2,\ \dots,\ p_{k-1}$ that are associated with the production rules $(X_1 \to \alpha_1 \cdot \beta_1,\ i_1,\ p_1), (X_2 \to \alpha_2 \cdot \beta_2,\ i_2,\ p_2), \dots$ that have been used so far to parse the string of tokens α.

3.3.4 Training Sequence Classifiers for SEQ2PARSE

Our next task is to *train* a model that can predict the error production rules that are needed to parse a given program $e_\perp$ (with syntax errors) according to a given grammar G, by using its (abstracted) program token sequence t^a. We define the function **predictDL** which takes as input a *pre-trained sequence classifier* DLModel and an abstracted token sequence t^a and returns as output a *small subset* of ErrorRules. We train the DLModel offline with the **trainDL** method with a dataset $[t^a \times \text{ErrorRules}]$ of token sequences t^a and the *exact small set* of error production rules

ErrorRules that the ECE-Parser used to generate the *user parse*. We build our classifier DLModel using classic *Deep Neural Networks (*DNN*s)* and parts of state-of-the-art *Sequence-to-Sequence (seq2seq)* models. We leave the high-level details of acquiring the dataset of labeled token sequences and using the predictor for new erroneous programs for subsection 3.3.5. In the next few paragraphs, we summarize the recent advances in machine learning that help as build the sequence classifier.

We encode the task of learning a function that will map token sequences of erroneous programs to a small set of error production rules as a *supervised multi-class classification (MCC)* problem. A *supervised* learning problem is one where, given a labeled training set, the task is to learn a function that accurately maps the inputs to output labels and generalizes to future inputs. In a *classification* problem, the function we are trying to learn maps inputs to a discrete set of two or more output labels, called *classes*. We use a *Transformer encoder* to encode the input sequences into abstract vectors that we then directly feed into a DNN classifier to build a *Transformer classifier*.

Neural Networks. A neural network can be represented as a directed acyclic graph whose nodes are arranged in layers that are fully connected by weighted edges. The first layer corresponds to the input features, and the final layer corresponds to the output. The output of an internal node is the sum of the weighted outputs of the previous layer passed to a non-linear *activation function*, which in recent work is commonly chosen to be the rectified linear unit (ReLU) [228]. In this work, we use relatively *deep neural networks* (DNN) that have proven to make more accurate predictions in recent work [280]. A thorough introduction to neural networks is beyond the scope of this work [141, 235].

Sequence Models. *Seq2seq* models aim to transform input sequences of one domain into sequences of another domain [305]. In the general case, these models consist of two major layers, an *encoder* and a *decoder*. The encoder transforms an input token sequence $x_1, x_2, \ldots, x_n$ into an *abstract vector* $V \in \mathbb{R}^k$ that captures all the essence and context of the input sequence. This vector does not necessarily have some physical meaning and is just an internal representation of the input sequence into a higher dimensional space. The abstract vector is then given as an input to the decoder, which in turn transforms it into an output sequence $y_1, y_2, \ldots, y_n$.

The simplest approach historically uses a Recurrent Neural Network (RNN) [272, 342], which is a natural next step from the classic neural networks. Each RNN unit operates on each input token x_t separately. It keeps an internal *hidden state* h_t that is calculated as a function of the input token x_t and the previous hidden state h_{t-1}. The output y_t is calculated as the product of the current hidden state h_t and an output weight matrix. The activation function is usually chosen as the standard *softmax* function [34, 116]. Softmax assigns probabilities to each output that must add up to 1. Finally, the loss function at all steps of the RNN is typically calculated as the sum of the cross-entropy loss of each step.

Transformers. The Transformer is a DNN architecture that deviates from the recurrent pattern (*e.g.*, RNNs) and is solely relying on *attention mechanisms*. Attention has been of interest lately [19, 170, 325] mainly due to its ability to detect dependencies in the input or output sequences regardless of the distance of the tokens. The nature of this architecture makes the Transformer significantly easier to parallelize and thus have a higher quality of predictions and sequence translations after a shorter training period.

The novel architecture of a Transformer [325] is structured as a *stack of N identical layers*. Each layer has two main sub-layers. The first is a *multi-head self-attention mechanism*, and the second is a position-wise fully connected neural network. The output of each sub-layer is $LayerNorm(x + SubLayer(x))$, where $SubLayer(x)$ is the function implemented by each sub-layer, followed by a residual connection around each of the two sub-layers and by layer normalization $LayerNorm(x)$. To facilitate these residual connections, all sub-layers in the model, as well as the input *embedding layers*, produce outputs of the same dimension d_{model}.

Transformer Classifier. For our task, we choose to structure DLModel as a *Transformer Classifier*. We use a state-of-the-art Transformer encoder to represent an abstracted token sequence t^a into an abstract vector $V \in \mathbb{R}^k$. The abstract vector V is then fed as input into a multi-class DNN. We use **trainDL** to train the DLModel given the training set $[t^a \times \text{ErrorRules}]$. The binary cross-entropy loss function is used per class to assign the loss per training cycle. DLModel predicts error production rules for a new input program t^a. Critically, we require that the classifier outputs *confidence scores* $\mathcal{C}$ that measure how sure the classifier is that a rule can be used to parse the associated input program $e_\perp$. The **predictDL** function uses the trained DLModel to predict the confidence scores $[\text{ErrorRules} \ \times \ \mathcal{C}]$ for all error production rules ErrorRules for a new unknown program $e_\perp$ with syntax errors. The ErrorRules are then sorted based on their predicted confidence score $\mathcal{C}$ and finally the *top-N* rules are returned for error-correcting parsing. N is a small number in the 10s that will give accurate predictions without making the ECE-Parser too slow, as we discuss in subsection 3.3.6.

3.3.5 Building Seq2Parse, a Fast Error-Correcting Parser

We show how Seq2Parse uses the abstracted token sequences from subsection 3.3.3 and the trained sequence models from subsection 3.3.4 to generate an *error-correcting parser* $(e_\perp \to e)$, that will parse an input program $e_\perp$ with syntax errors and produce a correct program e. We first describe how we extract a machine-learning-amenable training set from a corpus of fixed programs and finally how we structure everything to train our model.

3.3.5.1 Learning Error Production Rules

The **trainDL** method requires a dataset of token sequences t^a that is annotated with an *exact and small set* of error production rules, *i.e.* $[t^a \times \textsf{ErrorRules}]$. These ErrorRules are just a subset of all the possible error rules that are needed to parse and fix t^a. The straightforward approach is to use **ECEParse** with all possible error production rules for each program $e_\bot$ in the dataset. Then, when **ECEParse** returns with a successful parse, we extract the rules that where used to parse the program $e_\bot$. This approach generates a dataset with the smallest possible set of error rules as labels per program, since the original ECE-Parser returns the minimum-distance edit parse. However, this approach completely ignores the programmer's fix and takes an unreasonable amount of time to parse a dataset with millions of programs, due to the inefficient nature of the ECE-Parser.

We suggest using an $O(ND)$ difference algorithm [227] to get a small but still representative set of error production rules for each program $e_\bot$. We employ this algorithm to find the differences between the input *program token sequence* t^i, which is the lexed program $e_\bot$ and the *fixed token sequence* t^o, which is the lexed program e. This algorithm returns changes between token sequences in the form of *inserted or deleted tokens*. It is possible that this algorithm returns a sequence of deletions followed by a sequence of insertions, which can in turn be interpreted as a *replacement* of tokens. We map these three types of changes to the respective error production rules. Let t^i be a sequence $t^i_1, t^i_2, \ldots, t^i_n$ and t^o be the updated sequence $t^o_1, t^o_2, \ldots, t^o_m$. We map:

- an inserted output token t^o_j to a *deletion* error $E_{t^o_j} \rightarrow \epsilon$.
- a deleted input token t^i_k to an *insertion* error $Tok \rightarrow t^i_k$ and the helper rule $E_{t^i_{k+1}} \rightarrow Ins\ t^i_{k+1}$.
- a replaced token t^i_k with t^o_j to a *replacement* error $Repl \rightarrow t^i_k$ and the helper rule $E_{t^o_j} \rightarrow Repl$.

In the case of an insertion error, we also include the helper rules $Ins \rightarrow Tok$ and $Ins \rightarrow Ins\ Tok$, that can derive any nonempty sequence of insertions. To introduce (possible) insertion errors at the end of a program, we include the starting production rules $S' \rightarrow S$ and $S' \rightarrow S\ Ins$.

The above algorithm, so far, adds only the *terminal error productions*. We have to include the *non-terminal error productions* that will invoke the terminal ones. If $X \rightarrow a_0 b_0 a_1 b_1 \ldots a_m b_m,\ m \geq 0$, is a production in P such that a_i is in N^* and b_i is in Σ, then we add the error production $X \rightarrow a_0 X_{b_0} a_1 X_{b_1} \ldots a_m X_{b_m},\ m \geq 0$ to P', where each X_{b_i}, is either a new non-terminal E_{b_i} that was added with the previous algorithm, or just b_i again if it was not added.

Finally, we further refine the new small set of error productions for each program $e_\bot$ with the ECE-Parser, to create the final annotated dataset $[t^a \times \textsf{ErrorRules}]$. The changes that we extracted from the programmers' fixes might include irrelevant changes to the parse error fix, *e.g.* code clean-up. Therefore, filtering with the ECE-Parser (*e.g.*, running the ECE-Parser on the original program to determine the actual rules required) is still essential to annotate each program with

Algorithm 3.2 Training **Seq2Parse**'s model DLModel

Input: Probabilistic Grammar G, DataSet Ds
Output: Classifier $Model$

```
procedure TRAIN($G$, $Ds$)
    $D_{ML} \leftarrow \emptyset$
    for all $p_{err} \times p_{fix} \in Ds$ do
        $t^a \leftarrow$ PARTIALPARSE($G$, $p_{err}$)
        $rules \leftarrow$ DIFFRULES($p_{err} \times p_{fix}$)
        $D_{ML} \leftarrow D_{ML} \cup (t_a \times rules)$
    $Model \leftarrow$ TRAINDL($D_{ML}$)
    return $Model$
```

Algorithm 3.3 Predicting error production rules with **Seq2Parse**'s model DLModel

Input: Classifier $Model$, Probabilistic Grammar G, Program P
Output: Error Production Rules Rls

```
procedure PREDICT($Model$, $G$, $P$)
    $t^a \leftarrow$ PARTIALPARSE($G$, $P$)
    $Rls \leftarrow$ PREDICTDL($Model$, $t^a$)
    return $Rls$
```

the appropriate error production rules. We implement this error-rule-extracting approach in the function **diffRules**, which extracts the token differences between an erroneous program $e_\perp$ and a fixed program e and returns the appropriate error production rules.

3.3.5.2 Training and Using a Transformer Classifier

Given a (probabilistic) grammar G and a dataset Ds, Algorithm 3.2 extracts a machine-learning appropriate dataset D_{ML} to **train** a Transformer classifier $Model$ with **trainDL**. The classifier $Model$ can then be used to predict error rules for new erroneous programs p_{err}.

The dataset D_{ML} starts as an empty set. For each program pair $p_{err} \times p_{fix}$, we, first, employ **partialParse** with the PCFG G and an erroneous program p_{err} to extract the abstracted token sequence t^a. Second, we use the token difference algorithm **diffRules** to extract the specific error $rules$ that fix p_{err} based on p_{fix}. The abstracted sequence t^a is annotated with the label $rules$ and is added to D_{ML}. The Transformer classifier $Model$ is trained with **trainDL** and the newly extracted dataset D_{ML}, which is finally returned by the algorithm. Finally, the **train** procedure can be performed offline and thus won't affect the performance of the final program repair.

Having trained the Transformer classifier $Model$, we can now predict error rules Rls, that will be used by an ECE-Parser, by using the **predict** procedure defined in Algorithm 3.3. **predict** uses the same input grammar G to generate an abstracted token sequence t^a for the program P with the **partialParse** procedure. Finally, the **predictDL** procedure predicts a small set of error production rules Rls for the sequence t^a given the pre-trained $Model$.

3.3.5.3 Generating an Efficient Error-Correcting Parser

Algorithm 3.4 presents our *neurosymbolic* approach, SEQ2PARSE. This is the high-level algorithm that combines everything that we described so far in the last three sections. SEQ2PARSE first

Algorithm 3.4 Generating the Final ECEP

```
Input: Grammar $G$, DataSet $Ds$
Output: Error-Correcting Parser $Prs$
procedure Seq2Parse($G$, $Ds$)
    $ps \leftarrow$ Map($\lambda.p \rightarrow$ Snd($p$), $Ds$)
    $PCFG \leftarrow$ LearnPCFG($G$, $ps$)
    $Model \leftarrow$ Train($PCFG$, $Ds$)
    ERulePredictor $\leftarrow$ Predict($Model$, $PCFG$)
    $Prs \leftarrow (\lambda.p_{err} \rightarrow$ ECEParse(ERulePredictor($p_{err}$), $p_{err}$))
    return $Prs$
```

extracts the fixed programs ps from the dataset Ds to learn a probabilistic context-free grammar $PCFG$ for the input grammar G with **learnPCFG**. It then trains the Transformer classifier $Model$ to predict error production rules. We define an error rule predictor, ERulePredictor, using the **predict** procedure with the pre-trained $Model$ and grammar $PCFG$. Finally, the algorithm returns the ECE-Parser Prs, which we define as a function that takes as input an erroneous program p_{err} that uses the ERulePredictor to get the set of error rules needed by **ECEParse** to parse and repair it.

3.3.6 Seq2Parse: Evaluation Overview

We have implemented our approach in Seq2Parse: a system for repairing parse errors for Python in its entirety.[6] Next, we describe our implementation and an evaluation that addresses four questions:

- **RQ1**: How *accurate* are Seq2Parse's predicted error production rules? (subsection 3.3.7)
- **RQ2**: How *precisely* can Seq2Parse repair parse errors? (subsection 3.3.8)
- **RQ3**: How *efficiently* can Seq2Parse repair parse errors? (subsection 3.3.9)
- **RQ4**: How *useful* are Seq2Parse's suggested repairs? (subsection 3.3.10)

Training Dataset. For our evaluation, we use the same Python dataset that we used in our error data analysis in subsection 3.3.1, gathered from Python Tutor [130] between the years 2017 and 2018. The dataset has more than 1,100,000 usable erroneous Python programs and their respective fixes. The programs have an average length of *87 tokens*, while the abstracted token sequences have a much shorter average of *43 tokens*. We choose 15,000 random programs from the dataset for all our tests, and the rest we use as our training set.

[6] The code for Seq2Parse is publicly available at `https://github.com/gsakkas/seq2parse` and a simplified online demonstration is available at `http://seq2parse.goto.ucsd.edu/index.html`.

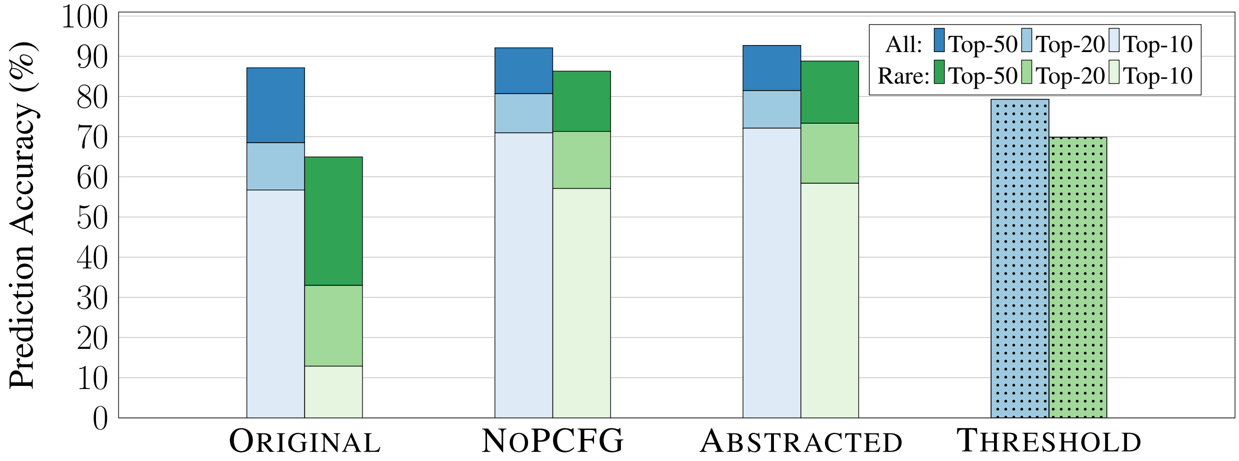


Figure 3.18: Error Production Rule Prediction Classifier Results: Results of our error production rule prediction classifiers for the simple original token sequences and their abstracted versions using the PCFG.

We first learn a PCFG on the training set of fixed programs to learn the probabilities for each production rule in the *full Python grammar*. SEQ2PARSE then extracts the abstracted token sequences for all programs in the training set. Next, while the full Python grammar has 455 possible terminal error production rules, in reality, only *340 error rules* are ever used in our dataset and are assigned as *labels*. We arrive at this set of error rules by parsing all the erroneous programs in the training set with the ECE-Parser and the "diff" error rules, as described in subsubsection 3.3.5.1.

Transformer Classifier. SEQ2PARSE's error rule prediction uses a Transformer classifier with *six* transformer blocks that each have a fully-connected hidden layer of 256 neurons and 12 attention heads. The output of the transformer blocks is then connected to a DNN-based classifier with *two* fully-connected hidden layers of 256 and 128 neurons respectively. The neurons use rectified linear units (ReLU) as their activation function, while the output layer uses the sigmoid function for each class. Additionally, there are *two input embedding layers* of a length of 128 units, one for input tokens and one for their positions in the sequence. We also limit the input abstracted token sequences to a length of 128 tokens, which covers 95.7% of the training set, without the need to prune them. Finally, the Transformer classifier was trained using an ADAM optimizer [171], a variant of stochastic gradient descent, on an NVIDIA GeForce RTX 3080 Ti for a total of 50 epochs.

3.3.7 RQ1: SEQ2PARSE Accuracy

Figure 3.18 shows the accuracy results of our error production rule prediction experiments. The y-axis describes the *prediction accuracy*, *i.e.* the fraction of test programs for which the correct *full set* of error rules to repair the program (extracted from the user fix) was predicted in the top-K sorted rules. The ORIGINAL version of our transformer classifier does not consider the abstracted token sequences and used the full ORIGINAL token sequences, whose results are presented in the first two bars of Figure 3.18. The next four bars show our final results using the ABSTRACTED token sequences to train the classifier with NOPCFG and with fully ABSTRACTED sequences. Finally, the last two dotted bars show the results for when a probability THRESHOLD is set in order to select the predicted error rules (instead of picking the static top-K ones) but using again the ABSTRACTED sequences as input. The predicted error rule set ranges between 1–20 elements.

The blue bars show the accuracy on the full test set of ALL 15,000 test programs, while the green bars show the results on a subset of RARE programs, *i.e.* programs that did not include any of the 50 most popular error rules. The RARE programs account for 1233 programs, 8% of our test set.

The ORIGINAL predictor, even with the Top-50 predicted error rules, is less accurate than the Top-20 predictions of the ABSTRACTED, with an accuracy of 87.13%, which drops to 68.48% and 56.71% respectively for the Top-20 and Top-10 predictions. The ABSTRACTED predictor significantly outperforms the ORIGINAL predictor with a 72.11% Top-10 accuracy, 81.45% Top-20 accuracy and 92.70% Top-50 accuracy.

The THRESHOLD predictions are almost as accurate as the ABSTRACTED Top-20 predictions with an accuracy of 79.28% and a median number of selected error rules of 14 (average 14.1). This could potentially mean that this predictor is a valid alternative for the static Top-20 predictions.

The classifiers are also not very *sensitive* in the PCFG probabilities used during abstraction, as shown in the accuracy of the NOPCFG predictor. The NOPCFG predictor has almost 2% less Top-10 and Top-20 accuracy, 70.94% and 80.69% respectively, and less than 1% for Top-50 predictions, with 92.11%.

Finally, we observe that our ABSTRACTED classifiers generalize efficiently for our dataset of erroneous Python programs and are almost as accurate for the RARE programs as the rest of the dataset with a 73.32% Top-20 accuracy (88.81% Top-50 accuracy). The same holds for the THRESHOLD predictions with a 69.83% RARE accuracy. The NOPCFG also has a drop of more than 2% accuracy, with a 71.29% Top-20 accuracy (86.29% Top-50 accuracy).

SEQ2PARSE's transformer classifier learns to encode programs with syntax errors and select candidate error production rules for them effectively, yielding *high accuracies*. By abstracting the tokens sequences, SEQ2PARSE is able to *generalize* better and make more accurate predictions with a *81.45% Top-20 accuracy*.

3.3.8 RQ2: SEQ2PARSE Repaired Program Preciseness

Next we evaluate SEQ2PARSE's end-to-end accuracy and preciseness when restricting SEQ2PARSE's parsing time to *5 minutes* and run our experiments on the 15,000-program test set. Additionally,

Error Rule Approach	Abstracted				NoPCFG			
	Parse Accuracy	*Rare Parse Accuracy*	*User Fix Accuracy*	*Median Parse Time*	*Parse Accuracy*	*Rare Parse Accuracy*	*User Fix Accuracy*	*Median Parse Time*
20 Most Popular	79.87%	65.01%	16.31%	7.0 sec	–	–	–	–
50 Most Popular	90.89%	81.26%	18.56%	13.6 sec	–	–	–	–
AllParses	61.46%	59.80%	**34.57%**	7.1 *(14.2)* sec	55.52%	56.42%	30.21%	20.3 *(24.3)* sec
MinimumCost	**94.25%**	**94.01%**	20.55%	5.3 *(12.9)* sec	91.63%	90.89%	17.89%	5.9 *(18.7)* sec
Threshold	94.19%	93.42%	21.19%	**2.1 *(7.0)* sec**	93.70%	91.09%	19.39%	2.5 *(9.1)* sec

Figure 3.19: Seq2Parse Experimental Results: The (*parenthesized*) numbers in the Median Parse Time columns represent the median time for larger programs, *i.e.* programs with more than 100 tokens.

we use here the highest-performing transformer classifiers, *i.e.* the Abstracted, NoPCFG and Threshold classifiers.

We compare *three versions* of our Seq2Parse implementation (AllParses, MinimumCost and Threshold) against two versions of the ECE-Parser with a static selection of the 20 and 50 most popular error production rules in our training set. We make this choice because we observe that the 50 most popular error rules are used as labels for as much as *86%* of the training set. For the AllParses, MinimumCost and Threshold versions, we run our experiments for the Abstracted and NoPCFG classifier predictions.

The MinimumCost ECE-Parser uses the *Top-20 predictions* from our Abstracted or NoPCFG classifier to parse and repair buggy programs. The AllParses and Threshold ECE-Parsers use the Threshold classifier's predicted set of error rules to repair programs.

The AllParses ECE-Parser keeps internally *all possible states* that arise from using the predicted error rules similarly to the original ECE-Parser described by Aho and Peterson [8]. We use a maximum repair cost of 3 edits (*i.e.*, a maximum of 3 insertions, deletions or replacements) to limit the search space. The MinimumCost version, however, always keeps the minimum-edit repair and discards all other states that may lead to a higher cost. This more efficient version of the ECE-Parser allows for a higher maximum cost of 10 edits. We use the same ECE-Parser and cost as in MinimumCost for our Threshold parser. The maximum cost for each parser is a hyperparameter to Seq2Parse and is set here arbitrarily to achieve a uniform run time across experiments while obtaining high-quality multiple-edit repairs. Finally, MinimumCost will always return the top 1 repair, while AllParses can generate a large number of repairs and we select to keep only the top 5 repairs after filtering with a static code checker (Pylint, `https://www.pylint.org/`) as most developers will consider only a few suggestions before falling back to manual debugging [176, 244].

Figure 3.19 shows the percentage of test programs that each of these five versions can parse successfully (*i.e.* the *parse accuracy*), the rare program parse accuracy, and the *user equivalent parse accuracy*, *i.e.* the amount of parses that match the one that the user compiled. We observe that the ABSTRACTED MINIMUMCOST parser *outperforms* every other option with 94.25% parse accuracy and 94.01% rare parse accuracy. It also generates the intended user parse for 20.55% of the set, *i.e.* over 1 out of 5 of the cases. The 20 MOST POPULAR parser with 79.87% parse accuracy and 65.01% rare parse accuracy is much less accurate, and is 4.24% less likely to generate the user parse, while the 50 MOST POPULAR is slightly less accurate with 90.89% and 81.26% accuracy, as expected from the usage of a large number of popular error rules. The 50 MOST POPULAR parser has also a high user fix accuracy of 18.56%. The ALLPARSES parser has the lowest parse accuracy of 61.46%. However, it manages to generate the user fix 34.57% of the time and also achieve a 59.80% rare accuracy. Finally, the THRESHOLD parser is almost as accurate as the efficient MINIMUMCOST parser with 94.19% and 93.42% parse and rare accuracy, while achieving a slightly higher user fix accuracy of 21.19%.

Additionally, the NOPCFG MINIMUMCOST parser achieves 2.6% lower parse accuracy than the ABSTRACTED version and 3.1% less rare parse accuracy. The NOPCFG THRESHOLD parser also performs only 0.5% less accurately than the ABSTRACTED version and has a 2.3% less rare parse accuracy. Finally, both NOPCFG parsers achieve 2.7% and 1.8% respectively lower user fix accuracy. These results further confirm that our ECE-Parsers are not very sensitive to the PCFG use in the abstraction phase. However, the NOPCFG ALLPARSES ECE-Parser performs much worse with 55.52% accuracy and 56.42% rare parse accuracy, which highlights the importance of abstracting token sequences with our full algorithm.

SEQ2PARSE can *parse and repair 94.25%* of programs with syntax errors. In addition, it generates *the exact user fix over 20% of the time*.

3.3.9 RQ3: SEQ2PARSE Efficiency

Next, we evaluate SEQ2PARSE's efficiency by measuring how many programs it can parse. We limit each ECE-Parser to 5 minutes (we conjecture that a longer timeout will diminish the practical usability for developers). We compare the efficiency of SEQ2PARSE for all the versions of Figure 3.19 using the full test set of 15,000 programs.

Figure 3.20 shows the cumulative distribution function of all SEQ2PARSE approaches' repair rates over their repair time. We observe that using THRESHOLD predictions with the MINIMUMCOST ECE-Parser is the most efficient and it maintains the highest parse accuracy at all times, with a repair rate of 83.04% within 20 seconds and a median parse time of 2.1 seconds. We also observe that the median parse time for *larger programs* is slightly higher with 7.0 seconds for programs with more than 100 tokens and increases a bit more for more than 500 tokens, with 10.8 seconds. While the ECE-parser uses dynamic programming that may not scale greatly for larger programs, SEQ2PARSE's scalability is *mostly* proportional to the predicted error rules and the number of

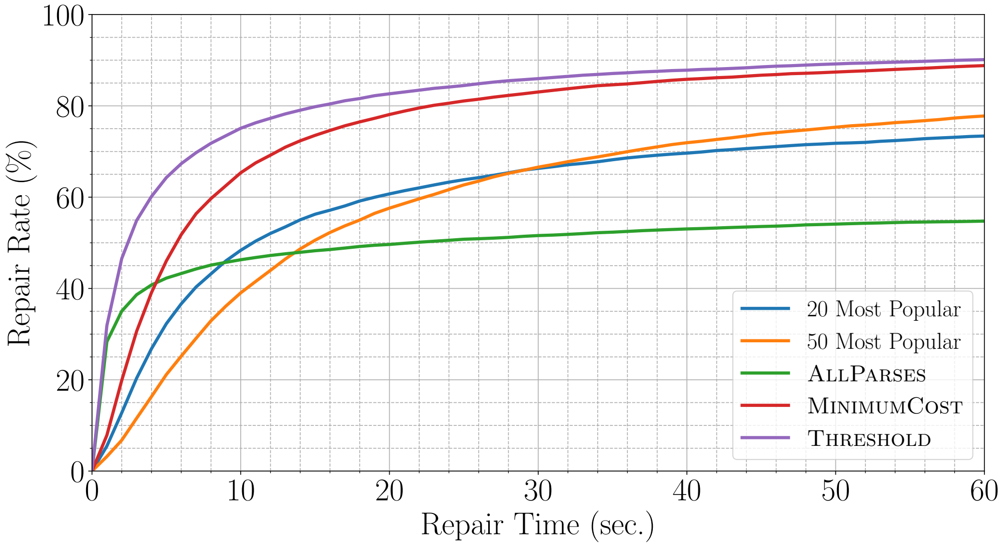


Figure 3.20: Seq2Parse Repair Rate vs. Repair Time: The repair rate for all the Abstracted approaches in Figure 3.19.

syntax errors, and therefore we don't get an exponential explosion in parse time for larger programs.

The MinimumCost with the top 20 error rule predictions is still very efficient with a repair rate of 78.10% within 20 seconds and a median parse time of 5.3 seconds. For larger programs the median parse time is 12.9 seconds for programs with more than 100 tokens and 50.8 seconds for more than 500 tokens. We observe that a fixed-length set of predicted error rules can hinder the ECE-parser, when inaccurate predictions are involved.

We observe that, using a fixed set of the 20 and 50 most popular rules, Seq2Parse (with the MinimumCost ECE-Parser) repairs 61.41% and 58.61% of the programs respectively within 20 seconds, and has median parse times of 7.0 and 13.6 seconds respectively. The 50 most popular rules admit parsing fewer programs quickly than the 20 most popular.

We also observe that Seq2Parse successfully parses around 49.90% of the programs with its AllParses approach in 20 seconds and has a median parse time of 7.1 and 14.2 seconds for all programs and programs with more than 100 tokens respectively. While this approach is much less efficient that the others, it is also able to generate the exact human repair in more than 1 out of 3 cases, representing a valuable quality tradeoff (subsection 3.3.8).

Seq2Parse can parse programs with syntax errors for the vast majority of the test set in under 20 seconds with a median parse time of 2.1 seconds.

3.3.10 RQ4: SEQ2PARSE Usefulness

As SEQ2PARSE is intended as an aid for programmers (especially novices) faced with parse errors, we are also interested in subjective human judgments of the quality and helpfulness of our repairs. Around 35% of repairs produced by SEQ2PARSE using its ALLPARSES approach are identical to the historical human repair and thus likely helpful for programmers. However, it may be that SEQ2PARSE's parses (and thus repairs) are still helpful for debugging even when they differ slightly from the human repair (*i.e. non-equivalent* repairs). To investigate this hypothesis, we conduct a human study of the quality and debugging helpfulness of SEQ2PARSE's non-equivalent repairs.

Human Study Setup. We recruited participants from two large public research institutions (UC San Diego and the University of Michigan) and through Twitter. The study was online, took around 30 minutes, and participants could enter a drawing for one of two $50 awards. In the study, participants were each asked to rate 15 debugging hints randomly selected from a corpus of 50 stimuli.[7]

We created the stimuli by selecting 50 buggy programs from our test set for which SEQ2PARSE and the human produced different fixes. Other than ensuring a wide array of difficulty (as assessed by how long the human took to fix the error), programs were selected randomly. Each stimulus consisted of a buggy program, its associated syntax error message, and a potential program fix presented as a *debugging hint*. For each stimulus, we produced two versions: one where the debugging hint was generated by SEQ2PARSE and one where the debugging hint was the historical human fix. Note that, in practice, the historical human fix would *not* be available to a struggling novice in real situations: it represents future or oracular information. Informally, in our comparison, the historical human fixes can be viewed as an upper bound.

Participants rated the quality and helpfulness of each debugging hint using a 1–5 Likert scale. They also indicated if the debugging hint provided helpful information beyond that in the Python error message. Participants were unaware of whether any given hint was generated by a human or SEQ2PARSE, and participants were never shown multiple fixes to the same program. To be included in the analysis, participants had to assess at least four stimuli. Overall, we analyze 527 unique stimuli ratings from $n = 39$ valid participants (246 for human fixes and 281 for SEQ2PARSE).

Overall Results. While humans in our study find that non-equivalent repairs produced by SEQ2PARSE are lower in both quality and debugging helpfulness than those produced manually (2.9/5 helpfulness for tool-produced repairs vs. 3.7/5 for human-produced repairs, $p < 0.001$), humans still often find SEQ2PARSE's fixes helpful for debugging. Participants found that SEQ2PARSE repairs contained helpful debugging information beyond that contained in the Python Error message

[7] All human study stimuli are included in our replication package at https://github.com/gsakkas/seq2parse and via the human study website https://dijkstra.eecs.umich.edu/~endremad/APR_HumanEval/.

48% of the time (134/281). This additional debugging information was helpful in terms of both the content (73% of the time) and location (55% of the time). Additionally, SEQ2PARSE fixes are helpful for easy and hard syntax errors alike: we found no statistically significant difference between the helpfulness or quality of SEQ2PARSE's repairs for easy (those repaired by the human in under 40 seconds) or hard parse errors (over 40 seconds). Overall, these results indicate that even when SEQ2PARSE repairs differ from historical human repairs, they can still be helpful for debugging.

Individual Stimuli. Beyond an analysis of SEQ2PARSE's overall quality, we also analyze the helpfulness of each stimulus. Of the 48 programs for which we collected sufficient data to permit statistical comparison, the historical repair was statistically more helpful for debugging than SEQ2PARSE's repair for 33% of stimuli (16/48, $p < 0.05$). However, we found that SEQ2PARSE's repair was actually *more helpful* for debugging than the human's repair for 15% of stimuli (7/48, $p < 0.05$). For the remaining 52% of stimuli, we found no evidence of a statistical difference in the debugging helpfulness of the two repairs.

Buggy Program

```
def gcdIter(a, b):
   for i in range(1, a+1):
      if a % i == 0:
      elif b % i == 0:
   return i
gcdIter(9, 12)
```

Human Fix

```
def gcdIter(a, b):
   for i in range(1, a+1):
      return a % i
gcdIter(9, 12)
```

SEQ2PARSE Fix

```
def gcdIter(a, b):
   for i in range(1, a+1):
      if a % i == 0: new_var
      elif b % i == 0: break
   return i
gcdIter(9, 12)
```

(a) SEQ2PARSE repair that is significantly more helpful: 4.3/5 vs 1.0/5, $p = 0.03$

Buggy Program

```
aList = [12, 'yz', 'ab'];
aList.reverse();
print "List : ", aList
```

Human Fix

```
aList = [12, 'yz', 'ab']
aList.reverse()
```

SEQ2PARSE Fix

```
aList = [12, 'yz', 'ab']
aList.reverse()
print("List : ", aList)
```

(b) SEQ2PARSE repair that is significantly more helpful: 4.75/5 vs 2.0/5, $p = 0.02$

Buggy Program

```
a = int(input(enter a))
print(a***3)
```

Human Fix

```
a = int(input("enter a"))
print(a**3)
```

SEQ2PARSE Fix

```
a = int(input(enter)(a))
print(a ** (* 3))
```

(c) Historical human repair that is significantly more helpful: 1.8/5 vs 4.75/5, $p = 0.01$

Figure 3.21: Example SEQ2PARSE Repairs: Three example buggy programs followed by their historical human and SEQ2PARSE repairs. For (a) and (b), SEQ2PARSE's repair was rated more helpful by participants. For (c), the human repair was more helpful.

To better contextualize these results, we provide examples of stimuli with statistically significant differences in debugging helpfulness. In Figure 3.21b, Seq2Parse's repair was significantly more helpful than the historical repair: Seq2Parse correctly adds parentheses to `print` while the human simply deletes the buggy line, perhaps out of confusion or frustration. Similarly, Figure 3.21a's Seq2Parse repair was also better than the human repair. In this case, the user appears to try to implement a function to calculate the greatest common divisor of two integers, but has empty `if` and `elif` statements. To "fix" this bug, the user deletes the `if` and `elif` and modifies the return statement. However, this fix does not correctly calculate the greatest common divisor. Seq2Parse, on the other hand, adds a template variable to the `if` and `break` to the `elif`. While this also does not implement greatest common divisor, it is viewed as more helpful than the user repair. This example also demonstrates the beneficial ability of our approach to conduct multi-edit repairs.

Figure 3.21c, on the other hand, shows an example of a more helpful human repair. In this case, the human correctly deletes the extraneous `*` in the power operator while Seq2Parse adds parentheses to make a more complex expression, the result of favoring one insertion over one deletion. Similarly, the human correctly transforms the argument to `input` on line 1 into a string by adding two quote (`"`) tokens, whereas Seq2Parse again makes a more complicated expression.

35% of Seq2Parse's repairs are equivalent to historical repairs. Of the remainder, our human study found 15% to be more useful than historical repairs and 52% to be equally useful. In total, including both equivalent and non-equivalent cases, Seq2Parse repairs are at least as useful as historical human-written repairs 78% of the time.

3.3.11 Qualitative Comparison of Seq2Parse to Related Work

Seq2Parse performs quite well compared to the published state of the art for the particular domain of novice programs, discussed in subsection 2.1.5. We note that most existing related work is on different benchmarks or datasets, permitting only an indirect comparison. However, we believe that Seq2Parse compares favorably in terms of *accuracy* and *efficiency*, since it completely repairs 94.25% of our tests within 2.1 seconds, while generating the exact user fix in more than 1 out of 3 of the cases, a metric that most papers ignore.

Specifically, DeepFix [132] uses a multi-layer seq2seq model to repair programs that may have up to 5 syntax errors, but initial results, although promising, yield error-free compilation for only *27%* out of the 6971 benchmark programs. *Lenient parsing* [5] leverages a large corpus of code and error seeding to train a transformer-based neural network, resulting in a broadly applicable approach, but one with potentially lower accuracy in our domain (a top-1 repair accuracy of only *32%* for real student code with up to 3 syntax errors). GGF [349] tries to encode program context in a novel way by using a graph neural network and partial parses, which leads to a higher repair accuracy of *58%* of the syntax errors in a real-world dataset. Lastly, CoCoNuT [201] is a recent state-of-the-art automated repair technique that depends on a different approach of context-aware NMTs and is evaluated on standard software defect benchmarks. While CoCoNuT can repair a broader range of defects than syntax errors, it only repairs 509 out of 4456 (11.42%) benchmark defects.

3.4 Lens 1 Conclusion: InFix and Seq2Parse

In our first lens into improving programmer productivity, we help non-traditional novices by providing efficient bug-fixing support in the form of an automatically generated repair. To work towards this goal, we first analyzed a large corpus of programs from non-traditional novice programmers, submitted to the free online programming environment Python Tutor [129, 130]. From this analysis, we identified input-related bugs and parse errors as two types of error types that are challenging for novices but overlooked by existing expert-focused automatic program repair. To help non-traditional novices fix these errors, this dissertation proposes and evaluates two tools, InFix for input-related bugs and Seq2Parse for parse errors. We briefly summarize each approach in turn.

In section 3.2, we presented InFix, a randomized template-based approach for automatically fixing buggy program inputs. InFix repairs input data rather than source code, requires no test cases, and requires no special annotations. InFix iteratively applies prioritized error-based templates (generated from a qualitative study of buggy novice inputs) and random mutations. We evaluated InFix on 25,995 unique input-related scenarios from Python Tutor. Overall, we found that InFix can repair 94.5% of novice input errors. The majority of these errors were repaired in under a second, facilitating real-time support. Beyond automated evaluation, we also find that programmers view InFix's repairs to be of high quality: in a human study with 97 participants, humans judged the output of InFix to be equally helpful and within 4% of the quality of human-generated repairs.

In section 3.3, we presented Seq2Parse, a language-agnostic neurosymbolic approach to automatically repair parse errors. Seq2Parse uses a dataset of ill-parsed programs and their fixed versions to train a classifier that accurately predicts EC-rules for ill-parsed programs. To make accurate predictions, we abstract the low-level program token sequences using partial parses and probabilistic grammars. Then, a small set of predicted EC-rules are used with an ECE-Parser to repair the program in a tractable and precise manner. We evaluated Seq2Parse using ill-parsed Python programs from Python Tutor. We found that Seq2Parse makes accurate EC-rule predictions 81% of the time when considering the top 20 EC-rules, and that the predicted EC-rules let us parse and repair over 94% of the test set in a median of 2.1 seconds, generating the user fix in almost 1 out of 3 cases. Finally, we conducted a user study with 39 participants, finding that

SEQ2PARSE's repairs are useful and helpful, even when not equivalent to the user's fix.

Through developing and evaluating INFIX and SEQ2PARSE, we show that targeted bug-fixing tools could enhance the productivity of non-traditional novice programmers. By addressing errors specific to novices (input-related bugs and parse errors) we provide practical repairs that are found to be of high quality by users. This work demonstrates the potential of combining programming language techniques with empirical insights to support novices in writing more correct code faster, ultimately contributing to a more inclusive and efficient software development environment.

CHAPTER 4

Improving Programming Outcomes with Cognitive Training

In our second lens into improving programmer productivity, we focus on using cognitive insights to develop effective developer training and help novices become more like experts faster. Building and fixing software are highly trained activities: for many aspiring software engineers, completing a first computer science course (CS1) is an essential step toward a technology-focused career. However, CS1 students often struggle [26, 172], and less affluent students with weaker preparatory education in relevant cognitive skills can struggle disproportionately [242]. This may contribute to a lack of diversity endemic in many modern software workplaces [262].

Recognizing these challenges, the software engineering research community has demonstrated increased interest in both supporting novice programmers [136, 258, 316] and understanding which cognitive skills are most important for software engineering success [104, 153, 293, 294]. As discussed in subsection 2.2.3, the increased use of neuroimaging in software engineering has led to new insights into the cognitive processes behind programming. At a high level, many of the neuroimaging studies in software engineering have compared programming to reading or spatial reasoning, two skills with well-understood cognitive structures. As broadly summarized in Table 4.1, such studies have generally found striking similarities between code comprehension and prose reading [104, 293, 294], and one study has found similarities between spatial reasoning and data structure manipulation [153]. For more information on related work connecting programming to reading and spatial reasoning, see subsection 2.2.5 and subsection 2.2.7.

