# Minimizing Technical Barriers to Learning Programming

By

MARTIN VELEZ

DISSERTATION

Submitted in partial satisfaction of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

Computer Science

in the

OFFICE OF GRADUATE STUDIES

of the

UNIVERSITY OF CALIFORNIA

DAVIS

Approved:

Zhendong Su, Chair

Premkumar Devanbu

Vladimir Filkov

Committee in Charge

2019

*To my parents, my wife, and my son.*

# CONTENTS

# LIST OF FIGURES

# List of Tables

## ABSTRACT

**Minimizing Technical Barriers to Learning Programming**

Software is an integral part of our lives. It controls the cars we drive every day, the ships we send into space, and even our toasters. It is everywhere and we can easily download more. Software solves many real-world problems and satisfies many needs. Thus, unsurprisingly, there is a rising demand for software engineers to maintain existing software and to design and build new systems. Unfortunately, there is a scarcity of software engineers. But thankfully, more and more people are opting or being encouraged to pursue a Computer Science education, and we are seeing an explosion in enrollment worldwide. At UC Davis, for example, an introductory programming course quadrupled in enrollment from about a hundred students to four hundred. Recently, official enrollment at UC Berkeley's introductory programming course was $1,762$. Massively open online courses (MOOCs) like Coursera and Udacity regularly sign up thousands of students for a single course.

Not surprisingly, however, not everyone who enrolls in CS courses succeeds. A study in 2014 found that, on average, 33% of students fail. Often, these same students decide to drop out instead. For example, Ireland reported that about a third of students drop out of Computer Science degree programs. Undoubtedly, there are many factors contributing to these issues. But perhaps, the most straightforward reason is that programming is simply difficult and challenging. And with larger classes, students are receiving less personalized instruction and intervention. Therefore, we need innovative tools and approaches to help students learn to program.

In this dissertation, we focus on *technical barriers to learning programming*. We define technical barriers as those challenges that are faced by programmers of all levels but are especially difficult to beginners. These technical barriers can cause students to waste time, to become frustrated, and even to quit. This dissertation describes three efforts addressing these technical barriers from different angles: simplifying the programming environment, assisting with compilation errors, and exploring a syntax-free programming paradigm.

Students often spend a considerable amount of time and effort installing and configuring programming tools and environments. This can frustrate, and distract them from more important learning objectives, particularly in introductory programming courses. A web integrated development environment (IDE) can serve as a low-threshold, ready-to-use programming environment, and reduce the time and effort needed to start practicing programming. Moreover, the uniform execution environment can facilitate better interactions between students and instructors.

We describe the design and deployment of KODETHON, a web IDE, at a large public university. KODETHON can support multiple programming languages, multi-file projects, and real-time collaboration. To date, more than 3,000 students have used KODETHON in at least 15 different courses to write over 15 million lines of code. We studied student adoption behavior and perceptions of KODETHON by analyzing server database and logs, and by deploying a user survey. We found that about a third of participants perceive KODETHON to be useful. We also found that students find "Web-based" and "No Installation Required" to be the two most useful features. We present lessons learned and provide advice for educators and researchers considering introducing a web IDE as a pedagogical tool.

Every programmer, from novices to professionals, makes compilation errors. Resolving compilation errors can be time-consuming, difficult, and frustrating. For decades, error messages have been identified as a source of this difficulty. A promising approach to help programmers is to augment error messages with *compilation repair examples*. The challenge is how to obtain and present these repair examples.

We present COMPASSIST, a system that generates and refines repair examples. Based on these repair examples, the system suggests possible patches to users when their program fails to compile. We evaluated COMPASSIST on a mainstream C++ compiler, and demonstrate that it can generate examples for more than half (867/1,686) of compiler errors. We also conducted a user study where participants found these synthetic repair examples to be helpful in a majority (5/9) of tasks involving real-world C++ compiler

programs.

Lastly, we focus on programming language syntax. Natural language is robust against noise. The meaning of many sentences survives the loss of words, sometimes many of them. Some words in a sentence, however, cannot be lost without changing the meaning of the sentence. We call these words “wheat” and the rest “chaff”. The word “not” in the sentence “I do not like rain” is wheat and “do” is chaff. For human understanding of the purpose and behavior of *source code*, we hypothesize that the same holds. To quantify the extent to which we can separate code into “wheat” and “chaff”, we study a large (100M LOC), diverse corpus of real-world projects in Java. Since methods represent natural, likely distinct units of code, we use the ~9M Java methods in the corpus to approximate a universe of “sentences.” We extract their wheat by computing the function’s *minimal distinguishing subset (*MINSET*)*. Our results confirm that functions contain much chaff. On average, MINSETS have 1.56 words (none exceeds 6) and comprise 4% of their methods. Beyond its intrinsic scientific interest, our work offers the first quantitative evidence for recent promising work on keyword-based programming and insight into how to develop a powerful, alternative programming model.