Importantly, neuroimaging has the potential to enhance our understanding of expertise and inform software engineering pedagogy, helping novices become experts more quickly (see Floyd *et al.* [104, Sec. II-D] for a summary). However, most (c.f., [246]) software engineering neuroimaging studies as of 2023 have only studied programming *experts* that are either professionals or students with multiple years of experience (*e.g.*, [294, Sec. 3.3]). Should patterns observed in experts continue to novices, supplementary training for relevant cognitive skills identified through neuroimaging may *transfer* to improved programming outcomes (see subsection 2.2.8). Such transfer for spatial

Experiment	Like reading?	Like spatial reasoning?	Novices?
Siegmund *et al.* (2014) [293]	✓		**?**
Siegmund *et al.* (2017) [294]	✓		**?**
Floyd *et al.* (2017) [104]	✓		**?**
Huang *et al.* (2019) [153]		✓	**?**

Table 4.1: Summary of Closest Existing Neuroimaging Work: "What is coding like in programmers' brains?" We use this informal summary of selected previous work to motivate and contextualize the experiments in this chapter of this dissertation.

reasoning has been observed in the educational literature [37, 71, 302, 320], but the benefit of training of other cognitive skills identified by neuroimaging (such as reading) is less understood.

In this dissertation, we take a first step toward realizing the potential for neuroimaging findings to influence software training in practice. We do so by first conducting a neuroimaging study on new programmers. We then carry out a semester-long controlled experiment, investigating if supplementary cognitive training can transfer to improved programming outcomes. We focus this initial investigation on the two aforementioned cognitive skills: reading and spatial visualization. We start this chapter with an overview of our experimental approach in section 4.1. We then present our findings from our neuroimaging experiment in section 4.2, and our transfer training experiment in section 4.3. Finally, we conclude and discuss the implications of our findings in section 4.4.

4.1 Lens 2: Overview

In this dissertation lens, we investigate if neuroimaging can help inform supplementary cognitive training that *transfers* to improved programming outcomes. We take a two-phase approach. *First*, we use fNIRS (a type of neuroimaging, see subsection 2.2.1) to conduct a controlled study with first-semester programming students. We compare participants' brain activation patterns while coding to those while reading prose or using spatial reasoning (*i.e.*, mentally rotating 3D objects, see subsection 2.2.6). The goal of this phase is both to model novice cognition and also to see if patterns observed with more expert developers continue. *Second*, we compare two skills-based CS1 interventions: spatial ability training and technical reading training. Specifically, *we propose a novel computer-science-focused technical reading training* for novice software engineers, and we *compare its effectiveness to that of an established spatial ability curriculum.*

We hypothesize that reading ability may sometimes be more critical than spatial ability for software engineering success. One of the most common skill-based interventions proposed and tested for STEM education is spatial ability training (see subsection 2.2.8). Supplemental training

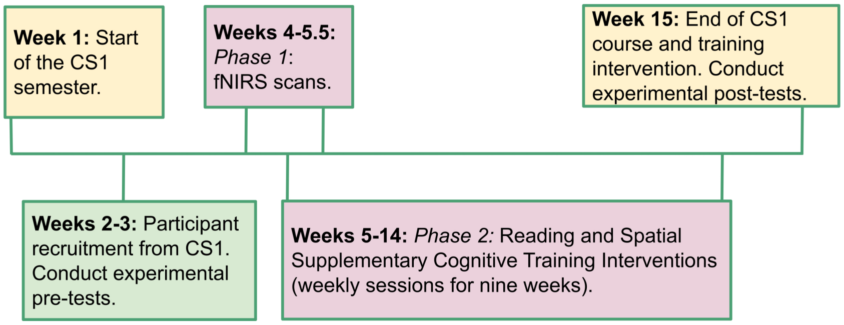

Week 1: Start of the CS1 semester.
Weeks 4-5.5: *Phase 1:* fNIRS scans.
Week 15: End of CS1 course and training intervention. Conduct experimental post-tests.
Weeks 2-3: Participant recruitment from CS1. Conduct experimental pre-tests.
Weeks 5-14: *Phase 2:* Reading and Spatial Supplementary Cognitive Training Interventions (weekly sessions for nine weeks).

Figure 4.1: Lens 2 Experimental Timeline: Overall timeline of the two-phase experiment presented in chapter 4 of this dissertation.

designed to help improve spatial ability has been shown to both improve outcomes in several key engineering classes and also increase engineering degree retention [300]. There is also evidence that spatial ability training directly improves novice programming outcomes [37, 71].

We observe, however, that there are many aspects of software that are not obviously spatial. For example, many essential programming tasks, such as code review, code summarization, and documentation, require strong technical reading skills (see subsection 2.2.5). Supporting this hypothesis, one neuroimaging study with experts found that reading code became more neurologically similar to reading prose as programmers gained even greater expertise [104]. It is possible, therefore, that technical reading ability may sometimes be more important for initial software engineering success.

4.1.1 Overall Experimental Design

We now discuss each phase of our overall experimental design in more detail. For clarity, Figure 4.1 contains an overview of our timeline for both phases.

Phase 1 (fNIRS)—Developing a Mathematical Model of Novice Programmer Cognition. To understand the cognitive processes of novice programmers, we first use functional near-infrared spectroscopy (fNIRS, see subsection 2.2.1) to conduct a controlled study with first-semester programming students. Our goal is to compare participants' brain activation patterns while coding, reading prose, and using spatial reasoning to determine if the cognitive processes observed in expert programmers are also present in novices.

We recruit participants who are just starting their first computing course (CS1), ensuring that they have minimal prior programming experience to maintain consistency in the study (for more information on our participant recruitment procedure, see subsection 4.1.2). At a high level, our fNIRS experiment involves contrasting the brain activation of novice programmers completing three tasks: coding, reading prose, and performing spatial reasoning exercises. We carefully design the stimuli for these tasks to be appropriate for novices, leveraging introductory computing syllabi to create coding problems and selecting relevant prose and spatial reasoning exercises. For a more detailed discussion of our fNIRS design and stimuli construction, see subsection 4.2.1.

Overall, we find that novice programmers exhibit distinct brain activation patterns for all three tasks. However, we also find that there are more significant and substantial differences between prose and coding than between spatial reasoning and coding. This result suggests that the brain activation patterns observed in expert programmers (*i.e.* the finding that reading code becomes less distinguishable from reading prose as programming expertise increases [104]) may extend to novices, providing a foundation for developing targeted cognitive training interventions. We discuss the implications of our findings in greater detail in section 4.4.

Phase 2 (Transfer Training)—Evaluating the Pedagogical Impact of Cognitive Training. Beyond modeling novice cognition, we also test if targeted cognitive training in either reading or spatial reasoning transfers to improved programming outcomes. To do so, we design a semester-long technical reading training course (see subsection 4.3.1) and compare its effectiveness with a validated spatial ability curriculum (see subsection 4.3.5). We hypothesize that explicitly teaching how to trace through scientific texts will transfer to tracing through programs, a key software activity. Across the course of a semester, we conducted a controlled experiment with 57 students enrolled in the same CS1 class used in phase 1. We randomly assign participants to one of two Treatments, a Reading Treatment and a Spatial Treatment.

In each Treatment, participants attend a supplementary training session once a week for nine weeks. In the Reading Treatment, participants receive our technical reading intervention, and in the Spatial Treatment, participants attend a standardized spatial training intervention. The goal of this experiment is to see if either Treatment *transfers* to improved programming outcomes. To facilitate comparing the two Treatments, we ensure equal time commitments and a similar structure for each. For more details on the controlled structure of our transfer training experiment, see subsection 4.3.2.

Overall, we find that technical reading training transfers to improved programming outcomes: CS1 students in our reading intervention demonstrated significantly greater programming gains on our experimental post-test than do those in the spatial intervention ($p < 0.05$). This result demonstrates that cognitive insights learned through neuroimaging can lead to direct productivity impacts in practice. We discuss the implications of our findings in greater detail in section 4.4.

4.1.2 Participant Recruitment and Population Contextualization

As illustrated in Figure 4.1, we use the same participants for both phase one (fNIRS) and phase two (transfer training). More specifically, our fNIRS population is a subset of those participants enrolled in the full transfer training study. In the rest of this section, we discuss our participant recruitment and contextualize our study population.

Table 4.2: Transfer Training Participant Overview: Profile of study participants by Treatment type. We only include racial and ethnic categories with at least 5 study participants. The pre-test scores are all average raw scores.

	Spatial	Reading
Total Participants	28	29
Female Participants	23	17
Male Participants	5	12
Native English Speakers	21	20
Native Chinese Speakers	4	8
White / Caucasian	10	10
Asian / Pacific Islander	15	12
Hispanic American	2	3
Pre-test PFT (out of 20)	13.1	15.1
Pre-test PSVT:R II (out of 30)	17.5	21.6
Pre-test GRE (out of 25)	8.0	9.6
Pre-test Programming (out of 12)	3.1	2.8

Overall Participant Recruitment. Participants were recruited from the same introductory CS course at the University of Michigan (EECS 183 in Winter 2020). A prerequisite for declaring a computer science major, the course is in C++ and Python. We recruited using email, forum posts, and in-class presentations. Participants had to be over 18, be able to attend at least six sessions, have no prior programming experience, and only be enrolled in the one programming course. Recruitment occurred during the second to third week of the fifteen-week semester.

Out of 736 students in CS1, 187 completed the pre-screening. Of those, 151 met the eligibility requirements, and 97 completed the pre-test. These 97 participants were then assigned randomly to one of the two Treatments (Reading Treatment or Spatial Treatment, see subsection 4.1.1).

For each two-hour session attended, participants were compensated $20 in cash ($220 for participants who attended all 11 weeks). Ultimately, 57 valid participants (29 in the Reading Treatment and 28 in the Spatial Treatment) completed at least 6/9 training sessions and took the post-test, a 58.7% study completion rate for pre-test takers. Four additional students who took the post-test and attended the requisite number of training sessions were eliminated due to invalid post-test scores; all four rushed through assessments, completing them faster than two standard deviations below the mean. We note that the majority of participant drop-out occurred in weeks four and five of the study, coinciding with the intensification of the COVID-19 pandemic (see subsection 4.3.2). Table 4.2 contains demographic breakdowns of the final 57 participants.

Phase 1 Subpopulation—fNIRS Recruitment. During the pre-screening process, participants were also asked if they were interested in attending the fNIRS scan. Of those that said yes, a subset of 40 were randomly selected to be included in the fNIRS sub-population, 20 from each Treatment group. Of these, 37 completed the fNIRS scan. All fNIRS scans were carried out in the fourth and fifth week of the semester. Thus, participants were in the first third of their first programming course while completing the scan. Participants received \$20 in compensation. In total, data from 31 participants passed our analysis quality threshold (see subsection 4.2.2). The final 31 participants (24 female, 7 male) ranged in age from 18 to 21.

Beyond the initial fNIRS scan, we also followed up with participants via email at the end of the semester, inviting them to attend the post-test (even if they had dropped out of the overall longitudinal study). Of our 31 participants, 23 participated in this written post-test. Post-test participants were compensated an additional \$20.

Ensuring No Incoming Programming Experience. During participant selection, we were keenly aware that confirming the absence of previous programming ability for a diverse population is a challenging task. To mitigate self-selection bias, we implemented checks for participant programming ability in three places: recruitment, pre-screening, and statistical validation of scores on a written programming test. During recruitment, both in-person and written, we emphasized that participants could have no prior programming experience of any kind. Prospective participants were explicitly told, in person, that even minimal practice or exposure to textual or visual programming languages (such as Scratch [264] or MIT App Inventor) counted as prior experience.

During prescreening, participants indicated if they "had any prior programming experience" with one of "Yes", "No", or "Other/unsure". We only retained participants who selected "No". The presence of the "Other/unsure" option mitigates some self-selection bias in experience reporting. We also asked potential participants to indicate concurrent and previous course enrollment from a list. Courses that contained "programming" or "code" in their syllabus or description precluded study participation. These questions eliminated 18% of pre-screening respondents.

Finally, at the same time as the demographics questionnaire, participants were given a brief programming test. The test consisted of twelve pseudocode multiple-choice questions and is a validated measure of CS1 concepts [242]. We expected low scores: novices should have no prior programming exposure (beyond the first few weeks of their current course). Indeed, we found an average score of 24.6% (random guessing yields 20%). We believe that these three mitigations help account for self-selection bias issues and give confidence that our pool contains novices (but acknowledge that the issue is reduced, rather than eliminated).

Population Demographics. We also observe that unusually for software engineering studies, a majority of our population were female. We believe that this is caused by a combination of a more gender-balanced population pool than most software engineering studies and self-selection bias. First, the course we recruited from is fairly gender balanced: around 45–50% of students identified as women in 2020, compared to 22% for CS overall at our university. Second, women often volunteer for college studies at higher rates than men and for different reasons [197]. In addition, men are more likely to have some prior programming experience [275] and thus be excluded from our study.

Population Homogeneity. We also checked for incoming skills-related differences between our Treatment populations. While we assigned participants randomly and all participants had no prior programming experience, one group could have greater reading, spatial, or programming skills than the other, yielding a potential advantage in CS1. We find no significant differences between the incoming reading scores or the incoming programming scores of the two Treatments. We do, however, find a significant difference between incoming spatial ability on both spatial assessments (PFT: $p = 0.01$, PSVT:R II: $p = 0.004$). The average raw pre-test scores are presented in Table 4.2. We account for this in our mathematical analyses of transfer training effectiveness (*e.g.*, by considering programming gains, see Figure 4.11).

Recruitment and Contextualization Summary. We recruited a substantial number of participants ($n > 50$), used multiple checks to ensure they were true novices with no prior programming experience, and achieved a reasonable demographic representation. This effort provides a solid foundation for the validity of our experimental results presented in this chapter.

4.2 Lens 2, Phase 1: Cognitive Modeling of Novice Programmers using fNIRS

We now focus on the first phase of our effort to improve software productivity by leveraging cognitive insights. For this phase, our primary question is: *Can we use neuroimaging to model novice programmer cognition?* As discussed in section 4.1, we focus on how two cognitive skills compare to novice programmer cognition: reading comprehension and spatial ability.

To better understand the cognitive processes of novice programmers, we use functional near-infrared spectroscopy (fNIRS, see subsection 2.2.1) to conduct a controlled neuroimaging study with 31 novice programmers. As described in subsection 4.1.2, all participants have no previous programming experience, and they are all enrolled in the same introductory computing class. We conduct these scans during the first third of the class. We compare participants' brain activation patterns while coding to those while reading prose or using spatial reasoning (*i.e.*, mentally rotating 3D objects). We also use a written assessment at the end of the semester to conduct a preliminary exploration of an aspect of learning, assessing if novices' brain activation patterns while coding can predict their future programming ability. For more detail on the motivation and design of this first experimental phase, see subsection 4.1.1.

We find that, for novices, coding is a working-memory-intensive task that is neurally distinct from both a spatial task and prose reading ($p < 0.01$, $q < 0.05$). Unlike previous work with experts which generally reports strong similarities between coding and reading [104, 293, 294], we observe more significant and substantial differences between coding and reading than we do for coding and spatial tasks. This indicates that novices rely heavily on visuospatial cognitive processes while coding. We also find that particular activation patterns at the start of a course can predict how well students perform on a final programming assessment 11 weeks later; the *less* similar the activation patterns are between coding and mental rotation, the *better* an individual performs ($r = 0.48$, $p = 0.006$). This may indicate that novices who use more problem-solving intensive strategies at the beginning of the semester (*i.e.*, find programming more challenging) make less progress.

In the rest of this section, we introduce our phase 1 fNIRS setup and stimuli design (subsection 4.2.1), analysis methodology (subsection 4.2.2), approach validation (subsection 4.2.3), and experimental results (subsection 4.2.4).

4.2.1 fNIRS Experimental Setup and Design

We now present our experimental design for understanding the neurological basis for novice programmers.[1] This section focuses on experimental design related to the first phase of Lens 2. For an overview of how this design fits into the full longitudinal transfer training study (phase 2), see subsection 4.1.1.

Our fNIRS experiment was conducted in two parts: an initial fNIRS scan and a written follow-up assessment.[2] During the fNIRS scan, participants were shown three types of stimuli: code comprehension, mental rotation, and prose reading. During the written post-test, participants completed a validated language-agnostic programming assessment.

All 31 participants were enrolled in the same 15-week introductory programming course. For more information on participant recruitment, see subsection 4.1.2. The initial fNIRS scans were held during the first third of the semester while the written post-test was held during the last week of the semester. This design allows the controlled exploration of the relationships and contrasts between reading, coding, and spatial reasoning for novice programmers. It also opens a preliminary investigation of neurological factors that might be predictive for programming. In the rest of this section, we outline our fNIRS data collection, experimental setup, and stimuli (subsubsection 4.2.1.1 and subsubsection 4.2.1.2); and describe our follow-up programming assessment (subsubsection 4.2.1.3).

[1] A replication package containing all of our fNIRS experimental materials can be found at `https://github.com/CelloCorgi/ICSE_fNIRS2021`.

[2] This is the same post-test described in the transfer training longitudinal study in phase 2, see section 4.3.

4.2.1.1 fNIRS Data Collection and Setup

Each of the 31 participants' fNIRS data was collected during a single session which lasted 1.5 hours. First, participants gave informed consent and filled out a short demographic survey. Next, participants watched a training video preparing them for the scan, a 30–45-minute process that involved fitting each participant for the fNIRS cap by moving hair to optimize sensor contact with the scalp, and thus, signal quality.

During the fNIRS scan, the participant sat in a chair facing a monitor wearing the fNIRS cap. The room was kept dim to reduce the amount of ambient light that could interfere with the fNIRS data collection. Participants were also instructed to stay as still as possible. Each participant was shown 90 stimuli: 30 mental rotation stimuli, 30 reading-based stimuli, and 30 coding-based stimuli. All stimuli asked the participant to choose one of two answers. Participants indicated their answers by pressing a corresponding key on a standard keyboard. For each stimulus, participants had up to 30 seconds to respond. The 90 stimuli were in randomized order and were broken into three blocks of 30 questions, each containing 10 stimuli of each type. Between stimuli, participants were shown a fixation cross for 2–10 seconds. Between blocks, participants had an optional longer break to rest and/or drink water.

We now present technical information about our fNIRS device and cap. We used a CW6 fNIRS system (TECHEN, Milford, MA, USA) with 690 nm and 830 nm wavelengths. It has fiber-optic cables which transmit light from the device to sensors connected to the participant's cap. These sensors are either transmitters that emit light or receivers that detect light. As a result, fNIRS collects a participant's hemodynamic response only along a set of pre-defined channels between transmitters and receivers. The location, number, and coverage of these channels is determined by the cap design. We used a probe configuration similar to that used in Huang *et al.* [153] with dense coverage of transmitter-receiver pairs in the occipital, frontotemporal, and frontal regions. In total, 15 receiver and 30 transmitter fibers were used, yielding 55 channels from which data were collected, covering Brodmann areas 6–9, 17–19, 21, 22, 39–41, and 44–47. The data from these channels were then analyzed using both NIRS Brain AnalyzIR Toolbox [279] and custom scripts written in MATLAB. A picture of our cap coverage is in Figure 4.2; it includes areas identified in previous mental rotation studies as well as key language areas.

Our number of channels is higher compared to many previously published fNIRS papers in software engineering, giving broader coverage [155, 229]. We used two cap sizes to accommodate different head circumferences (58 cm and 60 cm). Signals were sampled at 50 Hz.

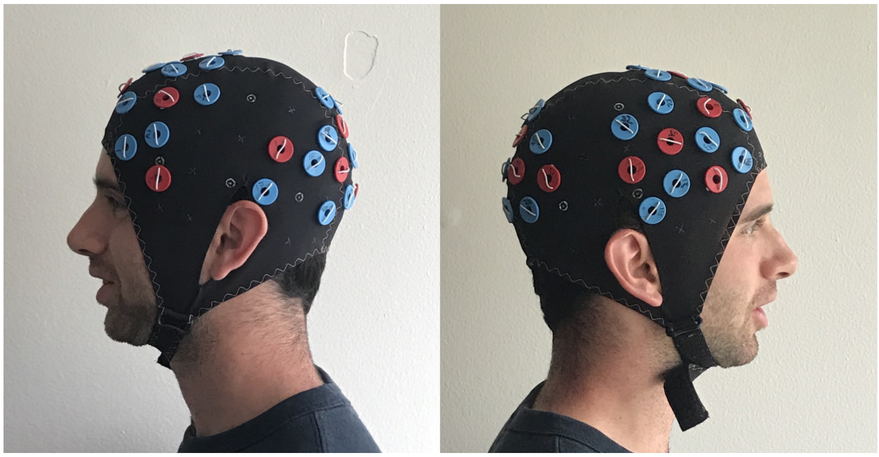
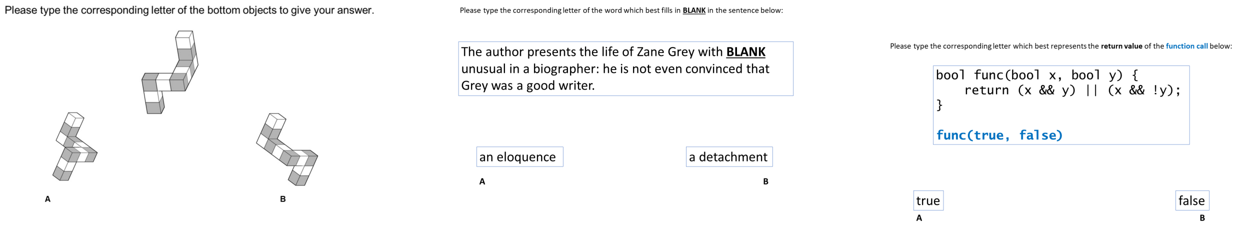


Figure 4.2: fNIRS Cap Photograph: fNIRS cap used in our experiments. Red circles are light transmitters while blue circles are light receivers. fNIRS is only able to observe brain activation on channels between nearby transmitters and receivers.



Please type the corresponding letter of the bottom objects to give your answer.

Please type the corresponding letter of the word which best fills in **BLANK** in the sentence below:

The author presents the life of Zane Grey with **BLANK** unusual in a biographer: he is not even convinced that Grey was a good writer.

Please type the corresponding letter which best represents the **return value** of the **function call** below:

```
bool func(bool x, bool y) {
    return (x && y) || (x && !y);
}

func(true, false)
```

(a) Mental Rotation: correct answer is “A”.

(b) Reading: correct answer is “B”.

(c) Code: correct answer is “A”.

Figure 4.3: Example fNIRS Stimuli: for mental rotation (a), reading (b), and coding (c) tasks.

4.2.1.2 fNIRS Stimuli

We now turn to a description of the content of our fNIRS stimuli. As mentioned in subsubsection 4.2.1.1, participants were shown three categories of stimuli: mental rotation, reading-based stimuli, and coding-based stimuli. All three types asked the participant to choose between two answers *A* and *B*.

We use mental rotation stimuli adapted from Peters and Battista’s Mental Rotation Stimulus Library [248]. These stimuli are designed to induce brain activation associated with mentally rotating 3D shapes, one facet of visuospatial cognition (See subsection 2.2.6). In each mental rotation stimulus, the participant was asked to choose which of two objects was a possible rotation of a third object (see Figure 4.3a for an example). To admit direct comparison with previous work, our mental rotation stimuli are the same stimuli used by Huang *et al.* when analyzing data structure manipulation brain activation with experienced software developers [153].

For the reading stimuli, we use sentence completion tasks adapted from official *Graduate Record*

Examination practice exam questions [98], an assessment required for admission to many graduate programs. For each stimulus, participants were asked to read a sentence and choose the appropriate word or phrase to fill a blank (see Figure 4.3b for an example).

For the coding stimuli, we created a corpus of short code snippets that use constructs familiar to introductory computing students such as boolean logic, while loops, for loops, and arrays. These are all early core concepts in most introductory curricula [309] and are also covered in our institution's CS1. Unfortunately, we were not able to directly reuse coding stimuli from previous neuroimaging studies as they are generally geared toward expert programmers and thus contain constructs unfamiliar to novices. For the array-based questions, however, we were able to adapt stimuli created by Huang *et al.* [153]. For each coding stimulus, participants were asked to choose either the correct output or return value of a short code snippet (see Figure 4.3c for an example).

4.2.1.3 Followup Programming Assessment

At the end of the full semester-long longitudinal transfer training study (10–12 weeks after the fNIRS scans, see subsection 4.1.1), participants who remained in the study completed a written programming test. Of our 31 participants, 23 participated in this written post-test. Along with the programming test, participants also completed a battery of cognitive, and behavioral assessments. For the programming assessment, we used the *Second CS1 Assessment* (SCS1), a validated language-agnostic measure of CS1 programming ability [242]. As discussed in more detail in subsection 4.3.3, the SCS1 contains 27 multiple-choice questions and takes one hour. It covers Boolean logic, while loops, for loops, arrays, if statements, functions, and recursion. There are three types of questions: definition questions, code tracing questions, and code replacement questions. Due to COVID-19, we were unable to hold the test in person. Rather, participants completed an online version of the SCS1 over a proctored video call. Responses were then checked for timing-based anomalies to ensure participants did not rush through the test.

4.2.2 fNIRS Analysis Methodology

We now go over our methodology for analyzing the fNIRS data. Broadly, there are three stages in our analysis pipeline: preprocessing, individual modeling, and group-level modeling.

1) Preprocessing: The raw data, in the form of light intensity values, were converted into optical density data by calculating the fluctuations in light absorption by the presence of either oxygenated (HbO) or deoxygenated (HbR) blood. The optical density data were then converted into an HbO/HbR signal using the Modified Beer-Lambert law. We ran a general linear model (GLM) with pre-whitening and robust least squares to fit the data [22].

2) Individual Subject Modeling: After the hemodynamic response was modeled for each subject, quality control checks were implemented to limit the amount of noise in the group-level model. The signal-to-noise ratio, anticorrelation of HbO and HbR, and brain-activation plots were considered when deciding whether to exclude individual blocks or whole participants. The signal-to-noise ratio was calculated as a ratio between the absolute signal mean and standard deviation, with a threshold set at 0.9. As a result, 16 blocks were excluded from further analysis (see subsubsection 4.2.1.1 for a discussion of our experimental block setup). In an ideal hemodynamic response function, the levels of HbO will increase as the levels of HbR decrease and vice-versa [75], so the correlation between these two levels should be negative. Thirteen additional blocks that did not show this pattern were excluded from further analysis.

Next, the brain activations estimated by the GLM for *All Conditions* $>$ *Rest* were plotted onto brain models using a photogrammetry-based localization method [151], at which point the loci of activity could be examined and scrutinized. We expected activity in the visual cortex (as this is a visual experiment), as well as in the inferior frontal gyrus during the reading task (as this is a well-documented region for language processing). Thirty-seven blocks that did not pass this activation pattern check were excluded. These criteria were not mutually exclusive. In total, 59 blocks from 31 participants were included in the group-level modeling.

3) Group Modeling: We used a linear mixed effects model for group-level analysis, contrasting *Task* $>$ *Baseline* activations to estimate task-related brain activations and behavior correlations. Lastly, we applied a false-discovery rate (FDR) threshold correction ($q < 0.05$) to account for the multiple-comparison issue.

4.2.3 fNIRS Validation

In this section, we validate that the brain activation patterns we observe during mental rotation and reading align with those established by previous work. Specifically, we present our results for the contrasts *Mental Rotation* $>$ *Rest* and *Reading* $>$ *Rest*. We provide brain activation visualizations for *Mental Rotation* $>$ *Rest* and *Reading* $>$ *Rest* in Figures 4.4 and 4.5 respectively. We also provide a tabular view of our results with specific t-values in Table 4.3 (see subsection 2.2.2 for an overview of t-values and $A > B$ notation).

In *Mental Rotation* $>$ *Rest*, we observe significant bilateral activation in the occipital and parietal lobes (BAs 7, 17, 18, 19, 39). The occipital lobe (BAs 17, 18, 19) is associated with the visual cortex and is responsible for tasks such as image and pattern recognition, both visuospatial processes. The parietal activation (BAs 7, 39) is also localized in regions associated with spatial tasks including spatial reasoning and mental rotation [69, 76].

Our results are therefore consistent with previous mental rotation neuroimaging; in his meta-review of mental rotation imaging studies, Zacks also identified BAs 7, 17, 19, and 39 as active

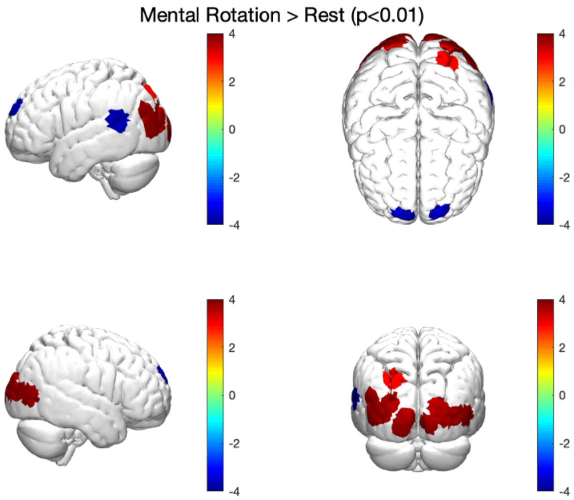


Figure 4.4: Baseline Mental Rotation Activation: Red indicates regions more activated during the mental rotation task while blue indicates regions more activated during rest. Note that all significant mental rotation activation is located in the back of the head in the parietal and occipital lobes.

during mental rotation tasks [356]. We do not observe significant supplementary motor cortex activation (BA 6), another area often active during mental rotation tasks. This is not too surprising, however, as the supplementary motor cortex is most strongly activated in tasks that encourage participants to use motor simulation strategies [356]; we did not prompt participants to use such strategies in our experiment.

In *Reading* > *Rest*, we observe significant occipital, parietal, and prefrontal activation lateralized to the left hemisphere, aligning with previous work. We observe significant activation in both Broca's and Wernicke's areas, widely considered two of the most important language areas [251]. We also observe significant activation in the left dorsolateral prefrontal cortex (See subsection 2.2.2, BA 46), a region associated with attention and working memory. Regarding the occipital activation, while we observe some bilateral activation, significant activation is substantially more widespread in the left hemisphere. The left occipital cortex has been found to correspond with word and letter-specific pattern recognition [68].

Our activations for rotation and reading align with previous work: significant occipital and parietal activation for rotation, and occipital and prefrontal activation for reading, including Broca's and Wernicke's areas. Rotation activation is bilateral while reading is primarily in the left hemisphere. This validation gives confidence for both construct and internal validity (*i.e.*, that our protocol measures what we think it measures and does so correctly).

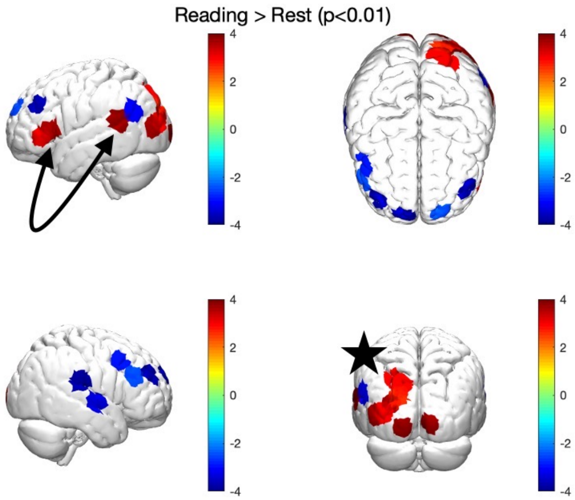


Figure 4.5: Baseline Reading Activation: Red indicates regions more activated during the reading task while blue is more activated during rest. Note the reading activation in Broca's and Wernicke's areas (the double arrow) as well as the left-lateralized occipital and parietal activation (the star).

4.2.4 fNIRS Experimental Results

We now present the results of our experiment probing the neurological connections between reading, mental rotation, and programming for novice programmers (See subsection 2.2.2 for an overview of relevant notation and vocabulary (*e.g.* $A > B$). We focus our analysis around three research questions:

- RQ1—Programming Activation: What areas of the brain activate when novice software engineers program?
- RQ2—Comparative Activation: How does the coding brain activation of novices compare to their brain activation during mental rotation and during reading?
- RQ3—Prediction: Are there connections between coding brain activation patterns at the beginning of CS1 and their programming performance at the end of the course?

4.2.4.1 fNIRS Experiment: RQ1—Programming Activation

To determine which areas of the brain activate when novice software engineers program, we present the results of the contrast *Coding* $>$ *Rest*. That is, we test which brain areas significantly distinguish programming from a resting state ($p < 0.01$, $q < 0.05$). We also discuss the functionality of the distinguishing brain regions. Figure 4.6 contains a brain activation visualization for *Code* $>$ *Rest*, and we provide a tabular view of our results with specific t-values in Table 4.3.

Brain Region	*Rot.* > *Rest*	*Read* > *Rest*	*Code* > *Rest*	*Code* > *Rot.*	*Code* > *Read*
Frontal Cortex					
Left DLPFC (BA 46)		3.32 – 3.32	3.30 – 3.95	2.78 – 2.96	3.76 – 3.76
Right DLPFC (BA 46)		−2.62 – −2.62	2.78 – 3.27	3.79 – 3.79	2.99 – 4.72
Broca's Area (Left BAs 44 and 45)		3.76 – 3.76			−3.54 – −3.54
IFG (Right BAs 44 and 45)			2.78 – 3.27	3.11 – 3.79	2.99 – 2.99
Left Supple. Motor Cortex (BA 6)			3.45 – 3.45	3.45 – 3.45	3.22 – 5.32
Right Premotor Cortex (BA 6)		−2.98 – −2.98			
Left BA 8		−4.07 – −4.07	3.30 – 3.30	2.96 – 2.96	−4.93 – 3.93
Right BA 8		−3.95 – −2.62			2.67 – 4.72
Left BA 9	−6.26 – −6.26	−4.07 – −2.63	−7.49 – −7.49		−5.10 – 3.93
Right BA 9	−7.46 – −7.46	−6.69 – −2.67	−6.13 – −6.13		2.67 – 4.72
Temporal Cortex					
Wernicke's Area (Left BAs 22, 40)	−6.44 – −6.44	5.91 – 5.91		6.51 – 6.51	−4.77 – −4.77
Right BA 21					2.79 – 2.79
Right BA 22		−3.65 – −3.65			3.15 – 3.15
Right Auditory Cortex (BA 41)		−5.81 – −5.81			6.70 – 6.70
Parietal Cortex					
Left BA 7	3.38 – 3.38	3.15 – 3.17	3.64 – 3.64	2.60 – 2.60	
Left Angular Gyrus (BA 39)	4.95 – 4.95	−3.19 – −3.19	−2.99 – 5.28		6.86 – 6.86
Right Angular Gyrus (BA 39)	6.83 – 6.83		3.68 – 5.15	−3.49 – 3.98	4.29 – 4.29
Occipital Cortex					
Left BA 17	5.78 – 6.36	8.11 – 8.11	2.75 – 2.83	−3.26 – −3.23	−4.48 – 4.78
Right BA 17	4.17 – 4.17	4.29 – 4.29		−2.84 – −2.84	−2.91 – −2.91
Left BA 18	4.69 – 6.36	3.06 – 8.11	2.75 – 3.84	−3.23 – −3.23	−4.48 – −4.48
Right BA 18	4.18 – 6.48	4.29 – 4.29	5.95 – 5.91	−2.84 – −2.84	−2.91 – 5.26
Left BA 19	4.52 – 7.76	2.71 – 4.16	3.84 – 5.46		
Right BA 19	3.95 – 6.83		3.68 – 5.95	−3.49 – −3.49	3.54 – 5.65

Table 4.3: Overall fNIRS t-value Statistics: t-value statistics for *Mental Rotation (Rot.)* > *Rest*, *Reading* > *Rest*, *Code* > *Rest*, *Code* > *Mental Rotation (Rot.)*, and *Code* > *Reading*. All results are significant ($p < 0.01$) and pass our false discovery threshold ($q < 0.05$). Blank cells indicate no significant effect was found in that region. Results closer to +8 or -8 indicate more significant activation or deactivation. We highlight results with t-values less than -5 or greater than +5.

While coding, novices exhibit significant occipital activation (BAs 17, 18, 19). While bilateral, we observe somewhat stronger right hemisphere activity. Functionally, the occipital cortex is associated with visual processing, and it includes areas such as the primary visual cortex (BA 17) and visual association area (BA 18). This occipital activation is the strongest activation we observe in *Coding* > *Rest*, with three out of the five channels with t-values greater than five.

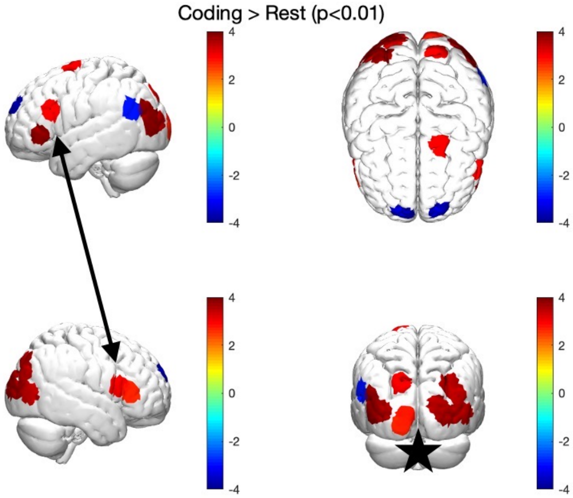


Figure 4.6: Baseline Coding Brain Activation: Red indicates regions more activated during the coding task while blue indicates regions more activated during rest. Note the widespread bilateral activation in both the DLPFC (the arrows) and the occipital and posterior parietal cortices (the star).

Beyond occipital activation, we also observe significant posterior parietal activation, primarily in the angular gyrus (BA 39). This activation is bilateral: the other two channels with t-values greater than five both cover BA 39, one in each hemisphere. In the left hemisphere, the angular gyrus is important for language-related tasks [48]. Some researchers also include BA 39 in Wernicke's area, one of the two main brain cortex regions associated with natural language processing. However, we do not observe activation while coding in the regions most commonly associated with Wernicke's area: BAs 22 and 40. The angular gyrus is also strongly associated with spatial cognition tasks including spatial orientation (*e.g.*, distinguishing left from right), spatial attention, numerical computation, and mental rotation [285, 356]. While the spatial functionality of the angular gyrus is bilateral, many spatial tasks, including mental rotation, are concentrated in the right hemisphere [285]. Therefore, the bilateral activation of the angular gyrus indicates that *novices use both language and spatial cognitive processes while programming*.

We also observe significant activation in the frontal cortex. Specifically, we observe bilateral activation in the DLPFC (BA 46) and activation in the left superior premotor cortex (BA 6). The premotor cortex is associated with motor processing, and it has also been found to activate during visuospatial tasks including mental rotation [356]. The DLPFC is associated with working memory; lower activation in this region corresponds with worse performance on working-memory intensive tasks such as complex problem solving [21]. The significant bilateral DLPFC activation, therefore, indicates that *novices find programming a challenging and working- memory-intensive task*.

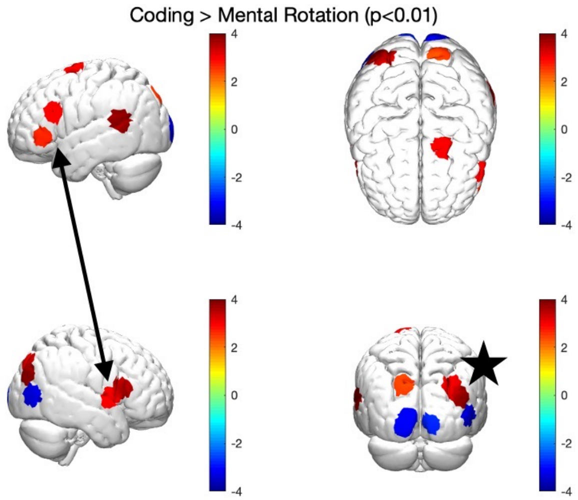


Figure 4.7: Brain Activation Contrast between Programming and Mental Rotation: Red indicates regions more activated during coding while blue indicates regions more activated during mental rotation. Note that compared to mental rotation, coding has stronger bilateral frontal activation (the arrow) and right posterior parietal activation in the angular gyrus (the star).

> Novices engage brain regions associated with language and spatial cognition, as well as regions associated with increased demand for attention and executive function (via neural activity in *Coding* > *Rest*, $p < 0.01$, $q < 0.05$).

4.2.4.2 fNIRS Experiment: RQ2—Significant Comparative Activation

We now compare novices' coding brain activation to their activation during mental rotation and reading by presenting our findings for the contrasts *Coding* > *Mental Rotation* and *Coding* > *Reading*. We provide brain activation visualizations for *Coding* > *Mental Rotation* and *Coding* > *Reading* in Figure 4.7 and Figure 4.8, and we provide a tabular view of our results with specific t-values in Table 4.3. Our high-level finding is that while coding, mental rotation, and reading are all neurally distinct tasks, *we observe more substantial differences between coding and reading than we do between coding and mental rotation*.

For *Coding* > *Mental Rotation*, we observe several significant differences in activation. First, when coding, participants exhibit more bilateral frontal activation than while mentally rotating objects. This comparative activation is in the DLPFC and the premotor cortex (BAs 6 and 46), areas commonly associated with working memory and spatial manipulation [21, 356]. Similarly, we observe comparatively high coding activation in the right angular gyrus, a region also connected with spatial reasoning [285]. We find it intriguing that many of the areas with more activity for coding

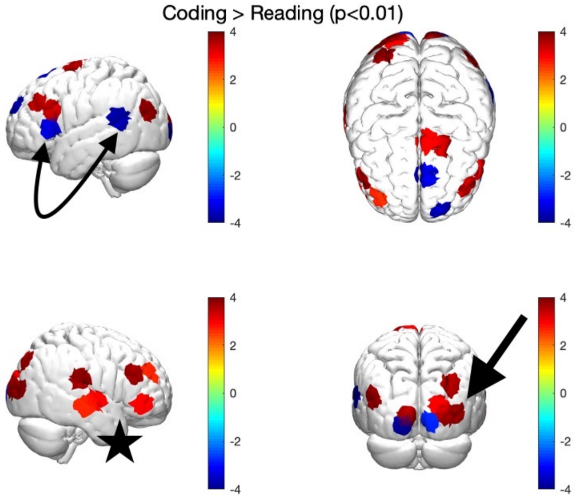


Figure 4.8: Brain Activation Contrast between Programming and Reading: Red indicates regions more activated during coding while blue indicates regions more activated during reading. Note that reading has comparatively more activation in Broca's and Wernicke's areas (the double arrow), while coding has substantially more right frontal activation (the star) and more right-lateralized occipital/parietal activation (the single arrow).

than for mental rotation are associated with spatial reasoning, ostensibly the process measured by the mental rotation stimuli. This may imply that coding is a challenging task that is actually *more* spatially intensive for novices than simple mental rotation.

When considering a contrast $A > B$, it is important to distinguish between a comparison that is positive because A is greater than B (called a "true activation") vs. one that is positive only because B is negative (a "strong deactivation"). All of the significant *Coding* > *Mental Rotation* differences discussed so far are channels with either positive activation or no significant activation in *Rotation* > *Rest*. Thus, they are all true differences between coding and mental rotation. On the other hand, while we also observe a very strong comparative activation ($t = 6.51$) in *Coding* > *Mental Rotation* in Wernicke's area, this is a facet of the same channel's strong deactivation in *Rotation* > *Rest* instead of a true activation.

For *Coding* > *Reading*, we also observe significant differences in activation: coding has comparatively stronger activation throughout the right hemisphere and in the premotor cortex while reading has stronger activation in Broca's and Wernicke's areas. All of the significant differences in the left hemisphere including Broca's and Wernicke's areas are true contrasts; that is, none of them are caused by a significant deactivation in one of the *Task* > *Rest* comparisons. In the right hemisphere, several of the apparent activations for coding are caused by strong deactivation in *Reading* > *Rest*. Even so, we observe true comparative right-hemisphere activations: the right occipital, angular gyrus, and DLPFC are all more activated while coding than while reading.

While we observe that Coding is neurologically distinct from both mental rotation and reading, we also observe more substantial differences in *Coding* $>$ *Reading* than in *Coding* $>$ *Mental Rotation*. In *Coding* $>$ *Reading*, there are four channels with t-values greater than $+5$ or less than -5, while in *Coding* $>$ *Mental Rotation*, there is only one (see Table 4.3). Furthermore, the strong channel difference in *Coding* $>$ *Mental Rotation* is not particularly compelling as it is caused by deactivation. Taken together with the previous functional analysis, this trend hints that for novices, coding is a more spatially-based and less language-based cognitive process.

> We find that, for novices, coding is neurally distinct from both reading and spatial reasoning. Coding engages regions associated with working memory more than either reading or rotation, indicating that programming is a more cognitively challenging task. However, we observe more significant substantial differences in *Coding* $>$ *Reading* than we do in *Coding* $>$ *Rotation*: novices may rely heavily on spatial reasoning while coding.

4.2.4.3 fNIRS Experiment: RQ3—Prediction

We now turn to a transitional analysis of connections between observed brain activation patterns and participants' final programming assessment scores (see our longitudinal training study, section 4.3).[3] To do so, we use *Representational Similarity Analysis* (RSA), a common Psychology approach, to correlate brain activity interactions with scores on the programming post-test (see Kriegeskorte *et al.* [179] for an introduction to RSA). We test if the brain activation similarity between mental rotation and coding (*Mental Rotation* × *Coding*) or the brain activation similarity between reading and coding (*Reading* × *Coding*) are correlated with SCS1 scores. We calculate these correlations for the right hemisphere, left hemisphere, right hemisphere frontal, left hemisphere frontal, and occipital regions for a total of five statistical tests per hypothesis and ten total test–hypothesis pairs.

We find a correlation that remains significant after applying the Bonferroni correction for multiple comparisons [63]. Specifically, we find a significant medium negative correlation between (*Mental Rotation* × *Coding*) in the right frontal region and programming post-test score ($r = -0.48$, $p = 0.0006$, adjusted Bonferroni $p = 0.006$): the *less similar* the neural activation patterns for coding and rotation, the *better* the final programming assessment outcome.

Notice that coding also elicited stronger right frontal activation than the mental rotation task. It is therefore possible that the more individuals engage the right frontal, in a way that is dissimilar from its engagement during spatial processing, the less progress they make over the course of the semester. The hypothesis is similar to the observation that the more dissimilar the activity

[3] We do not account for training Treatment group (see subsection 4.1.1) in this analysis due to participant drop-out decreasing our sample size in each Treatment.

between novice readers and their language, the less progress they make in learning to read [210]. In the psychology literature, one common approach for testing such a hypothesis would be to investigate correlations between significant channel activations and response time. However, while in some settings response time can be used as a proxy for difficulty, our coding stimuli were not designed with that consideration; we do not find a statistically significant correlation ($r = -0.095$, $p = 0.4866$) and lack enough information to either substantiate or refute that hypothesis.

In a transitional analysis relating initial brain activation patterns and post-test programming scores, we find that *less similar* patterns of activation for coding and mental rotation in the right frontal hemisphere at the start of the semester predicts better outcomes on the end-of-semester final programming assessment ($r = -0.482$, $p = 0.006$).

4.2.5 fNIRS Results Summary

In a study of 31 participants, we use fNIRS to compare the neural activity for introductory programming, reading, and spatial reasoning tasks in a controlled, contrast-based experiment. We find that **all three tasks—coding, prose reading, and mental rotation—are mentally distinct for novices**. This clarifies previous findings that they may be more similar in experts [104] or for complex data structures [153]. However, while those tasks are neurally distinct, we find **more significant and substantial differences between prose and coding than between mental rotation and coding**. Intriguingly, we find generally more activation in areas of the brain associated with spatial ability and task difficulty while coding compared to that reported studies with more expert developers. Finally, in an exploratory analysis, we find that **certain patterns of neural activity at the start of the semester are predictive of end-of-semester outcomes**, opening the door for future experiments to model such phenomena more directly. To the best of our knowledge, this is the first study to focus specifically on novice programmers and to make use of a time-delayed outcomes assessment. These findings both elaborate on previous results (*e.g.*, relating expertise to a similarity between coding and prose reading) and also provide a new understanding of the cognitive processes underlying novice programming.

4.2.6 fNIRS Experiment: Limitations and Threats to Validity

Our phase 1 fNIRS results may not generalize. We consider a number of threats to validity and discuss how our approach mitigates them.

We note that fNIRS experiments are dependent on transmitters and receivers for infrared light (see subsection 2.2.1): if no pairs are present for a relevant portion of the brain, activity there cannot be measured. We mitigate this potential source of false negatives in two ways. First, we adapt a validated cap design proposed by Huang *et al.* for use in software engineering and spatial ability studies [153]. Second, information from other medical imaging approaches, such as fMRI, which do not depend on transmitter placement, is used to determine which brain regions to measure [104, 181, 293].

We also consider issues of construct validity: are we measuring what we claim to be measuring (*e.g.*, spatial ability, introductory programming, *etc.*)? While there are multiple aspects to spatial ability, we use mental rotation, an established paradigm for investigating spatial ability, both in psychology in general [69, 76, 138] and in computer science in particular [37, 153]. For introductory programming, we make use of the SCS1, a validated assessment [242].

Finally, all of our subjects are students at the same large US university. This aspect of participation selection may limit the generality of our results to other populations.

4.3 Lens 2, Phase 2: Cognitive Transfer Training

We now turn to the second phase of our investigation into improving novices' software productivity by leveraging cognitive insights. For this phase, our primary question is: *Can supplementary cognitive training transfer to improved programming outcomes?* As discussed in subsection 4.1.1, we focus on the same two cognitive skills targeted by the fNIRS experiment in phase 1: reading comprehension and spatial ability.

In this second phase of Lens 2, we propose and design a novel CS-focused technical reading training, and we use a semester-long randomized trial with 57 participants to compare the efficacy of two CS1 interventions: our novel technical reading training and an established standardized spatial ability training. For more detail on our motivation and design for this transfer training experiment, see section 4.1. Our technical reading training is CS-focused to better cover CS-specific reading tasks, such as reading API documentation. Reading transfer training interventions have been carried out in other fields, but they are rare for STEM education. To the best of our knowledge, we are the first to test for transfer between technical reading ability and computer science.

Overall, we find that reading training participants had significantly larger programming gains than those in the spatial training ($p = 0.02$). We also find that the reading training's benefit is most significant for tasks that require tracing code. Furthermore, we observe that reading ability tends to correlate more strongly with final programming ability than spatial ability. The contributions of this second phase of this dissertation lens include:

- A novel CS-focused technical reading intervention.
- Results of an 11-week randomized control trial comparing the effects of our reading training and an established spatial training on novice programming ability.

- The finding that reading training participants exhibit significantly larger CS1 programming gains than spatial participants, primarily during code-tracing tasks.

- A replication of prior findings that both spatial ability and reading ability correlate with programming performance.

- An analysis of cognitive and demographic features predictive of CS1 programming ability.

- A discussion of the implications of our findings on CS1 education and software engineering in general.

In the rest of this section we detail the training materials used in each intervention (subsubsection 4.3.1.1), detail our experimental design (subsection 4.3.2) and assessment stimuli (subsection 4.3.3), validate our training curricula (subsection 4.3.4), and overview our transfer training experimental results (subsection 4.3.5).

4.3.1 Transfer Training Materials

In this section, we discuss the structure and content of the two transfer training interventions. In subsubsection 4.3.1.1, we present our novel CS-focused technical reading intervention (Reading Treatment), and in subsubsection 4.3.1.2 we describe the standardized spatial training (Spatial Treatment).

4.3.1.1 Technical Reading Training

For the Reading Treatment, we designed a novel computer science-focused technical reading curriculum. We had several key design goals. First, for direct comparison with the Spatial Treatment (see subsubsection 4.3.1.2), we required that our Reading Treatment consist of nine two-hour weekly training sessions and include workbook practice problems, group work, and physical props—snap-blocks for the Spatial Treatment and vocabulary flashcards for the Reading Treatment. We also required that the Reading Treatment use active learning techniques such as think-pair-share, group work, and in-class practice problems, as these have been found to enhance learning [252]. Finally, we required that our training be CS-focused to increase participant engagement and programming relevance. To accomplish this, we integrated computer science research papers and API documentation into the course.

Overall Session Structure. Each session followed the same general lesson plan: a 20-minute ice-breaker and workbook warm-up, general (not CS-specific) vocabulary, a short lecture covering a technical reading strategy, and finally reading and analysis practice. This practice typically accounted for half of each training session.

Warm-up and Vocabulary. The initial ice-breaker and workbook warm-up served to get students on task and excited for the rest of the session [91]. For the workbook, students completed pages from *Reading Comprehension Skills and Strategies: Level 8*, a commercially available workbook [276]. We included vocabulary practice due to the established benefit of emphasizing vocabulary when teaching reading [118]. Vocabulary words were selected from a curated list of common GRE words. Generally, we emphasized words that we believed were prevalent in technical or scientific writing. For the lecture portion, we focused on teaching established technical reading strategies.

Lecture Topic Selection. During the lecture portion, we covered various technical reading strategies. The majority of these strategies focused on using structural cues to quickly and accurately scan texts to retrieve and understand key points. Topics included focusing on outlines when reading to improve comprehension, understanding figures and charts in scientific writing, and strategies for understanding persuasive technical proposals (*e.g.*, Heilmeier's Catechism). Our focus on teaching structural cues and patterns to efficiently skim texts was motivated by findings that experienced programmers tend to read code non-linearly, focusing on high-level features [52, 267].

CS-Paper Selection. All but two of the reading and analysis practice sections involved computer science research papers selected using several criteria. First, papers were selected from the reading lists of various software engineering and computer science education courses at a top-tier public university. All papers were published in reputable computer science journals or conferences. Second, to ensure the material was understandable for CS1 students, we deliberately selected accessible research papers that were interesting to first-year students; in particular, many readings involved computer science education so the content was relatable. Finally, for more complex readings, students were only asked to read a curated subset of the paper such as the introduction, related work, and conclusion. We asked students to summarize readings and to write and share short reading responses; previous work has found that asking learners to put technical information in their own words results in increased comprehension and broader learning for transfer [212]. The other two sessions contained general review (roots and affixes) or API documentation reading strategies.

Training Materials. For replication purposes, we have made our reading training materials publicly available in our replication package.[4] This replication package contains session slide decks and a list of all lecture topics and all CS papers covered. We also provide study recruitment materials and data analysis scripts.

[4] Replication package: `https://github.com/CelloCorgi/FSE2021_To_Read_or_to_Rotate`.

4.3.1.2 Spatial Training

For our Spatial Treatment, we used a spatial ability course developed at Michigan Technical University by Sorby and Baartmans [301]. Intended to help incoming engineers improve their spatial ability, Sorby and Baartmans's curriculum has been shown to improve not only students' spatial skills [302], but also their grades in several key engineering courses [300]. The course consists of 10 modules and covers the following topics: surfaces and solids of revolution, combining solid objects, isomorphic sketching, orthographic sketching, orthographic projections, flat pattern folding, 3D shape rotation around single and multiple axes, object reflection or symmetry, and mental cutting. Teaching materials were provided, including training videos, software, and individual practice workbooks.

We taught the material in Sorby and Baartmans's curriculum in nine two-hour weekly sessions. Each session involved showing the provided lecture videos, working through the software in pairs, and completing workbook problems. We also gave each participant snap-blocks, as recommended by the provided training materials, to help them visualize challenging workbook problems. To ensure student engagement, we held the training sessions in person on campus until COVID-19 necessitated remote instruction. We discuss the impact of COVID-19 on our experiment in more detail in subsection 4.3.2.

4.3.2 Transfer Training: Experimental Design

In this section, we cover our transfer training experimental design and protocol.

Design Overview. To compare the effects of our CS-focused technical reading training with the standardized spatial training, we conducted an 11-week controlled study with 57 participants. As discussed in subsection 4.1.1, participants were enrolled in the same CS1 course at the University of Michigan during Winter 2020, and they were randomly assigned to either the Spatial Treatment or the Reading Treatment. These two treatments required participants to make the same weekly time commitment for the same number of weeks, and all participants were compensated the same monetary amount. Participants were not aware of their Treatment group when they took the pre-test. We measured training effectiveness using a validated CS1 assessment [242]. For more detail on our recruitment process and population demographics, see subsection 4.1.2.

Both Treatments had nine two-hour in-person weekly training sessions with the same instructors. The study started in the fourth week of term and lasted until the end of the semester. Participants had to complete at least 6/9 training sessions to be included in our analysis. Participants also attended two additional two-hour assessment sessions, a pre-test the week before the start of the study and a post-test the week after the last training session. At the two assessments, participants took spatial ability, reading ability, and programming ability tests. They also completed a demographics questionnaire during the pre-test and a qualitative survey during the post-test. We describe the assessments in detail in subsection 4.3.3.

Design Motivation. An alternate study design for evaluating our reading training's effectiveness would have been to compare the programming gains of participants to those in a "no treatment" group (*i.e.*, students in the same CS1 course who were not in the reading treatment). We choose instead to compare two interventions: our CS-focused technical reading training and an extant spatial training of the same duration and intensity. The main factor behind this choice was to mitigate self-selection bias. In studies of educational interventions with supplementary instruction, self-selection into the treatment group can introduce significant bias, even after controlling for demographic factors [81, 186]. Thus, we designed our study such that participants first self-selected into the study, and only then were they randomly assigned to a treatment group.

We also considered comparing both interventions against a weaker placebo course as a control, but decided not to as participant drop-out rate (and thus our potential statistical power) was a real concern. We hypothesized that drop-out would be significantly higher with any control appearing unrelated to CS. Finally, we note that we emailed a post-test invitation to all participants who were pre-tested but later dropped out of the study to form a potential additional no-treatment comparison group. Unfortunately, likely due to COVID-19 and exam timing, only five participants responded. While we observed lower programming gains in this group than those in either treatment, five responses were not enough to support meaningful analysis.

COVID-19 Adjustments. Our experiment was concurrent with the COVID-19 pandemic (Winter 2020), and our institution suspended in-person research activities during the study's fifth week. Therefore, we moved the remainder of the training sessions and the post-test to a virtual format; students attended a proctored two-hour video conference each week instead of the in-person training session. They then emailed scans of their completed work to the research team. For the post-test, the participants took online Qualtrics versions of each assessment. Like the virtual sessions, the virtual post-test was proctored over a video conference. Beyond session format, the COVID-19 epidemic affected study retention. During the last week with full in-person sessions before university-wide COVID-19 measures, the study had 82 attendees, while in the first full week of virtual sessions, the study had only 61 participants, a 25.6% drop.

4.3.3 Transfer Training: Experimental Instruments

In this section, we describe the instruments that we used during the study's pre-test and post-test. We used these instruments to collect demographics and assess participants' spatial, reading, and programming abilities.

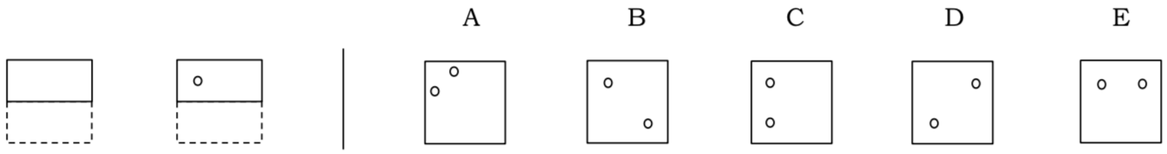


Figure 4.9: Example Paper Folding Test Problem: Participants select the choice that corresponds to the paper on the left unfolded. Answer is “C.”

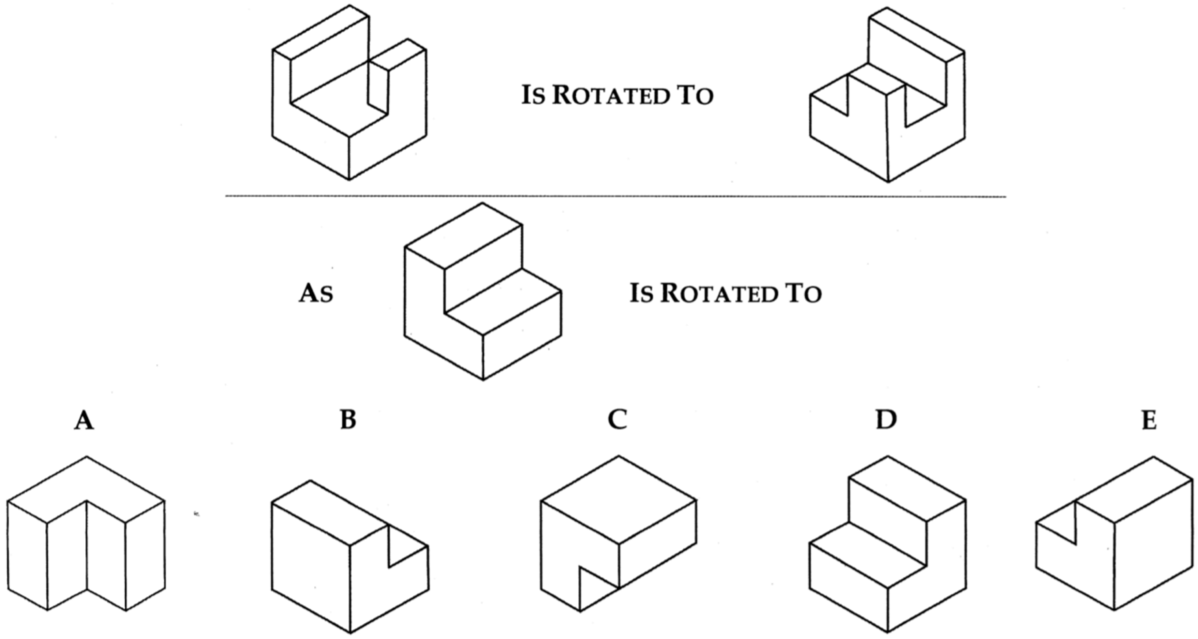


Figure 4.10: Example Revised Purdue Spatial Visualization Test Problem: (PSVT:R II) Participants select the shape that is rotated in the same way as the top shape. Answer is “D.”

Spatial Ability Assessments. We administered two spatial ability assessments that are designed to be taken as paper and pencil tests: the *Paper Folding Test* (PFT) [345] and the *Revised Purdue Spatial Visualization Test* (PSVT:R II) [355]. Both are validated standard assessments of different facets of spatial ability. The PFT measures mental folding and contains 20 multiple-choice questions split into two halves, each with a three-minute time limit. Participants took the PFT during both the pre-test and post-test. An example problem is shown in Figure 4.9. The PSVT:R II measures mental rotation via 30 multiple-choice questions of increasing difficulty, and takes 20 minutes. Participants took the PSVT:R II during the pre- and post-tests. An example problem is shown in Figure 4.10.

Reading Assessment. To assess reading ability, we gave students verbal sections of the *Graduate Record Examination* (GRE) [98], a required exam for admission to many graduate programs. We chose verbal sections from the official GRE practice test, using sections from the old (pre-2010) GRE format to minimize the chance of participants taking a test they had seen before. Each GRE verbal section had 25 multiple-choice questions and a 35-minute time limit. The test includes vocabulary questions and reading comprehension questions. Students were randomly assigned one of two GRE verbal sections during the pre-test and took the other GRE section during the post-test to avoid re-testing effects. We checked for difficulty differences between the two test versions by comparing the means and medians of both versions' pre-test and post-test scores. We did not find any significant difference in difficulty between the two versions. We note that the GRE is a general measure of reading ability rather than a measure of CS-focused technical reading. While there is a CS version of the GRE, it primarily measures programming ability rather than technical reading. To the best of our knowledge, there does not yet exist a validated measure of CS-focused reading.

Programming Assessment. To assess programming ability, we used the Second CS1 Assessment (SCS1) developed by Parker *et al.* [242]. Written in pseudocode, the SCS1 is a validated language-agnostic measure of CS1 programming ability. It is also the same programming assessment used by Bockmon *et al.* for evaluating their spatial training intervention [37]. An updated version of the FCS1 [310], the SCS1 consists of 27 multiple choice questions evenly divided into three question types: definition questions which ask students to recall the function of a programming construct, trace-based questions that require students to mentally walk through code before choosing the answer, and code-completion questions which ask students to choose a code snippet that causes a given program to have a specified behavior when inserted into a specified program location. The SCS1 has a one-hour time limit. Along with the full-length version, Parker *et al.* created a shorter 12-question subset. Due to students' limited initial programming ability, we used this shortened version during the pre-test and had participants complete the full SCS1 during the post-test.

Demographic and Qualitative Instruments. We also collected a variety of demographic and qualitative data. During the pre-test, students took a demographic survey for gender, race, ethnicity, native language, intended major(s), and computing attitudes. We also included a version of the Financial Affluence Survey III, a validated measure estimating socioeconomic status [317]. During the post-test, we had participants self-report anticipated CS1 course grades and other measures of programming ability. We also asked students for feedback on the Treatments including which modules they found the most challenging or helpful and if they thought that the training helped them in CS1. Finally, we asked for short-answer reflections on their experiences in the study.

4.3.4 Training Effectiveness Validation

We first analyzed the effectiveness of both Treatments in their respective domains. Especially due to COVID-19, it is important to verify the interventions' effectiveness. We used a multiple regression approach, with pre-test score as a predictor to control for individual variation in spatial and reading ability, and included Treatment Group as a predictor as well.[5]

We observed main effects of pre-test score for all three measures (PSVT:R, PFT, and GRE) suggesting that pre-test scores significantly predicted participants' post-test scores (see Table 4.4). We did not, however, find significant differences between Treatment Groups for the PSVT:R, PFT, nor the GRE Reading Assessment at post-test (see Table 4.4); we would expect a main effect of Treatment Group for each of these regression analyses. While it was surprising that our interventions did not appear to improve spatial ability or GRE reading scores, a closer examination of the data makes clear why we do not see significant improvements. For one, the students in our study had relatively high spatial ability. For example, in the study upon which we base our spatial ability training [300], the mean PSVT:R pre-test score of intervention participants was 52%, while the average pre-test PSVT:R score of our participants was 65%. A one-sample t-test comparing our pre-test data on the PSVT:R to a .522 suggested that our pre-test scores were significantly higher than those in the original study ($t(54) = 4.99, p < .001$).

Furthermore, our reading intervention could target different skills than those assessed by the GRE. We note that the GRE is a general assessment of reading ability and potentially not the best measure of skills taught in a CS-focused Reading Treatment. To the best of our knowledge, there does not yet exist a validated measure of CS-focused reading ability that does not also test programming ability. We hope the efficacy of our proposed Reading Treatment may help facilitate the future development of such a tool.

4.3.5 Transfer Training: Experimental Results

We now present our analysis of our transfer training experiment (see subsection 4.3.2). This experiment compares two potential CS1 skill-training interventions: Spatial Ability Training and our CS-focused Technical Reading Training. We center our investigation around the following five questions:

- **RQ1—***Efficacy*: Did Reading Treatment participants perform better than Spatial participants on the final programming test?
- **RQ2—***Question Type*: Were the effects of the Treatments more pronounced for some programming question types than others?
- **RQ3—***Subpopulation*: Were participant subpopulations better supported by either Treatment?

[5] For all regressions, we report (1) the regression coefficient B, the direction and strength of the relationship between the predictor and dependent variable; and (2) the uncertainty around that estimate $SE(B)$, the outcome of a t-test $t(x)$ with x degrees of freedom, and the significance level of this test.

Table 4.4: Training Validation Results: Regression analysis on the effectiveness of the training interventions.

Post-test Score **Predictor**	**Estimate (B)**	**SE(B)**	**t(54)**	**P-value**
PSVT:R II				
Intercept	0.29	0.07	4.14	<0.001
Pre-test	0.63	0.09	6.87	<0.001
Training Type	0.01	0.04	0.24	0.810
PFT				
Intercept	0.16	0.07	2.24	0.030
Pre-test	0.85	0.09	9.43	<0.001
Training Type	−0.01	0.03	−0.33	0.750
GRE				
Intercept	0.18	0.04	4.15	<0.001
Pre-test	0.02	< 0.01	6.35	<0.001
Training Type	−0.04	0.04	−1.13	0.270

- **RQ4—***Correlation*: Which participant demographic- and skill-based traits were most predictive of, and/or correlated with, programming success regardless of Treatment?

- **RQ5—***Perception*: How do participants subjectively describe their experiences in both Treatments?

In this evaluation section, we use the phrase *Final Programming Outcomes* to refer to scores on the final programming assessment.

Notes on Statistical Methods. When analyzing our transfer training results, most statistics were conducted with R statistical software, primarily using the `lm` function for regression analysis. All analysis scripts, including tests for violations of statistical assumptions and graph generation with `ggplot2`, are available in the replication package

For each of the first three research questions, we specified multiple linear regression models. We specified separate multiple linear regression models for each question containing only the variables that we report with the results for each research question. For example, the model for RQ1 contains pre-test programming scores (to control for participant variation) and participant Treatment Group. We did not specify an overarching model between research questions that includes all of the individual differences explored in RQ4 for two reasons. First, there is no effect of these variables on treatment effectiveness (as shown in RQ3, see subsubsection 4.3.5.3). Second, adding them would increase model complexity in RQ1 and RQ2 and possibly introduce a multicollinearity issue [12], as confirmed by the variance inflation factors we observed for each model.

Beyond multiple linear regression models, we also compute a correlation matrix for RQ4 and perform chi-square tests for RQ5. To measure the effect size of our results, we compute Cohen's f^2 [67]. Finally, we note that correction for multiple statistical tests is necessary when multiple dependent variables are used to search for an effect on at least some variable as a function of an independent variable. As a result, we employ false discovery rate correction when calculating the correlation matrix for RQ4. For RQ1, however, we have only one dependent variable of interest. Similarly, RQ2 acts as a breakdown of this dependent variable and thus serves to provide more information about what students learn. As a result, it is not necessary to control for multiple comparisons in RQ1 and RQ2. Finally, RQ3 and RQ5 contain no statistically significant tests even without correction, making further correction unnecessary.

4.3.5.1 Transfer Training RQ1: Treatment Efficacy

To answer our main question of whether the reading training would improve programming abilities, we compared Final Programming Outcomes between participants in the Spatial and Reading Treatments. Using a multiple regression model with pre-test programming score to control for individual variation and Treatment Group as predictors, we found a significant main effect of pre-test score ($B = .37$, $SE(B) = .14$, $t(54) = 2.68$, $p < .01$), suggesting that participants' pre-test score predicted their post-test score.

More importantly, we also found a significant main effect of the Treatment Group ($B = -0.09$, $SE(B) = 0.14$, $t(54) = -2.33$, $p = .02$): the students in the Reading Treatment performed significantly better on the post-test programming test than students in the Spatial Treatment (see Figure 4.11). This is our primary result, and it indicates that technical reading ability may facilitate programming for novice software engineers more than spatial ability in some cases. We discuss further implications of this result in the context of novice programmer cognition in section 4.4.

> Reading Treatment participants perform better than Spatial Treatment participants on Final Programming Outcomes ($p = 0.02$), a small effect ($f^2 = 0.10$).

4.3.5.2 Transfer Training RQ2: Treatment Effects by Question Type

In this section, we analyze if the Reading Treatment's benefit varies based on the programming question type. As mentioned in subsection 4.3.3, our programming assessment (the SCS1) has three types of questions: definitional, tracing, and code-completion [242]. We computed a multiple regression model for each type of question, with pre-test scores on the subtype questions and Treatment Group as predictors. We do so to determine if the Reading Treatment significantly improved more on specific question subtypes in comparison to the Spatial Treatment. We found evidence that the Reading Treatment improved more on the tracing questions than the Spatial

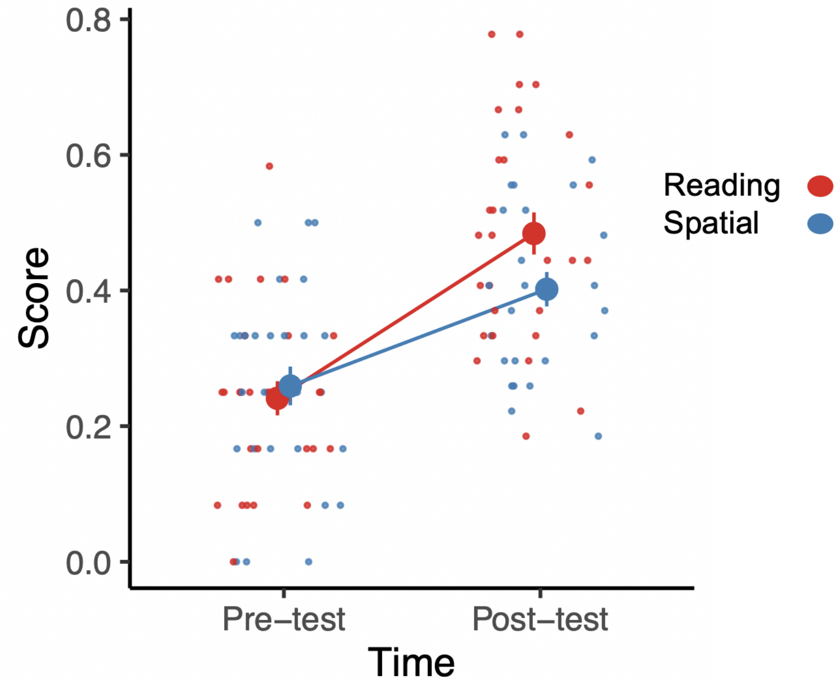


Figure 4.11: Percentage Programming Gains by Training Treatment: Reading Treatment participants exhibit significantly larger gains than Spatial Treatment participants.

Treatment ($B = -1.04, SE(B) = 0.45, t(54) = -2.31, p = .03$), a small-sized effect ($f^2 = 0.10$). However, the Reading Treatment did not improve more on the definitional ($B = 0.41, SE(B) = 0.23, t(54) = 1.76, p = .33$) nor code-completion ($B = .27, SE(B) = .32, t(54) = -1.61, p = 0.11$) questions (see Figure 4.12). We consider hypotheses for the effect of the Reading Treatment on Tracing questions in the context of our findings on novice programmer cognition in section 4.4.

The Reading Treatment was especially helpful for code-tracing questions ($p = 0.03$, $f^2 = 0.10$). There were no significant performance differences on definitional ($p = 0.33$) nor code-completion ($p = 0.11$) questions.

4.3.5.3 Transfer Training RQ3: Participant Subpopulation Effects

In this section, we analyze variations in both Treatments' effectiveness by participant sub-population. To avoid spurious effects due to multiple comparisons, we performed a limited number of sub-population analyses. In particular, we considered incoming spatial ability, incoming reading ability, native language, and gender. To examine whether one treatment was more effective than the other for different subpopulations, we ran a series of multiple regression analyses. In each analysis, we consider the post-test programming score as the dependent variable and include the individual difference of interest, pre-test programming score, Treatment Group, and the individual difference

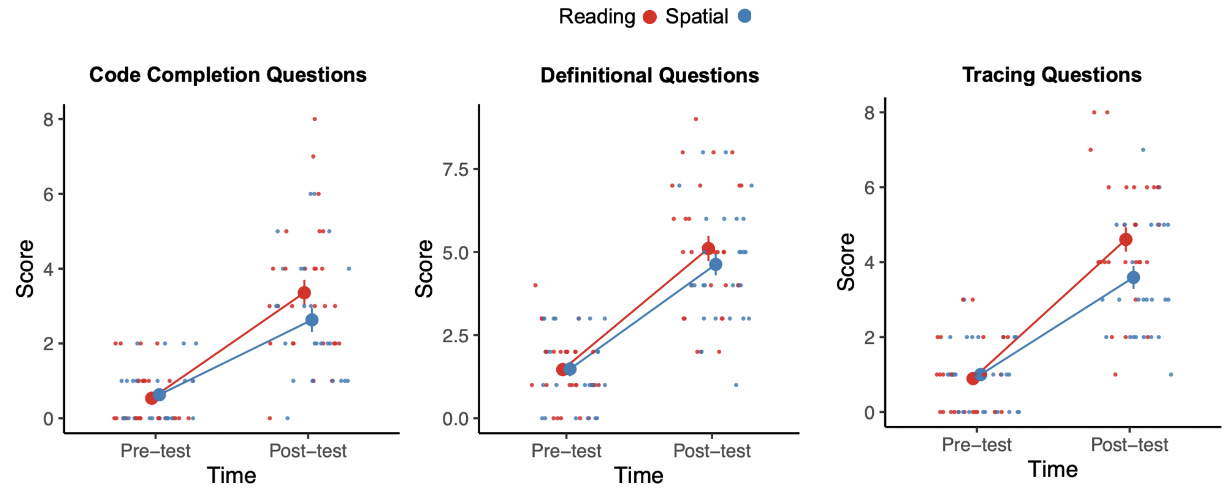


Figure 4.12: Programming Question-type Specific Programming Gains by Training Treatment: Reading Treatment participants exhibit significantly larger gains than Spatial Treatment participants on the Tracing questions but not on the Code Completion or Definitional questions. For each graph, the score is out of a maximum of 9.

Table 4.5: Transfer Training Subpopulation Analysis: Results of regression tests for individual differences.

Interaction: **Treatment Group * X**	**Estimate (B)**	**SE(B)**	**t(53)**	***p*-value**
Incoming PSVT:R II	−0.04	0.20	−0.18	0.86
Incoming PFT	−0.27	0.24	−1.12	0.27
Incoming Reading	−0.06	0.19	−0.30	0.76
Native Language	−0.13	0.08	−0.30	0.94
Gender	−0.09	0.09	−0.99	0.33

× Treatment Group interaction as predictors.

If one treatment were more effective for improving programming scores than another for a specific sub-population, we would expect to see a significant interaction. However, we were unable to see any variations in Treatment effectiveness by subpopulation for any of the models (see Table 4.5). This is not surprising given the sizes of our subpopulations. It is difficult to analyze individual difference data without a large sample size. Future work might replicate the current study with a larger sample size to determine if sub-population variation in treatment effectiveness exists.

We did not find evidence that variation in the effectiveness of the Treatments existed by participant sub-population when considering incoming spatial ability, incoming reading ability, native language, and gender.

4.3.5.4 Transfer Training RQ4: Treatment-Agnostic Correlations

We also analyze which skills-based and demographic features across both Treatments predict Final Programming Outcomes using correlations to capture how assessment scores relate to each other. Figure 4.13 contains a Spearman's rank correlation matrix for pre-test and post-test scores and select demographic features for all 57 study participants.[6] In particular, it contains the spatial ability measures (PFT and PSVT:R II), reading test (GRE), programming test (SCS1), gender, and native language. We report only significant correlations that pass false discovery rate correction ($p < 0.05$, $q < 0.05$).

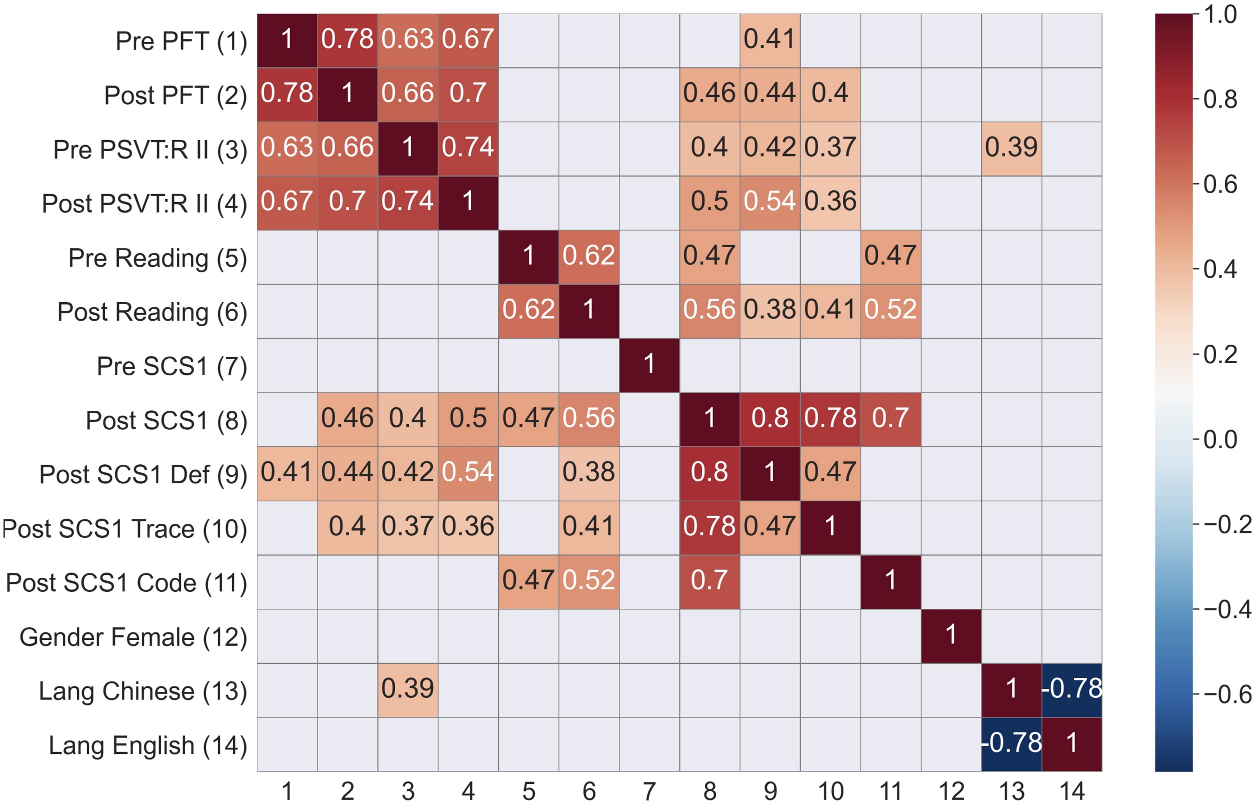
Perhaps unsurprisingly, the pre-tests and post-tests for both spatial measures were positively correlated ($0.63 < r < 0.78$). We note that cross-assessment (PFT vs. PSVT:R II) correlations trend lower than pre-test and post-test correlations on the same assessment. The pre-test and post-test reading scores also correlate ($r = 0.62$). We do not observe any significant correlations between either spatial test with the reading assessment.

We also observe that all reading and spatial assessments other than the pre-test PFT correlated significantly with Final Programming Outcomes (see row 8 in Figure 4.13). The correlations with post-tests for reading and PSVT:R II were particularly strong, both with $r \geq 0.50$. This indicates that while reading and spatial ability are separate skills, they are both important for software engineering. We also note that the PSVT:R II tended to correlate more strongly with coding than the PFT. It appears that mental rotation is more connected to introductory programming than mental folding.

Additionally, we observe a trend that reading may correlate more with programming than spatial ability; reading was both the most correlated pre-test and the most correlated post-test with Final Programming Outcomes (r = 0.47 and 0.56 respectively). When considering only code-replacement questions, only reading was significantly correlated with programming. This trend may indicate that reading is more predictive of software engineering success than spatial ability. However, replication with larger statistical power is required to confirm our observation. We did not observe any significant correlations between gender or native language with spatial ability, reading ability, or programming scores.

Spatial ability and reading ability, while distinct, are both correlated with programming ability. We also observe a trend that reading ability may be more correlated with programming ability than spatial ability.

[6] We chose Spearman's rank correlation instead of Pearson's because Spearman's captures both linear and non-linear monotonic relationships.



	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Pre PFT (1)	1	0.78	0.63	0.67					0.41					
Post PFT (2)	0.78	1	0.66	0.7				0.46	0.44	0.4				
Pre PSVT:R II (3)	0.63	0.66	1	0.74				0.4	0.42	0.37			0.39	
Post PSVT:R II (4)	0.67	0.7	0.74	1				0.5	0.54	0.36				
Pre Reading (5)					1	0.62		0.47			0.47			
Post Reading (6)					0.62	1		0.56	0.38	0.41	0.52			
Pre SCS1 (7)							1							
Post SCS1 (8)		0.46	0.4	0.5	0.47	0.56		1	0.8	0.78	0.7			
Post SCS1 Def (9)	0.41	0.44	0.42	0.54		0.38		0.8	1	0.47				
Post SCS1 Trace (10)		0.4	0.37	0.36		0.41		0.78	0.47	1				
Post SCS1 Code (11)					0.47	0.52		0.7			1			
Gender Female (12)												1		
Lang Chinese (13)			0.39										1	-0.78
Lang English (14)													-0.78	1

Figure 4.13: Transfer Training Test Score Correlation Matrix: Contains significant Spearman's rank correlations for pre-tests, post-tests, and select demographic traits for all participants.