## Acknowledgments

First, I would like to thank my parents for their love and support. Thirty years ago, they left their home immigrated to the United States to give my siblings and me a chance at a better education. They taught me much including kindness and the joy of learning.

I would like to thank my advisor Professor Zhendong Su for his support. Zhendong has been an amazing role model to me because he is kind, thoughtful, and hard-working. I thank him for believing in me, and for guiding me for many years. I am going to miss our weekly meetings where we dreamt big together.

I also thank Prof. Earl Barr for mentoring me as an undergraduate student while he was a Postdoctoral Researcher at UC Davis and into the first years of graduate school although he was already on the other side of the Atlantic working at UCL.

I also want to thank all of the professors who mentored and advised me throughout the years: Prof. Prem Devanbu, Prof. Vladimir Filkov, Prof. Cindy Rubio-Gonzalez, Prof. Philip Rogaway, Prof. Patrice Koehl, Prof. Amin Alipour. Every one taught me many interesting lessons on all sorts of subjects that I hope to carry with me. They were always available and enthusiastic to discuss my projects and ideas.

I also want to thank all of the instructors who allowed me to demonstrate Kodethon to their students: Sean Davis, Matthew Butner, Rob Gyselt, and Prof. Dave Doty. A big part of this dissertation was made possible thanks to their enthusiasm for new technologies.

I also want to thank all of the co-authors of the chapters in this disseration: Dong Qiu, You Zhou, Michael Yen, Mathew Le, Chengnian Sun, and Nima Johari.

I would like to thank all of the labmates who came and went throughout the years. I enjoyed our daily coffee runs, our lively discussions, and the encouragement we gave each other: Dr. Qirun Zhang, Dr. Mehrdad Afshari, Dr. Liang Xu, Dr. Fangqi Sun, Dr. Vu Le, Dr. Andreas Saebjornsen, Dr. Tianxiao Gu, Dr. Zhaopeng Tu, Dr. Ke Wang.

Last but not least, I would like to thank my wife Rocio for her daily support and encouragement. For many years, she has been the biggest cheerleader in my life. Thanks to her, I am able to focus on research.

# Chapter 1

# Introduction

Over the last several decades, we have come to realize that the right software can solve many important real-world problems and help us achieve our grandest dreams. We write software to guide our spaceships into outer space and to explore the solar system. [1] We write software to assist us in driving our cars more safely and efficiently. As we steer and press pedals, software is invisibly and continuously optimizing fuel consumption, minimizing emissions, maximizing traction, and avoiding collisions. [2] We build a variety of web and mobile applications to connect with family and friends in ways that may have seemed impossible only a century ago. We even put software in toasters to toast bread to perfection and notify us on our smartphones when our toast is ready. [3] Indeed, whenever many of us face a new problem, one of the first questions we ask is “Is there an app for that?”. If there isn’t, we write it. In this manner, software has rapidly become an integral part of our lives.

Correspondingly, there is high demand for software developers who can build new software systems and maintain existing ones. The U.S. Bureau of Labor Statistics estimates that there were 1,256,200 Software Developer jobs in 2016 in the US [Bureau of Labor Statistics, 2018, Adams, 2016]. The number of software developer jobs is expected to grow by 302,500 (24%) from 2016 to 2026, which is nearly thirty thousand new jobs every year, “much faster than the average for all occupations”. More impor-

[1] https://software.nasa.gov/
[2] https://www.embedded.com/design/operating-systems/4442406/Software-in-cars
[3] https://www.engadget.com/2017/01/04/griffin-connects-your-toast-to-your-phone/

tantly, by most reports, there is a shortage of software developers to fill the current openings [Torres, 2018, Society, 2018]. In such a job market, software developers enjoy a low unemployment rate, 1.9% in 2017 compared to 4.1% overall [Torres, 2018]. The average salary is also much higher than in most other occupations, ranging from $92,592 to $104,425, depending on the report [DataUSA, 2018, Bureau of Labor Statistics, 2018, Society, 2018]. Pursuing a career in Computer Science seems to be a safe choice for a lot of people [Adams, 2014] .

Thankfully, due in part to this attractive job market and to a global movement led by organizations like Code.org[4] and the previous White House administration [Mechaber, 2014], more people are enrolling in Computer Science courses [Soper, 2014, Lazowska et al., 2014]. Many people choose to enroll in traditional four-year universities causing typical introductory class sizes to increase. At University of California, Davis, for example, an introductory programming course quadrupled in enrollment from about a hundred students to four hundred in the last several years. Recently, official enrollment at University of California, Berkeley's introductory programming course reached $1,762$ [Kim, 2017]. Many students have to watch the lecture from another room on a projector screen. Millions of other people are enrolling in massively open online courses (MOOCs) offered by companies like Coursera and Udacity [Shah, 2014]. There is so much demand that dozens of coding boot camps have sprung up all over the world offering short-term computer science education. [5].

As a society, we want and need all of these students to succeed; unfortunately, many do not. For example, in a 2014 study of an introductory programming course (called CS1) taught in 15 different countries across 51 different institutions, researchers found that only about 67% students pass the course, which means that about 33% fail or dropout [Watson and Li, 2014]. A study reported a year later a similar pass rate (77%) at the University of Toronto [Horton and Craig, 2015]. Worse, students can fail to progress and drop out of CS programs entirely [Pappas et al., 2016]. In Europe, for example, a report estimated that 19% of students drop out [Hüsing et al., 2013] of

[4]`https://code.org/`

[5]`https://www.coursereport.com/reports/2017-coding-bootcamp-market-size-research`

Information and Communication Technology programs. In some countries, the dropout rate can be as high as 32% [Kori et al., 2015].

There are many more reasons why computer science students struggle. One popular explanation is that learning to program is inherently difficult, and, perhaps, one of the most challenging human endeavors. It involves understanding and solving problems precisely, often down to individual bits. It requires working knowledge of data structures and algorithms. It demands one to manage the ever-growing complexity of function/module interactions. It requires breadth and depth of knowledge; writing a video game is different from writing a spreadsheet program. As Dijkstra put it:

> "The art of programming is the art of organizing complexity, of mastering multitude and avoiding its bastard chaos as effectively as possible."

Numerous socioeconomic factors also affect a student's preparedness, performance, and eventual success [Pappas et al., 2016]. Historically, poorer children have generally had less access to computers and to schools that taught basic computer science skills.

Large class sizes can also negatively impact students. At the current levels of enrollment, in both, traditional university courses and MOOCs, effective instruction and learning becomes extremely challenging. Students lose opportunities to ask questions during class. There simply is not enough time during lecture to answer the questions of hundreds of students or, in the case of MOOCs, thousands to even hundreds of thousands. Students also lose opportunities to ask for help during instructor and teaching assistant office hours. Lines for office hours become longer and longer. Some of their questions may never be brought up and thus answered. On the other hand, instructors and teaching assistants, due to more students, have to spend more of their time conducting tedious activities like grading instead of providing better feedback to individual students. In general, students miss out on personalized help.

We need innovative tools and approaches to help students learn to program. In this dissertation, we focus on ***technical barriers to learning programming***. We define technical barriers to be those difficulties related to programming languages and tools that tend to cause students to waste time and to become frustrated. Technical barriers can be faced by programmers of all levels but are especially difficult to beginners. In this dissertation, we identify technical barriers in developer tools, in compiler feedback, and in language syntax. We propose approaches to minimize these barriers to help students learn to program.

***Developer Tools:*** Students often spend a considerable amount of time and effort installing and configuring programming tools and environments. This can frustrate, and distract them from more important learning objectives, particularly in introductory programming courses. A web integrated development environment (IDE) can serve as a low-threshold, ready-to-use programming environment, and reduce the time and effort needed to start practicing programming. Moreover, the uniform execution environment can facilitate better interactions between students and instructors.

We describe the design and deployment of KODETHON, a web IDE, at a large public university. KODETHON can support multiple programming languages, multi-file projects, and real-time collaboration. To date, more than 3,000 students have used KODETHON in at least 15 different courses to write over 15 million lines of code. We studied student adoption behavior and perceptions of KODETHON by analyzing server database and logs, and by deploying a user survey. We found that about a third of participants perceive KODETHON to be useful. We also found that students find “Web-based” and “No Installation Required” to be the two most useful features. We present lessons learned and provide advice for educators and researchers considering introducing a web IDE as a pedagogical tool.

***Compiler Feedback:*** Every programmer, from novices to professionals, makes compilation errors. Resolving compilation errors can be time-consuming, difficult, and frustrating. For decades, error messages have been identified as a source of this difficulty. A promising approach to help programmers is to augment error messages with *compilation repair examples*. The challenge is how to obtain and present these repair examples.

We present COMPASSIST, a system that generates and refines repair examples. Based on these repair examples, the system suggests possible patches to users when their program fails to compile. We evaluated COMPASSIST on a mainstream C++ compiler, and demonstrate that it can generate examples for more than half (867/1, 686) of compiler errors. We also conducted a user study where participants found this synthetic repair examples to be helpful in a majority (5/9) of tasks involving real-world C++ compiler programs.