4.3.5.5 Transfer Training RQ5: Perception

Finally, we present a brief analysis of participants' experiences in each Treatment. After the skills-based post-test, participants filled out a questionnaire to capture self-reported assessments of training efficacy and programming competency (see subsection 4.3.3). Participants also wrote short-answer explanations for their self-reported assessments. We focus on two questions: (1) "Did participating in the Transfer Training Study increase or decrease your desire to major or minor in computer science?" and (2) "Do you personally feel that the training sessions helped you with CS1? Why or why not?."

Perception Question 1—Major Declaration. In both Treatments, a majority of participants reported that our study did not change their desire to major or minor in computer science (75.1% of Reading vs. 81.5% of Spatial). Similar proportions of each Treatment reported that participating increased their desire to continue (18.5% of Spatial vs. 17.9% of Reading). A Chi-square test compared the proportion of different responses to this question between groups, and there were no significant differences ($\chi^2(2) = 2.01, p = .37$).

Perception Question 2—Perceived Helpfulness. We also compared the proportion of responses

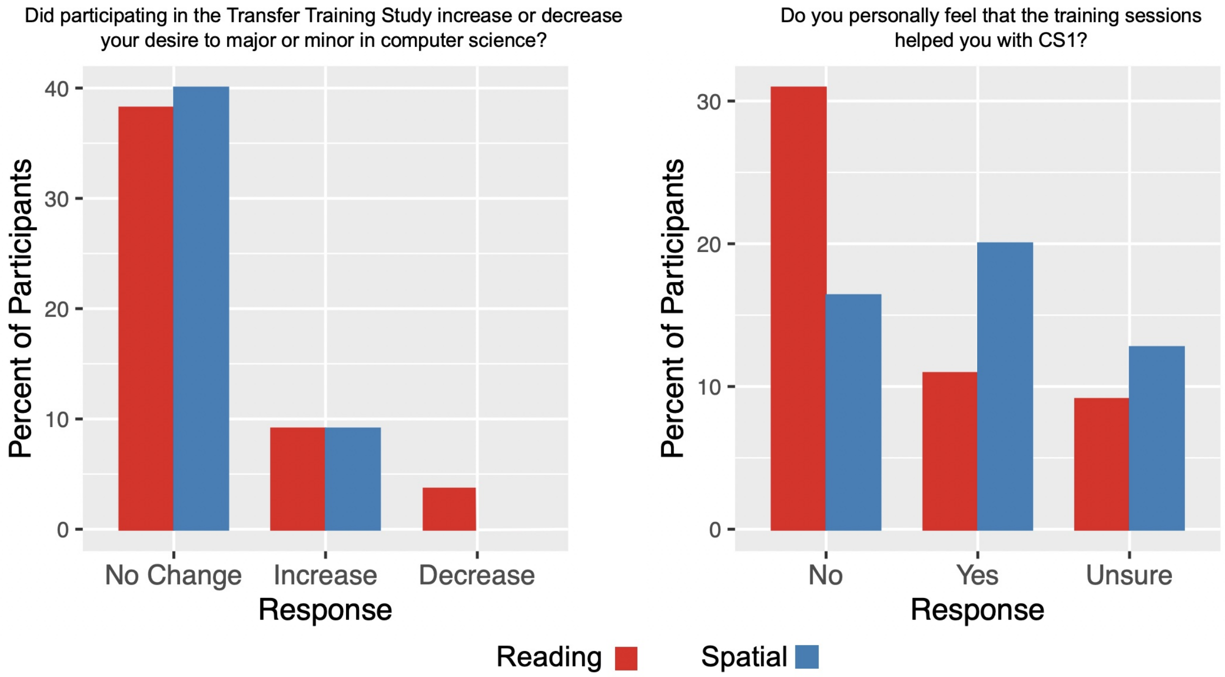


Figure 4.14: Subjective Participant Experiences of Transfer Training: participant response counts regarding their subjective experiences in both Treatments

between Treatments for whether students thought the training helped with CS1. We find no statistically significant differences between groups ($\chi^2(2) = 4.25, p = .12$). Although this test did not reach significance, the Spatial Treatment appeared to be perceived to be more effective than the Reading Treatment (40.2% vs 17.9%). All but one Spatial Treatment participant who described the training as useful also described their programming activities as spatial tasks. As a particularly demonstrative example, one subject explained that they "really like to visualize how the computer processes the information. This often requires imagining data traveling in space from one place to another and learning more about spatial thinking, I thought, really helped me in doing so." Students who did not find the Spatial Treatment helpful generally reported that they saw no clear connection between the topics.

While less common, several Reading Treatment participants also found the training helpful for CS1. For instance, one participant thought that the Reading Treatment "helped me to become more logical" while another participant commented that "it gave me an insight to computer science that I have been looking for." For those students who found the Reading Treatment less helpful, a common theme was that they wanted more hands-on programming practice directly related to CS1 ($\approx 70\%$). For instance, one student found that "it wasn't really helpful for [CS1] because we didn't practice coding concepts." Other students wanted more in-depth coverage of topics such as API documentation and less focus on vocabulary.

We find this difference between student perception of training helpfulness and observed quantitative helpfulness intriguing. We encourage future studies to investigate this effect further using a methodology, such as interviews, that enables the collection of more robust qualitative data.

Although not statistically significant, students perceived the Spatial Treatment to be more helpful than the Reading Treatment, a discrepancy between student perception and quantitative effect. Treatments had no significant negative effect on students' intentions to continue CS.

4.3.6 Transfer Training: Evaluation Summary

We evaluated our cognitively-informed CS-focused technical reading intervention using a controlled semester-long longitudinal study, comparing CS1 programming outcomes between our Reading Treatment and a standardized Spatial Treatment. We found that **Reading Treatment participants exhibit larger programming gains than Spatial Treatment participants** ($p = 0.02$), and that training type has a small size effect on final programming score ($f^2 = 0.10$). We also observed that our **Reading Treatment is most helpful for programming problems that require code tracing** ($p = 0.03$). Finally, we found that while both spatial ability and reading ability correlate with CS1 programming ability, they are not strongly correlated with each other. This aligns with our fNIRS results (see subsection 4.2.4), and further indicates that spatial ability and reading ability may be discrete cognitive skills that are both important for software engineering success.

4.3.7 Transfer Training: Limitations and Threats to Validity

In this section, we address various limitations and threats to the validity of our transfer training experiment. In particular, we address threats to our experimental control, generalizability concerns, threats concerning our training validation step, and limitations imposed by COVID-19. We also discuss our efforts to mitigate these threats.

One threat to validity is that despite the random assignment of participants to Treatments, our populations were not homogeneous in incoming spatial ability. Reading Treatment participants' average incoming spatial ability was higher than that of Spatial Treatment participants (see subsection 4.3.2). We mitigate this by using linear regression models that account for pre-existing individual differences.

Furthermore, as noted in subsection 4.3.4, we did not observe significant differences between treatment groups for gains on the spatial and reading assessments. This raises the concern that our interventions could be ineffective at teaching spatial ability and technical reading. However, also as indicated in subsection 4.3.4, the lack of difference in spatial ability between groups is likely due to a ceiling effect as our population already had high incoming spatial ability scores. As for the lack of difference in reading scores between treatments, we note the limitations in using a generic reading assessment, the GRE, as a proxy for CS-focused technical reading ability. We note that this threat does not lessen the core result that our Reading Treatment benefited students more than the Spatial Treatment. However, we acknowledge it is possible that our choice of test may affect our within-domain Reading Treatment assessment.

Our work also has generalizability limitations. While our overall results were statistically significant, our population of 57 students is only a small portion of all students taking a single CS1 course at a single American university. As a result, our results may not generalize to other CS1 curricula. Also, all students in the study had time to attend multiple two-hour training sessions. It is possible our results will not generalize to CS1 students who would not elect to participate in such a study. Furthermore, our participants also started CS1 with comparatively high spatial ability. Therefore, our results may not generalize to low-spatial populations.

COVID-19 did not directly impact our praxis until the fifth week when we transferred training sessions from in-person to virtual. We mitigated the effects of transitioning online by requiring all training material be completed while in a video chat with a proctor, sending participants physical training materials when possible, and having participants email proctors scans of their work. Even so, COVID-19 impacted the generalizability and power of our results. For instance, the online sessions contained fewer active learning activities, potentially lowering student engagement and learning [252]. Furthermore, participant dropout may have skewed our population. That is, students with the means and time to continue after the start of COVID-19 may be different than the initial study population. While any such difference is likely to affect both treatment groups equally, it may affect the generalizability of our findings to all CS1 students.

Finally, we note that while our results are intriguing, our effect sizes are small. We encourage future work to attempt to replicate given these small effects and the limitations of our research.

4.4 Lens 2: Discussion and Conclusion

In this chapter, we explored our second lens into improving programmer productivity: using cognitive insights to develop effective training for new programmers. To do so, we used a two-phase approach. We first used fNIRS to develop a mathematical model of novice programmer cognition (see section 4.2). We then evaluated if supplemental skills training can transfer to improved programming outcomes (see section 4.3). Throughout this productivity lens, we focused on two cognitive skills commonly associated with programming: reading and spatial ability.

In this section, we conclude and discuss the implications of our experimental results from both phases. In particular, we consider how the brain activation patterns of novice programmers compare to those of more experienced software developers, as well as the importance of our finding that technical reading training can transfer to programming ability.

Brain activation patterns while programming: novices vs. experts. We first discuss how our fNIRS results from phase 1 compare to those observed in neuroimaging studies with experts. Generally, we observe that novices have more right-hemisphere activation, more engagement of visuospatial processes, and less engagement of language processes than is seen in experts. For example, all significant activation areas observed by Siegmund *et al.* were in the left hemisphere, a majority coinciding with established language regions [293]. Similarly, Floyd *et al.* found that for experts, programming becomes increasingly less distinguishable from reading, a left-lateralized cognitive activity. In this context, our work helps establish an experience-based lateralization shift: novice programmers generally exhibit bilateral activation, especially in regions associated with visuospatial processing, while expert developers see increasingly left-lateralized activation centered in language-associated regions.

However, not all extant studies of experts observe a strong connection between reading and programming: in their study examining *code writing* (as opposed to code reading, the focus of other neuroimaging studies including our own), Krueger *et al.* observed significantly more right-brain activation in spatial areas during code writing than prose writing [181]. More investigation is needed to see if code writing exhibits a similar lateralization trend with expertise. However, it is possible that code writing consistently remains a more spatial activity.

Our fNIRS results also have implications on how programming in general compares to established neurological theories for learning in other disciplines. A number of theories and hypotheses about how humans learn various subjects, from second languages [177] to musical instruments [187], have been posited in the literature. We believe that it will be fruitful to investigate whether a sequential model or a more spatial encoding strategy (see Margulieux [207]) best describes learning to program. Based on our fNIRS results in phase 1, our preliminary speculation is that spatial encoding is indeed a key general strategy employed by novices that may decrease in importance over time. This speculation is supported by our finding in phase 2 that our reading training transfers to improved programming outcomes.

Implications of our finding that reading training can transfer to programming outcomes. We now discuss the implications of our primary result from phase 2 that first-year computer science students may benefit more from our CS-focused technical reading training than standardized spatial training. A diverse, and sometimes contradictory, array of models have been proposed as cognitive theories for learning programming. Some focus primarily on math or visuospatial processes [207], while others build on models explaining comprehension of natural language texts [299]. In their survey of proposed program comprehension models through the lens of CS education, Schulte *et al.* posit that reading strategies may be a distinct and important element of the knowledge base required to learn programming [281]. Our finding supports this supposition, lending support to program comprehension models that emphasize reading strategies. Our work also motivates additional research on the relationship between technical reading strategies and programming.

We do not claim, however, that spatial ability is unhelpful for software engineering. In fact, we observe a positive correlation between spatial ability and programming scores. Rather, we claim that technical reading may in some cases be a *more* transferable skill, especially when a population already has high incoming spatial ability. We also note that while both spatial ability and reading ability correlate with programming, they are at most weakly correlated with each other. Therefore, spatial ability and technical reading may be distinct skills that are both critical for software engineering.

However, if as mentioned above, spatial encoding is a general strategy used by novices that decreases in importance over time, then we hypothesize that technical reading may in contrast become *even more important* for experienced software engineers. For novices, the ability to trace through code and describe its function in natural language is highly predictive of programming ability [200, 224]. While useful for novices, code tracing also remains essential for experienced developers; software engineers are frequently required to read and understand other programmers' code to both contribute to large multi-programmer projects [109] and also for code reviews [17]. Interestingly, our Reading Treatment's effect was largest for questions that required code tracing, a result that underscores the connection between code tracing and natural language facility. Therefore, exploring technical reading-based training for more expert programmers is one future direction. In this context, our results may appear less surprising given that the CS1 testing primarily focused on code comprehension and tracing (see subsection 4.3.3) rather than code writing (*cf.* [181]), and that others have found links between CS performance and comprehension (*e.g.*, [200, 212, 224]).

Conclusion for Productivity Lens 2. Through our two-phase approach involving using fNIRS to model novice programmer cognition and a controlled evaluation of targeted cognitive training in a computing context, we show that technical reading training can enhance the productivity of novice programmers. We find that our novel, cognitively-based, CS-focused technical reading training curriculum is an effective and actionable intervention that can improve programming outcomes in practice. As a result, this dissertation demonstrates the potential of combining neuroimaging insights with software engineering pedagogy to support novices in developing essential programming skills faster, ultimately contributing to a more inclusive and efficient software development environment.

CHAPTER 5

How External Factors Impact Software Productivity: A Case Study on Psychoactive Substances

For our final research lens, we investigate how external factors can impact programmer productivity via a case study of cannabis use in a programming context. As discussed in subsection 2.3.1, researchers have identified various cultural or environmental factors that may influence programming productivity [105, 226]. However, many external factors with anecdotal connections with programming remain understudied in the scientific literature. In addition, while many studies identify or correlate various external factors with productivity, controlled experimental evaluations of their productivity impacts in practice are less common (*cf.* [203]).

In this dissertation, we conduct a case study on how one external factor, psychoactive substance use, impacts programming productivity. We focus on one substance in particular, *cannabis*. Cannabis is frequently connected to programming via online forum posts (see subsection 2.3.4). Despite these myriad anecdotal connections and the potential physiological impacts of cannabis on programming (see subsection 2.3.4), as of 2023 no previous empirical studies have directly investigated the prevalence of cannabis use in software. Nor are there any controlled observational studies of the actual effects of cannabis on programming ability. We hope the findings in this dissertation help fill this gap, allowing for both evidence-based corporate drug policies and also for developers to make more informed decisions regarding when and if to use cannabis while programming.

In the rest of this chapter, we first overview our approach for investigating the intersection of cannabis and programming in section 5.1. We then present our survey of cannabis use while programming in section 5.2, and our observational study on the impacts of cannabis on programming in section 5.3. Finally, we summarize and discuss the implications of our results in section 5.4.

5.1 Lens 3: Overview

In this dissertation, we use a two-phase approach to study the intersection of cannabis and programming. Specifically, we use a large-scale survey of programmers followed by a controlled observational study of cannabis use while programming. We consider two primary research questions: 1) Do programmers use psychoactive substances, such as cannabis, while programming? If so, when and why do they do so? and 2) Does cannabis intoxication while programming impact program correctness, programming speed, or programming creativity? We briefly introduce each phase in more depth.

First in section 5.2, we survey over 800 programmers to gain an empirical understanding of the prevalence and perceptions of cannabis in software. As we would like our survey results to be useful to company policymakers, our population must include professional developers. However, as cannabis can be illegal or explicitly prohibited by corporate policy (*e.g.*, see [66] for an example), it is also imperative that we maintain participant privacy and confidentiality in our study to avoid the risk of workplace retaliation. To balance these considerations, we focus recruitment on top contributors to open source projects (who are often professionals [266]) rather than on software engineering company contacts. We find that some programmers *do* regularly use cannabis while programming: 35% of our sample has tried programming while using cannabis, and 18% currently do so at least once a month (see subsubsection 5.2.4.1). Furthermore, we find that this cannabis usage is primarily motivated by a perceived enhancement to certain software development skills (such as brainstorming or getting into a programming zone) rather than medicinal reasons (such as pain relief, see subsubsection 5.2.4.3).

Second in section 5.3, we conduct an observational study of the actual effects of cannabis on programmer productivity. As discussed in subsection 2.3.4 and confirmed via our survey (see section 5.2), many programmers use cannabis while programming, generally holding positive views of its impacts. In broader contexts, however, views may be more negative. Cannabis can impair decision-making consistency, motor control, and reaction times [180, 270], impacts that inform both general and software-specific regulation. However, the efficacy of such policies [232, Sec. IX.B] and whether they are needed or beneficial [166] have been questioned in a modern context. To help resolve this conflict, we conduct a controlled observational study of the impact of cannabis on programming. In contrast to some anecdotal reports, we find evidence that cannabis intoxication impairs programming productivity, in terms of code correctness and programming speed (see Table 5.6). We hope our findings allow for evidence-based corporate policies and informed decisions by developers regarding when and if to use cannabis while programming.

In the rest of this chapter, we present our survey on cannabis use by programmers in section 5.2, and report on our observational study of the impact of cannabis on programming performance in practice in section 5.3. Finally, in section 5.4, we summarize the main results of this productivity lens, both in terms of the intersection of psychoactive substance use and programming and the importance of studying external factors more generally.

5.2 A Survey of Cannabis Use and Programming

In this section, we conduct and report on our survey of programmers' cannabis usage. We introduce and motivate our survey in subsection 5.2.1, overview our survey methodology in subsection 5.2.2, and contextualize our survey population in subsection 5.2.3. Finally, we present our results in subsection 5.2.4 and discuss the limitations of our approach in subsection 5.2.6.

5.2.1 Cannabis and Programming Survey: Motivation

Cannabis sativa is the world's most common illicit substance, used by more than 192 million people in 2018 [308]. The global cannabis market in 2020 was estimated at 20.5 billion USD, and it is estimated to grow to 90.4 billion by 2026 [348]. Globally, cannabis legality is changing rapidly, with many countries (*e.g.*, the United Kingdom, Colombia, and Malawi) legalizing medical cannabis, and a subset (*e.g.*, Uruguay, Mexico, and Canada) also legalizing cannabis for recreational adult use.[1] In the US as of 2023, however, cannabis is classified as a Schedule I drug, which criminalizes cannabis and defines it as having no accepted therapeutic value and a high abuse potential [230]. This classification has hampered research on its therapeutic effects [230], and prohibition is contrary to popular opinion: 91% of Americans believe cannabis should be legal for medical or recreational purposes [322]. Similarly, 81% of Americans believe that cannabis has at least one benefit [168]. These benefits are mostly medical, but 16% and 11% cited improved creativity and focus, respectively. For more background on cannabis's usage and acute impacts, see subsection 2.3.2.

Despite legal concerns, anecdotes of cannabis use intersecting with software engineering abound. Questions inquiring about cannabis's effects on programming are common on online forums such as Reddit,[2] Quora,[3] Hacker News,[4] and Dev,[5] often inspiring numerous conflicting answers. Similarly, popular tech-related media sites cover the topic, with one claiming that "folks in the tech mecca that is Silicon Valley are clearly getting their fair share of medical (or otherwise) marijuana", positing

[1] https://en.wikipedia.org/wiki/Legality_of_cannabis/

[2] https://www.reddit.com/r/computerscience/comments/dbzp5v/how_many_of_you_found_that_smoking_weed_gets_you/

[3] https://www.quora.com/Are-there-any-pothead-programmers-Can-you-code-while-youre-high

[4] https://news.ycombinator.com/item?id=509614

[5] https://dev.to/damcosset/coding-and-cannabis-3f8e

that cannabis may help with chronic pain associated with long hours of programming [174]. This claim aligns with epidemiological trends, as chronic pain is the most common reason for medical cannabis licensure in the US [38], and many people report that cannabis is useful for managing chronic pain symptoms [40]. Congruent with popular opinion, other posts claim that cannabis products enhance programming through increasing focus [32] or creativity [338]. Subsection 2.3.4 overviews extant work at the intersection of cannabis and programming, including summarizing relevant cognitive and physiological effects. However, to the best of our knowledge, no prior work has investigated the motivations of focus, creativity, or wellness for cannabis use while programming.

Cannabis is often prohibited in the workplace, a policy frequently enforced through mandatory drug testing for metabolites of Δ-9-tetrahydrocannabinol (THC), which causes intoxication. A 2018 report found that 63% of US organizations conduct drug screening, with two-thirds not accommodating medical cannabis use or lacking a medical cannabis policy [148]. In software engineering workplaces, drug tests remain common: we find that 29% of programmers have taken a drug test for a programming-related job (see subsubsection 5.2.4.1).

This prohibition of cannabis use in software engineering has contributed to a widely reported hiring shortage for certain US government programming jobs [64, 166, 193]. In 2014, after struggling to meet hiring goals, then-director of the American Federal Bureau of Investigation James Comey stated that while he had "to hire a great workforce to compete with . . . cyber criminals[,] . . . some of those kids want to smoke weed on the way to the interview", a behavior counter to the FBI's current policy that "prohibits anyone working for it who has used cannabis in the past three years" [166].

Despite such evidence highlighting the intersection of cannabis and programming, little empirical research has been conducted on cannabis use in software development (see subsection 2.3.4). Thus, it remains unclear if, when, or why people currently use cannabis while programming, let alone how job type or company policy may play a role. Without a grounded understanding of this intersection, companies cannot effectively evaluate the utility of extant anti-cannabis policies. To help fill this gap, we conduct the first large-scale empirical survey of cannabis use in software engineering, reporting data from 803 programmers, including 450 full-time developers (via an online, anonymized, and confidential survey). In this dissertation, we present findings from this survey. These findings include:

- We find that some developers (18% of our survey sample) use cannabis while programming at least once a month, with many even choosing to use it for work-related tasks.

- We find that cannabis use while programming is more commonly motivated by perceived programming-related skill enhancement than by medical reasons. This aligns with perceptions among a subset of students and younger people that cannabis use may enhance creativity or cognitive performance [15, 107].

- We find that cannabis use while programming occurs at similar rates for programming employees, managers, and students despite differences in perception of cannabis approval level and cannabis visibility between the three groups.

- Despite contrary anecdotal reports, we find that anti-cannabis policies and screening remain common for programming-related jobs, with 29% of respondents reporting having taken a drug test for a programming-related job.

5.2.2 Cannabis and Programming Survey: Design and Methodology

We now discuss the design of our survey in more detail. To understand programmers' cannabis usage, perception, and motivation, we conducted an online survey of 803 programmers. We desired a design that was *time-efficient* (allowing for a large number of responses), *low risk* (free of legal or employment-related consequences), and structured to permit *statistical* comparisons (within sub-populations such as employment status, age, and gender).

Thus, we designed our survey to take under 30 minutes, to be anonymous and confidential, and to use best practices from drug survey construction in other fields, including adapting questions from previously validated surveys when possible. We scoped this survey to United-States-based developers. To obtain diverse responses, we recruited participants from several sources including GitHub, social media, and the University of Michigan (a large public American university). We did not recruit participants directly from software companies to avoid risks of work retaliation. Recruitment materials indicated that previous cannabis use was *not* necessary to participate.

We now describe the construction (subsubsection 5.2.2.1), ethical considerations (subsubsection 5.2.2.2), and distribution (subsubsection 5.2.2.3) of our survey. Our replication package is available online with our full survey, recruitment materials, and IRB protocol.[6]

5.2.2.1 Survey Construction

Our survey included questions on demographics, programming background, cannabis attitudes, and cannabis usage. For the cannabis sections, we asked questions about general cannabis usage and cannabis usage in a programming context. We employed display logic as appropriate to minimize exposure to irrelevant questions. When possible, we based our questions on previous studies about cannabis [79, 135, 322] or software development [312] to allow comparisons with prior work.

[6] Replication package with survey instruments, data analysis scripts, and our IRB protocol: https://github.com/CelloCorgi/HashingItOut_ICSE2022.

Demographics. Querying age, gender, and employment status, our demographics questions were adapted from those asked by Boehnke *et al.* in their recent survey of cannabidiol use for fibromyalgia [40]. To help ensure participant safety, we did *not* collect identifying information such as names or IP addresses. As a result, we included additional validity checks as described in subsubsection 5.2.2.3.

Programming background. Participants reported how long they had programmed, their programming education, and their programming-related job history. We also asked participants with programming jobs if they self-identified as a manager, and we asked participants to indicate how often they conducted various software-engineering-related tasks such as brainstorming and requirements elicitation, adapted from those used by Tilley *et al.* [312, Sec. 2].

Cannabis attitudes. We asked questions regarding attitudes toward cannabis, both in general and also in programming-specific contexts. To gauge general cannabis attitudes, we asked questions about cannabis legalization and perceived risk drawn from previous American national surveys from the Pew Research center [322] and the National Survey on Drug Use and Health [59, 135]. For programming-specific cannabis attitudes, we adapted questions previously used to assess cannabis attitudes or perceptions from other contexts, *e.g.*, from a study of high school seniors' disapproval of individuals carrying out cannabis-related activities [18].

Cannabis usage. We asked questions regarding general and programming-specific cannabis use. For *general cannabis use*, we asked questions from the *Daily Sessions, Frequency, Age of Onset, and Quantity of Cannabis Use Inventory* (DFAQ-CU) [79], a validated measure of cannabis use behavior and history. We included questions that quantify current use frequency, past periods of heavy use, medicinal vs. recreational cannabis, and the percentage of the time participants use cannabis products with THC (see subsection 2.3.2). As this survey was conducted in 2021, we also asked how COVID-19 affected cannabis usage patterns. To measure cannabis use *while programming*, we adapted questions from the DFAQ-CU, adding the phrase "while programming, coding, or completing any other software engineering-related task?". We also asked for which types of programming projects or tasks participants are likely to use cannabis, and we asked participants about their motivation(s) for using cannabis with choices reflecting those we observed in anecdotal online posts as well as a free text option.

5.2.2.2 Ethical Considerations when Surveying Cannabis Use

In the US (the source of our survey population), cannabis remains illegal at a federal level and in many states (see subsection 2.3.2), so use can result in fines or incarceration. Further, cannabis is often explicitly prohibited by corporate policy, potentially resulting in employee termination. As such, we worked closely with our IRB to minimize legal risks and ensure participants felt comfortable answering honestly. First, we made our survey anonymous and confidential: we did not collect names or IP addresses, even though doing so necessitates additional data quality checks. Second, all participants gave informed consent, and all questions were optional. Third, we focused our recruitment of professional developers on open-source projects rather than through software engineering company contacts to avoid the risk of workplace retaliation or coercion. Fourth, we collected emails for our optional incentive on a separate platform where they could not be connected back to survey responses. Finally, we are unable to publish our full data set: although anonymous, it contains demographic data that may inadvertently identify participants.

5.2.2.3 Survey Distribution

Survey platform. When choosing a survey platform, we wanted to ensure the data we collected was anonymous and confidential while still ensuring data quality. Through consultation with our IRB, we used the Qualtrics XM Platform which enabled anonymous and confidential collection as well as data-quality options such as preventing multiple submissions and bot detection.

As mentioned in subsection 5.2.2, we scoped this initial survey of cannabis use and programming to United-States-based developers due to the high variance of cannabis laws and cultures worldwide. All recruitment materials stated that our survey was optional, anonymous, and confidential. To help mitigate participant self-selection bias, we also clarified that prior cannabis use was not required to participate. Finally, survey participation was encouraged through an optional drawing for one of five $100 awards. All data was collected between March 31 and May 2, 2021.

Survey recruitment. To encourage diverse responses, we recruited from several populations: open-source GitHub developers, current and former computing students at a large public university, and social media users.

For participants recruited from GitHub, we used GitHub's REST API to obtain the top 1000 developers and top 100 repositories associated with each of 25 "popular" programming languages (as identified in the GitHub interface) and 8 additional common languages such as MATLAB and R. For each repository, we pulled the profiles of the top 25 contributors. We filtered for profiles with a public email, resulting in 31,259 potentials. Using regular expressions combined with manual review, we identified 7,372 with a US location. Eliminating an additional 1,613 using DNS verification, we sent 5,759 emails, of which 36 failed, and received 440 valid responses (7.7%), a rate similar to previous studies of open-source developers [152, 297]. This use of GitHub profiles for research is permitted by the terms of service.[7]

As for the university-recruited participants, we sent 5,638 emails to all current and former

[7] https://docs.github.com/en/github/site-policy/github-acceptable-use-policies, see Section 6.

undergraduates who took a programming course for CS majors (*e.g.*, CS2 or CS3) at the University of Michigan between Fall 2018 and Fall 2019, receiving 283 responses (5.0%, 12 failed). As this study was conducted in 2021, this strategy recruited a mix of more senior CS undergraduates and young industrial developers rather than only current students. We also emailed CS graduate students for 56 responses. Finally, we posted the survey on Twitter, yielding 24 responses. While our study is not a random sample, we note that convenience samples are common in the cannabis and hidden populations literature [220] (see subsection 5.2.6).

Survey data validation. 1045 participants started our survey. To ensure we analyzed only high-quality data, we implemented several post-collection checks. First, 236 partial responses were removed. While participants could skip individual questions due to the sensitivity of the survey topic, valid responses must have answered at least: 1) if they had programmed; 2) if they had used cannabis, and; 3) if they had used cannabis while programming. To mitigate the threat to data quality from participants rushing through the survey, we removed responses completed faster than 1.5 standard deviations below the median. Finally, we checked for consistency between reported age, years of programming experience, professional programming experience, and cannabis use. Our completion time threshold and consistency checks eliminated 6 participants, leaving 803 valid participants for analysis.

5.2.3 Cannabis and Programming Survey: Population Contextualization

We now present indirect evidence that our participants, while not a random sample, are similar in many ways to previous random samples or studies. A true random sample would not have been ethically permitted, but we gain confidence in our results' generalizability by contextualizing participants' gender, age, and employment.

Table 5.1 overviews the gender, age, and programming-related employment of our survey population by recruitment pool (*e.g.*, how they were contacted to participate in this study). The majority (83%) of our population are men. This percentage is higher in those recruited from GitHub (91%) than those from the university (72%) and social media (83%). While this gender gap is large, it is similar to what we would expect of our sample population as a whole. For instance, Vasilescu *et al.* found that, of public GitHub profiles with ascertainable gender, 91% were men [324], the same percentage observed in our sample. We see a similar correspondence with university-recruited participants: 26% are women, close to the 24% of our CS undergraduate population overall. Because of this correspondence, our observed gender ratios give confidence in the generalizability of our results even though we could not collect a random sample of all developers.

Population	**Overall**	**GitHub**	**University**	**Social Media**
All Responses	803	440	339	24
Man	666 (83%)	403 (91%)	243 (72%)	20 (83%)
Woman	112 (14%)	21 (5%)	87 (26%)	4 (17%)
Non-binary	20 (2%)	12 (3%)	8 (2%)	0 (0%)
No Answer	5 (<1%)	4 (<1%)	1 (<1%)	0 (0%)
Median age	26	34	21	29
Average age	29.2	34.9	21.7	31.0
Min /Max age	15 / 70	15 / 70	18 / 61	21 / 52
Employment Status (could select multiple)				
Full-time CS job	450	357	75	17
Student	290	34	251	5
Unemployed	54	19	34	1
Part-time CS job	50	18	30	2
Programming-Related Job Title (could select multiple)				
Software Engin.	311	232	70	9
Developer	270	190	70	10
Systems Engin.	72	54	15	3
CS Researcher	53	23	25	5
CS Instructor	49	29	20	0
Data Scientist	49	25	24	0
Product Manager	22	17	4	1
IT	21	16	5	0

Table 5.1: Population Demographics of the Psychoactive Substances and Programming Survey: Demographics overview of our survey population broken down by recruitment pool. We note that participants in the 'University' column are those recruited from university emails. However, by the time of recruitment, many such participants have already graduated and thus have full-time CS jobs. Finally, we note that for job titles, participants could, and commonly did, select multiple descriptors. Thus, numbers in this section may add to more than the number of participants with jobs (*e.g.*, many selected both Software Engineer and Developer).

The ages of our participants also align with our sampled population. Our participants range in age from 15 to 70, with an average age of 29.2. As expected, GitHub-recruited developers were generally older than university-recruited participants, with an average age of 34.9 compared to 21.7. This age of our GitHub participants is comparable to the 30 reported in a previous study of open source developers [185]. Similarly, the average age of our university-recruited participants matches a typical US senior undergraduate.

As for employment, the majority (56%) of our sample are currently full-time employees at a programming-related job while an additional 6% are part-time and 36% are students. Of those who currently have a programming-related job, we observe a wide range of reported titles. While the most common titles were software engineer and developer (30% and 34% of our sample respectively), a significant number of participants identified as a systems engineer, computer science researcher, computer science instructor, data scientist, project manager, or information technician (between 2–10% of our sample for each).[8] This wide array of jobs indicates that our sample contains a diverse sample of programmers in various fields.

As for self-identified students, the majority came from university-recruited emails. Due to survey time constraints, we did not include a functional test of programming ability for students. However, all university-recruited participants had taken a programming course for CS majors 1–3 years before participating. This gap was intentional: it resulted in recruiting young computing professionals rather than only students. Since some in this set had graduated, the "University" descriptor in Table 5.1 refers to the email list source rather than current enrollment. To verify that students were indeed more advanced, we asked about general and professional programming experience. For general experience, results match expectations for more advanced undergraduates and graduates: both the first quartile and median student participant reported 3–5 years of experience while the third quartile reported 6–10 years. 67% of students also reported professional programming experience. Of these students, the median was 1–3 years, with some students reporting 6–10. This high level of professional experience may reflect the 21% who were graduate students.

Overall, the gender and age of our participants aligned with their populations. Even though we only collect data from university emails and open-source users, the high percentage of professional developers and the wide array of programming-related jobs indicate that our sample contains diverse types of programmers. We thus gain confidence in the generalizability of our findings.

5.2.4 Cannabis and Programming Survey: Research Questions

We organize our analysis of our survey around the following questions:

- *RQ1—Usage:* Do programmers use cannabis while programming? If so, how often?
- *RQ2—Context:* In what contexts do programmers use cannabis?
- *RQ3—Motivation:* Why do programmers use cannabis?

[8] Participants could select multiple job titles. We do note that 27 respondents (around 3% of our sample) self-reported as only CS Instructors. As suggested by our reviewers, we conducted an additional sub-population analysis removing these participants from our sample. This removal resulted in no changes to our overall significance results and analysis conclusions. For example, the number of developers who have tried cannabis while programming is 35.4% with educators removed and 34.8% with educators included. We include the full results of this additional analysis in our replication package.

- *RQ4—Perception:* How do opinions of cannabis use while programming vary between managers, employees, and students?

Survey Statistical Methods. Our analysis was conducted in a Python Jupyter Notebook using `Pandas` [343]. For our statistical analyses, we primarily used `SciPy` [331] and `Statsmodels` [284]. When testing the significance of a difference between continuous variables (*e.g.*, age) or Likert scores (*e.g.*, 5-point scale) of two independent sub-populations, we use the Student's t-test. While Likert scores are ordered categorical variables, previous research shows that with large samples, parametric tests are sufficiently robust for analyzing Likert data even though it is ordinal and normality cannot be assumed [237]. Thus, the Student's t-test is best statistical practice. For testing the significance of a difference between two binary variables, we use the n-1 χ^2-test (*i.e.*, the proportions z-test) [58].

We consider results significant if $p < 0.05$. As this is a large survey study, we investigate multiple research questions and conduct multiple statistical tests. To avoid fishing and p-hacking, we defined our primary research direction when designing the survey and we report the results for all initial research questions and analyses. Within each research question, we also correct for multiple comparisons using a Benjamini-Hochberg False Discovery Threshold of $q = 0.05$: unless otherwise noted, all significant results pass this multiple comparisons threshold. Most of our findings produced p-values well below 0.0001, increasing confidence.

5.2.4.1 Survey RQ1: Cannabis Usage While Programming

We first investigate if and how often programmers in our sample use cannabis while coding. We analyze usage trends in our sample overall and by gender, age, and recruitment pool.

Overall cannabis usage while programming. We find that 35% (280/803) of our participants have tried cannabis while programming or completing another software engineering-related task, around half of those who tried cannabis in general (69% = 557/803). Of those that have used cannabis while programming, 73% (205/280, 26% of our population overall) did so in the last year. While not a perfect comparison, we observe higher cannabis use than that in recent national surveys: 35% of Americans ages 18–25 and 15% of those 26 and older report using cannabis in the last year [135] compared to 54% of our sample. However, considering our population (and open source developers in general) skews young and male, higher reported use is expected.

Cannabis usage frequency. We also investigate *how often* participants currently use cannabis while programming. In the last year, 53% (147/280, 18% of our full sample) reported using cannabis while programming at least 12 times (monthly). Furthermore, 27% (76/280) reported using while programming at least twice a week (100 times per year), and 11% (30/280, 4% of our sample as a

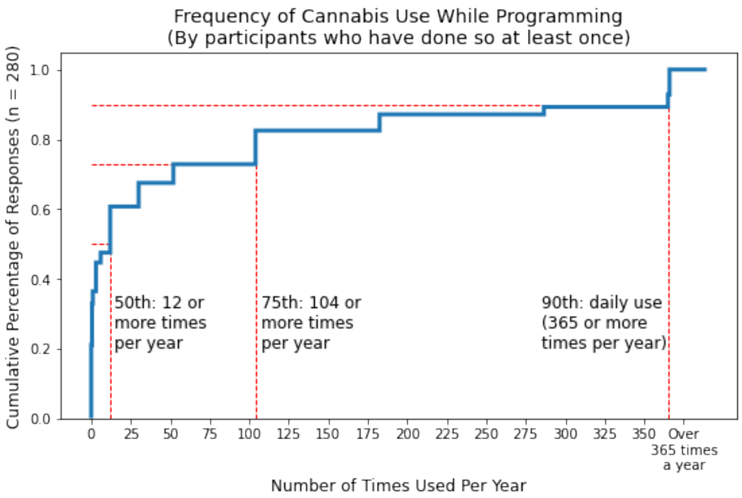


Figure 5.1: Frequency of Cannabis Use While Programming: "Which of the following best captures the average frequency you currently use cannabis while programming, coding, or completing any other software engineering task?" responses by programmers who have used while programming at least once. Converted from options such as "1 time a week" to times used per year for clarity. The dotted lines show the 50th, 75th, and 90th percentiles of use frequency. For example, programmers who use cannabis while programming at least 104 times per year are in the 75th percentile of use.

whole) reported using on a near daily basis. While those frequencies speak to *current* usage over the last year, these trends also occur over a longer term: 46% of our sample of cannabis-using programmers (128/280) also report that they have, at a point in the past, regularly used cannabis while programming (2 or more times per month for at least one six-month span). Our results regarding cannabis use frequency while programming are visualized in Figure 5.1.

These findings give the first formal insight into the prevalence of cannabis in programming communities, and they have important implications for drug tests in software engineering. Urine-based drug tests detect cannabis up to 30 days after use [133], much longer than the interval between cannabis sessions reported by many developers. Additionally, programmers' cannabis products typically contain THC, the compound detected by most drug tests [133]: on average, developers reported that 87% of their cannabis had THC, with the median reporting 100% of products included the compound. Simultaneously, we found that drug tests remain common for software: 29% of our sample reported that they had taken a drug test for a programming-related job. Thus, cannabis-using developers may avoid applying to jobs with drug tests, limiting application pools.

Cannabis use demographic context. We further contextualize our results by investigating variance by gender, age, and recruitment pool. Regarding gender, we do not observe a significant difference between men and women in the percentage of participants who have tried cannabis (70% vs. 69%, $p = 0.87$). However, we do find that men are more likely than women to have tried cannabis while programming (36% vs. 25%, $p < 0.0001$). This aligns with surveys of general populations not limited to programmers which find that men use cannabis more frequently than women [78]. Even though our sample size is smaller, we also observe that non-binary and transgender participants (not broken down in Table 5.1 for space) are significantly more likely to have tried cannabis while programming (57% vs. 34%, $p = 0.01$) than the rest of the population.

We also find a small but significant positive correlation between age and current frequency of cannabis use while programming ($r = 0.21$, $p = 0.003$):[9] of those who currently use cannabis while programming, older programmers tend to use it slightly more often. When plotting this correlation, we observe the bulk of this increase can be attributed to increases in usage likelihood before the age of 35. After that, usage frequency appears to level off or drop slightly.

We also examine our two main recruitment pools: GitHub and university (see subsection 5.2.3). GitHub participants were not significantly more likely to have tried cannabis (71% vs. 65%, $p = 0.065$). However, they were more likely to have used while programming (39% vs. 28%, $p = 0.001$). This aligns with observed demographics as GitHub participants are older and more likely to be men. However, it also may indicate that cannabis is particularly prevalent in open-source communities, a result that motivates future investigation of open-source cannabis culture.

Over one-third of our sample have used cannabis while completing a programming or software engineering-related task, of which half currently use cannabis at least once or more a month. Programmers typically use cannabis products that contain THC, and 11% of programmers who have used cannabis while programming report currently doing so on a near daily basis, behaviors very likely to be detected by most drug-test-related policies. We find that cannabis use while programming is particularly common for non-binary or transgender participants (57%) and participants recruited from GitHub (39%).

5.2.4.2 Survey RQ2: Cannabis Use Programming Context

We now investigate in what programming-related contexts programmers use cannabis, including both high-level project qualities (*e.g.*, personal projects vs. work-related projects) and also software-engineering task types (*e.g.*, refactoring, debugging, or requirements elicitation). We also analyze the impact of remote work.

In which programming contexts is cannabis used? We first investigate which programming project types (*e.g.*, personal or work projects) developers are most likely to choose to complete while using cannabis. We provide our full results in Table 5.2, but we emphasize our result that 95 participants (34% of cannabis-using programmers and 12% of our population overall) sometimes use cannabis

[9] Calculated using Spearman's r which detects all monotonic relationships (as opposed to Pearson's r which detects only linear relationships).