***Language Syntax*** Lastly, we focus on programming language syntax. Natural language is robust against noise. The meaning of many sentences survives the loss of words, sometimes many of them. Some words in a sentence, however, cannot be lost without changing the meaning of the sentence. We call these words "wheat" and the rest "chaff". The word "not" in the sentence "I do not like rain" is wheat and "do" is chaff. For human understanding of the purpose and behavior of *source code*, we hypothesize that the same holds. To quantify the extent to which we can separate code into "wheat" and "chaff", we study a large (100M LOC), diverse corpus of real-world projects in Java. Since methods represent natural, likely distinct units of code, we use the ~9M Java methods in the corpus to approximate a universe of "sentences." We extract their wheat by computing the function's *minimal distinguishing subset (*MINSET*)*. Our results confirm that functions contain much chaff. On average, MINSETS have 1.56 words (none exceeds 6) and comprise 4% of their methods. Beyond its intrinsic scientific interest, our work offers the first quantitative evidence for recent promising work on keyword-based programming and insight into how to develop a powerful, alternative programming model.

***Contributions:*** In this dissertation, we make the following contributions:

- We present Kodethon, a web-based integrated development environment used by students in Computer Science and Linguistics courses at University of California, Davis. Students can program in Python, C, C++, Java, Lisp, Prolog, and other programming languages. Programs are stored in the cloud but can be accessed from a variety devices including tablets and smartphones. The built-in editor provides convenient features like syntax highlighting and auto completion. To execute Unix commands, e.g., ssh and scp, students can use a new easy-to-use shell or a full Unix terminal — right within the browser. Kodethon also supports real-time collaboration which facilitates pair-programming and live teaching assistance.

- We present the first large scale study of deployment and adoption of a web IDE at a large public university. We show that about a third of students adopted Kodethon to write their programming assignments. We also show that about a third find Kodethon useful.

- We present FUZZ-AND-REDUCE, a novel technique to synthesize compilation repair examples from a seed collection of compilable programs. Using this technique, we have been able to synthesize compilation repair examples for 51% of all Clang++ C++ compilation errors.

- We present COMPASSIST, a novel prototype tool that suggest repairs for compilation errors ranking patches that are more likely to repair the error first and showing examples that are most similar to the given uncompilable program.

- We show that synthetic compilation repair examples can be helpful to programmers seeking help in resolving compiler errors.

- We introduce the concept of *lexical distinguishability of source code*, and formulate the *Wheat and Chaff Hypothesis*, which states that units of code consists of 1) "wheat", important lexical features that preserve meaning, and "chaff", and 2) the "wheat" is small compared to "chaff".

- We formalize the problem of separating the wheat from the chaff in source code as a new NP-Complete problem, which we call The MINSET Problem. We prove that this is an NP-Complete problem by producing a polynomial-time reduction from the Hitting Set problem.

- Through an empirical study of over 100M Java LOC, we show that source code can be separated into wheat and chaff. In other words, we show that it is lexically distinguishable. We specifically show that as much as 96% of source code is potentially chaff. We present an exploration over various lexicons that shows that only a few lexical element suffice to distinguish Java methods from one another with a natural, minimal lexicon.

***Thesis Structure:*** We structure the remainder of the dissertation as follows. Chapter 2 discusses our approach in building a low-threshold web programming environment called Kodethon and our large scale study at a large public university. Chapter 3 presents our approach for helping students with compilation errors through our novel FUZZ-AND-REDUCE technique for synthesizing compilation errors and our prototype tool COMPASSIST. Chapter 4 presents our first steps towards a minimal syntax-free keyword-programming paradigm. Chapter 5 summarizes and presents our conclusions.

# Chapter 2

# Student Adoption and Perceptions of a Web Integrated Development Environment

Students often spend a considerable amount of time and effort installing and configuring programming tools and environments. This can frustrate, and distract them from more important learning objectives, particularly in introductory programming courses. A web-based integrated development environment (web IDE) can serve as a low-threshold, ready-to-use programming environment, and reduce the time and effort needed to start practicing programming.

We report our experience of developing and deploying a web IDE at a large public university in North America. Over the last three years, it has been used by thousands of students in diverse Computer Science courses as an *optional* programming tool. We designed and conducted a survey to understand students' usage of and perceptions toward the web IDE and its features. We also explored potential correlations between student demographic and behavioral traits and adoption of the web IDE. In this chapter, we also describe broader lessons for educators resulting from interactions with students and instructors and supported by our findings.

## 2.1 Introduction

Programming involves several steps such as coding, compiling, linking, testing and debugging [Kelleher and Pausch, 2005]. Each step requires proper installation and configuration of the corresponding tools and environments. Integrated development environments (IDEs) attempt to integrate and present all tools needed for programming in a unified interface, but still they require users to configure the system and install individual tools. For example, an IDE for Python programming language will provide syntax highlighting for Python programs, but users may still need to install the Python interpreters and configure the IDE to use the right installation (*e.g.* python 2.7 or 3.6).