Project Type	**Number**	**Percent**
Personal programming projects	175	63.0
Non-urgent programming tasks	133	47.8
Work-related programming tasks	95	34.2
School-related programming tasks	76	27.3
Deadline-critical programming tasks	25	9.0

Table 5.2: Projects for which Participants Sometimes Use Cannabis while Programming: Percentages are out of the 280 respondents who have used cannabis while programming before.

for work-related tasks. While we anticipated our finding that personal programming projects were the most common project type completed while high, we hypothesized that the percentage using cannabis for work-related projects would be lower than observed. This indicates that cannabis routinely interacts with professional software engineering environments, underlining the potential impact of corporate drug policies and motivating future studies of cannabis in software engineering.

For which software tasks do programmers use cannabis? We now investigate how likely participants are to use cannabis while completing common software engineering tasks adapted from Tilley *et al.* [312]. Our full results are in Figure 5.2: programmers reported a higher likelihood of using cannabis while brainstorming or prototyping and a lower likelihood of using cannabis while performing quality assurance, requirements elicitation, or tasks with an imminent deadline. These results indicate that developers may self-regulate cannabis use for when it is most beneficial (*i.e.*, for creative, open-ended tasks) while avoiding use for time- or safety-critical tasks. We note that participants who are unable to self-regulate cannabis use (*e.g.*, are dependent on cannabis) may be unlikely to admit so in our survey. Even so, our results call into question the usefulness of blanket anti-cannabis policies. We investigate the motivations of these choices further in subsubsection 5.2.4.3.

Cannabis use and remote work. Many developers work remotely. Also, the COVID-19 pandemic was ongoing during recruitment. We find that 52% (145/280) of cannabis-using programmers report they are somewhat or a lot more likely to use cannabis for work-related tasks when working from home compared to only 5% (13/280) who report that they are somewhat or a lot less likely to do so ($p < 0.0001$). Similarly, 29% (82/280) report increased programming cannabis use since the onset of COVID-19 compared to only 10% (27/280) who report a decrease ($p < 0.0001$), a result in line with other populations such as medicinal cannabis users [39]. This indicates that workplace culture, environment, and policies can tangibly affect the frequency of cannabis use while programming.

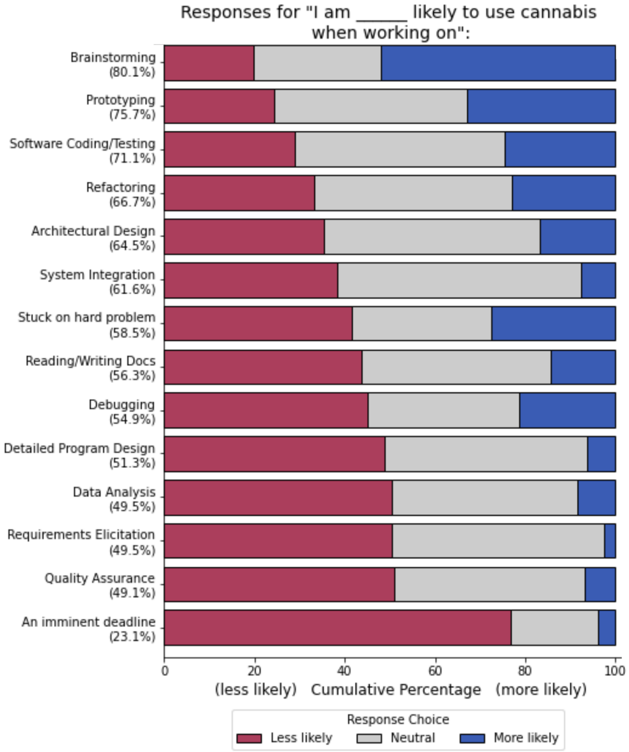


Figure 5.2: Cannabis Use Likelihood by Programming Task: Chart of participant cannabis use likelihood while completing various common programming-related tasks on a 3-point scale. Tasks are ordered by the combined percentage of more likely and neutral to use cannabis. Percentages are the sum of more likely and neutral responses for each task.

Developers most commonly choose to use cannabis during personal programming projects (63%). However, over a third of cannabis-using programmers also sometimes choose to use cannabis during work-related tasks, use that is more common during remote work. Programmers also self-regulate when they use cannabis: cannabis is more likely during creative open-ended software tasks vs. time- or safety-critical tasks.

5.2.4.3 Survey RQ3: Cannabis Use Motivation

Having established that some developers regularly use cannabis while completing both personal and work-related programming projects, we now investigate *why* programmers use cannabis. Understanding why developers use cannabis is important because it can help inform company drug policies and developer support.

Overall Motivation Results. We report our cannabis use motivation results in Table 5.3. We found that programmers were more likely to report enjoyment or programming enhancement motivations than wellness motivations: the most common reasons were “to make programming-related tasks more enjoyable” (61%) and “to think of more creative programming solutions” (53%). In fact, all programming enhancement reasons were selected by at least 30% of respondents. In contrast, wellness-related reasons (such as mitigating pain and anxiety) were all cited by less than 30% of respondents: while wellness does motivate some cannabis use while programming, it is not the most common motivation. This result is further corroborated by only 19% (54/280) of cannabis-using programmers indicating that they have a physician's recommendation to use cannabis medicinally. Additionally, of those that have such a recommendation, two-thirds report using cannabis for both medicinal and recreational reasons. This is important because it indicates that any cannabis policy should consider medicinal, recreational, and performance-enhancing use.

We also investigated cannabis-use motivations by population pool (*i.e.*, GitHub-recruited vs. University-recruited participants)—while percentages varied slightly, the top four rationales were the same regardless of recruitment pool. Wellness responses were also consistently below programming-enhancement motivations.

Additional Qualitative Responses. While we leave a formal qualitative analysis to future work, we also observe the emphasis on programming-enhancement-related reasons for cannabis use while programming in the textual free-response section. For instance, one participant said that when using cannabis while programming, they are "able to better connect ideas and think about things on a broader level, which typically leads to more well-rounded solutions." Similarly, another participant stated that cannabis use while programming helped brainstorming, allowing them "to organize [their] thoughts better and keep them separate, which helps [them] follow threads further and come up with new paths to follow".

Beyond cannabis's apparent usefulness during programming itself, participants also cite its usefulness during adjacent tasks. For example, one participant said that they used cannabis "to stay awake/focused when . . . grading 70 programming assignments. If [I] have a deadline and need to binge work for many hours, it is easier to keep going if I periodically get high. When your life is mostly work, cannabis is something that makes it bearable." Finally, some participants indicated reasons other than enjoyment, wellness, or programming enhancement for using cannabis while programming. For instance, some participants indicated that cannabis use while programming was only a coincidence from their regular cannabis use, while a few others said they tried cannabis to enhance programming but did not observe any effect.

Finally, while we observed little quantitative evidence of negative experiences of cannabis use while programming, we did observe a few qualitative reports. For example, one participant stated: "I generally don't use [cannabis] while programming because [. . .] it affects my short-term memory, which is a huge part of programming for me [. . .] My managers wouldn't have an issue with me using cannabis during my job, but I do have an issue just due to the nature of cannabis." Similarly, another participant stated "I wanted to see if [cannabis] would help, but all it did was make it harder to keep track of what I was doing. I'm glad I tried it, but I wouldn't do it again." These quotes show

Reason for cannabis use while programming	**Cat**	**Count**	**%**
To make programming tasks more enjoyable	E	148	60.9
To think of more creative programming solutions	P	128	52.7
To get in a programming zone	P	117	48.2
To make programming-related tasks less tedious	E	103	42.4
To enhance brainstorming	P	96	39.5
To focus on programming-related tasks	P	80	32.9
To gain insight or understanding	P	79	32.5
To help with work-related anxiety	W	67	27.6
To have fun in social programming settings	E	35	14.4
Other (please describe)	N/A	33	13.6
To mitigate programming-related pain	W	32	13.2
To help with work-related social anxiety	W	32	13.2
To improve social interactions in the workplace	E	18	7.4
For non-programming related medical conditions	W	17	7.0
To help with work-related migraines	W	12	4.9
I am unable to think as clearly without cannabis	W	11	4.5

Table 5.3: Why programmers use cannabis while programming: "Why do you use cannabis while programming, coding, or completing other software engineering-related tasks?" Participants could select multiple choices. "Cat" delineates a particular motivation as programming enhancement (P), enjoyment (E), or wellness (W). % is out of the 243 who selected at least one option.

that even among cannabis-using programmers, there is a wide array of experiences and reactions, a variance that invites further and more in-depth qualitative analysis.

Implications. These conflicting experiences raise the question of if the programming-related benefits of cannabis perceived by some developers translate to verifiable programming enhancement. For example, even though many participants report using cannabis to enhance programming creativity, it is unclear if this actually occurs. We note this concern was raised by some non-cannabis-using programmers. For instance, one such programmer wrote that they "had a series of developers work for [them] that used cannabis to varying degrees. All of them fully believed that it made them better engineers, that it sparked creativity and capability. From the outside, however, the results have consistently not been that way. Not just from less code productivity, but far more often inability to work as well with others, and code quality issues." Thus, our results motivate our observational study of the effects of cannabis on programming (see section 5.3).

We found that programmers who use cannabis were more likely to be motivated by potential programming ability enhancement or programming enjoyment than wellness-related reasons. This pattern was observed in both open-source developers and university participants, and it motivates future work investigating if the perceived programming benefits of cannabis manifest in a more rigorous observational study.

5.2.4.4 Survey RQ4: Perception of Cannabis Use

We also investigate how programmers perceive cannabis use. Understanding this is important because if perception varies from actual usage, this may result in sub-optimal cannabis-related policies or biases in programming environments. We investigate cannabis perceptions in general and programming-specific contexts. For the former, we compare to national cannabis attitudes surveys. For the latter, we analyze attitude differences between programming students, employees, and software managers. We then compare any differences to each group's cannabis usage.

General Cannabis Perceptions. Programmers in our sample have more positive attitudes toward cannabis than the population overall. For example, 91% of our participants say that marijuana use should be legal for both recreational and medicinal use compared to 60% of the general United States population in 2021 [322]. Similarly, only 5% of our population views smoking marijuana once or twice a week to be of "great risk" as opposed to 29% of the US population [135]. This difference is likely explained by the demographic differences in age, gender, and political leaning between programmers in our sample and the population overall (see subsection 5.2.3), or by self-selection bias.

Programming and Cannabis Perceptions Setup. To understand cannabis perceptions in a programming context, we first ask participants to rate their approval or disapproval of someone who uses cannabis while working with them on software engineering tasks such as programming, brainstorming, debugging, or security testing, on a 5-point Likert scale (from -2 for disapproval to +2 for approval). We average these responses into an overall "cannabis approval score". This approval/disapproval format was adapted from previous research on cannabis attitudes [18]. Second, we indirectly asked about cannabis use *visibility* by asking if respondents knew of a colleague who regularly used cannabis while programming on a 5-point Likert scale (from -2 for a solid no to +2 for a solid yes).

The exact wording varied for student, employee, or manager participants. Differentiating between groups admits analyzing more nuanced perception differences. For example, we ask employees if they think their manager would disapprove of cannabis use while we ask managers if they actually would disapprove. Even though this is not a perfect comparison, as managers in our sample do not necessarily manage employees in our sample, it still allows a first investigation into perception differences between these two groups overall. Table 5.4 lists our population-specific questions.

Q ID	Perception Question	Answer Choices	Population	Num	Score
	Perceived approval/disapproval of cannabis use while programming:				
1	Would you approve/disapprove of a partner using cannabis while ____ for school-related tasks?	5-point Likert: disapprove/approve	Programming Students	237	-0.25
2	Would you approve/disapprove of a colleague using cannabis while ____ for work-related tasks?	5-point Likert: disapprove/approve	Non-manager Employees	329	-0.36
3	Would your manager approve/ disapprove of you using cannabis while ____ for work-related tasks?	5-point Likert: disapprove/approve	Non-manager Employees	306	-1.04
4	Would you approve/disapprove of a supervisee using cannabis while ____ for work-related tasks?	5-point Likert: disapprove/approve	Programming Managers	189	-0.51
	Perceived visibility of cannabis use while programming:				
5	Do you know a fellow student who regularly uses cannabis while completing programming-related tasks?	5-point Likert (no...unsure...yes)	Programming Students	236	0.36
6	Do you know a colleague who regularly uses cannabis while completing programming-related tasks?	5-point Likert (no...unsure...yes)	Programming Workers	519	-0.25
7	Do you know a manager who regularly uses cannabis while completing programming-related tasks?	5-point Likert (no...unsure...yes)	Programming Workers	519	-0.72

Table 5.4: Perceptions of cannabis use while programming: Student, employee, and manager perception questions (wordings modified slightly for clarity). For Q ID 1–4, participants gave approval for programming, brainstorming, debugging, and security testing, then aggregated for an overall participant score. The score column represents the average Likert score (from -2 to +2) for all “Response” participants.

Perceptions of Professionals vs. Students. For student programmers, we considered those who were students in computer science, software engineering, or another programming-related field. For professional programmers, we consider those non-students who were either full-time employees, part-time employees, or self-employed with a programming job. We hypothesized that students would approve more of cannabis use. However, we find no evidence of a difference between employee approval of colleagues (question 2) and student approval of fellow students (question 1) for using cannabis while completing programming tasks: the average from both groups was between neutral (0) and slight disapproval (-1) (-0.25 for student on students and -0.36 for employee on colleagues, $p = 0.26$). This indicates that perceptions of cannabis use while programming are similar in academic and professional contexts.

For visibility, however, we do observe a significant difference. Responses to questions 5 and 6 show that students are significantly more likely to know another student who regularly uses cannabis while programming than a professional programmer is to know a colleague who does the same ($p < 0.0001$): 48% of students report knowing or probably knowing a fellow student who regularly uses cannabis while programming compared to only 23% of professional programmers. However, we observe no significant differences in cannabis usage prevalence or frequency between the two groups, though fewer students than professionals report cannabis use while programming (32% vs 38%, $p = 0.09$). This finding may represent cultural differences despite similar approval and usage levels between the two groups, differences perhaps driven by higher levels of support for cannabis legalization among younger Americans [322] or the fact that some students were under the age of 21, the most common age limit for legal cannabis use in the United States.

Perceptions of Managers vs. Employees. For programming employees, we consider full-time, part-time, and self-employees with a programming job. Managers were those who reported they were managers. The average level of *expected* manager disapproval of cannabis use by employees was between slight and strong. However, managers reported an *actual* disapproval level between neutral and slight—a significant difference (questions 3 and 4, $p < 0.0001$). We found no significant difference between manager disapproval of their supervisees and employee disapproval of their colleagues (questions 2 and 4, $p = 0.15$), both reporting between neutral and slight disapproval. We also found no significant cannabis usage differences between employees and managers. Additionally, managers rarely report witnessing negative cannabis-related effects: while 27% (51/189) of managers suspect a supervisee uses cannabis, less than 3% (5/189) report that such programmers were less productive, and only one reported reprimanding an employee for cannabis use. Thus, our results indicate a perception mismatch of programming cannabis use between employees and managers: employees expect managers to disapprove of such use more than they actually do.

This mismatch between managers and employees is further compounded by a difference in cannabis-use visibility. Professional programmers are more likely to know or probably know a colleague who regularly uses cannabis while programming than they are to know a manager who does the same (23% vs. 8%, $p < 0.0001$). One potential implication is that if managers are ambivalent about cannabis use while programming, then corporate policies might be adjusted to avoid repercussions for cannabis use as long as work performance remains reasonable.

We found that US-based programmers have more favorable views on general cannabis use than the American population overall. In programming-specific contexts, we found differences in perception of cannabis use while programming between programming students, employees, and managers despite finding no significant differences in cannabis usage between the three groups. For example, we found that managers disapprove of cannabis less than employees expect them to. These results may indicate a mismatch between perception and reality.

5.2.5 Cannabis and Programming Survey: Results Summary

In our survey of 803 programmers, we find that **some programmers regularly use cannabis while programming** (18% of our sample do so at least once a month), many choosing to use cannabis for **both personal and work-related projects**. Furthermore, we find that cannabis use while programming is primarily motivated by **perceived enhancement to programming-related skills** and increased enjoyment rather than by medicinal reasons. Finally, we find that programming employees, managers, and students use cannabis while programming at similar rates, despite differences in cannabis perceptions and visibility. We also find that such cannabis usage **conflicts with anti-drug policies** currently enacted for many software engineering jobs: **29% of our sample reported they had taken a drug test** for a programming-related job, a hiring practice that may limit developer application pools. Thus, our results have implications for programming workplaces that currently have anti-drug policies and motivate our observational study of the effects of cannabis use while programming (see section 5.3.)

5.2.6 Cannabis and Programming Survey: Limitations and Threats

Limitations of our survey approach include self-selection bias and data quality concerns, common for survey research in general and self-reported drug research in particular [139]. Due to interest, cannabis-using programmers may be more likely than non-users to participate. Simultaneously, cannabis users may be less willing to report use due to retaliation worries. To mitigate these biases, we made it clear that prior cannabis use was not necessary (see subsection 5.2.2), and our survey was anonymous and confidential to encourage honesty. To check for self-selection bias, we validated that key demographics in our sample match those of our recruitment populations (see subsection 5.2.3). We also note that online cannabis use surveys typically produce high-quality and internally consistent data [259]. Even so, these biases are important when interpreting our findings and may result in our findings overestimating the prevalence of cannabis use while programming.

One additional limitation is timing during COVID-19: our findings may be influenced by this period of primarily remote work. We mitigate this threat by asking about cannabis-use behavior changes resulting from the onset of COVID-19 (see subsubsection 5.2.4.2). We note, however, that while this is a potential limitation, it also admits timely and useful analysis in light of our finding that remote work increases cannabis use while programming for work-related tasks.

5.3 Observational Study of Cannabis and Programming

We now go beyond surveying developers, and conduct a controlled observational study of the impact of cannabis on programming productivity. While our survey of cannabis-using programmers shows that cannabis use while programming is relatively common, with many developers reporting programming enhancement (see section 5.2), it alone does not capture the actual impacts of cannabis on programming. This is important because using cannabis while programming in a work context can conflict with software company policy [232]. In addition, claims, biases, and folk wisdom about the actual effects of cannabis intoxication on programming do not agree (including those reported by our survey participants, see subsubsection 5.2.4.3). This lack of a firm understanding prevents individuals and companies alike from making informed policy decisions and accurately balancing risk and reward.

Despite anecdotal reports of programming enhancement, there is reason to believe that cannabis may impair programming productivity (for an overview of relevant physiological and cognitive effects, see subsection 2.3.4). For example, cannabis intoxication can impair decision-making accuracy and consistency, resulting in losses being under-estimated ("treating each loss as a constant and minor negative outcome regardless of the size of the loss") [108]. It can also impair motor control and reaction times; "acute cannabis intoxication is associated with a statistically significant increase in motor vehicle crash risk" [270]. These more negative views often inform both general and software-specific regulations. However, questions have been raised about the efficacy of such policies [232, Sec. IX.B] and whether they are either needed or beneficial [166] in a modern context.

Understanding the impacts of cannabis intoxication on particular aspects of software development (rather than on general cognitive skills) would help fill this knowledge gap, allowing for evidence-based corporate policies and informed decisions by developers regarding when and if to use cannabis while programming. An effective understanding would be based on an indicative sample size, ecologically-valid conditions, and quantitative and qualitative aspects of produced software. In addition, any such study must be conducted ethically, given the rapidly changing legal landscape around cannabis use.

In this dissertation, we conduct the first rigorous observational study of the effects of cannabis intoxication on programming. We assess $n = 74$ participants from multiple American metropolitan areas across four states and used *preregistered hypotheses* to mitigate researcher bias. Each participant completed two sessions on different days, one while sober and one while using cannabis, in a *randomly assigned* order. This design permits both within-subject and between-subject comparisons. In each session, participants completed both short targeted programming tasks as well as multiple LeetCode [27] problems using a standard development environment (Visual Studio) on their personal computers. In the cannabis condition, participants were asked to use the dosage they would normally use while programming, an *ecologically-valid* context that allows us to learn actionable insights. The contributions of this dissertation section are:

1. The first rigorous observational study of programming in both sober and cannabis-intoxicated conditions.

2. The finding that ecologically-valid cannabis intoxication has a *small to medium effect on program correctness*: programs written by cannabis-intoxicated programmers exhibit *more bugs*, failing 10% more test cases, on average ($p < 0.05$).

3. The finding that cannabis-intoxicated programmers take *more time* to write non-trivial functions than do sober programmers (11% more on time average, $p = 0.39$).

4. The finding that cannabis-intoxicated programmers exhibit *different typing patterns*, including deleting and rewriting code more frequently and pausing for longer without typing ($p \leq 0.003$, $d \geq 0.35$).

5. Despite anecdotes of cannabis improving creativity, we observe no evidence that cannabis-intoxicated programmers make different algorithmic or stylistic programming choices.

6. The finding that cannabis-using programmers accurately recognize their programming performance, even when intoxicated, $r = 0.59$.

To the best of our knowledge, this dissertation presents the first study of how cannabis intoxication impacts programming. We believe aspects of our design (preregistered hypotheses, ecologically-valid settings, broad participant pool, *etc.*) make it generalizable and useful for informing policies and individual developer decisions surrounding cannabis. In the rest of this section, we first present our study design (subsection 5.3.1), describe our programming stimuli (subsection 5.3.2), and overview our study population (subsection 5.3.3. We then overview our analysis methodology (subsection 5.3.4), present our results (subsection 5.3.5), and discuss potential threats to validity (subsection 5.3.7).

5.3.1 Observational Study: Study Design

To understand the impact of cannabis intoxication on programming performance, we conduct a controlled observational study with 74 participants. Eligible participants were at least 21, had used cannabis in the last year, and had smoked or vaped cannabis before. We also required Python familiarity and programming experience comparable to that of a senior undergraduate.

When designing our study, we had three main design considerations: achieving sufficient statistical power for our preregistered hypotheses (see subsubsection 5.3.4.1), balancing ecological validity with experimental control, and maximizing participant privacy and safety.

Overall Study Design. To maintain statistical power, we use a within-subjects design. Participants completed a 20-minute questionnaire with demographics, cannabis usage history, programming history, and a four-minute training video introducing the study platform. Next, they attended two structurally identical programming sessions: one cannabis-intoxicated and one sober. The session order was counterbalanced: participants were randomly assigned the cannabis-first or sober-first condition (35/71 cannabis first vs. 36/71 sober first). This counterbalanced and within-subject design mitigates the impact of individual differences in programming ability, cannabis tolerance, and session ordering effects in our analysis.

Study Session Structure. All programming sessions lasted 1.5 hours and included two cognitive assessments, a series of short programming problems, and three "interview-style" coding questions. We included cognitive assessments for data validation (see replication package), short programming problems for controlled observations of the impact of cannabis on programming, and "interview-style" coding questions to capture more complex coding algorithmic options. The short programming problems permit a controlled investigation of the impact of cannabis on specific aspects of programming while the "interview-style" questions enable a holistic and ecologically-valid analysis at the expense of statistical power and experimental control. Subsection 5.3.2 details all experimental tasks.

The short programming problems were administered via an online Qualtrics survey. Qualtrics permits randomization and timing collection via custom JavaScript. For the "interview-style" questions, participants wrote and executed code using a browser-based instance of VSCode (a popular programming text editor) via a GitHub Codespace configured to collect keystrokes, terminal/compiler interactions, and program file contents. Participants were also permitted to search (Google) for help with syntax errors. This design allowed our programming environment to have higher ecological validity while remaining controlled enough to permit straightforward statistical analysis. Sessions were conducted remotely (via Zoom) and participants were required to screen share.

Cannabis Session Logistics. Participants used cannabis 10–15 minutes before the start of the session, then uploaded pictures of the product and indicated the amount. Participants were instructed to consume cannabis via vaping or smoking, rather than by taking an edible, to reduce variability: edibles can have very different effects than vaping or smoking [51].

Participants who had used cannabis while programming before were asked to use the amount they would typically use when programming. If they had not, they were asked to use a mild to medium dosage, consistent with the amount they use when not programming. Allowing participants to choose the amount of cannabis to consume is different than the approach taken by many studies of cannabis use on behavior or cognition (*cf.* [77]). However, allowing participants to self-select their usual cannabis dosage improves ecological validity. Informally, in this dissertation, we are not interested in learning the amount of impairment per milligram of cannabis, but instead in the amount of impairment an average cannabis-intoxicated programmer faces while programming. We view the latter as significantly more actionable (*e.g.*, to managers).

Ethical Considerations. The varying legality of cannabis requires special care in study design. Throughout the design and implementation of this study, we worked with our IRB (ethical review board) to ensure participant privacy and safety. Additional safety precautions were incorporated into our protocol including requiring that participants have used cannabis in the last year, and participating using a personally owned computer (*e.g.*, not a company-distributed laptop that may have tracking software) from a location where they would not have to travel for several hours.

Ethical research practice also influenced our selection of a remote study design, rather than an in-person lab study. Design decisions such as directly administering specific amounts of cannabis at a central location or analyzing blood samples to assess intoxication were considered and rejected given our focus on ecological validity, as well as for reasons related to logistics and privacy.

5.3.2 Observational Study Instruments and Stimuli: Content and Metrics

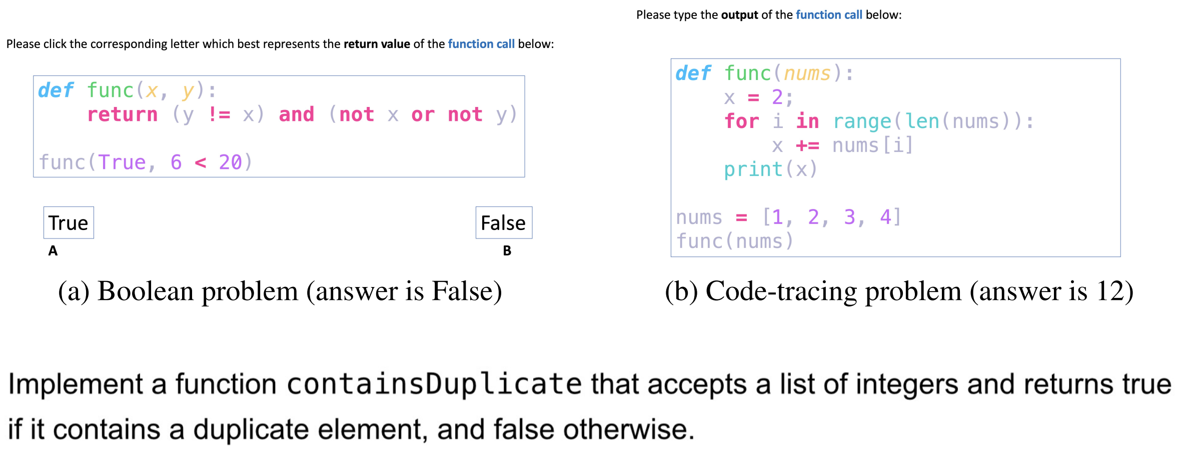
We now describe our survey instruments, programming tasks, and metrics. All surveys and stimuli are in our replication package.[10]

5.3.2.1 Demographics

We collected demographics such as gender, age, and employment status. We also asked about programming experience, cannabis usage history, and prior experiences using cannabis while programming. For general cannabis usage, we used the validated *Daily Sessions, Frequency, Age of Onset, and Quantity of Cannabis Use Inventory* (DFAQ-CU) [79]. This is the same validated assessment that we used in our survey of developers (see subsubsection 5.2.2.1).

[10] https://github.com/CelloCorgi/CannabisObservationalStudy

5.3.2.2 Short Programming Problem Stimuli



Please click the corresponding letter which best represents the **return value** of the function call below:

```
def func(x, y):
    return (y != x) and (not x or not y)

func(True, 6 < 20)
```

True
A

False
B

(a) Boolean problem (answer is False)

Please type the **output** of the function call below:

```
def func(nums):
    x = 2;
    for i in range(len(nums)):
        x += nums[i]
    print(x)

nums = [1, 2, 3, 4]
func(nums)
```

(b) Code-tracing problem (answer is 12)

Implement a function `containsDuplicate` that accepts a list of integers and returns true if it contains a duplicate element, and false otherwise.

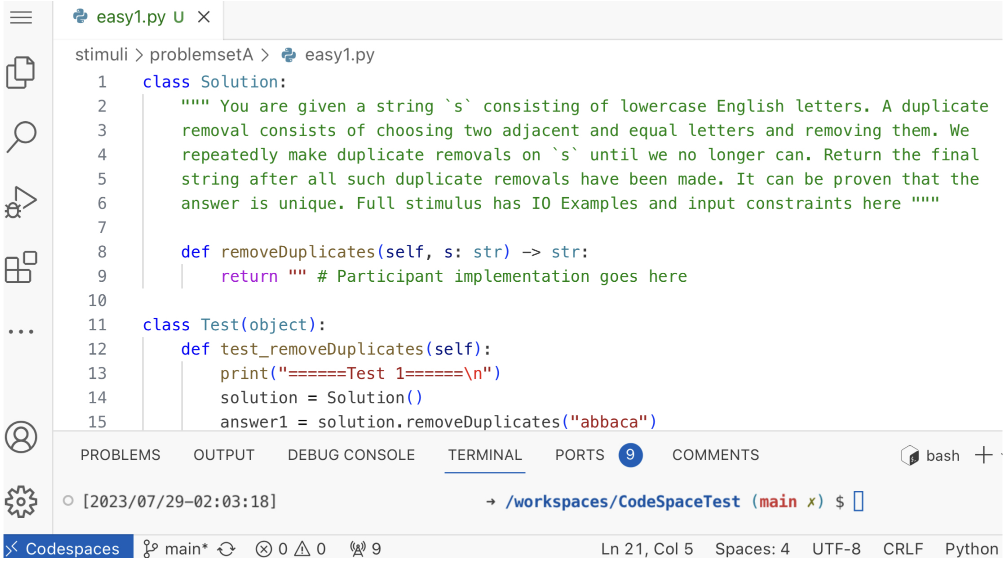
(c) Code writing problem

Figure 5.3: Example Short Programming Stimuli: adapted from the program comprehension literature.

Each session included a series of short programming questions. We adapted stimuli from the programming comprehension literature, specifically those that use neuroimaging [95, 181]. We chose to do this because such stimuli are explicitly designed to study targeted programming aspects in a controlled manner while still being completed quickly. We included three types of programming tasks: Boolean questions, code-tracing questions, and code-writing questions (see Figure 5.3 for examples). The Boolean and code-tracing stimuli were adapted from those that we used in our fNIRS study of program comprehension in lens 2 (see subsubsection 4.2.1.2), while the code-writing stimuli were adapted from a study of code writing and prose writing [181].

Consistent with their design and use in program comprehension studies, each stimulus was timed: participants had at most 30 seconds for each Boolean problem, 45 seconds for each code-tracing problem, and 90 seconds for each code-writing problem. In each session, each participant completed six problems from each sub-type that were randomly sampled from our corpus.

Metrics and Scoring. Boolean problems were graded automatically. Code-tracing and code-writing problems were assessed manually by marking responses as either correct (full score), partially correct (half score), or incorrect (zero). A manual assessment of responses was done without knowing if the response was produced while high or sober. Our full rubric is available in our replication package. Graded responses were aggregated into percent-correct values per sub-type per programming session for use in analysis.



```python
class Solution:
    """ You are given a string `s` consisting of lowercase English letters. A duplicate
    removal consists of choosing two adjacent and equal letters and removing them. We
    repeatedly make duplicate removals on `s` until we no longer can. Return the final
    string after all such duplicate removals have been made. It can be proven that the
    answer is unique. Full stimulus has IO Examples and input constraints here """

    def removeDuplicates(self, s: str) -> str:
        return "" # Participant implementation goes here

class Test(object):
    def test_removeDuplicates(self):
        print("======Test 1======\n")
        solution = Solution()
        answer1 = solution.removeDuplicates("abbaca")
```

```
○ [2023/07/29-02:03:18]                    → /workspaces/CodeSpaceTest (main ✗) $
```

Figure 5.4: Example Interview-Style Programming Problem: presented in the study platform (shortened for space).

5.3.2.3 “Programming Interview” Style Stimuli

Each session included three “Programming Interview” problems. These were taken directly from LeetCode, a popular platform for practicing technical interview skills. Performance on these coding challenges has been found to reflect a software engineer’s fundamental computer science knowledge, ability to find efficient and scalable algorithms for unknown problems, and skill at testing or debugging a short piece of code [215]. Each problem consisted of a natural language specification, a function stub, and 2–3 basic tests. They admit implementation in a file where programmers can type, run, and edit code. We recorded the state of each program every 15 seconds, along with all keystrokes and terminal interactions. Figure 5.4 shows an example of the study platform and an indicative stimulus.

To choose our stimuli, we first selected 20 potential problems labeled “easy” or “medium” on LeetCode. We avoided those marked “hard” due to the time constraints imposed by our study design. We categorized these problems into three groups by their primary data structure: a 1D-array, a 2D-array, or a recursive data structure (list or tree). Through a series of pilot experiments, we selected six problems (two of each type) that were non-trivial but could be completed in the available time. In each session, participants completed one each of the two 1D-array, recursive, and 2D-array problems. A participant never saw the same problem twice. Problem order was randomized across sessions and study conditions to minimize between-problem variance and learning effect impacts. We partially controlled the difficulty of the problem pairs using the LeetCode problem difficulty and solve rate. For example, the two 1D-array and two recursive problems were “easy” on LeetCode, while the two 2D-array problems were marked as “medium”. Participants had 15 minutes for each easy problem (1D-array, recursive) and 20 minutes for the medium problem (2D-array).

Automated Metrics and Scoring. We assessed correctness via held-out test suites. As LeetCode does not publish its own hidden tests, we constructed our own held-out *correctness tests*. The number of correctness test cases ranged from 24 to 34 per problem. All held-out test suites achieved full branch coverage on the published LeetCode solution. We analyzed the maximum correctness score across all saved file versions for a given solution. We use the best score rather than the final score because in cases where a participant ran out of time, the last score is often much lower because the participant was mid-edit. We also made 11 additional *efficiency tests* per problem. These consisted of inputs of increasing size (including very large inputs) and were run separately from the correctness tests to ascertain the run-time efficiency of correct solutions.

Manual Annotation. We qualitatively analyzed the 1D-array problem solutions to permit a nuanced analysis of design and code style factors that may differentiate high and sober programmers. We annotate algorithmic method choices (*e.g.*, brute force, dynamic programming, *etc.*), and code style features (*e.g.*, comments, helper functions, *etc.*). To determine algorithmic method categories, one author manually clustered the most up-voted Python solutions on LeetCode. This initial set was validated by another author. A third author manually assigned participant solutions to these clusters. Solutions about which the annotator was unsure were shown to the others and final categorizations were confirmed via consensus.

Problem Difficulty. We validated that the two alternate problems in each pairing had similar difficulties for our population, regardless of LeetCode labels, as a potential source of bias. We used a within-subjects t-test of participant scores. For the 1D-array and recursive pairings, we find no significant differences in difficulty ($p = 0.82, 0.66$ respectively). For the 2D-array problems, however, there is a significant difference in difficulty (68% vs. 34% on average, $p < 0.00001$). We consider this discrepancy when interpreting our results for the 2D-array problems in subsubsection 5.3.5.1 and subsubsection 5.3.5.2.

5.3.3 Observational Study: Participant Overview

We overview our recruitment process and then describe the demographics, programming background, and cannabis usage history of our final $n = 74$ participants to contextualize our results.

5.3.3.1 Recruitment Process

Participants were recruited via flyers posted around four American metropolitan areas (San Francisco Bay Area, Ann Arbor, Seattle, and New Haven). Each area is in a US state where recreational

Table 5.5: Cannabis Observational Study's Participant Demographics: Demographics and experience for the $n = 74$ cannabis observational study participants.

Gender	
Man	53 (72%)
Woman	15 (20%)
Non-binary	6 (8%)
Age and Programming Experience (Average, (Min–Max))	
Age	24, (20–49)
Programming experience (years)	5,[11] (1–30)
Has 1+ years of professional programming experience	48 (65%)
Computing-related employment status (could select multiple)	
Currently Employed at a CS-related job	28 (38%)
Undergraduate Student in CS-related field	37 (50%)
Graduate Student in CS-related field	12 (16%)
Unemployed or N/A	3 (4%)

cannabis is legal (California, Michigan, Washington, and Connecticut). Within these areas, posters were placed primarily near the offices of technology companies and on university campuses. Prospective participants were directed to an online pre-screening form for eligibility requirements (subsection 5.3.1). This verification included eight programming questions to ensure sufficient Python performance. Of the 640 who completed the pre-screening, 247 obtained the perfect score that was required to participate. Participants were contacted in batches on a first-come-first-serve basis, with a preference for those with more professional programming experience. In total, we sent 205/247 invitation emails before closing recruitment.

Of the 205 invited potential participants, 85/205 finished the initial survey and scheduled programming sessions. This is a response rate of 42%. Of these, 74 attended at least one session and 71 attended both, a study retention rate of 84%. This overall rate aligns with previous software engineering studies with multiple sessions [94, 105]. Upon completion of the study, participants were compensated with an 80 USD gift card to the company of their choice. All data collection occurred in 2023.

5.3.3.2 Population Contextualization

To better contextualize our results, we now describe the demographics, programming experiences, and cannabis use histories of our population. A summary of these numbers is available in Table 5.5.

Participants ranged in age from 20 to 49 (average 24), with 72% men, 20% women, and 8% non-binary. Our population had a mix of students and full-time professional developers. About half

(37/74) were undergraduates in a computing field. The remainder were either graduate students (16%, 12/74) or professional programmers (38%, 28/74). A few reported both student status and current programming employment. This split between students and professionals is consistent with the locations where we put up recruitment posters.

Participants reported a range of programming experiences. While all had at least one year of programming experience (a requirement to participate), eight participants had over 10 years of experience, and the median was 5.[11] As for professional programming experience, participants ranged between none and over 20 years. The majority of participants in our sample had professional programming experience (94%), with the median participant reporting between 1 and 5 years: 65% had at least one year of professional experience. Of those who had at least one year, the most common job titles reported were "software engineer" or "software developer".

Cannabis Usage History. All eligible participants had used cannabis in the last year. Specific usage levels varied substantially, ranging from once a year to more than once every day. The median participant reported using cannabis twice a week. In addition, the majority of our participants had experience programming while high (66%, 49/74). The frequency of this use ranged from under once a year to five or more days a week. The wide array of cannabis usage histories, both with programming and without, allows us to systematically investigate if the magnitude of prior cannabis use mediates the impacts of cannabis intoxication on programming performance.

5.3.4 Observational Study: Research Questions and Analysis Overview

We structure our analysis in two parts: a primary hypothesis-driven analysis and a secondary exploratory analysis. The primary hypotheses and analysis plan were preregistered to mitigate biases and increase confidence in our results. However, as investigating the impact of cannabis on programming is relatively novel, we perform additional exploratory analysis to glean insights for further study. In the rest of this section, we present our research questions for both analysis parts and outline our statistical methods.

5.3.4.1 Primary Analysis: Preregistered Questions

Following more recent best practices in both software engineering (*e.g.*, the "Registered Reports" track of Mining Software Repositories [291]) and psychology and the social sciences (*e.g.*, [295]), we *preregistered* our hypotheses before conducting our analysis.

In preregistration, the "research rationale, hypotheses, design and analytic strategy" are submitted before beginning the study [115]. As a result, biases associated with researchers choosing which

[11] Estimate: years of experiences was reported in ranges (*e.g.*, 1–2, 3–5, 6–10, *etc.*)

results to present after the fact may be mitigated: “preregistration can prevent or suppress HARKing, p-hacking, and cherry-picking since hypotheses and analytical methods have already been declared before experiments are performed” [351]. Similarly, preregistration may mean that “researchers will not be motivated to engage in practices that increase the likelihood of making a type I error” [115].

For this study, we preregistered four research questions and associated hypotheses with the Open Science Framework (OSF), along with our data collection strategy and statistical analysis methods:[12]

- **RQ1—Program Correctness:** How does cannabis intoxication while programming impact program correctness?
  - *Hypothesis:* Programs will be less correct when written by cannabis-intoxicated programmers.
- **RQ2—Program Speed:** How does cannabis intoxication while programming impact programming speed?
  - *Hypothesis:* Cannabis-intoxicated programmers will take longer to write programs.
- **RQ3—Program Method Divergence:** Does cannabis use influence programmer algorithmic method choice?
  - *Hypothesis 1:* Correct programs by high participants will run slower than those by sober participants (*i.e.* are less efficient or have higher algorithmic complexity).[13]
  - *Hypothesis 2:* Solutions to free-form programming problems by cannabis-intoxicated programmers will exhibit greater method choice divergence and diversity.
- **RQ4—Cannabis Use History:** Does cannabis usage history mediate the effect of cannabis intoxication on programming outcomes?
  - *Hypothesis:* The impact of cannabis use while programming will be lessened for heavy vs. moderate users.

[12] OSF preregistration is available here: https://osf.io/g6fds. We note that in the preregistration, *RQ3* mentions *creativity* instead of *program method divergence*. We changed this name in this dissertation to better match our methods.

[13] In our preregistered hypotheses, this was listed under RQ2. We present it under RQ3 for thematic and narrative clarity.

5.3.4.2 Exploratory Analysis

We also consider two exploratory research questions:

- **E-RQ1—Code Style:** Does cannabis intoxication impact stylistic code properties (*e.g.*, code comments, *etc.*)?
- **E-RQ2—Self Perception:** Are programmers able to accurately assess how cannabis impacts programming performance?

5.3.4.3 Statistical Methods

Our analysis was primarily conducted in a Python Jupyter Notebook using `Pandas` [343]. Some analyses, especially those informed by data visualization, were done using Excel. For statistical tests, we primarily used the `SciPy` [331] and `Statsmodels` [284] libraries.

Statistical Significance. We consider results significant if $p < 0.05$. When testing for a significant difference between sober and cannabis-intoxicated programming using continuous variables (*e.g.*, percent correctness scores or response time such as in RQ1 and RQ2), we use a paired samples t-test unless otherwise noted. Assumptions of normality are confirmed through the inspection of histograms. While we primarily use paired tests (as is appropriate with our within-subjects design), in some cases (*e.g.*, missing data, *etc.*) we use a non-paired test and note the specific test used in the text.

For categorical values (*e.g.*, program method choice in RQ3), we use a χ^2-test. For the difference between two binary variables (such as if a solution has comments or not in E-RQ1), we use the n-1 χ^2-test (*i.e.*, the proportions z-test) [58]. We treat the responses to Likert questions (*e.g.*, self-perception of cannabis impact in E-RQ2) as continuous variables.[14] The Student's t-test is thus appropriate.

Multiple Comparisons. We investigate multiple research questions and conduct multiple statistical tests per research question. To avoid fishing or p-hacking, we preregistered our primary hypotheses and analysis plan and we report results for each. Within each research question, we correct for multiple comparisons for all tests used to accept or reject the null hypothesis. We use Benjamini-Hochberg Correction, with a false discovery threshold of $q = 0.05$: unless stated otherwise, all significant results pass this threshold.

Effect Size. We use Cohen's d (with pooled standard deviation) to assess the size of differences tested by t-tests. We consider $0.2 < r \leq 0.5$ a small effect, and values above 0.5 a medium effect.

[14] Although they are ordered categorical variables and normality cannot be assumed, with large samples, parametric tests are sufficiently robust for analysis [237].

	Sober	High	Diff	p	BH-p	d	Sober	High	Diff	p	BH-p	d
RQ1: Short Programming Problems, Correctness Scores							*RQ2: Average Stimulus Time (in seconds)*					
Boolean	81.5%	81.0%	-0.5%	0.846	0.846	0.03	14.2	14.7	+0.5	0.310	0.465	0.10
Code-tracing	62.3%	52.1%	-10.2%	<0.001	**0.003**	0.42	31.7	32.1	+0.4	0.656	0.656	0.06
Code-writing	56.9%	46.4%	-10.6%	<0.001	**0.003**	0.44	67.4	70.4	+3.0	*0.065*	*0.130*	0.23
RQ1: "Programming Interview" Problems, Correctness Scores							*RQ2: Average Overall Time (in min)*					
Problem 1: String/1D-array	65.9%	56.4%	-9.5%	0.033	**0.049**	0.28	9.7	11.1	+1.4	0.012	**0.039**	0.32
Problem 2: Recursive List/Tree	48.5%	34.5%	-14.0%	0.012	**0.024**	0.35	11.8	13.0	+1.2	0.013	**0.039**	0.33
Problem 3: 2D-array	53.9%	48.4%	-5.5%	0.383	0.460	0.15	16.2	16.8	+0.6	0.500	0.656	0.11

Table 5.6: Overall Impacts of Cannabis Intoxication on Programming: The impact of cannabis intoxication on programming correctness (RQ1) and speed (RQ2). All significance tests are paired t-tests, while effect size calculations use Cohen's d. Cells that are bold and highlighted in green are those differences that are significant with Benjamini-Hochberg correction ($p < 0.05$, $q = 0.05$). Cells in italics and highlighted in yellow indicate a trend that did not reach significance. Notice that all differences, even those that did not reach significance, indicate decreased performance while high: cannabis-intoxicated programmers take more time to write more incorrect programs.

For correlations, we use Pearson's r, with $0.1 < r \leq 0.3$ a weak correlation, $0.3 < r \leq 0.5$ a moderate correlation, and $0.5 < r$ a strong correlation.

5.3.5 Observational Study: Analysis Results

We present the results of our preregistered (RQ1-RQ4 subsubsection 5.3.4.1) and exploratory (E-RQ1, E-RQ2, subsubsection 5.3.4.2) questions. Subsection 5.3.2 details experimental tasks, including the metrics used. A discussion of our statistical methods is in subsubsection 5.3.4.3.

5.3.5.1 Observational Study: RQ1 — Impacts on Program Correctness

We first investigate how programming while intoxicated impacts program correctness. We do this by using paired t-tests to compare sober vs. cannabis session percent correctness for each short program comprehension task type and "interview style" coding question. This results in six total significance tests.

At a high level, we find strong evidence supporting our hypothesis that cannabis-intoxicated written programs are less correct. For all correctness score comparisons, participants' cannabis correctness scores were *lower*, on average, than sober scores. For four out of the six comparisons, this difference was statistically significant ($0.0005 < p < 0.05$) with a small to small-medium effect ($0.28 \leq d \leq 0.44$). Table 5.6 summarizes our top-level results.

Short Programming Problems. Participants completed several Boolean logic problems, code-tracing problems, and code-writing problems during each session. For the Boolean task, we do not

(a) Code produced by participant when sober

```python
def is_sorted(integers):
  for i in range(len(integers)-1):
    if integers[i] > integers[i+1]:
      return False
  return True
```

(b) Code from the same participant when intoxicated

```python
def is_sorted(input_list):
  return helper(None, input_list)

def helper(min_val, input_list):
  if len(input_list) == 0: return True
  if min_val > input_list[0]: return False
  return helper(input_list[0], input_list[1:])
```

Figure 5.5: Code While Sober Vs. Code While High: Indicative example comparing code produced while high vs. sober by the same participant for the same problem. The intoxicated code is more complicated and contains a bug.

find evidence of cannabis intoxication impairing performance ($p = 0.85$). However, for both the code-tracing task and the code-writing task, we find that cannabis intoxication has a significant negative impact on performance ($p = 0.0009$, $p = 0.0005$ respectively). For both of these tasks, the negative impact was of a small-medium effect ($d = 0.42$, $d = 0.44$). To put this in perspective, correctness scores were, on average, 10% lower when intoxicated (52% vs. 62% for code-tracing and 46% vs. 57% for code-writing). Cannabis impairment at ecologically-valid levels can have a negative impact on two fundamental software development tasks: code reading via tracing and code writing. Additionally, the presence of an effect for the two more cognitively demanding tasks, but not for the simpler Boolean task, may indicate that cannabis intoxication impacts scale with task complexity; if so, this trend could relate to known effects of cannabis on working memory [180].

To understand the factors driving this decreased performance, we perform an informal qualitative analysis of the types of errors that cannabis-intoxicated programmers are likely to make on the code-tracing and code-writing tasks. We observe that high programmers often complicate their solutions and add extraneous conditionals while still missing edge cases. Figure 5.5 shows an indicative example: code produced by the same participant for the same problem (on different days) while high and sober. The code produced while intoxicated features a more complicated structure (recursion via a helper function) as well as more opportunities for simple mistakes (*e.g.*, three indexing operations vs. two, two conditional branches vs. one, *etc.*). The intoxicated solution contains one such error, comparing `None` with a number, which raises a `TypeError`.

"Interview style" Problems. While the short problems highlight the impact of cannabis on specific programming aspects, for a more holistic understanding we consider the three longer "interview style" problems. Participants never completed the same interview problem twice: problems were paired and counterbalanced across sessions by algorithm type and difficulty to permit within-subjects analysis. The first problem always involved 1D-arrays, the second recursive trees or lists, and the third 2D-arrays.

We find that cannabis significantly impaired correctness for the 1D-array problems and recursive problems. For the 1D-array problems, participants passed 10% fewer correctness tests in the cannabis-intoxicated session than in the sober session (56% vs 65%, $p = 0.033, d = 0.28$). For the recursive problems, participants passed 14% fewer correctness tests in the cannabis-intoxicated session than in the sober session (34% vs. 48%, $p = 0.012, d = 0.35$).

For the 2D-array problems, while participants passed 6% fewer correctness tests in the cannabis session, this trend did not rise to the level of significance (48% vs 54%, $p = 0.383, d = 0.15$). We note, that in contrast to the 1D-array and recursive problems, the two 2D-array problems that we chose were not of equivalent difficulty for our population (Figure 5.3.2.3). This confound may explain the lack of an observed significant difference on this problem pair.

In our preregistered analysis plan, we stated that "The null hypothesis will be rejected if high programmers have lower scores on the majority of interview-style[15] problems when high." As we found a significant difference in 2/3 interview-style problems, we reject the null hypothesis and conclude that cannabis intoxication has a significant negative effect on program correctness. Specifically, this result shows that the impairment we observed on the controlled short programming tasks persists when implementing more complicated functions. We discuss implications in section 5.4.

We find support for our hypothesis that **cannabis use decreases program correctness** with a small-medium effect ($0.0005 < p < 0.05$, $0.28 \leq d \leq 0.44$, 10–14% fewer passed tests). Cannabis impairs *writing* and *tracing* through programs.

5.3.5.2 Observational Study RQ2 — Impacts on Programming Speed

We next investigate the impact of cannabis on programming speed. We hypothesized that cannabis-intoxicated people will program more slowly. For the short programming problems, we use a paired t-test to compare the average stimulus completion time per task type (Boolean, code-tracing, code-writing) per condition (sober or intoxicated). For the "interview-style" questions, we compare each problem type's total completion time. Our results are in Table 5.6.

Short Programming Problems. We find no significant evidence that cannabis intoxication impacts programming time for simpler programming tasks. For the code-writing task (the most complicated

[15] The original references LeetCode explicitly. We use "interview-style" in this dissertation.

of the three), we observe a trend toward significance with task completion taking longer when intoxicated with a small effect ($p = 0.065$, $d = 0.23$). However, this difference neither reaches significance nor passes our multiple comparison threshold.

"Interview Style" Problems. In contrast, we find a significant programming speed difference for 2/3 of the more complex "interview style" problems. For the 1D-array problems, high participants spent an average of 84 more seconds than sober participants (11.1 minutes vs. 9.7 minutes, $p = 0.01, d = 0.32$). For the recursive problems, high participants spent an average of 83 more seconds than sober participants (13.0 minutes vs. 11.8 minutes, $p = 0.01$, $d = 0.33$). We do not observe a significant difference between completion times for the 2D-array problems. However, as with correctness (subsubsection 5.3.5.1), this lack of result may be attributable to uneven difficulty matching for this problem pair.

Why Are Programmers Slower? We investigate the factors driving the difference for the "interview style" questions: are intoxicated programmers slower because they physically type slower, because they make more typing corrections, or because they have less "active typing time" (time spent not actively programming, but rather thinking or searching online for help)?

We find that all three factors contribute! We compared participants' overall typing speed in both sessions using a paired t-test. Sober participants typed 6 more characters per minute than cannabis-intoxicated participants on average (84 chars/min vs. 90 chars/min, $p = 0.0004, d = 0.32$). We excluded the time when participants were not actively typing, defined as any period between two keystrokes longer than 8s, from the total time used to calculate typing speed. For corrections, cannabis-intoxicated participants delete more of their keystrokes than sober participants (20.9% vs. 18.5%, $p = 0.00003, d = 0.35$). This result, along with the slower typing speed, aligns with work on general cannabis impairment, which finds a negative impact on fine motor control [180]. Finally, we find that cannabis-intoxicated participants spend more of their total time not actively typing code (64.9% vs. 60.6%, $p = 0.003, d = 0.36$). We visualize these typing-related differences between the sober and cannabis sessions for a single indicative participant in Figure 5.6.

For "interview-style" tasks, **cannabis use impairs programming speed** ($p < 0.04$, $d = 0.3$, 10–14% slower). This decrease in speed is associated with typing slower, deleting more characters, and more time spent not typing.

5.3.5.3 Observational Study RQ3 — Method Choice and Divergence

We now investigate if high and sober programmers *choose to solve the same programming problem in different ways*. We consider both the *efficiency* of solutions and also the *algorithmic method* implemented. We have two hypotheses: first, correct programs by high participants will be less efficient than those of sober participants. Second, we hypothesize high participants will show more

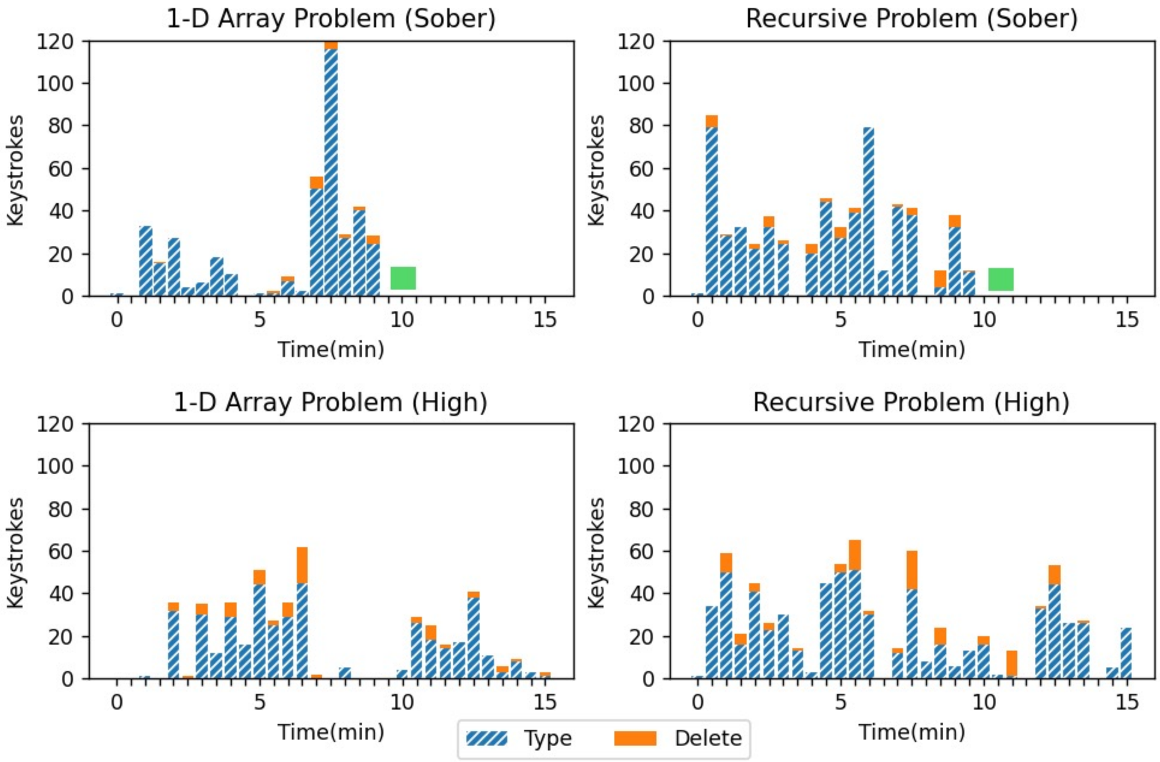


Figure 5.6: Programming Process Wile Sober Vs. While High: Histograms with typed characters (dark blue with lines) and deletions (orange) over time for the same participant while sober and high. The high condition features longer pauses and more deletions. The participant also had a higher maximum typing speed sober (120 keystrokes in 30 seconds vs. only 65 while high). This participant finished both problems early when sober (noted by the green box) but ran out of time when high.

divergent choices in algorithmic or methodological approaches (one potential aspect of "creativity"), compared to sober participants.

We focus on the "interview-style" problems, as those tasks are complex enough for a meaningful algorithmic analysis. For RQ3 we analyze all six problems separately (*i.e.*, two 1D-array, two recursion, and two 2D-array problems). This is done because the running times of difficulty-paired problems may vary significantly. For example, for the 1D-array problem type, one instance features a 1D-array as a Python list while the other uses a Python string. While conceptually similar, Python treats these very differently from an efficiency standpoint (*e.g.*, list instances are mutable while strings are not). The statistical comparisons in this research question thus use non-paired tests unless otherwise noted.

Program Efficiency. We measure the efficiency of correct solutions on very large program inputs (subsubsection 5.3.2.3, average of three trials). All program running times were generated using the same multi-core Linux server and were run sequentially. Despite generous timeouts for the efficiency tests, some particularly inefficient correct solutions failed to terminate for our biggest inputs. For these problems, we assign them our maximum timeout of 60 seconds.

We compare the efficiency test runtimes for correct solutions written while high to those written while sober for each of the six "interview-style problems". For 5/6 problems, the differences are not significant. For one of the two 1D-array problems, the difference is significant before multiple comparison correction ($p = 0.03$, $d = 1.03$). For this problem, the solutions by sober participants are actually *less efficient* than those by high participants (5.0 seconds vs. 21.7 seconds). While intriguing, this result does not survive correction for multiple comparisons (corrected $p = 0.18$). Additionally, few high and sober participants correctly implemented the problem (9 and 12 respectively), leading to low statistical power. We cannot reject the null hypothesis regarding efficiency.

Solution Divergence. We manually annotate the solutions to both 1D-array problems to obtain a more nuanced understanding of method choice differences between high and sober participants. Unlike our efficiency analysis, we manually annotate the solutions for all participants, even those who did not arrive at the correct solution. This is so because there is qualitative evidence that cannabis use might improve programming *creativity* [232], a theme also observed in our survey (see Table 5.3). If this is the case, even if solutions produced by cannabis-intoxicated programmers contain more bugs, other benefits may offset this cost; informally, a developer might generate a solution while using cannabis and then come back the next day to fix any errors.

To the best of our knowledge, while creativity is an important aspect of the software development process, a robust metric remains an open problem [124]. One approach used by prior work is to measure *divergence* in computational patterns [31]. Divergence tests have long been the basis of common creativity measurements [273]. We adapt this use of solution divergence to method choice.

The possible method choices were specific to the first or second problem instance and included categories such as brute force using a loop, brute force using recursion, or a stack data structure. For both problem instances, a couple of submissions did not fall into any category and were labeled other, or were categorized as completely incorrect. We overview our method annotation process in more detail in subsubsection 5.3.2.3. We applied a χ^2 test with the groups as sober and intoxicated and the categories as the different methods for each problem. For the first problem instance, $\chi^2 = 1.68, p = 0.89$. For the second problem instance, $\chi^2 = 8.44, p = 0.077$. We find no evidence that method choice differed significantly between high and sober participants. As a result, it does not make sense to investigate if methods chosen by high participants were more diverse because there was no significant difference between the two distributions. Overall, we find no evidence to support our hypothesis that high programmers generate more diverse or more creative programming solutions. We discuss implications in section 5.4.

We found **no statistically significant evidence that cannabis intoxication impacts solution efficiency or implementation divergence** ($p \geq 0.077$). We do not reject the null hypothesis that programmers using cannabis exhibit the same divergence of method choice as sober counterparts.

5.3.5.4 Observational Study: RQ 4 — Influence of Cannabis Use History

For our last preregistered hypothesis, we investigate if cannabis use history mediates the negative impact of cannabis intoxication on program correctness. We hypothesize the impact of cannabis use while programming will be lessened for those who are heavy cannabis users vs. those who are moderate users.

We divide participants into two groups for analysis: heavy users and light users, classified by if their aggregated $z-$transformed scores on the DFAQ-CU use frequency questions are positive or negative [79].[16] 38 participants are classified as light users while 33 are classified as heavy users (roughly, those participants who use cannabis more than two times a week). We then calculate the per-participant difference between high and sober sessions for all correctness scores for which we observed significant general impairment: code-tracing problems, code-writing problems, the 1D-array problem type, and the recursive-data structure problem type. We use an independent t-test to compare the performance differences of light and heavy users on each test to see if one group experiences significantly more cannabis-related impairment than the other.

We find no significant differences in cannabis-related impairment between heavy and light users ($0.35 \leq p \leq 0.88$). To confirm this null result, we additionally compute pairwise correlations between inter-session performance correctness differences and two cannabis-related features: self-reported current intoxication level and lifetime cannabis usage amount. We find no significant correlations with these comparisons ($-0.26 < r < 0.13$ for all correlations).

We find **no significant evidence that cannabis impacts heavy users less than others** ($p \geq 0.35$). We do not reject the null hypothesis and instead conclude that cannabis impairs all programmers equally, regardless of cannabis use history.

5.3.5.5 E-RQ1 — Impacts on Code Style

We explore the impact of cannabis use on code style. While annotating participant method choices for the two 1D-array problems, we also annotated responses for various stylistics features. In particular, we marked if a participant added comments, print statements, helper functions, or additional test cases to their implementation. We also counted the number and maximum nesting

[16] In the preregistered hypotheses, we said we would use three groups: light cannabis users (at most 3–4 times per month), moderate cannabis users (1–2 times per week), and heavy cannabis users (2+ times per week). We use an aggregated score instead after a closer inspection of the DFAQ-CU assessment's scoring instructions [79].

depth of loops and conditionals to get a measure of the branch and loop complexity of the code. We compare proportions for stylistic features between high and sober participants using the n-1 χ^2-test. This analysis determines if the correctness impairment of cannabis extends to stylistic properties which, while non-functional, facilitate software readability and maintainability [99, 110].

We find no significant style differences between programs written while high vs. sober ($0.20 \leq p \leq 0.85$). While exploratory, this may mitigate concerns about cannabis substantially compromising code clarity and ease of understanding, which are critical for successful collaboration and future program modifications.

We find **no significant evidence that cannabis impacts programming style** (*e.g.*, comments, helpers, *etc.*) ($p \geq 0.20$).

5.3.5.6 E-RQ2 — Self Perception of Impact

For our second exploratory analysis, we investigate if participants are able to correctly perceive their relative programming performance, even when intoxicated. We explore the answer to this question because prior studies have reported that cannabis-using developers self-assess task contexts and potential impairment when making usage decisions [232]. To answer this question, we first investigate how participants perceived their programming performance while high vs. sober. At the end of their second session, we asked all participants about their performance on the "interview style" questions in that session compared to the previous one. 63% thought they performed worse during their cannabis session, 20% could not tell either way, and only 17% thought they performed better while high. Figure 5.7 breaks down those reports.

While participants did, on average, have decreased performance while high, this was not universal. We thus investigate *if participant perceptions of their performance while high is accurate*: we correlate perceived performance difference (a 7-point Likert scale) with actual performance difference (average difference between percent correctness scores for all three "interview style" problems). We find a strong, positive correlation between self-reported relative performance while high and actual performance: $r = 0.59$. Of the 49/75 participants who showed decreased performance on the interview problems while high, only four of them (8%) incorrectly perceived increased performance. Together, these results indicate that programmers are generally able to accurately judge their relative programming performance, even when under the influence of cannabis. This has implications for policy (see section 5.4).

Most programmers **can accurately judge relative programming performance while high** ($r = 0.59$).

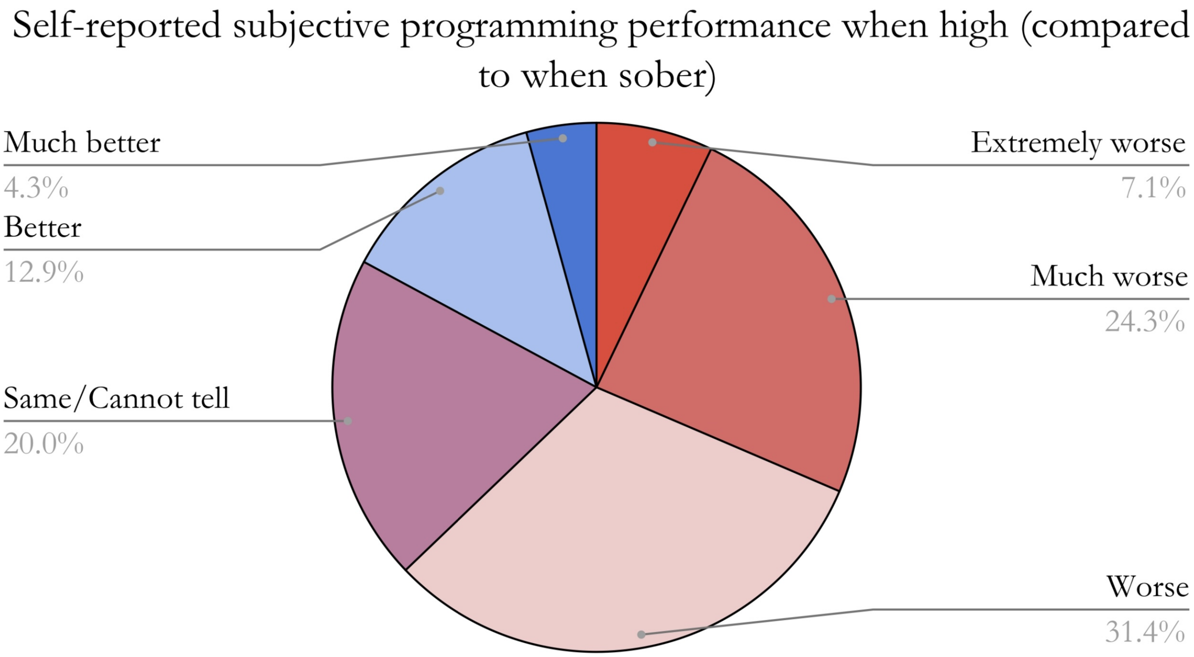


Figure 5.7: Self-perceived Programming Performance while High: Self-reported subjective programming performance in cannabis-using sessions compared to sober sessions. Most participants report perceiving decreased performance when high (63%) compared to only 17% who perceived improvement.

5.3.6 Observational Study: Results Summary

In our controlled observational study with 74 participants, we find that at ecologically-valid dosages, **cannabis intoxication has a significant small-medium impairment on both program correctness and programming speed** ($p < 0.5, 0.22 \leq d \leq 0.44$). We did not find evidence of cannabis increasing solution divergence. We also did not find that past cannabis usage history significantly mediates programming impairment. However, even when under the influence of cannabis, programmers correctly perceive differences in their programming performance ($r = 0.59$). We hope our results contribute to the development of evidence-based policies and assist software developers in making informed decisions.

5.3.7 Observational Study: Threats to Validity

Our results regarding the impact of cannabis use on programming may not generalize to other populations. We highlight a number of considerations.

First, our participants are not a random sample of the population. Our selection may be biased toward those interested in cannabis-related studies or with a positive perception of cannabis. We partially mitigate this by assessing the cannabis usage history of our participants (see subsection 5.3.2).

In addition, we are interested in understanding how programmers who routinely use cannabis are affected by it in development settings: in that context, a participant who has not used cannabis before is less indicative of the daily impact on a company. Similarly, the legal status of cannabis in some locations may deter participation in our study. We partially mitigated this by recruiting in four US states where this sort of cannabis use is legal.

Second, our larger programming tasks were taken from the LeetCode repository of skills-based interview questions. These questions may not be indicative of industrial practice [27]. This is partially mitigated by the fact that they are indicative of programming tasks people carry out and study for in the hiring process.

Third, our notions of code quality and divergence may not generalize. We assess code correctness via tests and assess divergence and style by expert annotation. There are other useful notions of utility (*e.g.*, formally proving correctness or using other static analyses, measuring maintenance efforts, *etc.*) that we do not capture. We partially mitigate this concern, noting that automated regression testing remains a dominant activity in SE and that manual assessment is relevant for both code reviewing and hiring decisions. There are other indicators revealing creativity in software engineering problem solving [124], and other factors linked to programming creativity (*e.g.*, knowledge [145], personality [13, 124]), but we only measure divergence in products.

Fourth, we are unable to control the amount of cannabis affecting participants. Our IRB protocol did not permit *dispensing* cannabis, directing participants to take a particular amount, or collecting blood samples — instead, our *observational* study involves participants using cannabis anyway. We partially mitigate this via photographs of cannabis products used (and include this self-reported amount of marijuana and THC in our replication materials), and by restricting attention to one delivery method (smoking/vaping, but not edibles). Experienced and novice cannabis users may make different dosage decisions and have different tolerances (*e.g.*, [41]), something our approach does not capture.

5.4 Lens 3 Discussion and Conclusion

In this productivity lens, we conducted a case study on how one external factor (cannabis use) can impact programming productivity. We first surveyed over 800 programmers and we found that some programmers use cannabis while programming, both in personal and professional projects, with many reporting perceived programming enhancement from its use (section 5.2). Next, we conducted the first controlled observational study of the impact of cannabis on programming, finding evidence that cannabis impairs both code correctness and programming speed (section 5.3). However, our work is only a first step toward understanding this intersection, and we briefly discuss the implications of this dissertation on company drug policy. We also consider the implication of our case study on the importance of using controlled observational studies to verify anecdotal claims about programming productivity.

Cannabis Use and Company Policies. Our findings have implications for software company anti-drug policies. While perhaps less prevalent than in other industries (notably, many FAANG (Facebook, Apple, Amazon, Netflix, Google) companies do not drug-test employees), anti-cannabis regulations remain common in programming environments (*e.g.*, [66, p. 12] and [154, pp. 12–13]). These policies are often enforced through drug testing: 29% of our survey participants reported taking a drug test for a programming job.

In our survey (see section 5.2), we also found that cannabis use is common among programmers, with 35% of our sample having used cannabis while programming, 34% of which sometimes do so for work-related tasks. Cannabis-using programmers typically use THC products, the chemical detected by most drug tests. Furthermore, perceptions of cannabis use while programming are mostly ambivalent from both managers and employees, and we observed limited reports of cannabis use negatively impacting programming work-places: while 27% of programming managers suspect cannabis use by a supervisee, less than 3% report that those programmers were less productive, and only one manager in our sample reported reprimanding an employee for cannabis use. Thus, our survey results indicate that software company anti-drug policies may conflict with the preferred practice of many developers, a dissonance that may lead to smaller application pools and hiring difficulties for drug-testing jobs (as evidenced by the 2014 FBI cybersecurity hiring shortage [166]).

At the same time, we observe significant impairment associated with ecologically-valid cannabis use while programming (10% fewer correct tests, 10% slower programming), and we do not observe a significant method divergence benefit (*e.g.*, creativity). As a result, *our findings paint a more nuanced picture, especially for situations without a robust mechanism to catch bugs or with deadline pressure.*

We consider the conflicts between our survey and observational study results more directly. In our survey, programmers most commonly reported using cannabis to either increase programming enjoyment or to enhance perceived programming performance (*i.e.*, improved focus and creativity, see Table 5.3). During the observational study, we asked participants what was different when programming while high (after completing both observational study sessions, see subsection 5.3.1). In contrast to our survey results, the majority emphasized that it was harder to focus and easier to get distracted. However, some participants did indicate more enjoyment, fewer worries, and increased insight into alternative perspectives. Even so, we did not observe that cannabis increases method divergence. Some of this apparent conflict may come from the design of our observational study. For example, when analyzing programs produced by observational study participants, we only considered some software solution divergence aspects (*e.g.*, we do not assess architecture or design creativity, *etc.*). "Interview-style" questions may be too structured to admit certain creative freedoms. This is relevant because in our survey programmers self-report self-regulating cannabis use by software task, using more when tasks are open-ended (see Figure 5.2). Participants also reported stress from timing and researcher observation.

In addition, although the variance we observe in outcomes for cannabis intoxication is consistent and significant, we note that it is *much less than the productivity variance already found in new hires*. A classic study reported 16–25× differences in coding times and 26–28× differences in debugging times for programmers [274, Tab. III], with no correlations to class grades or other hiring distinctions. This general pattern has continued, with a recent Microsoft study reporting that the time to first code check-in (in weeks), a job-relevant productivity metric, was 57% lower in some geographic locations than others [261, Sec. 4.1]. A 10% difference is not large compared to such an already-existing variance. In addition, some programmers in our observational study received full correctness scores even while high, or performed better when high. Most were able to accurately recognize their own cannabis-related impairment or the lack of it. Blanket employment policy may thus *not* be well-motivated.

Anecdotally, we note that several participants reached out during the study to reschedule because they had an upcoming drug screening for a new job. This aligns with the qualitative results of Newman *et al.* who found that developers view drug policies in software as ineffective and easy to circumvent [232]. Additionally, the mere existence of an anti-drug policy can serve as a deterrent for hiring and retention [232]. This, combined with the low observed magnitude of cannabis impairment, may indicate that strict drug policies might not be optimal uses of resources.

Lens 3: Overall Conclusion. Through a survey of over 800 programmers combined with a controlled observational study, we show that cannabis use is prevalent in professional software contexts and its use also impacts programming productivity by impairing both code correctness and programming speed. This research provides a case study of how one external factor, cannabis use, influences software developer productivity, filling a gap in scientific literature previously dominated by anecdotal claims. We hope this lens highlights the importance of controlled studies to verify anecdotal claims about programming productivity. Regarding cannabis use in particular, our findings suggest that company policies need to balance the prevalent use of cannabis with its actual impact on productivity.

CHAPTER 6

Discussion and Conclusions

Understanding and supporting programmer productivity remains key to improving the efficacy and efficiency of software development. Many ways to improve developer productivity have been proposed and implemented. However, without considering the various ways developers interact with their environment, productivity interventions can fail to deliver their intended benefits in practice.

In this dissertation, we present a set of systematic studies to both identify previously understudied but important factors that influence software productivity, and also provide rigorous human-focused evidence about the magnitude of their impacts. We use this approach to gain actionable productivity insights that generalize across programming populations. We focus on conducting human-focused productivity research on improving software productivity that 1) provides theoretically grounded and actionable insights, 2) includes empirical or objective measures, 3) minimizes scientific bias to support generalizability, and 4) supports diverse developers.

Overall, this dissertation's thesis is:

> *We can combine empirical evidence, theoretical modeling, and human-centered evaluation to develop and assess actionable interventions that improve programmer productivity in practice.*

We support this thesis with three key insights. First, we note that large-scale empirical evaluations can uncover previously unrecognized productivity barriers in software used by diverse programmers, drawing from collective wisdom shared in non-formal settings (*e.g.*, online programming environments or developer forums). Second, we argue that rigorous experimental design is essential for obtaining evidence-based conclusions to complex human-centered productivity questions, as it can help detect true signals in noisy real-world data. Third, we contend that interventions targeting real-world productivity barriers are more likely to be adopted. By providing empirical evidence and validating solutions in diverse contexts, we strive to offer evidence-based interventions that address specific programmer needs.

In this dissertation, we combine these insights into a set of systematic studies that are organized into three lenses into understanding and improving programmer productivity. For each lens, we conduct an initial exploratory phase and a subsequent rigorous human evaluation and/or actionable solution. Each lens also combines interdisciplinary methodologies with core software engineering techniques. Informally, we believe that more controlled methods from fields outside of software engineering can improve programming productivity for diverse groups. In this dissertation, we leverage techniques and insights from programming languages, machine learning, psychology, and medicine. In the rest of this conclusion, we detail the contributions of each lens (section 6.1) and discuss recommended directions for future investigations in each lens (section 6.2).

6.1 Summary of Contributions

In this dissertation, we presented three lenses into improving programmer productivity. In *Lens 1* (Chapter 3) we designed two efficient bug-fixing techniques to help non-traditional novices programmers write more correct code faster. In *Lens 2* (Chapter 4) we leveraged cognitive insights to develop effective developer training that helps novices program more like experts faster. Finally in *Lens 3* (Chapter 5) we showed how anecdotal external factors can influence programming productivity via a case study on how one such factor, cannabis use, influences software developer productivity. We briefly summarize the two primary contributions of each lens in more detail.

1. *Improved bug-fixing support (Productivity Lens 1)—Section 3.2:* **The first identification of, and support for, input-related bugs as a productivity barrier for novice programmers.** Our approach, INFIX, iteratively applies prioritized error-based templates (generated from a qualitative study of buggy novice inputs) and random mutations to automatically fix buggy program inputs. It is effective and efficient—INFIX can repair 94.5% of novice input errors with a median repair time of under a second, facilitating real-time support. In addition, we find that programmers view INFIX's repairs to be of high quality: in a human study with 97 participants, humans judged the output of INFIX to be equally helpful and within 4% of the quality of human-generated repairs.

2. *Improved bug-fixing support (Productivity Lens 1)—Section 3.3:* **A novel neurosymbolic approach for efficiently fixing novice parse errors.** Our approach SEQ2PARSE uses a classifier to accurately predict relevant Error Correcting Earley Parser rules for ill-parsed programs, allowing the parser to repair the program in a tractable and precise manner. SEQ2PARSE makes accurate EC-rule predictions 81% of the time when considering the top 20 EC-rules, and these predicted EC-rules let us parse and repair over 94% student parse errors in a median of 2.1 seconds, generating the user fix in almost one out of three cases. Finally, in our user study with 39 participants, we found that SEQ2PARSE's repairs are useful and helpful, even when not equivalent to the user's fix.

3. *Cognitively-informed programming training (Productivity Lens 2)—Section 4.2:* **A novel model of the cognition of new programming, developed via neuroimaging (fNIRS).** In a study of 31 participants, we compare the neural activity for introductory programming to reading and spatial reasoning tasks (mental rotation) in a controlled, contrast-based experiment. We find that all three tasks are mentally distinct for novices, but we find more significant and substantial differences between prose and coding than between mental rotation and coding. Furthermore, we find more activation in areas of the brain associated with spatial ability and task difficulty while coding compared to that reported in studies with more expert developers. These findings elaborate on previous results, relating expertise to a similarity between coding and prose reading.

4. *Cognitively-informed programming training (Productivity Lens 2)—Section 4.3:* **The first evaluation of the practical use of neuroimaging findings on software training via a controlled experiment comparing our novel CS-focused technical reading training curriculum with a standardized spatial training.** In our reading training, we teach strategies for summarizing scientific papers and understanding scientific charts and figures. We find that those in our reading training exhibit larger programming ability gains than those in the spatial training ($p = 0.02$, $f^2 = 0.10$). We also find that reading-trained participants perform particularly well on programming problems that require tracing through code ($p = 0.03$, $f^2 = 0.10$). Our results suggest that technical reading training could be beneficial for novice programmer productivity.

5. *External Factors Case Study (Productivity Lens 3)—Section 5.2:* **The first systematic survey of cannabis use while programming.** In our survey of 803 programmers, we find that some programmers regularly use cannabis while programming (18% of our sample do so at least once a month), many choosing to use cannabis for both personal and work-related projects. This use is primarily motivated by perceived programming enhancement and increased enjoyment. We also find that this cannabis use conflicts with anti-drug policies in software: 29% of our sample reported they had taken a drug test for a programming-related job, a hiring practice that may limit developer application pools.

6. *External Factors Case Study (Productivity Lens 3)—Section 5.3:* **The first controlled observational study of the impact of cannabis intoxication on programming.** In our controlled observational study with 74 participants, we find that at ecologically valid dosages, cannabis intoxication has a significant small-medium impairment on both program correctness and programming speed ($p < 0.05, 0.22 \leq d \leq 0.44$). We did not find evidence of cannabis increasing solution divergence. However, even when under the influence of cannabis, programmers correctly perceive differences in their programming performance ($r = 0.59$). These

Venue	Publication Title
Lens 1	
ASE 2019	INFIX*: Automatically Repairing Novice Program Inputs* [96]
OOPSLA 2022	SEQ2PARSE*: Neurosymbolic Parse Error Repair* [278]
Lens 2	
ICSE 2021	*Relating Reading, Visualization, and Coding for New Programmers: A Neuroimaging Study* [95]
FSE 2021	*To Read or To Rotate? Comparing the Effects of Technical Reading Training and Spatial Skills Training on Novice Programming Ability* [94]
Lens 3	
ICSE 2022	*Hashing It Out: A Survey of Programmers' Cannabis Usage, Perception, and Motivation* [92]
ICSE 2024	*High Expectations: An Observational Study of Programming and Cannabis Intoxication* [142]

Table 6.1: Supporting Peer-Reviewed Publications: Major peer-reviewed publications supporting this dissertation

results may contribute to the development of evidence-based policies and assist software developers in making informed decisions.

For all three lenses in this dissertation, we make relevant source code and data publicly available (when ethically permitted). The contributions in this dissertation are supported by the peer-reviewed publications listed in Table 6.1.

6.2 Future Directions

The studies included in this dissertation have improved our understanding of various aspects of programming productivity, and propose various interventions that may be helpful in practice. However, significant room remains for further exploration and improvement. Here, we summarize potential future directions and open challenges for each lens in this dissertation.

Lens 1—Improving Productivity Through Better Bug-Fixing Support. This dissertation demonstrated how expert-focused automatic program repair approaches could be extended to develop effective bug-fixing support for errors commonly encountered by novices (*e.g.*, input-related bugs or parse errors). In our human evaluations, we found that programmers judge these repairs to be helpful and of high quality. However, subjective self-reporting may not wholly capture the pedagogical and learning impacts of these bug-fixing tools. Therefore, one potential future direction could involve large-scale evaluations of bug-fixing support in more ecologically valid educational contexts. For example, researchers could use techniques such as alpha and beta testing or controlled longitudinal studies to validate the effectiveness of these tools in real-world environments.

Additionally, examining non-technical factors such as trust and/or bias associated with bug-fixing tools in pedagogical contexts could offer additional insight into their adoption and effectiveness. For instance, programming novices may have varying levels of trust in automated suggestions based on their prior experiences or the transparency of the tool's decision-making process. Understanding these dynamics could lead to the design of more intuitive and trustworthy tools. In addition, exploring how these tools are perceived and used by different demographics within programming communities may yield valuable findings, particularly in addressing diversity and inclusivity in software engineering practices.

Overall, there is a need for larger, more rigorous human-centered evaluations of program repair tools and programming language techniques. Such research could provide a deeper understanding of their educational impacts and guide the development of tools that are both effective and widely adopted, leading to larger productivity gains for a more diverse set of programmers. We hope that future studies will address these challenges and contribute to the advancement of programming productivity and usable bug-fixing support.

Lens 2—Leveraging Cognitive Insights to Improve Programmer Training. This dissertation presents one of the first studies to use medical imaging to explicitly investigate novice programmers' brain activation patterns and their correlates. In addition, we demonstrate the practical usefulness of these insights through a longitudinal transfer training study where we find that supplementary technical reading training leads to improved programming outcomes. As a result, beyond replications and meta-analyses, which are more common in psychology (*e.g.*, [327]) but not yet as prevalent in Computer Science, many potential future directions relate to building on the baselines established by our results. We focus on two dimensions: how programmers learn other computing activities at a cognitive level, and how these insights can inform the development of more effective programming tools and educational strategies.