Installing and configuring programming tools and environments can be a frustrating and error-prone task, especially for a student that is learning programming, and may also distract the student from the primary learning objectives [Jenkins et al., 2010]. For example, at University of California, Davis, there is an upper division "Programming Languages" course where students learn programming language theory and concepts by studying different languages, namely C++, Java, Lisp, and Prolog. Since the course is only ten weeks long (due to the quarter system), the assigned programming projects are limited in length and difficulty. Nonetheless, students have reported spending a significant portion of their time and effort (hours, and even days) installing and configuring programming tools for each new assignment.

A web-based integrated development environment can provide a uniform, simple programming interface and require no installation or configurations on the local machine [Goldman et al., 2011, van Deursen et al., 2010, Jenkins et al., 2010]. A web IDE is particularly desirable in classrooms because (1) it can reduce time for installation, configuration and troubleshooting of tools and environments, allowing students and instructors to focus on the primary learning objectives, and (2) it can provide a unified, reproducible execution environment, which can improve communicating programming problems between students and teaching staff [Jenkins et al., 2010].

We developed and deployed KODETHON, a web IDE at University of California, Davis. We have operated and maintained KODETHON for more than *three years*. KODETHON supports all of the programming languages used to teach courses at our university, including Python, C, C++, Java, Lisp, and Prolog. Its built-in editor provides convenient features like syntax highlighting and auto-completion. Student can also use an easy-to-use shell or a full Unix terminal—within the browser—to execute Unix commands, such as, `ssh` and `scp`. KODETHON also supports real-time collaboration which facilitates pair-programming and live teaching assistance.

In this chapter, we discuss adoption of the web IDE by students, and their perceptions toward it. To date KODETHON has been used by *more than 3,000 students* in 15 courses as an *optional tool*. Thus far, students have used KODETHON to write more than 15 million lines of code. We analyze survey responses from 140 students who took a course recently to understand student usage and perceptions of KODETHON.

We found that 48% of survey participants use the web IDE often. About a third of participants opted to use the web IDE to write their programs, and, coincidentally, about a third perceived the web IDE to be useful. Students ranked "Web-based" and "No Installation Required" as the two most useful features – consistent with the premise of web IDEs as a low-threshold, ready-to-use programming environment. We found that class standing has a strong correlation with adopting the web IDE, as novice students are more likely to adopt the system. We found students that use the web IDE more often tend to not use the alternative stand-alone IDEs or editors. However, we did not find strong correlations between adoption of the web IDE and adoption of authoring web applications like Google Docs.

The main contributions of this chapter are:

- We describe KODETHON, a web IDE that has had widespread use across diverse university CS courses, and its deployment (Section 2.3).
- We designed and conducted a user survey of student usage and perceptions of the IDE, and analyzed the results (Section 2.4).
- We explored correlations between adoption of a web IDE and student characteristics (Section 2.5).

- We discuss broader lessons for CS educators and researchers regarding web IDEs (Section 2.6).

## 2.2 Related Work

**Web IDEs for classrooms:** Perhaps, the work most related to ours is a recent study by Benotti *et al.* that describes a web IDE to support teaching functional programming in Haskell, and evaluated the students' attitude toward it [Benotti et al., 2018]. In their study, students strongly agreed that the web IDE makes them more productive, and a better Haskell programmer. They surveyed 86 students in two institutions. Barr *et al.* developed CodeLab as a Web IDE for introductory programming courses. They observed that the average grade of students increased moderately after adopting CodeLab (0.2 points in a 4-scale grading) [Barr and Trytten, 2016]. Note that in these studies students were *required* to use the web IDE for programming while we did not requires students to use KODETHON.