Recent medical imaging research on programming has focused on program comprehension and code review, with a lesser emphasis on data structures and code writing [104, 153, 181, 293, 294]. Other activities remain unexplored. For example, it is unknown whether Boolean logic has a significant spatial cognitive component. While general logic has been studied, particular paradigms such as circuit design may be processed differently by humans, potentially suggesting alternate training or tool-support approaches. Additionally, experimental paradigms like priming, masking, and recall, which are foundational in psychology, are relatively unexplored in software engineering neuroimaging research. For instance, building on the influential work of Chase and Simon [296], psychologists have studied the relationship between chess expertise and the ability to recall or reason about briefly-presented random chess boards [114]. Using similar paradigms to tease apart programming expertise neurologically remains unexplored. In addition, neuroimaging techniques such as Transcranial Magnetic Stimulation (TMS) could be employed to investigate causality rather than correlation, providing deeper insights into the cognitive processes underlying programming expertise.

Our novel CS-technical reading training led to more significant gains than standardized spatial training, suggesting practical applications for helping students. However, this finding also raises questions about when and how to use different cognitive training approaches. For example, the students in our longitudinal study started with relatively high incoming spatial ability compared to other spatial ability training studies, which may have contributed to the reading training being more effective. Future research could thus investigate how supplementary cognitive training helps students with weaker incoming cognitive skills. One related hypothesis that we argue merits future investigation is that low-spatial students may benefit more from spatial training than reading training; the participants in this dissertation may have already had “enough” spatial ability for the programming tasks at hand. Conversely, novice programmers with low spatial ability may use different, more reading-related program comprehension strategies than their highly spatial-skilled counterparts. If, as posited by SpES, spatial ability helps students develop general strategies for mentally encoding programming problems [207], then low-spatial programmers might compensate by using alternate problem-solving strategies better supported by technical reading.

Overall, our results underscore the importance of reading skills in programming education and suggest that future productivity training should emphasize developing these skills. Specifically, we recommend that future researchers investigate when reading is most helpful while programming and what tool support can best help novice comprehension from a cognitive perspective. It is possible that understanding the cognitive processes involved in programming can also inform the design of programming tools that better support novice programmers, potentially improving their productivity and learning outcomes. Overall, integrating cognitive insights into the development of programming tools and educational strategies holds promise for enhancing programmer training and productivity. Future research could continue to explore these connections, building on the foundations established in this dissertation to create more effective and supportive environments for learning and practicing programming.

Lens 3—External Factors and Programming Productivity, a Cannabis Case Study. In this dissertation, we used cannabis use as a case study to highlight the importance of controlled experimentation in understanding the impact of anecdotal external factors on programming productivity. There is substantial room for future work, both in exploring the intersections of cannabis and programming and in applying our general approach to other external factors.

Regarding cannabis use in particular, the work in this dissertation has broader implications on software engineering ecosystems and cannabis-connected human-computer interactions. The intersection of cannabis use and human-computer interaction remains unexplored, in both general and software contexts. As cannabis becomes more acceptable, it may interact more with socio-technical issues including accessibility, end-user security, data privacy, and programming identity. This possibility was emphasized with “CHI-nnabis”, a recent panel at *Computer Human Interaction* (CHI), which called for more studies at the intersection of cannabis and technology [165].

While we focused on cannabis use, other understudied mind-altering substances may also interact with programming. For instance, while psychedelics have anecdotal connections to the nascent commercial computing industry [209, 337], little is known about their current prevalence in software engineering, let alone the accuracy of any perceived benefits. Similarly, while college students often abuse prescription stimulants as “study drugs” [323], little is known about the use of and culture surrounding such drugs by computing students or in a programming context. Overall, we hope this dissertation spurs research on the intersection of programming and mind-altering substances from multiple angles including programming productivity and socio-technical considerations.

Beyond substance use, we hope that in the future, researchers investigate other external factors that are currently understudied, but have the potential to have a significant impact on programmer productivity and wellbeing. These factors include neurodiversity, location, salary, and mental health. For instance, understanding how neurodiverse individuals engage with programming tasks could lead to the development of more inclusive tools and practices. Additionally, exploring the impact of economic factors like salary on programming productivity and job satisfaction could provide insights into workforce retention and performance. Overall, there is a need for comprehensive research on how various external factors influence programming productivity. We hope the approach used in this dissertation contributes to a more holistic understanding of the factors that shape programming practices and outcomes.

Cross-Lens Future Directions. While we discuss potential future directions for each lens separately, the techniques and insights explored in each of the three dissertation lenses also have the potential to influence and complement each other in future research. For example, neuroimaging (Lens 2) could be used to develop cognitively-informed bug-fixing support tools (Lens 1), enhancing the effectiveness of these tools by aligning them with the cognitive processes involved in programming. Similarly, insights from neuroimaging could inform the understanding of the cognitive effects of substances like cannabis (Lens 3), potentially identifying how such substances impact programming performance at a neurological level.

External factors (Lens 3) could also provide valuable perspectives on individual differences regarding the design of inclusive programmer training (Lens 2) or inclusive bug-fixing support (Lens 1). For instance, understanding how different external factors, such as stress or substance use, affect programming productivity could lead to more tailored and effective training programs that accommodate these influences. Additionally, tools such as those developed in Lens 1 could be designed to support developers in various cognitive states, helping them make fewer errors or complete tasks more efficiently (Lens 3).

In conclusion, the integration of insights and techniques across these three lenses has the potential to advance our understanding of programming productivity. By exploring the intersections between cognitive processes, external factors, and tool development, we hope that future researchers can create more effective and inclusive strategies that support programmers in writing more correct code and being happier while doing so.

BIBLIOGRAPHY

[1] Rasmus Aamand, Thomas Dalsgaard, Yi-Ching Lynn Ho, Arne Moller, Andreas Roepstorff, and Torben Lund. A NO way to BOLD?: Dietary nitrate alters the hemodynamic response to visual stimulation. *NeuroImage*, 83, 07 2013.

[2] Thomas Acton and Willie Golden. Training the knowledge worker: a descriptive study of training practices in Irish software companies. *Journal of European industrial training*, 27(2/3/4):137–146, 2003.

[3] Alireza Ahadi, Raymond Lister, Shahil Lal, and Arto Hellas. Learning programming, syntax errors and institution-specific factors. In *Proceedings of the 20th Australasian Computing Education Conference*, ACE '18, pages 90–96, New York, NY, USA, 2018. Association for Computing Machinery.

[4] Hammad Ahmad, Zachary Karas, Kimberly Diaz, Amir Kamil, Jean-Baptiste Jeannin, and Westley Weimer. How do we read formal claims? Eye-tracking and the cognition of proofs about algorithms. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 208–220. IEEE, 2023.

[5] Toufique Ahmed, Premkumar T. Devanbu, and Vincent J. Hellendoorn. Learning lenient parsing & typing via indirect supervision. *Empir. Softw. Eng.*, 26(2):29, 2021.

[6] Umair Z. Ahmed, Pawan Kumar, Amey Karkare, Purushottam Kar, and Sumit Gulwani. Compilation error repair: for the student programs, from the student programs. In Patricia Lago and Michal Young, editors, *International Conference on Software Engineering*, pages 78–87, 2018.

[7] A. V. Aho and S. C. Johnson. LR parsing. *ACM Comput. Surv.*, 6(2):99–124, jun 1974.

[8] Alfred V. Aho and Thomas G. Peterson. A minimum distance error-correcting parser for context-free languages. *SIAM J. Comput.*, 1(4):305–312, 1972.

[9] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley series in computer science / World student series edition. Addison-Wesley, 1986.

[10] Maizam Alias, Thomas R Black, and David E Gray. Effect of instruction on spatial visualization ability in civil engineering students. *International Education Journal*, 3(1):1–12, 2002.

[11] Muath Alkhalaf, Abdulbaki Aydin, and Tevfik Bultan. Semantic differential repair for input validation and sanitization. In *International Symposium on Software Testing and Analysis*, pages 225–236, 2014.

[12] Michael Patrick Allen. The problem of multicollinearity. *Understanding regression analysis*, pages 176–180, 1997.

[13] Aamir Amin, Shuib Basri, Mobashar Rehman, Luiz Fernando Capretz, Rehan Akbar, Abdul Rehman Gilal, and Muhammad Farooq Shabbir. The impact of personality traits and knowledge collection behavior on programmer creativity. *Information and Software Technology*, 128:106405, 2020.

[14] Paul Ammann and Jeff Offutt. *Introduction to software testing*. Cambridge University Press, 2008.

[15] Amber Anthenien, Mark Prince, Gemma Wallace, Tiffany Jenzer, and Clayton Neighbors. Cannabis outcome expectancies, cannabis use motives, and cannabis use among a small sample of frequent using adults. *Cannabis*, 4(1):69–84, 2021.

[16] Bernard J Baars and Nicole M Gage. *Cognition, brain, and consciousness: Introduction to cognitive neuroscience*. Academic Press, 2010.

[17] Alberto Bacchelli and Christian Bird. Expectations, outcomes, and challenges of modern code review. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 712–721. IEEE, 2013.

[18] Jerald G Bachman, Lloyd D Johnston, Patrick M O'Malley, and Ronald H Humphrey. Explaining the recent decline in marijuana use: Differentiating the effects of perceived risks, disapproval, and general lifestyle factors. *Journal of Health and Social Behavior*, 29(1):92–112, 1988.

[19] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *CoRR*, abs/1409.0473, 2015.

[20] Thomas Ball, Byron Cook, Vladimir Levin, and Sriram K Rajamani. SLAM and static driver verifier: Technology transfer of formal methods inside Microsoft. In *Integrated Formal Methods: 4th International Conference, IFM 2004, Cnaterbury, UK, April 4-7, 2004. Proceedings 4*, pages 1–20. Springer, 2004.

[21] Aron K Barbey, Michael Koenigs, and Jordan Grafman. Dorsolateral prefrontal contributions to human working memory. *cortex*, 49(5):1195–1205, 2013.

[22] Jeffrey W Barker, Ardalan Aarabi, and Theodore J Huppert. Autoregressive model based algorithm for correcting motion and serially correlated errors in fNIRS. *Biomedical optics express*, 4(8):1366–1379, 2013.

[23] Tiffany Barnes and John C. Stamper. Toward automatic hint generation for logic proof tutoring using historical student data. In *Intelligent Tutoring Systems*, volume 5091, pages 373–382, 2008.

[24] Eric P Baron, Philippe Lucas, Joshua Eades, and Olivia Hogue. Patterns of medicinal cannabis use, strain analysis, and substitution effect among patients with migraine, headache, arthritis, and chronic pain in a medicinal cannabis cohort. *The journal of headache and pain*, 19(1):1–28, 2018.

[25] Osbert Bastani, Rahul Sharma, Alex Aiken, and Percy Liang. Synthesizing program input grammars. In *Programming Language Design and Implementation*, pages 95–110, 2017.

[26] Theresa Beaubouef and John Mason. Why the high attrition rate for computer science students: some thoughts and observations. *ACM SIGCSE Bulletin*, 37(2):103–106, 2005.

[27] Mahnaz Behroozi, Chris Parnin, and Titus Barik. Hiring is broken: What do developers say about technical interviews? In Justin Smith, Christopher Bogart, Judith Good, and Scott D. Fleming, editors, *2019 IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC 2019, Memphis, Tennessee, USA, October 14-18, 2019*, pages 1–9. IEEE Computer Society, 2019.

[28] Moritz Beller, Niels Spruit, Diomidis Spinellis, and Andy Zaidman. On the dichotomy of debugging behavior among programmers. In Michel Chaudron, Ivica Crnkovic, Marsha Chechik, and Mark Harman, editors, *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, pages 572–583. ACM, 2018.

[29] Bachir Bendrissou, Rahul Gopinath, and Andreas Zeller. ”Synthesizing input grammars”: a replication study. In Ranjit Jhala and Isil Dillig, editors, *PLDI ’22: 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, San Diego, CA, USA, June 13 - 17, 2022*, pages 260–268. ACM, 2022.

[30] Craig M Bennett, Michael B Miller, and George L Wolford. Neural correlates of interspecies perspective taking in the post-mortem atlantic salmon: An argument for multiple comparisons correction. *Neuroimage*, 47(Suppl 1):S125, 2009.

[31] Vicki E. Bennett, KyuHan Koh, and Alexander Repenning. Computing creativity: Divergence in computational thinking. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education*, page 359–364, 2013.

[32] Marc Berman. How CBD oil can help programmers focus. https://programminginsider.com/how-cbd-oil-can-help-programmers-focus/, 1 2020. Accessed: 2021-03-07.

[33] Sahil Bhatia and Rishabh Singh. Automated correction for syntax errors in programming assignments using recurrent neural networks, 2016.

[34] Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*, pages 209–210. Springer-Verlag, Berlin, Heidelberg, 2006.

[35] Alice A Black. Spatial ability and earth science conceptual understanding. *Journal of Geoscience Education*, 53(4):402–414, 2005.

[36] David A Boas, Clare E Elwell, Marco Ferrari, and Gentaro Taga. Twenty years of functional near-infrared spectroscopy: introduction for the special issue, 2014.

[37] Ryan Bockmon, Stephen Cooper, William Koperski, Jonathan Gratch, Sheryl Sorby, and Mohsen Dorodchi. A CS1 spatial skills intervention and the impact on introductory programming abilities. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education*, pages 766–772, 2020.

[38] Kevin F Boehnke, Saurav Gangopadhyay, Daniel J Clauw, and Rebecca L Haffajee. Qualifying conditions of medical cannabis license holders in the United States. *Health Affairs*, 38(2):295–302, 2019.

[39] Kevin F Boehnke, Jenna McAfee, Joshua M Ackerman, and Daniel J Kruger. Medication and substance use increases among people using cannabis medically during the COVID-19 pandemic. *International Journal of Drug Policy*, 92:103053, 2021.

[40] Kevin F. Boehnke, J. Ryan Scott, Evangelos Litinas, Suzanne Sisley, Daniel J. Clauw, Jenna Goesling, and David A. Williams. Cannabis use preferences and decision-making among a cross-sectional cohort of medical cannabis patients with chronic pain. *The Journal of Pain*, 20(11):1362–1372, 2019.

[41] Kevin F. Boehnke, J. Ryan Scott, Evangelos Litinas, Suzanne Sisley, David A. Williams, and Daniel J. Clauw. Pills to pot: Observational analyses of cannabis substitution among medical cannabis users with chronic pain. *The Journal of Pain*, 20(7):830–841, 2019.

[42] Curtis J. Bonk, Mimi Miyoung Lee, Xiaojing Kou, Shuya Xu, and Feng-Ru Sheu. Understanding the self-directed online learning preferences, goals, achievements, and challenges of MIT OpenCourseWare subscribers. *Educational Technology & Society*, 18(2):349–365, 2015.

[43] Claus Brabrand, Nanna Inie, and Paolo Tell. Programming under the influence: On the effect of heat, noise, and alcohol on novice programmers. *J. Syst. Softw.*, 210:111887, 2024.

[44] Korbinian Brodmann. *Brodmann's localisation in the cerebral cortex*. World Scientific, 1999.

[45] Seth A. Brown. Standardized measures for substance use stigma. *Drug and Alcohol Dependence*, 116(1):137–141, 2011.

[46] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners, 2020.

[47] Sara E Brownell, Jordan V Price, and Lawrence Steinman. A writing-intensive course improves biology undergraduates' perception and confidence of their abilities to read scientific literature and communicate science. *Advances in Physiology Education*, 37(1):70–79, 2013.

[48] Sonia LE Brownsett and Richard JS Wise. The contribution of the parietal lobes to speaking and writing. *Cerebral Cortex*, 20(3):517–523, 2010.

[49] Samantha J. Broyd, Hendrika H. van Hell, Camilla Beale, Murat Yücel, and Nadia Solowij. Acute and chronic effects of cannabinoids on human cognition—a systematic review. *Biological Psychiatry*, 79(7):557–567, 2016. Cannabinoids and Psychotic Disorders.

[50] Michael G. Burke and Gerald A. Fisher. A practical method for LR and LL syntactic error diagnosis. *ACM Trans. Program. Lang. Syst.*, 9(2):164–197, 1987.

[51] Thomas S. Burt, Timothy L. Brown, Gary Milavetz, and Daniel V. McGehee. Mechanisms of cannabis impairment: Implications for modeling driving performance. *Forensic Science International*, 328:110902, 2021.

[52] Teresa Busjahn, Roman Bednarik, Andrew Begel, Martha Crosby, James H Paterson, Carsten Schulte, Bonita Sharif, and Sascha Tamm. Eye movements in code reading: Relaxing the linear order. In *2015 IEEE 23rd International Conference on Program Comprehension*, pages 255–265. IEEE, 2015.

[53] Teresa Busjahn, Carsten Schulte, and Andreas Busjahn. Analysis of code reading to gain more insight in program comprehension. In *Proceedings of the 11th Koli Calling International Conference on Computing Education Research*, pages 1–9, 2011.

[54] Cory J. Butz, Shan Hua, and R. Brien Maguire. A web-based bayesian intelligent tutoring system for computer programming. *Web Intell. Agent Syst.*, 4(1):77–97, 2006.

[55] Richard B Buxton, Kâmil Uludağ, David J Dubowitz, and Thomas T Liu. Modeling the hemodynamic response to brain activation. *Neuroimage*, 23:S220–S233, 2004.

[56] Antonio Byrd. Between learning and opportunity: A study of African American coders' networks of support. *Literacy in Composition Studies*, 7(2):31–56, 2019.

[57] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. EXE: automatically generating inputs of death. *ACM Trans. Inf. Syst. Secur.*, 12(2):10:1–10:38, 2008.

[58] Ian Campbell. Chi-squared and Fisher–Irwin tests of two-by-two tables with small sample recommendations. *Statistics in medicine*, 26(19):3661–3675, 2007.

[59] Hannah Carliner, Qiana L. Brown, Aaron L. Sarvet, and Deborah S. Hasin. Cannabis use, attitudes, and legal status in the U.S.: A review. *Preventive Medicine*, 104:13–23, 2017. Special issue: Behavior change, health, and health disparities 2017.

[60] Beth M Casey, Eric Dearing, Marina Vasilyeva, Colleen M Ganley, and Michele Tine. Spatial and numerical predictors of measurement performance: The moderating effects of community income and gender. *Journal of educational psychology*, 103(2):296, 2011.

[61] Joao Castelhano, Isabel C Duarte, Carlos Ferreira, Joao Duraes, Henrique Madeira, and Miguel Castelo-Branco. The role of the insula in intuitive expert bug detection in computer code: an fMRI study. *Brain imaging and behavior*, 13(3):623–637, 2019.

[62] Nigel P. Chapman. *LR parsing - theory and practice*. Cambridge University Press, 1987.

[63] Shi-Yi Chen, Zhe Feng, and Xiaolian Yi. A general introduction to adjustment for multiple comparisons. *Journal of thoracic disease*, 9(6):1725, 2017.

[64] Max Cherney. The FBI says it can't find hackers to hire because they all smoke pot. https://www.vice.com/en/article/d737mx/the-fbi-cant-find-hackers-that-dont-smoke-pot, 5 2014. Accessed: 2021-03-07.

[65] Esther K. Choo, Sarah W. Feldstein Ewing, and Travis I. Lovejoy. Opioids Out, Cannabis In: Negotiating the Unknowns in Patient Care for Chronic Pain. *JAMA*, 316(17):1763–1764, 11 2016.

[66] Cisco. 2019 code of business conduct. https://www.cisco.com/c/dam/en\\\\_us/about/cobc/2019/english-2019.pdf, 2019. Accessed: 2021-08-09.

[67] Jacob Cohen. *Statistical power analysis for the behavioral sciences*. Routledge, 2013.

[68] Laurent Cohen, Olivier Martinaud, Cathy Lemer, Samson Lehéricy, Yves Samson, Michaël Obadia, Andrea Slachevsky, and Stanislas Dehaene. Visual word recognition in the left and right hemispheres: anatomical and functional correlates of peripheral alexias. *Cerebral cortex*, 13(12):1313–1333, 2003.

[69] Mark S Cohen, Stephen M Kosslyn, Hans C Breiter, Gregory J DiGirolamo, William L Thompson, AK Anderson, SY Bookheimer, Bruce R Rosen, and JW Belliveau. Changes in cortical activity during mental rotation a mapping study using functional MRI. *Brain*, 119(1):89–100, 1996.

[70] Michael Collins. Probabilistic context-free grammars (PCFGs). *Lecture Notes*, 2013.

[71] Stephen Cooper, Karen Wang, Maya Israni, and Sheryl Sorby. Spatial skills training in introductory computing. In *International Computing Education Research*, pages 13–20, 2015.

[72] Rafael Corchuelo, José A Pérez, Antonio Ruiz, and Miguel Toro. Repairing syntax errors in LR parsers. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 24(6):698–710, 2002.

[73] Bas Cornelissen, Andy Zaidman, Arie van Deursen, Leon Moonen, and Rainer Koschke. A systematic survey of program comprehension through dynamic analysis. *IEEE Trans. Software Eng.*, 35(5):684–702, 2009.

[74] Will Crichton, Maneesh Agrawala, and Pat Hanrahan. The role of working memory in program tracing. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*, CHI '21, New York, NY, USA, 2021. Association for Computing Machinery.

[75] X Cui, S Bray, and AL Reiss. Functional near infrared spectroscopy (NIRS) signal improvement based on negative correlation between oxygenated and deoxygenated hemoglobin dynamics. *Neuroimage*, 49(4):3039–3046, 2010.

[76] Jody C Culham and Nancy G Kanwisher. Neuroimaging of cognitive functions in human parietal cortex. *Current opinion in neurobiology*, 11(2):157–163, 2001.

[77] C. Cuttler, E.M. LaFrance, and A. Stueber. Acute effects of high-potency cannabis flower and cannabis concentrates on everyday life memory and decision making. *Sci. Rep.*, 11(13784), 2021.

[78] Carrie Cuttler, Laurie K Mischley, and Michelle Sexton. Sex differences in cannabis use and effects: A cross-sectional survey of cannabis users. *Cannabis and Cannabinoid Research*, 1(1):166, 2016.

[79] Carrie Cuttler and Alexander Spradlin. Measuring cannabis consumption: psychometric properties of the daily sessions, frequency, age of onset, and quantity of cannabis use inventory (DFAQ-CU). *PLoS One*, 12(5):e0178194, 2017.

[80] MS Darshan, Rajesh Raman, TS Sathyanarayana Rao, Dushad Ram, and Bindu Annigeri. A study on professional stress, depression and alcohol use among Indian IT professionals. *Indian journal of psychiatry*, 55(1):63, 2013.

[81] Phillip Dawson, Jacques van der Meer, Jane Skalicky, and Kym Cowley. On the effectiveness of supplemental instruction: A systematic review of supplemental instruction and peer-assisted study sessions literature between 2001 and 2010. *Review of educational research*, 84(4):609–639, 2014.

[82] Nicola Dell, Vidya Vaidyanathan, Indrani Medhi, Edward Cutrell, and William Thies. ”yours is better!” participant response bias in HCI. In *Proceedings of the sigchi conference on human factors in computing systems*, pages 1321–1330, 2012.

[83] Paul Denny, Andrew Luxton-Reilly, and Dave Carpenter. Enhancing syntax error messages appears ineffectual. In *Innovation and Technology in Computer Science Education*, pages 273–278, 2014.

[84] Paul Denny, Andrew Luxton-Reilly, and Ewan Tempero. All syntax errors are not equal. In *Proceedings of the 17th ACM Annual Conference on Innovation and Technology in Computer Science Education*, ITiCSE ’12, pages 75–80, New York, NY, USA, 2012. Association for Computing Machinery.

[85] Julie S. Downs, Mandy B. Holbrook, Steve Sheng, and Lorrie Faith Cranor. Are your participants gaming the system?: screening Mechanical Turk workers. In *Human Factors in Computing Systems*, pages 2399–2402, 2010.

[86] João Duraes, Henrique Madeira, João Castelhano, Catarina Duarte, and M Castelo Branco. Wap: understanding the brain at software debugging. In *2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE)*, pages 87–92. IEEE, 2016.

[87] Susan Dynarski. Online courses are harming the students who need the most help. In *https://www.nytimes.com/2018/01/19/business/online-courses-are-harming-the-students-who-need-the-most-help.html*, 2018.

[88] Jay Earley. An efficient context-free parsing algorithm. *Commun. ACM*, 13(2):94–102, February 1970.

[89] Ezekeial Edwards and Brooke Madubuonwu. A tale of two countries: Racially targeted arrests in the era of marijuana reform, 6 2020.

[90] Ann-Christine Ehlis, Sabrina Schneider, Thomas Dresler, and Andreas J Fallgatter. Application of functional near-infrared spectroscopy in psychiatry. *Neuroimage*, 85:478–488, 2014.

[91] Jim Eison. Using active learning instructional strategies to create excitement and enhance learning, 2010.

[92] Madeline Endres, Kevin Boehnke, and Westley Weimer. Hashing it out: A survey of programmers' cannabis usage, perception, and motivation. In *International Conference on Software Engineering*, page 1107–1119, 2022.

[93] Madeline Endres, André Brechmann, Bonita Sharif, Westley Weimer, and Janet Siegmund. Foundations for a new perspective of understanding programming (Dagstuhl seminar 22402). *Dagstuhl Reports*, 12(10):61–83, 2022.

[94] Madeline Endres, Madison Fansher, Priti Shah, and Westley Weimer. To read or to rotate? comparing the effects of technical reading training and spatial skills training on novice programming ability. In *Foundations of Software Engineering*, pages 754–766, 2021.

[95] Madeline Endres, Zachary Karas, Xiaosu Hu, Ioulia Kovelman, and Westley Weimer. Relating reading, visualization, and coding for new programmers: A neuroimaging study. In *International Conference on Software Engineering*, pages 600–612, 2021.

[96] Madeline Endres, Georgios Sakkas, Benjamin Cosman, Ranjit Jhala, and Westley Weimer. Infix: Automatically repairing novice program inputs. In *34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019*, pages 399–410. IEEE, 2019.

[97] Katryn Nicole B Enriquez, Andrea Monique S Hidalgo, Ryan Francis T Quina, Nicole Julia L Valencia, and James Romulus M Buzon. The effect of gender inequality on job satisfaction, productivity, and career progression of female IT and software professionals. *Millennium Journal of Humanities and Social Sciences*, 2023.

[98] ETS.org. GRE home, 2020.

[99] Sarah Fakhoury, Yuzhan Ma, Venera Arnaoudova, and Olusola Adesope. The effect of poor source code lexicon and readability on developers' cognitive load. In *International Conference on Program Comprehension*, 2018.

[100] Davide Falessi, Natalia Juristo, Claes Wohlin, Burak Turhan, Jürgen Münch, Andreas Jedlitschka, and Markku Oivo. Empirical software engineering experts on the use of students and professionals in experiments. *Empirical Software Engineering*, 23:452–489, 2018.

[101] Evelina Fedorenko, Anna Ivanova, Riva Dhamala, and Marina Umaschi Bers. The language of programming: a cognitive perspective. *Trends in cognitive sciences*, 23(7):525–528, 2019.

[102] Marco Ferrari and Valentina Quaresima. A brief review on the history of human functional near-infrared spectroscopy (fNIRS) development and fields of application. *Neuroimage*, 63(2):921–935, 2012.

[103] Sally Fincher, Anthony Robins, Bob Baker, Ilona Box, Quintin Cutts, Michael de Raadt, Patricia Haden, John Hamer, Margaret Hamilton, Raymond Lister, et al. Predictors of success in a first programming course. In *Proceedings of the 8th Australasian Computing Education Conference (ACE 2006)*, volume 52, pages 189–196. Australian Computer Society Inc., 2006.

[104] Benjamin Floyd, Tyler Santander, and Westley Weimer. Decoding the representation of code in the brain: An fMRI study of code review and expertise. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 175–186. IEEE, 2017.

[105] Denae Ford, Margaret-Anne D. Storey, Thomas Zimmermann, Christian Bird, Sonia Jaffe, Chandra Shekhar Maddila, Jenna L. Butler, Brian Houck, and Nachiappan Nagappan. A tale of two cities: Software developers working from home during the COVID-19 pandemic. *ACM Trans. Softw. Eng. Methodol.*, 31(2):27:1–27:37, 2022.

[106] Andrew Forward and Timothy C Lethbridge. The relevance of software documentation, tools and technologies: a survey. In *Proceedings of the 2002 ACM symposium on Document engineering*, pages 26–33, 2002.

[107] Andreas G. Franke, Patrik Roser, Klaus Lieb, Jochen Vollmann, and Jan Schildmann. Cannabis for cognitive enhancement as a new coping strategy? results from a survey of students at four universities in germany. *Substance Use & Misuse*, 51(14):1856–1862, 2016. PMID: 27607062.

[108] Daniel J. Fridberg, Sarah Queller, Woo-Young Ahn, Woojae Kim, Anthony J. Bishara, Jerome R. Busemeyer, Linda Porrino, and Julie C. Stout. Cognitive mechanisms underlying risky decision-making in chronic cannabis users. *Journal of Mathematical Psychology*, 54(1):28–38, 2010. Contributions of Mathematical Psychology to Clinical Science and Assessment.

[109] Thomas Fritz, Jingwen Ou, Gail C Murphy, and Emerson Murphy-Hill. A degree-of-knowledge model to capture source code familiarity. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pages 385–394, 2010.

[110] Zachary P. Fry, Bryan Landau, and Westley Weimer. A human study of patch maintainability. In Mats Per Erik Heimdahl and Zhendong Su, editors, *International Symposium on Software Testing and Analysis, ISSTA 2012, Minneapolis, MN, USA, July 15-20, 2012*, pages 177–187. ACM, 2012.

[111] Zachary P. Fry and Westley Weimer. A human study of fault localization accuracy. In *International Conference on Software Maintenance*, pages 1–10. IEEE Computer Society, 2010.

[112] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: directed automated random testing. In *Programming Language Design and Implementation*, pages 213–223, 2005.

[113] Patrice Godefroid, Michael Y. Levin, and David A. Molnar. SAGE: whitebox fuzzing for security testing. *Commun. ACM*, 55(3):40–44, 2012.

[114] Y Gong, KA Ericsson, and JH Moxley. Recall of briefly presented chess positions and its relation to chess skill. *PLoS ONE*, 10(3):e0118756, 2015.

[115] Joseph E. Gonzales and Corbin A. Cunninghham. The promise of pre registration in psychological research. *American Psychological Association*, 2015.

[116] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*, pages 180–184. MIT Press, 2016. http://www.deeplearningbook.org.

[117] Rahul Gopinath, Björn Mathis, Matthias Höschele, Alexander Kampmann, and Andreas Zeller. Sample-free learning of input grammars for comprehensive software fuzzing. *CoRR*, abs/1810.08289, 2018.

[118] William Grabe. Research on teaching reading. *Annual review of applied linguistics*, 24:44, 2004.

[119] Arthur C Graesser, Mark W Conley, and Andrew Olney. Intelligent tutoring systems. 2012.

[120] Daniel Graziotin and Fabian Fagerholm. Happiness and the productivity of software engineers. In Caitlin Sadowski and Thomas Zimmermann, editors, *Rethinking Productivity in Software Engineering*, pages 109–124. Apress open / Springer, 2019.

[121] Daniel Graziotin, Fabian Fagerholm, Xiaofeng Wang, and Pekka Abrahamsson. What happens when software developers are (un)happy. *Journal of Systems and Software*, 140:32–47, 2018.

[122] BOB Green, David Kavanagh, and Ross Young. Being stoned: a review of self-reported cannabis effects. *Drug and alcohol review*, 22(4):453–460, 2003.

[123] Wouter Groeneveld, Hans Jacobs, Joost Vennekens, and Kris Aerts. Non-cognitive abilities of exceptional software engineers: A Delphi study. In Jian Zhang, Mark Sherriff, Sarah Heckman, Pamela A. Cutter, and Alvaro E. Monge, editors, *Proceedings of the 51st ACM Technical Symposium on Computer Science Education, SIGCSE 2020, March 11-14, 2020*, pages 1096–1102, Portland, OR, USA, 2020. ACM.

[124] Wouter Groeneveld, Laurens Luyten, Joost Vennekens, and Kris Aerts. Exploring the role of creativity in software engineering. In *43rd IEEE/ACM International Conference on Software Engineering: Software Engineering in Society, ICSE (SEIS) 2021, May 25-28, 2021*, pages 1–9, Madrid, Spain, 2021. IEEE.

[125] World Bank Group. Digital progress and trends, 2023. https://openknowledge.worldbank.org/server/api/core/bitstreams/95fe55e9-f110-4ba8-933f-e65572e05395/content, 2024. Accessed: 2024-06-25.

[126] Paloma Guenes, Rafael Tomaz, Marcos Kalinowski, Maria Teresa Baldassarre, and Margaret-Anne D. Storey. Impostor phenomenon in software engineers. In *Proceedings of the 46th International Conference on Software Engineering: Software Engineering in Society, ICSE-SEIS2024, Lisbon, Portugal, April 14-20, 2024*, pages 96–106. ACM, 2024.

[127] Sumit Gulwani, Ivan Radicek, and Florian Zuleger. Automated clustering and program repair for introductory programming assignments. *Programming Language Design and Implementation*, 2018.

[128] Ananda Gunawardena, Aaron Tan, and David Kaufer. Encouraging reading and collaboration using classroom salon. In *Proceedings of the fifteenth annual conference on Innovation and technology in computer science education*, pages 254–258, 2010.

[129] Philip Guo. Ten million users and ten years later: Python Tutor's design guidelines for building scalable and sustainable research software in academia. In Jeffrey Nichols, Ranjitha Kumar, and Michael Nebeling, editors, *UIST '21: The 34th Annual ACM Symposium on User Interface Software and Technology, Virtual Event, USA, October 10-14, 2021*, pages 1235–1251. ACM, 2021.

[130] Philip J. Guo. Online Python Tutor: embeddable web-based program visualization for CS education. In Tracy Camp, Paul T. Tymann, J. D. Dougherty, and Kris Nagel, editors, *The 44th ACM Technical Symposium on Computer Science Education, SIGCSE 2013, Denver, CO, USA, March 6-9, 2013*, pages 579–584. ACM, 2013.

[131] Abhilash Gupta, Rahul Gopinath, and Andreas Zeller. Clifuzzer: mining grammars for command-line invocations. In Abhik Roychoudhury, Cristian Cadar, and Miryung Kim, editors, *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2022, Singapore, Singapore, November 14-18, 2022*, pages 1667–1671. ACM, 2022.

[132] Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish K. Shevade. DeepFix: Fixing common C language errors by deep learning. In *Conference on Artificial Intelligence*, pages 1345–1351, 2017.

[133] Scott E Hadland and Sharon Levy. Objective testing–urine and other drug tests. *Child and adolescent psychiatric clinics of North America*, 25(3):549, 2016.

[134] Sonia Haiduc, Jairo Aponte, and Andrian Marcus. Supporting program comprehension with source code summarization. In *2010 acm/ieee 32nd international conference on software engineering*, volume 2, pages 223–226. IEEE, 2010.

[135] Ben Han. Key substance use and mental health indicators in the United States: results from the 2019 national survey on drug use and health, 2020.

[136] Christoph Hannebauer, Marc Hesenius, and Volker Gruhn. Does syntax highlighting help programming novices? *Empirical Software Engineering*, 23(5):2795–2828, 2018.

[137] Momchil Hardalov, Ivan Koychev, and Preslav Nakov. Towards automated customer support. In Gennady Agre, Josef van Genabith, and Thierry Declerck, editors, *Artificial Intelligence: Methodology, Systems, and Applications - 18th International Conference, AIMSA 2018, Varna, Bulgaria, September 12-14, 2018, Proceedings*, volume 11089 of *Lecture Notes in Computer Science*, pages 48–59. Springer, 2018.

[138] Irina M Harris, Gary F Egan, Cynon Sonkkila, Henri J Tochon-Danguy, George Paxinos, and John DG Watson. Selective right parietal lobe activation during mental rotation: a parametric pet study. *Brain*, 123(1):65–73, 2000.

[139] Lana D Harrison and Arthur Hughes. *The validity of self-reported drug use: Improving the accuracy of survey estimates*, volume 167. US Department of Health and Human Services, National Institutes of Health, Washington, D.C., US, 1997.

[140] Björn Hartmann, Daniel MacDougall, Joel Brandt, and Scott R. Klemmer. What would other programmers do? Suggesting solutions to error messages. *Conference on Human Factors in Computing Systems - Proceedings*, 2:1019–1028, 2010.

[141] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer Series in Statistics. Springer New York, 2009.

[142] Wenxin He, Manasvi Parikh, Westley Weimer, and Madeline Endres. High expectations: An observational study of programming and cannabis intoxication. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024*, pages 193:1–193:12. ACM, 2024.

[143] Regina Hebig, Truong Ho-Quang, Rodi Jolak, Jan Schröder, Humberto Linero, Magnus Ågren, and Salome Honest Maro. How do students experience and judge software comprehension techniques? In *Proceedings of the 28th International Conference on Program Comprehension*, pages 425–435, 2020.

[144] Mary Hegarty and Maria Kozhevnikov. Types of visual–spatial representations and mathematical problem solving. *Journal of educational psychology*, 91(4):684, 1999.

[145] Reshma Hegde and Gursimran Walia. How to enhance the creativity of software developers: A systematic literature review. *International Conference on Software Engineering and Knowledge Engineering*, pages 229–234, 2014.

[146] Rik NA Henson, Cathy J Price, Michael D Rugg, Robert Turner, and Karl J Friston. Detecting latency differences in event-related bold responses: application to words versus nonwords and initial versus repeated face presentations. *Neuroimage*, 15(1):83–97, 2002.

[147] Crosby Hipes, Jeffrey Lucas, Jo C. Phelan, and Richard C. White. The stigma of mental illness in the labor market. *Social Science Research*, 56:16–25, 2016.

[148] HireRight. Hireright employment screening benchmark report 2018, 2018.

[149] Jay Holland, Antonija Mitrovic, and Brent Martin. J-LATTE: a constraint-based tutor for Java. In *International Conference on Computers in Education*, pages 1–5, 2009.

[150] Matthias Höschele and Andreas Zeller. Mining input grammars with AUTOGRAM. In *International Conference on Software Engineering*, pages 31–34, 2017.

[151] Xiao-Su Hu, Neelima Wagley, Akemi Tsutsumi Rioboo, Alexandre F DaSilva, and Ioulia Kovelman. Photogrammetry-based stereoscopic optode registration method for functional near-infrared spectroscopy. *Journal of Biomedical Optics*, 25(9):095001–095001, 2020.

[152] Yu Huang, Denae Ford, and Thomas Zimmermann. Leaving my fingerprints: Motivations and challenges of contributing to OSS for social good. In *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, 22-30 May 2021*, pages 1020–1032, Madrid, Spain, 2021. IEEE.

[153] Yu Huang, Xinyu Liu, Ryan Krueger, Tyler Santander, Xiaosu Hu, Kevin Leach, and Westley Weimer. Distilling neural representations of data structure manipulation using fMRI and fNIRS. In *International Conference on Software Engineering*, pages 396–407, 2019.

[154] IBM. Business conduct guidelines. https://www.ibm.com/investor/att/pdf/BCG_accessible_2019.pdf, 2018. Accessed: 2021-08-09.

[155] Yoshiharu Ikutani and Hidetake Uwano. Brain activity measurement during program comprehension with NIRS. In *15th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD)*, pages 1–6. IEEE, 2014.

[156] Andrew F. Jarosz, Gregory J.H. Colflesh, and Jennifer Wiley. Uncorking the muse: Alcohol intoxication facilitates creative problem solving. *Consciousness and Cognition*, 21(1):487–493, 2012. Beyond the Comparator Model.

[157] Frederick Jelinek, John D Lafferty, and Robert L Mercer. Basic methods of probabilistic context free grammars. In *Speech Recognition and Understanding*, pages 345–360. Springer, 1992.

[158] Miguel Jiménez, Mario Piattini, and Aurora Vizcaíno. Challenges and improvements in distributed software development: A systematic review. *Advances in Software Engineering*, 2009, 2009.

[159] Cindy Larison John P Hoffman, Angela Brittingham. *Drug Use Among US Workers: Prevalence & Trends by Occupation & Industry Categories*. DHHS Publishing, Chicago, IL, USA, 1996.

[160] James A Jones, Mary Jean Harrold, and John Stasko. Visualization of test information to assist fault localization. In *Proceedings of the 24th international conference on Software engineering*, pages 467–477, 2002.

[161] Sue Jones and Gary Burnett. Spatial ability and learning to program. *Human Technology: An Interdisciplinary Journal on Humans in ICT Environments*, 2008.

[162] Dan Jurafsky and James H. Martin. Speech and language processing: an introduction to natural language processing, computational linguistics, and speech recognition, 3nd edition draft, 2024.

[163] Steven J Kass, Robert H Ahlers, and Melissa Dugger. Eliminating gender differences through practice in an applied visual spatial task. *Human Performance*, 11(4):337–349, 1998.

[164] Harmanpreet Kaur, Alex C. Williams, Daniel McDuff, Mary Czerwinski, Jaime Teevan, and Shamsi T. Iqbal. Optimizing for happiness and productivity: Modeling opportune moments for transitions and breaks at work. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*, CHI '20, page 1–15, New York, NY, USA, 2020. Association for Computing Machinery.

[165] Brian C. Keegan, Patricia A. Cavazos-Rehg, Anh Ngoc Nguyen, Saiph Savage, Jofish Kaye, Munmun De Choudhury, and Michael J. Paul. Chi-nnabis: Implications of marijuana legalization for and from human-computer interaction. In Gloria Mark, Susan R. Fussell, Cliff Lampe, m. c. schraefel, Juan Pablo Hourcade, Caroline Appert, and Daniel Wigdor, editors, *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems, Denver, CO, USA, May 06-11, 2017, Extended Abstracts*, pages 1312–1317, Denver, CO, US, 2017. ACM.

[166] Leo Kelion. FBI 'could hire hackers on cannabis' to fight cybercrime. https://www.bbc.com/news/technology-27499595, 5 2014. Accessed: 2021-03-07.

[167] Marisa Kendall. Hacking the brain: Silicon Valley entrepreneurs turn to fasting and smart drugs. *The Mercury News*, 9, 2016.

[168] Salomeh Keyhani, Stacey Steigerwald, Julie Ishida, Marzieh Vali, Magdalena Cerdá, Deborah Hasin, Camille Dollinger, Sodahm R Yoo, and Beth E Cohen. Risks and benefits of marijuana use: a national survey of US adults. *Annals of internal medicine*, 169(5):282–290, 2018.