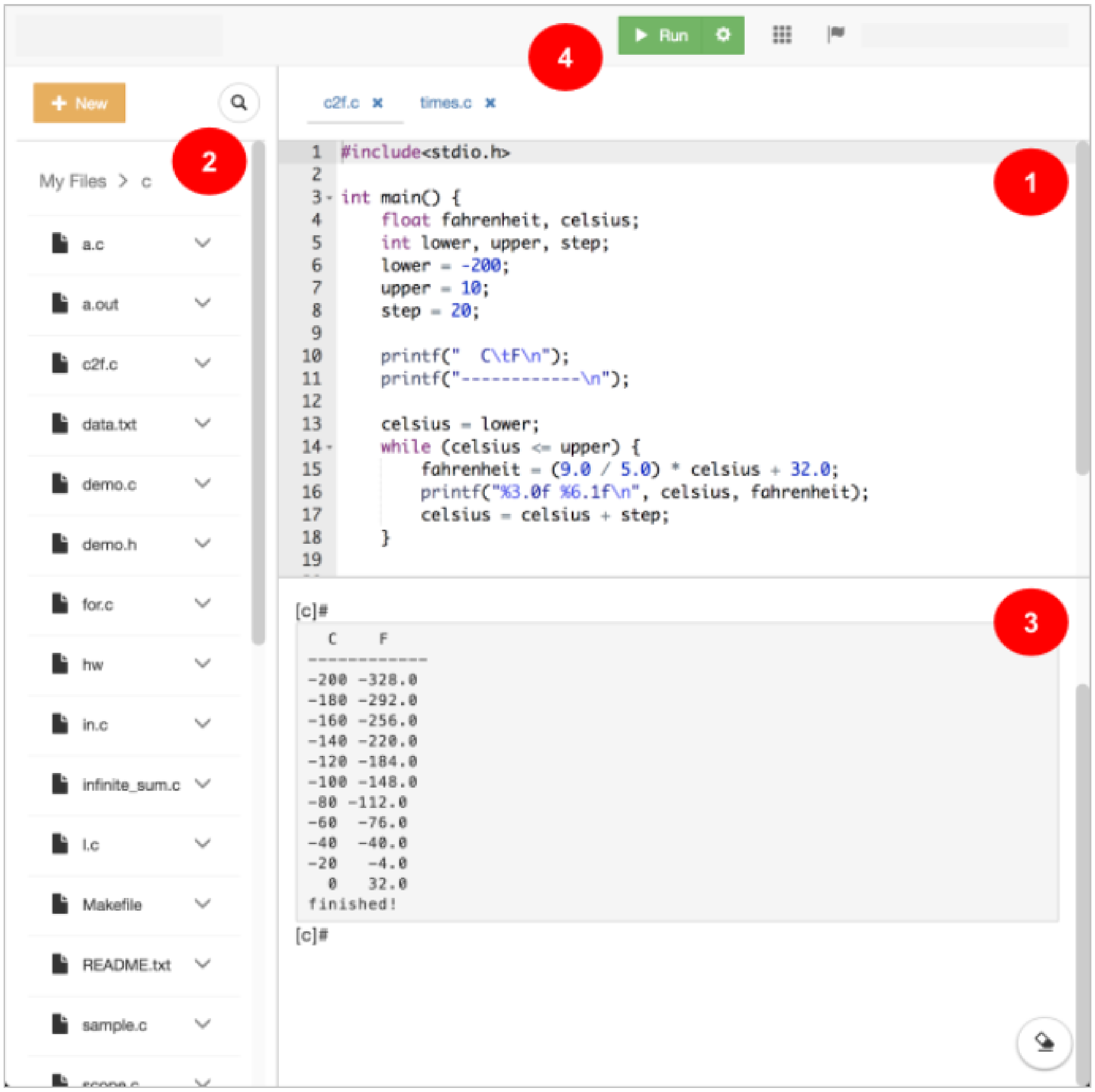
There are a number of web IDEs for classrooms. For example, PythonTutor is a popular Python tutoring system that in addition to execution of single-file programs, visualizes the heap of programs [Guo, 2013] . Helminen *et al.* developed a web IDE for Python programming and ran a user study [Helminen et al., 2013]. In their study, more than 40% of students reported that they used the system frequently, and a large portion of students indicated that the web IDE is useful and should be used in future course offerings. The major complaints were that students were used to other editors or IDE, and the system was slow.

**Pedagogical stand-alone IDEs:** While sophisticated features such as build, test, or integration of programs in professional IDEs help professional developers, they can be barriers in educational environments, especially in early programming classes where most students have little or no programming experience. Rich, sophisticated features can be intimidating to students, and distract them from the main learning objectives. Pedagogical IDEs attempt to address these barriers by abstracting and hiding irrelevant features of professional IDEs. For example, to teach object-oriented concepts, such as classes and relationships between them, BlueJ IDE provides visual tools to design a class and define the relationships between a class and other classes [Kolling et al., 2003]. BlueJ has been extended to be used in data structure courses [Paterson et al., 2005], to accommodate collaboration [Fisker et al., 2008], to teach design patterns [Paterson and Haddow, 2007], and in programming embedded systems [Altadmri et al., 2015]. A survey of computing education research community in 2006 shows that more than 25% respondents use BlueJ IDE in introductory courses [Schulte and Bennedsen, 2006].

Some pedagogical IDEs target particular needs of students. For example, Jenuity tools provides an *efficient* Java IDE that can run on outdated machines for students that do not have powerful computers [van Tonder et al., 2008]. Boyd *et al.* [Boyd and Allevato, 2012] developed an Eclipse plug-in to address the problem of configuration of assignment files on students' machines. Their plug-in directly downloads required files for assignments and configures the environment. Brune *et al.* found that, in web design courses, proper configuration of web servers and databases consumes a large portion of students' time [Brune et al., 2014]. Therefore, they created an IDE, where they wrap the functionality of a web server with a single Java class, therefore students can treat the execution of a web server as the execution of a simple Java program.

**Feedback in IDEs:** In addition to editing and building programs, IDEs can also provide hints to assist students to improve their programming skills. For example, ASIDE [Zhu et al., 2013], and ESIDE [Whitney et al., 2015] nudges students to adopt secure programming practices, or DevEventTracker [Kazerouni et al., 2017] evaluates how students adhere to principle of incremental programming and how much they procrastinate. Some IDEs also track students' activities to identify at-risk students [Munson, 2017, Dyke, 2011, Ford and Staley, 2016].

**Web IDEs in MOOCs:** With the advent of MOOC and large scale learning, web IDEs are becoming commonplace. For example, Khan Academy provides a simple web IDE in their programming lessons [Academy, 2017]. Most introductory programming courses on edX are also accompanied by a webIDE. Some web IDEs can offer automated feedback to students to fix the errors [Wang et al., 2018, Wang et al., 2017a]. Web-based



```
#include<stdio.h>

int main() {
    float fahrenheit, celsius;
    int lower, upper, step;
    lower = -200;
    upper = 10;
    step = 20;

    printf("  C\tF\n");
    printf("------------\n");

    celsius = lower;
    while (celsius <= upper) {
        fahrenheit = (9.0 / 5.0) * celsius + 32.0;
        printf("%3.0f %6.1f\n", celsius, fahrenheit);
        celsius = celsius + step;
    }

```

```
[c]#
   C    F
------------
-200 -328.0
-180 -292.0
-160 -256.0
-140 -220.0
-120 -184.0
-100 -148.0
-80 -112.0
-60  -76.0
-40  -40.0
-20   -4.0
  0   32.0
finished!
[c]#
```

Figure 2.1. KODETHON User Interface: File navigator, editor, and CDE Shell.

development environments also serve as a platform for massive collection of data from users actions. The data can be used to develop various predictive models. For example, Wang *et al.* use data collected from EdX's web IDE to predict the success of students in arriving at a correct code, given the intermediate steps they take [Wang et al., 2017b].

## 2.3 KODETHON Description and Adoption

KODETHON is a web integrated development environment—it requires no installation—and users can immediately start writing, executing, and storing programs. It is designed to be easy-to-use by university students, and to be useful in completing their coursework. To scale to classes of hundreds of students, we designed KODETHON as a distributed system that can scale horizontally. However, building a web IDE that scales to many users, has high availability, and fast response time is challenging task. We first describe the user interface features. Next, we describe how we addressed several technical challenges. Namely, how to support multiple programming languages? How to support multi-file programs?

### 2.3.1 Main Features

Figure 2.1 shows the main user interface which consists of an editor, a file navigator, a CDE shell, and a smart run button.

1. *Editor:* The editor is an instance of the open-source ACE editor which provides syntax highlighting for multiple programming languages and basic auto- completion. Users can personalize the editor by changing the theme, font, and indentation settings. ACE even supports editing in VIM mode which some users find to be a more efficient mode of editing. We added support to open multiple files and switch between them using tabs.

2. *File Navigator:* Users can create files and organize files in folders. They also can rename, move, copy, upload, and download files. This allows students to access their files from anywhere and from any device.

3. *CDE Shell:* This is KODETHON-specific shell that allows users to run common Unix commands like `ls` and `cd` but also allows users to run KODETHON-specific commands like `terminal` which opens a standard Unix terminal for more advanced users. The user can also compile and run programs using this shell.

4. *Smart Run Button:* A common point of initial confusion for students is: “How

do I run this program?" KODETHON employs a best-effort strategy for executing programs. For example, if the file end in `.c`, it will search for a `Makefile` and if it finds one, then it runs `make`. Otherwise, it will compile the current file, and run the resulting `a.out` program. Another example, if the file ends in `.l`, it will interpret it using `clisp`. In our experience, this helps students interact with programs much quicker. Users can customize build and run settings.

KODETHON also provides:

- *Multi-language Support:* It provides over a dozen programming environments, distinct sets of compilers, interpreters, and programming tools. For example, to program in Lisp, a user simply selects the `lisp` environment which includes the `clisp` interpreter. The environments are defined as Docker images. All user commands are executed in Docker containers. For scalability and performance, we limit resources per user such as disk space, RAM, and CPU usage.

- *Real-time Collaboration:* By default, a user starts with an always-private *project*. To collaborate with others, he/she must create a *shared project* and add collaborators. All collaborators will have access to that project. KODETHON allows users to set read/write/execute permissions for collaborators (think Linux groups), and for the public. We have observed that some users create a project for each homework while some create a project for the entire course. In a shared project, two or more people can edit the same file at the same time. We employ Firepad[1] to provide this functionality. But we also added support for File History, real- time chat, and comments.

- *Learning management system:* Instructors create programming assignments. Students upload submissions to KODETHON. The submissions are automatically graded. Students receive feedback on their submissions by running the provided test suite.