[169] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. Automatic patch generation learned from human-written patches. In *International Conference on Software Engineering*, pages 802–811, 2013.

[170] Yoon Kim, Carl Denton, Luong Hoang, and Alexander M. Rush. Structured attention networks. *CoRR*, abs/1702.00887, 2017.

[171] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In Yoshua Bengio and Yann LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.

[172] Päivi Kinnunen and Lauri Malmi. Why students drop out of a CS1 course? In *Proceedings of the second international workshop on Computing education research*, pages 97–108, 2006.

[173] Aniket Kittur, Jeffrey V. Nickerson, Michael S. Bernstein, Elizabeth Gerber, Aaron D. Shaw, John Zimmerman, Matt Lease, and John J. Horton. The future of crowd work. In *Computer Supported Cooperative Work*, pages 1301–1318, 2013.

[174] Matthew Klickstein. Does pot enhance your ability to code? https://www.itechpost.com/articles/6198/20130307/pot-enhance-ability-code.html, 7 2013. Accessed: 2021-01-07.

[175] Donald E. Knuth. On the translation of languages from left to right. *Information and Control*, 8(6):607–639, 1965.

[176] Pavneet Singh Kochhar, Xin Xia, David Lo, and Shanping Li. Practitioners' expectations on automated fault localization. In *International Symposium on Software Testing and Analysis*, pages 165–176. ACM, 2016.

[177] I. Kovelman, S. A. Baker, and L. A. Petitto. Bilingual and monolingual brains compared: a functional magnetic resonance imaging investigation of syntactic processing and a possible "neural signature" of bilingualism. *Journal of Cognitive Neuroscience*, 20(1):153–169, 2008.

[178] Mikael A Kowal, Arno Hazekamp, Lorenza S Colzato, Henk van Steenbergen, Nic JA van der Wee, Jeffrey Durieux, Meriem Manai, and Bernhard Hommel. Cannabis and creativity: highly potent cannabis impairs divergent thinking in regular cannabis users. *Psychopharmacology*, 232(6):1123–1134, 2015.

[179] Nikolaus Kriegeskorte, Marieke Mur, and Peter A Bandettini. Representational similarity analysis-connecting the branches of systems neuroscience. *Frontiers in systems neuroscience*, page 4, 2008.

[180] Emese Kroon, Lauren Kuhns, and Janna Cousijn. The short-term and long-term effects of cannabis on cognition: recent advances in the field. *Current Opinion in Psychology*, 38:49–55, 2021. Cannabis.

[181] Ryan Krueger, Yu Huang, Xinyu Liu, Tyler Santander, Westley Weimer, and Kevin Leach. Neurological divide: An fMRI study of prose and code writing. In *International Conference on Software Engineering*, 2020.

[182] Neil Kulkarni, Caroline Lemieux, and Koushik Sen. Learning highly recursive input grammars. In *36th IEEE/ACM International Conference on Automated Software Engineering, ASE 2021, Melbourne, Australia, November 15-19, 2021*, pages 456–467. IEEE, 2021.

[183] Sarah K. Kummerfeld and Judy Kay. The neglected battle fields of syntax errors. In *Proceedings of the Fifth Australasian Conference on Computing Education - Volume 20*, ACE '03, pages 105–111, AUS, 2003. Australian Computer Society, Inc.

[184] Emily M. LaFrance and Carrie Cuttler. Inspired by mary jane? mechanisms underlying enhanced creativity in cannabis users. *Consciousness and Cognition*, 56:68–76, 2017.

[185] Karim R Lakhani and Robert G Wolf. Why hackers do what they do: Understanding motivation and effort in free/open source software projects, 9 2003.

[186] Robert E Larzelere, Brett R Kuhn, and Byron Johnson. The intervention selection bias: an underrecognized confound in intervention research. *Psychological bulletin*, 130(2):289, 2004.

[187] Robert L. Lathrop. How students learn music: The psychology of music and music education. *Music Educators Journal*, 56(6):47–145, 1970.

[188] Timotej Lazar and Ivan Bratko. Data-driven program synthesis for hint generation in programming tutors. In *Intelligent Tutoring Systems*, volume 8474 of *Lecture Notes in Computer Science*, pages 306–311, 2014.

[189] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In Martin Glinz, Gail C. Murphy, and Mauro Pezzè, editors, *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*, pages 3–13. IEEE Computer Society, 2012.

[190] Claire Le Goues, Neal Holtschulte, Edward K. Smith, Yuriy Brun, Premkumar T. Devanbu, Stephanie Forrest, and Westley Weimer. The ManyBugs and IntroClass benchmarks for automated repair of C programs. *IEEE Trans. Software Eng.*, 41(12):1236–1256, 2015.

[191] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. Genprog: A generic method for automatic software repair. *IEEE Trans. Software Eng.*, 38(1):54–72, 2012.

[192] Christine M Lee, Clayton Neighbors, Christian S Hendershot, and Joel R Grossbard. Development and preliminary validation of a comprehensive marijuana motives questionnaire. *Journal of studies on alcohol and drugs*, 70(2):279–287, 2009.

[193] Charles Levinson. Comey: FBI 'grappling' with hiring policy concerning marijuana. `https://www.wsj.com/articles/BL-LB-48089`, 5 2014. Accessed: 2021-03-07.

[194] Martin A Lindquist, Ji Meng Loh, Lauren Y Atlas, and Tor D Wager. Modeling the hemodynamic response function in fMRI: efficiency, bias and mis-modeling. *Neuroimage*, 45(1):S187–S198, 2009.

[195] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F. Bissyandé. TBar: revisiting template-based automated program repair. In Dongmei Zhang and Anders Møller, editors, *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2019, Beijing, China, July 15-19, 2019*, pages 31–42. ACM, 2019.

[196] Sarah Lloyd-Fox, Anna Blasi, and CE Elwell. Illuminating the developing brain: the past, present and future of functional near infrared spectroscopy. *Neuroscience & Biobehavioral Reviews*, 34(3):269–284, 2010.

[197] Lucas Lobato, Jeffrey Michael Bethony, Fernanda Bicalho Pereira, Shannon Lee Grahek, David Diemert, and Maria Flávia Gazzinelli. Impact of gender on the decision to participate in a clinical trial: a cross-sectional study. *BMC Public Health*, 14(1):1–9, 2014.

[198] Fan Long, Vijay Ganesh, Michael Carbin, Stelios Sidiroglou, and Martin C. Rinard. Automatic input rectification. In *International Conference on Software Engineering*, pages 80–90, 2012.

[199] Fan Long and Martin Rinard. Automatic patch generation by learning correct code. In *Principles of Programming Languages*, pages 298–312, 2016.

[200] Mike Lopez, Jacqueline Whalley, Phil Robbins, and Raymond Lister. Relationships between reading, tracing and writing skills in introductory programming. In *Proceedings of the fourth international workshop on computing education research*, pages 101–112, 2008.

[201] Thibaud Lutellier, Hung Viet Pham, Lawrence Pang, Yitong Li, Moshi Wei, and Lin Tan. Coconut: Combining context-aware neural translation models using ensemble for program repair. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2020, pages 101–114, New York, NY, USA, 2020. Association for Computing Machinery.

[202] Lucy Ellen Lwakatare, Aiswarya Raj, Jan Bosch, Helena Holmström Olsson, and Ivica Crnkovic. A taxonomy of software engineering challenges for machine learning systems: An empirical investigation. In *Agile Processes in Software Engineering and Extreme Programming: 20th International Conference, XP 2019, Montréal, QC, Canada, May 21–25, 2019, Proceedings 20*, pages 227–243. Springer International Publishing, 2019.

[203] Yimeng Ma, Yu Huang, and Kevin Leach. Breaking the flow: A study of interruptions during software engineering activities. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024*, pages 185:1–185:12. ACM, 2024.

[204] Neil Maiden, Suzanne Robertson, and James Robertson. Creative requirements: Invention and its role in requirements engineering. In *Proceedings of the 28th International Conference on Software Engineering*, page 1073–1074, New York, NY, USA, 2006. Association for Computing Machinery.

[205] Guillaume Marceau, Kathi Fisler, and Shriram Krishnamurthi. Mind your language: on novices' interactions with error messages. In *Symposium on New Ideas in Programming and Reflections on Software, Onward!*, pages 3–18, 2011.

[206] Alexandru Marginean, Johannes Bader, Satish Chandra, Mark Harman, Yue Jia, Ke Mao, Alexander Mols, and Andrew Scott. SapFix: automated end-to-end repair at scale. In Helen Sharp and Mike Whalen, editors, *Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice, ICSE (SEIP) 2019, Montreal, QC, Canada, May 25-31, 2019*, pages 269–278. IEEE / ACM, 2019.

[207] Lauren E. Margulieux. Spatial encoding strategy theory: The relationship between spatial skill and STEM achievement. In *Proceedings of the 2019 ACM Conference on International Computing Education Research*, ICER '19, page 81–90, 2019.

[208] Paul Dan Marinescu and Cristian Cadar. KATCH: high-coverage testing of software patches. In Bertrand Meyer, Luciano Baresi, and Mira Mezini, editors, *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26, 2013*, pages 235–245. ACM, 2013.

[209] John Markoff. *What the dormouse said: How the sixties counterculture shaped the personal computer industry*. Penguin Group, New York, NY, USA, 2005.

[210] Rebecca A. Marks, Ioulia Kovelman, Olga Kepinska, Myriam Oliver, Zhichao Xia, Stephanie L. Haft, Leo Zekelman, Priscilla Duong, Yuuko Uchikoshi, Roeland Hancock, and Fumiko Hoeft. Spoken language proficiency predicts print-speech convergence in beginning readers. *NeuroImage*, 201:116021, 2019.

[211] Matias Martinez, Laurence Duchien, and Martin Monperrus. Automatically extracting instances of code change patterns with AST analysis. In *2013 IEEE International Conference on Software Maintenance, Eindhoven, The Netherlands, September 22-28, 2013*, pages 388–391. IEEE Computer Society, 2013.

[212] Richard E Mayer. The psychology of how novices learn computer programming. *ACM Computing Surveys (CSUR)*, 13(1):121–141, 1981.

[213] Paul W. McBurney and Collin McMillan. Automatic documentation generation via source code summarization of method context. In Chanchal K. Roy, Andrew Begel, and Leon Moonen, editors, *22nd International Conference on Program Comprehension, ICPC 2014, Hyderabad, India, June 2-3, 2014*, pages 279–290. ACM, 2014.

[214] Alice McCarthy. Most popular MITx MOOC reaches 1.2 million enrollments. In *http://news.mit.edu/2018/first-mitx-mooc-reaches-enrollment-milestone-0830*, 2018.

[215] Gayle Laakmann McDowell. *Cracking the coding interview: 189 programming questions and solutions*. CareerCup, LLC, 2015.

[216] Philippe McLean and R. Nigel Horspool. A faster Earley parser. In Tibor Gyimóthy, editor, *Compiler Construction, 6th International Conference, CC'96, Linköping, Sweden, April 24-26, 1996, Proceedings*, volume 1060 of *Lecture Notes in Computer Science*, pages 281–293. Springer, 1996.

[217] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. Angelix: scalable multiline program patch synthesis via symbolic analysis. In *International Conference on Software Engineering*, pages 691–701, 2016.

[218] Barbara S Mensch and Denise B Kandel. Do job conditions influence the use of drugs? *Journal of Health and Social Behavior*, 29(2):169–184, 1988.

[219] Douglas C. Merrill, Brian J. Reiser, Shannon K. Merrill, and Shari Landes. Tutoring: Guided learning by doing. *Cognition and Instruction*, 13(3):315–372, 1995.

[220] Peter G Miller and Anders L Sønderlund. Using the internet to research hidden populations of illicit drug users: a review. *Addiction*, 105(9):1557–1567, 2010.

[221] Martin Monperrus. The living review on automated program repair. Technical Report hal-01956501, HAL/archives-ouvertes.fr, 2018.

[222] Meredith Ringel Morris, Andrew Begel, and Ben Wiedermann. Understanding the challenges faced by neurodiverse software engineering employees: Towards a more inclusive and productive technical workforce. In *SIGACCESS Conference on Computers & Accessibility*, page 173–184, 2015.

[223] Simon Moser and Oscar Nierstrasz. The effect of object-oriented frameworks on developer productivity. *Computer*, 29(9):45–51, 1996.

[224] Laurie Murphy, Sue Fitzgerald, Raymond Lister, and Renée McCauley. Ability to 'explain in plain english' linked to proficiency in computer-based programming. In *Proceedings of the ninth annual international conference on International computing education research*, pages 111–118, 2012.

[225] Laurie Murphy, Gary Lewandowski, Renée McCauley, Beth Simon, Lynda Thomas, and Carol Zander. Debugging: the good, the bad, and the quirky – a qualitative analysis of novices' strategies. In J. D. Dougherty, Susan H. Rodger, Sue Fitzgerald, and Mark Guzdial, editors, *Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education, SIGCSE 2008, Portland, OR, USA, March 12-15, 2008*, pages 163–167. ACM, 2008.

[226] Emerson R. Murphy-Hill, Ciera Jaspan, Caitlin Sadowski, David C. Shepherd, Michael Phillips, Collin Winter, Andrea Knight, Edward K. Smith, and Matthew Jorde. What predicts software developers' productivity? *IEEE Trans. Software Eng.*, 47(3):582–594, 2021.

[227] Eugene W Myers. An O(ND) difference algorithm and its variations. *Algorithmica*, 1(1-4):251–266, 1986.

[228] Vinod Nair and Geoffrey E. Hinton. Rectified linear units improve restricted Boltzmann machines. In Johannes Fürnkranz and Thorsten Joachims, editors, *Proceedings of the 27th International Conference on Machine Learning (ICML-10), June 21-24, 2010, Haifa, Israel*, pages 807–814. Omnipress, 2010.

[229] Takao Nakagawa, Yasutaka Kamei, Hidetake Uwano, Akito Monden, Kenichi Matsumoto, and Daniel M German. Quantifying programmers' mental workload during program comprehension based on cerebral blood flow measurement: a controlled experiment. In *Companion proceedings of the 36th international conference on software engineering*, pages 448–451, 2014.

[230] National Academies of Sciences, Engineering, and Medicine. *The Health Effects of Cannabis and Cannabinoids: The Current State of Evidence and Recommendations for Research.* The National Academies Press, Washington, DC, 2017.

[231] John C. Nesbit, Olusola O. Adesope, Qing Liu, and Wenting Ma. How effective are intelligent tutoring systems in computer science education? In *IEEE 14th International Conference on Advanced Learning Technologies, ICALT 2014, Athens, Greece, July 7-10, 2014*, pages 99–103. IEEE Computer Society, 2014.

[232] Kaia Newman, Madeline Endres, Westley Weimer, and Brittany Johnson. From organizations to individuals: Psychoactive substance use by professional programmers. In *International Conference on Software Engineering*, pages 665–677, 2023.

[233] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. SemFix: program repair via semantic analysis. In David Notkin, Betty H. C. Cheng, and Klaus Pohl, editors, *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, pages 772–781. IEEE Computer Society, 2013.

[234] NIDA. Prescription stimulants drugfacts. https://nida.nih.gov/publications/drugfacts/prescription-stimulants, 2018. Accessed: 2022-08-28.

[235] Michael A Nielsen. *Neural Networks and Deep Learning*. Determination Press, 2015.

[236] Yannic Noller, Ridwan Shariffdeen, Xiang Gao, and Abhik Roychoudhury. Trust enhancement issues in program repair. In *Proceedings of the 44th International Conference on Software Engineering*, pages 2228–2240, 2022.

[237] Geoffrey Norman. Likert scales, levels of measurement and the "laws" of statistics. *Advances in health sciences education : theory and practice*, 15(5):625–32, 02 2010.

[238] Hellmuth Obrig. NIRS in clinical neurology — a 'promising' tool? *Neuroimage*, 85:535–546, 2014.

[239] Sofia Ouhbi and Nuno Pombo. Software engineering education: Challenges and perspectives. In *2020 IEEE Global Engineering Education Conference (EDUCON)*, pages 202–209. IEEE, 2020.

[240] Sagar Parihar, Ziyaan Dadachanji, Praveen Kumar Singh, Rajdeep Das, Amey Karkare, and Arnab Bhattacharya. Automatic grading and feedback using program repair for introductory programming courses. In *Innovation and Technology in Computer Science Education*, pages 92–97, 2017.

[241] Miranda Parker, Amber Solomon, Brianna Pritchett, David Illingworth, Lauren Margulieux, and Mark Guzdial. Socioeconomic status and computer science achievement: Spatial ability as a mediating variable in a novel model of understanding. In *Proceedings of the 2018 ACM Conference on International Computing Education Research*, pages 97–105, 08 2018.

[242] Miranda C. Parker, Mark Guzdial, and Shelly Engleman. Replication, validation, and use of a language independent CS1 knowledge assessment. In *Proceedings of the 2016 ACM Conference on International Computing Education Research*, ICER '16, page 93–101, 2016.

[243] Jack Parkinson and Quintin Cutts. Investigating the relationship between spatial skills and computer science. In *Proceedings of the 2018 ACM Conference on International Computing Education Research*, pages 106–114, 2018.

[244] Chris Parnin and Alessandro Orso. Are automated debugging techniques actually helping programmers? In *Proceedings of the 2011 international symposium on software testing and analysis*, pages 199–209, 2011.

[245] Norman Peitek, Sven Apel, Chris Parnin, André Brechmann, and Janet Siegmund. Program comprehension and code complexity metrics: An fMRI study. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 524–536, 2021.

[246] Norman Peitek, Annabelle Bergum, Maurice Rekrut, Jonas Mucke, Matthias Nadig, Chris Parnin, Janet Siegmund, and Sven Apel. Correlates of programmer efficacy and their link to experience: a combined EEG and eye-tracking study. In Abhik Roychoudhury, Cristian Cadar, and Miryung Kim, editors, *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2022, Singapore, Singapore, November 14-18, 2022*, pages 120–131. ACM, 2022.

[247] Norman Peitek, Janet Siegmund, Chris Parnin, Sven Apel, Johannes C. Hofmeister, and André Brechmann. Simultaneous measurement of program comprehension with fMRI and eye tracking: a case study. In Markku Oivo, Daniel Méndez Fernández, and Audris Mockus, editors, *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM 2018, Oulu, Finland, October 11-12, 2018*, pages 24:1–24:10. ACM, 2018.

[248] Michael Peters and Christian Battista. Applications of mental rotation figures of the Shepard and Metzler type and description of a mental rotation stimulus library. *Brain and cognition*, 66(3):260–264, 2008.

[249] Chantel S Prat, Tara M Madhyastha, Malayka J Mottarella, and Chu-Hsuan Kuo. Relating natural language aptitude to individual differences in learning programming languages. *Scientific reports*, 10(1):1–10, 2020.

[250] Jeffrey R Pribyl and George M Bodner. Spatial ability and its role in organic chemistry: A study of four organic courses. *Journal of research in science teaching*, 24(3):229–240, 1987.

[251] Cathy J Price. A review and synthesis of the first 20 years of PET and fMRI studies of heard speech, spoken language and reading. *Neuroimage*, 62(2):816–847, 2012.

[252] Michael Prince. Does active learning work? a review of the research. *Journal of engineering education*, 93(3):223–231, 2004.

[253] Luisa Prochazkova, Dominique P Lippelt, Lorenza S Colzato, Martin Kuchar, Zsuzsika Sjoerds, and Bernhard Hommel. Exploring the effect of microdosing psychedelics on creativity in an open-label natural setting. *Psychopharmacology*, 235(12):3401–3413, 2018.

[254] Yewen Pu, Karthik Narasimhan, Armando Solar-Lezama, and Regina Barzilay. sk_p: a neural program corrector for MOOCs, 2016.

[255] Yizhou Qian and James Lehman. Students' misconceptions and other difficulties in introductory programming: A literature review. *ACM Trans. Comput. Educ.*, 18(1), oct 2017.

[256] Kia Rahmani, Mohammad Raza, Sumit Gulwani, Vu Le, Daniel Morris, Arjun Radhakrishna, Gustavo Soares, and Ashish Tiwari. Multi-modal program inference: A marriage of pre-trained language models and component-based synthesis. *Proc. ACM Program. Lang.*, 5(OOPSLA), oct 2021.

[257] Sanguthevar Rajasekaran and Marius Nicolae. An error correcting parser for context free grammars that takes less than cubic time. In Adrian-Horia Dediu, Jan Janousek, Carlos Martín-Vide, and Bianca Truthe, editors, *Language and Automata Theory and Applications - 10th International Conference, LATA 2016, Prague, Czech Republic, March 14-18, 2016, Proceedings*, volume 9618 of *Lecture Notes in Computer Science*, pages 533–546. Springer, 2016.

[258] Václav Rajlich. Teaching developer skills in the first software engineering course. In *2013 35th international conference on software engineering (ICSE)*, pages 1109–1116. IEEE, 2013.

[259] Danielle E Ramo, Howard Liu, and Judith J Prochaska. Reliability and validity of young adults' anonymous online reports of marijuana use and thoughts about use. *Psychology of Addictive Behaviors*, 26(4):801, 2012.

[260] Ramdas Ransing, Pedro A de la Rosa, Victor Pereira-Sanchez, Jibril IM Handuleh, Stefan Jerotic, Anoop Krishna Gupta, Ruta Karaliuniene, Renato de Filippis, Eric Peyron, Ekin Sönmez Güngör, et al. Current state of cannabis use, policies, and research across sixteen countries: cross-country comparisons and international perspectives. *Trends in psychiatry and psychotherapy*, 44, 2022.

[261] Ayushi Rastogi, Suresh Thummalapenta, Thomas Zimmermann, Nachiappan Nagappan, and Jacek Czerwonka. Ramp-up journey of new hires: Do strategic practices of software companies influence productivity? In *Proceedings of the 10th Innovations in Software Engineering Conference*, page 107–111, 2017.

[262] Alison D Rayome. 5 eye-opening statistics about minorities in tech, 2018.

[263] Amanda Reiman, Mark Welty, and Perry Solomon. Cannabis as a substitute for opioid-based pain medication: patient self-report. *Cannabis and cannabinoid research*, 2(1):160–166, 2017.

[264] Mitchel Resnick, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay Silver, Brian Silverman, and Yasmin Kafai. Scratch: Programming for all. *Commun. ACM*, 52(11):60–67, November 2009.

[265] Kelly Rivers and Kenneth R. Koedinger. Data-driven hint generation in vast solution spaces: a self-improving Python programming tutor. *I. J. Artificial Intelligence in Education*, 27(1):37–64, 2017.

[266] Jeffrey A. Roberts, Il-Horn Hann, and Sandra A. Slaughter. Understanding the motivations, participation, and performance of open source software developers: A longitudinal study of the Apache projects. *Management Science*, 52(7):984–999, 2006.

[267] Paige Rodeghero and Collin McMillan. An empirical study on the patterns of eye movement during summarization tasks. In *2015 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–10. IEEE, 2015.

[268] Paige Rodeghero, Collin McMillan, Paul W McBurney, Nigel Bosch, and Sidney D'Mello. Improving automated source code summarization via an eye-tracking study of programmers. In *Proceedings of the 36th international conference on Software engineering*, pages 390–401, 2014.

[269] Ma. Mercedes T. Rodrigo and Ryan Shaun Joazeiro de Baker. Coarse-grained detection of student frustration in an introductory programming course. In *International Workshop on Computing Education Research*, pages 75–80, 2009.

[270] Ole Rogeberg and Rune Elvik. The effects of cannabis intoxication on motor vehicle collision revisited and revised. *Addiction*, 111(8):1348–1359, 2016.

[271] Debopriyo Roy. Using concept maps for information conceptualization and schematization in technical reading and writing courses: A case study for computer science majors in Japan. In *2008 IEEE International Professional Communication Conference*, pages 1–12. IEEE, 2008.

[272] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning Representations by Back-propagating Errors. *Nature*, 323(6088):533–536, 1986.

[273] Mark A Runco and Shawn M Okuda. Problem discovery, divergent thinking, and the creative process. *Journal of youth and adolescence*, 17(3):211–220, 1988.

[274] H. Sackman, W. J. Erikson, and E. E. Grant. Exploratory experimental studies comparing online and offline programming performance. *Commun. ACM*, 11(1):3–11, jan 1968.

[275] Marian Gunsher Sackrowitz and Ann Parker Parelius. An unlevel playing field: Women in the introductory computer science courses. *ACM SIGCSE Bulletin*, 28(1):37–41, 1996.

[276] Saddleback˜Publishing. *Reading Comprehension Skills and Strategies Level 8*. High-Interest Reading Comprehension Skills and Strategies Series. Saddleback Publishing, 2002.

[277] Georgios Sakkas, Madeline Endres, Benjamin Cosman, Westley Weimer, and Ranjit Jhala. Type error feedback via analytic program repair. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 16–30, 2020.

[278] Georgios Sakkas, Madeline Endres, Philip J Guo, Westley Weimer, and Ranjit Jhala. Seq2Parse: neurosymbolic parse error repair. *Proceedings of the ACM on Programming Languages*, 6(OOPSLA2):1180–1206, 2022.

[279] H. Santosa, X. Zhai, F. Fishburn, and T. Huppert. The NIRS brain AnalyzIR toolbox. *Algorithms*, 11(5), May 2018.

[280] Jürgen Schmidhuber. Deep learning in neural networks: An overview. *Neural Networks*, 61:85–117, Jan 2015.

[281] Carsten Schulte, Tony Clear, Ahmad Taherkhani, Teresa Busjahn, and James H. Paterson. An introduction to program comprehension for computer science educators. In Alison Clear and Lori Russell-Dag, editors, *Proceedings of the 2010 ITiCSE working group reports, ITiCSE-WGR 2010, Ankara, Turkey, June 28-30, 2010*, pages 65–86. ACM, 2010.

[282] Eric M. Schulte, Jonathan DiLorenzo, Westley Weimer, and Stephanie Forrest. Automated repair of binary and assembly programs for cooperating embedded devices. In *Architectural Support for Programming Languages and Operating Systems*, pages 317–328, 2013.

[283] Scicurious. IgNobel prize in neuroscience: The dead salmon study. *Scientific American Blog Network*, Sep 2012.

[284] Skipper Seabold and Josef Perktold. statsmodels: Econometric and statistical modeling with Python. In *9th Python in Science Conference*, pages 92–96, Austin, TX, US, 2010. SciPy.

[285] Mohamed L Seghier. The angular gyrus: multiple functions and multiple subdivisions. *The Neuroscientist*, 19(1):43–61, 2013.

[286] Eric L. Seidel, Huma Sibghat, Kamalika Chaudhuri, Westley Weimer, and Ranjit Jhala. Learning to blame: localizing novice type errors with data-driven diagnosis. *Proc. ACM Program. Lang.*, 1(OOPSLA):60:1–60:27, 2017.

[287] Koushik Sen, Darko Marinov, and Gul Agha. CUTE: a concolic unit testing engine for C. In *Foundations of Software Engineering*, pages 263–272, 2005.

[288] Dhawal Shah. A product at every price: A review of MOOC stats and trends in 2017. In *Edsurge.com*, 2018.

[289] Ridwan Shariffdeen, Martin Mirchev, and Abhik Roychoudhury. Program repair competition. In *IEEE/ACM International Workshop on Automated Program Repair, APR@ICSE 2023, Melbourne, Australia, May 16, 2023*, pages 19–20. IEEE, 2023.

[290] Roger N Shepard and Jacqueline Metzler. Mental rotation of three-dimensional objects. *Science*, 171(3972):701–703, 1971.

[291] Emad Shihab, Patanamon Thongtanunam, and Bogdan Vasilescu. Mining software repositories. *IEEE*, 2023.

[292] Valerie J Shute. Who is likely to acquire programming skills? *Journal of educational Computing research*, 7(1):1–24, 1991.

[293] Janet Siegmund, Christian Kästner, Sven Apel, Chris Parnin, Anja Bethmann, Thomas Leich, Gunter Saake, and André Brechmann. Understanding understanding source code with functional magnetic resonance imaging. In *Proceedings of the 36th International Conference on Software Engineering*, pages 378–389, 2014.

[294] Janet Siegmund, Norman Peitek, Chris Parnin, Sven Apel, Johannes Hofmeister, Christian Kästner, Andrew Begel, Anja Bethmann, and André Brechmann. Measuring neural efficiency of program comprehension. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 140–150, 2017.

[295] Joseph P. Simmons, Leif D. Nelson, and Uri Simonsohn. Pre-registration: Why and how. *J. Society for Consumer Psychology*, December 2020.

[296] H A Simon and W G Chase. Perception in chess. *Cognitive Psychology*, 4(1):55–81, 1973.

[297] Janice Singer, Susan Elliott Sim, and Timothy C. Lethbridge. Software engineering data collection for field studies. In Forrest Shull, Janice Singer, and Dag I. K. Sjøberg, editors, *Guide to Advanced Empirical Software Engineering*, pages 9–34. Springer, College Park, MD, US, 2008.

[298] Rishabh Singh, Sumit Gulwani, and Armando Solar-Lezama. Automated feedback generation for introductory programming assignments. In *Programming Language Design and Implementation*, pages 15–26, 2013.

[299] Elliot Soloway and Kate Ehrlich. Empirical studies of programming knowledge. *IEEE Trans. Software Eng.*, 10(5):595–609, 1984.

[300] Sheryl Sorby, Norma Veurink, and Scott Streiner. Does spatial skills instruction improve STEM outcomes? the answer is 'yes'. *Learning and Individual Differences*, 67:209–222, 2018.

[301] Sheryl A Sorby and Beverly J Baartmans. The development and assessment of a course for enhancing the 3-d spatial visualization skills of first year engineering students. *Journal of Engineering Education*, 89(3):301–307, 2000.

[302] Sheryl A. Sorby, Edmund Nevin, Avril Behan, Eileen Mageean, and Sarah Sheridan. Spatial skills as predictors of success in first-year engineering. In *IEEE Frontiers in Education Conference*, pages 1–7, 2014.

[303] Margaret-Anne Storey, Thomas Zimmermann, Christian Bird, Jacek Czerwonka, Brendan Murphy, and Eirini Kalliamvakou. Towards a theory of software developer job satisfaction and perceived productivity. *IEEE Transactions on Software Engineering*, 47(10):2125–2142, 2019.

[304] Substance Abuse and Mental Health Services Administration. Key substance use and mental health indicators in the United States: results from the 2020 national survey on drug use and health. *Center for Behavioral Health Statistics and Quality, Substance Abuse and Mental Health Services Administration*, 2020.

[305] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. Sequence to sequence learning with neural networks, 2014.

[306] Mohammad Tahaei and Kami Vaniea. Recruiting participants with programming skills: A comparison of four crowdsourcing platforms and a CS student mailing list. In Simone D. J. Barbosa, Cliff Lampe, Caroline Appert, David A. Shamma, Steven Mark Drucker, Julie R. Williamson, and Koji Yatani, editors, *CHI '22: CHI Conference on Human Factors in Computing Systems, New Orleans, LA, USA, 29 April 2022 - 5 May 2022*, pages 590:1–590:15. ACM, 2022.

[307] Toon W. Taris and Paul J.G. Schreurs. Well-being and organizational performance: An organizational-level test of the happy-productive worker hypothesis. *Work & Stress*, 23(2):120–136, 2009.

[308] United Nations Press Team. UNODC world drug report 2020: Global drug use rising; while COVID-19 has far reaching impact on global drug markets, 2020.

[309] Allison Elliott Tew and Mark Guzdial. Developing a validated assessment of fundamental CS1 concepts. In *Proceedings of the 41st ACM technical symposium on Computer science education*, pages 97–101, 2010.

[310] Allison Elliott Tew and Mark Guzdial. The FCS1: A language independent assessment of CS1 knowledge. In *Proceedings of the 42nd ACM Technical Symposium on Computer Science Education*, SIGCSE '11, page 111–116, 2011.

[311] Richard A Thompson. Language correction using probabilistic grammars. *IEEE Transactions on Computers*, 100(3):275–286, 1976.

[312] Thomas Tilley, Richard Cole, Peter Becker, and Peter W. Eklund. A survey of formal concept analysis support for software engineering activities. In Bernhard Ganter, Gerd Stumme, and Rudolf Wille, editors, *Formal Concept Analysis, Foundations and Applications*, volume 3626 of *Lecture Notes in Computer Science*, pages 250–271, College Park, MD, US, 2005. Springer.

[313] Nikolai Tillmann and Jonathan de Halleux. Pex—white box test generation for .NET. In Bernhard Beckert and Reiner Hähnle, editors, *Tests and Proofs - 2nd International Conference, TAP 2008, Prato, Italy, April 9-11, 2008. Proceedings*, volume 4966 of *Lecture Notes in Computer Science*, pages 134–153. Springer, 2008.

[314] Nikolai Tillmann, Jonathan de Halleux, Tao Xie, and Judith Bishop. Pex4Fun: A web-based environment for educational gaming via automated test generation. In Ewen Denney, Tevfik Bultan, and Andreas Zeller, editors, *2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013, Silicon Valley, CA, USA, November 11-15, 2013*, pages 730–733. IEEE, 2013.

[315] Nikolai Tillmann, Jonathan de Halleux, Tao Xie, and Judith Bishop. Constructing coding duels in Pex4Fun and code hunt. In *International Symposium on Software Testing and Analysis*, pages 445–448, 2014.

[316] Nikolai Tillmann, Jonathan de Halleux, Tao Xie, Sumit Gulwani, and Judith Bishop. Teaching and learning programming and software engineering via interactive gaming. In David Notkin, Betty H. C. Cheng, and Klaus Pohl, editors, *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, pages 1117–1126. IEEE Computer Society, 2013.

[317] Torbjørn Torsheim, Franco Cavallo, Kate Ann Levin, Christina Schnohr, Joanna Mazur, Birgit Niclasen, Candace Currie, FAS Development Study Group, et al. Psychometric validation of the revised family affluence scale: a latent variable approach. *Child Indicators Research*, 9(3):771–784, 2016.

[318] Dyanne M Tracy. Toys, spatial ability, and science and mathematics achievement: Are they related? *Sex roles*, 17(3-4):115–138, 1987.

[319] Adam Trendowicz and Jürgen Münch. Factors influencing software development productivity—state-of-the-art and industrial experiences. *Advances in computers*, 77:185–241, 2009.

[320] David H Uttal, Nathaniel G Meadow, Elizabeth Tipton, Linda L Hand, Alison R Alden, Christopher Warren, and Nora S Newcombe. The malleability of spatial skills: A meta-analysis of training studies. *Psychological bulletin*, 139(2):352, 2013.

[321] P. van der Spek, N. Plat, and C. Pronk. Syntax error repair for a Java-based parser generator. *SIGPLAN Not.*, 40(4):47–50, April 2005.

[322] Ted Van Green. Americans overwhelmingly say marijuana should be legal for recreational or medical use. *Pew Research Center*, 2021.

[323] Matthew D Varga. Adderall abuse on college campuses: a comprehensive literature review. *Journal of evidence-based social work*, 9(3):293–313, 2012.

[324] Bogdan Vasilescu, Daryl Posnett, Baishakhi Ray, Mark G. J. van den Brand, Alexander Serebrenik, Premkumar T. Devanbu, and Vladimir Filkov. Gender and tenure diversity in GitHub teams. In Bo Begole, Jinwoo Kim, Kori Inkpen, and Woontack Woo, editors, *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems, CHI 2015, Seoul, April 18-23, 2015*, pages 3789–3798, Seoul, Republic of Korea, 2015. ACM.

[325] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. Attention is all you need. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30, pages 5998–6008. Curran Associates, Inc., 2017.

[326] Alison Vekshin. Silicon Valley is high on innovation. and pot, 2 2013.

[327] Philip R Ventura Jr. Identifying predictors of success for an objects-first CS1. *Computer Science Education*, 2005.

[328] Gust Verbruggen, Vu Le, and Sumit Gulwani. Semantic programming by example with pre-trained models. *Proc. ACM Program. Lang.*, 5(OOPSLA), oct 2021.

[329] Mathieu Vigneau, Virginie Beaucousin, Pierre-Yves Herve, Hugues Duffau, Fabrice Crivello, Olivier Houde, Bernard Mazoyer, and Nathalie Tzourio-Mazoyer. Meta-analyzing left hemisphere language areas: phonology, semantics, and sentence processing. *Neuroimage*, 30(4):1414–1432, 2006.

[330] Piia Maria Vilenius-Tuohimaa, Kaisa Aunola, and Jari-Erik Nurmi. The association between mathematical word problems and reading comprehension. *Educational Psychology*, 28(4):409–426, 2008.

[331] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C J Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1.0 Contributors. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17:261–272, 2020.

[332] Daniel Voyer, Susan Voyer, and M Philip Bryden. Magnitude of sex differences in spatial abilities: a meta-analysis and consideration of critical variables. *Psychological bulletin*, 117(2):250, 1995.

[333] Greg Wadley. How psychoactive drugs shape human culture: A multi-disciplinary perspective. *Brain Research Bulletin*, 126:138–151, 2016. Neurobiology of emerging psychoactive drugs.

[334] Stefan Wagner and Melanie Ruhe. A systematic review of productivity factors in software development. *arXiv preprint arXiv:1801.06475*, 2018.

[335] Jonathan Wai, David Lubinski, and Camilla P Benbow. Spatial ability for STEM domains: Aligning over 50 years of cumulative psychological knowledge solidifies its importance. *Journal of Educational Psychology*, 101(4):817, 2009.

[336] Matthew Wall. How long will you wait for a shopping website to load. In `https://www.bbc.com/news/business-37100091`, 2016.

[337] Charlotte Walsh. Drugs, the internet and change. *Journal of psychoactive drugs*, 43(1):55–63, 2011.

[338] Mary Walton. Programming and cannabis — 5 things to know. `https://simpleprogrammer.com/programming-and-cannabis/`, 4 2019. Accessed: 2021-03-07.

[339] Ke Wang, Rishabh Singh, and Zhendong Su. Search, align, and repair: Data-driven feedback generation for introductory programming exercises. In *Programming Language Design and Implementation*, pages 481–495, 2018.

[340] Christopher Watson, Frederick WB Li, and Jamie L Godwin. Bluefix: Using crowd-sourced feedback to support programming students in error diagnosis and repair. In *International Conference on Web-Based Learning*, pages 228–239. Springer, 2012.

[341] Ming Wen, Junjie Chen, Rongxin Wu, Dan Hao, and Shing-Chi Cheung. Context-aware patch generation for better automated program repair. In Michel Chaudron, Ivica Crnkovic, Marsha Chechik, and Mark Harman, editors, *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, pages 1–11. ACM, 2018.

[342] Paul J. Werbos. Backpropagation through time: what it does and how to do it. *Proc. IEEE*, 78(10):1550–1560, 1990.

[343] Wes McKinney. Data Structures for Statistical Computing in Python. In Stéfan van der Walt and Jarrod Millman, editors, *Proceedings of the 9th Python in Science Conference*, pages 56–61, Austin, TX, US, 2010. SciPy.

[344] WHO. Drugs (psychoactive), 2022. Accessed: 2022-08-25.

[345] John Winslow, Ruth B Ekstrom, and Leighton A Price. *Kit of reference tests for cognitive factors*. Educational Testing Service, 1963.

[346] Amanda A Wolkowitz and Jeffrey A Kelley. Academic predictors of success in a nursing program. *Journal of Nursing Education*, 49(9):498–503, 2010.

[347] Evan Wood, Daniel Werb, Brandon D L Marshall, Julio S G Montaner, and Thomas Kerr. The war on drugs: a devastating public-policy disaster. *Lancet (London, England)*, 373(9668):989—990, March 2009.

[348] Laura Wood. Global cannabis market (2020 to 2026) - emergence of cannabis legalization in Asia-Pacific presents opportunities - researchandmarkets.com, 2 2021.

[349] Liwei Wu, Fei Li, Youhua Wu, and Tao Zheng. GGF: A graph-based method for programming language syntax error correction. In *ICPC '20: 28th International Conference on Program Comprehension, Seoul, Republic of Korea, July 13-15, 2020*, pages 139–148. ACM, 2020.

[350] Xin Xia, Lingfeng Bao, David Lo, Zhenchang Xing, Ahmed E Hassan, and Shanping Li. Measuring program comprehension: A large-scale field study with professionals. *IEEE Transactions on Software Engineering*, 44(10):951–976, 2017.

[351] Yuki Yamada. How to crack pre-registration: Toward transparent and open science. *Frontiers in Psychology*, 9(1831), 2018.

[352] Eun-Mi Yang, Thomas Andre, Thomas J Greenbowe, and Lena Tibell. Spatial ability and the impact of visualization/animation on learning electrochemistry. *International Journal of Science Education*, 25(3):329–349, 2003.

[353] Yanming Yang, Xin Xia, David Lo, and John C. Grundy. A survey on deep learning for software engineering. *ACM Comput. Surv.*, 54(10s):206:1–206:73, 2022.

[354] Jooyong Yi, Umair Z. Ahmed, Amey Karkare, Shin Hwei Tan, and Abhik Roychoudhury. A feasibility study of using automated program repair for introductory programming assignments. In Eric Bodden, Wilhelm Schäfer, Arie van Deursen, and Andrea Zisman, editors, *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*, pages 740–751. ACM, 2017.

[355] So Y. Yoon. *Psychometric properties of the Revised Purdue Spatial Visualization Tests: Visualization of Rotations (the Revised PSVT:R)*. PhD thesis, Purdue University, 2011.

[356] Jeffrey M Zacks. Neuroimaging studies of mental rotation: a meta-analysis and review. *Journal of cognitive neuroscience*, 20(1):1–19, 2008.

[357] J.M. Zelenski, S.A. Murphy, and D.A. Jenkins. The happy-productive worker thesis revisited. *J. Happiness Studies*, 9:521–537, 2008.

[358] Andreas Zeller. Yesterday, my program worked. Today, it does not. Why? In *Foundations of Software Engineering*, pages 253–267, 1999.

[359] Kim Zetter. Steve jobs' pentagon file: Blackmail fears, youthful arrest and LSD cubes, 6 2012.

[360] Stuart Zweben and Betsy Bizot. 2022 Taulbee survey. *Computing Research News: CRA*, 2023.