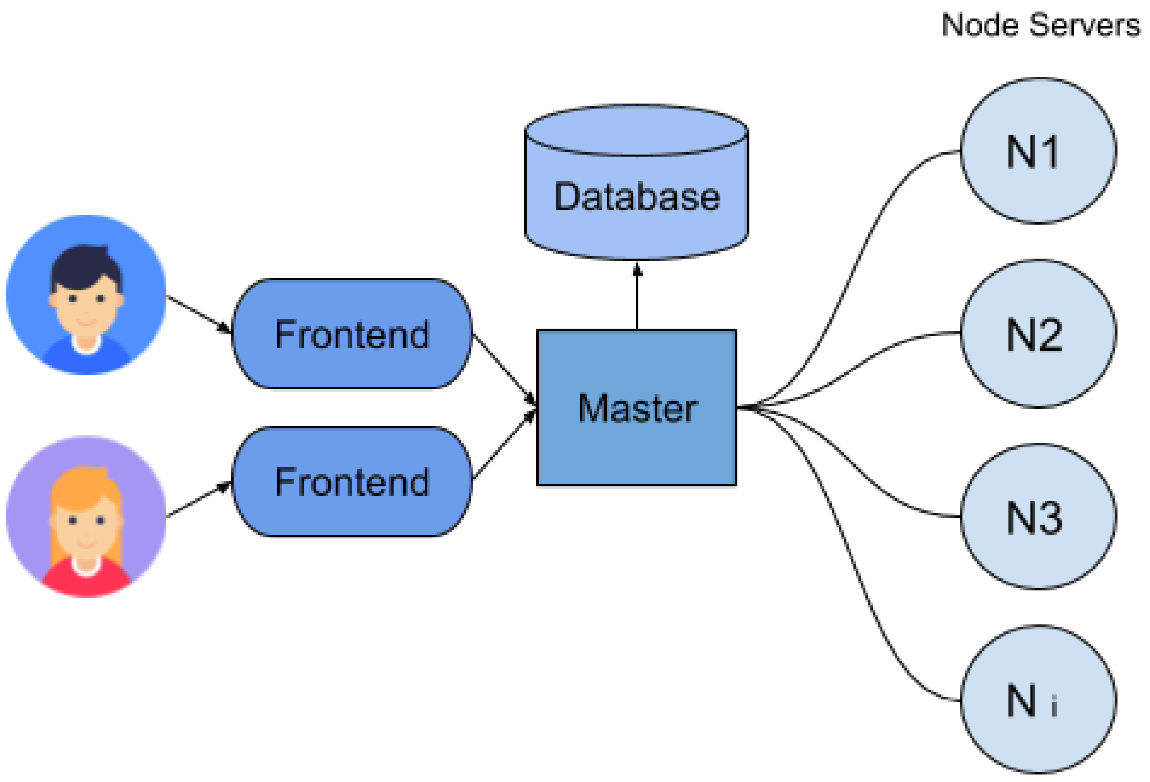


Figure 2.2. KODETHON Architecture: The user interacts with an AngularJS frontend. The frontend communicates with a master server. The master server handles user requests and orchestrates worker nodes in the cluster.

### 2.3.2 Architecture

KODETHON is a designed as distributed system. Figure 2.2 shows an overview of the KODETHON architecture. The user interacts with the front-end (Figure 2.1), which is an AngularJS application. The front-end sends HTTP request to the master server, which is a Ruby-on-Rails server application. The master routes user requests to the slave servers. The cluster consists of one or more slave servers, which store user files and execute user programs. The master reads and updates the database. The master is responsible for monitoring and orchestrating the entire cluster.

Each user is assigned to a slave server and all of his/her files are created and stored as real files and directories in this server. By assigning a user to a specific server, we can provide high availability of the user's files. In other words, the user does not need to

[1]https://firepad.io/

wait for files to be downloaded or moved around. When the user logs in, his complete file tree is usually retrieved and displayed in less than 500 milliseconds, even if the user has tens of thousands of files.

When a user opens/reads a file, the front-end requests it from the master server which in turn must first queries the database to determine the assigned server. Then, the master requests the file content from the slave server directly. The slave server reads the file, returns the content to the master server, as part of HTTP response, which returns it to the user interface. The user interface stores the received file content in a local buffer of the user's browser. All of this is usually completed in under 300 milliseconds for files of typical size, under 100KB. Of course, the actual time depends on file size and network speed. Since the user may edit the file in another browser tab or device, the front-end occasionally polls for new file contents. If the file on the server is different from the one stored in the active buffer, the user is asked which one he wants to keep.

Writing/editing a file works similarly. When the user edits and saves, the front-end sends the new file content stored in the local buffer to the master server which forwards it to the user's assigned server. The slave server replaces the old file contents with the new content.

Since students make mistakes, we also implemented a *File History* feature which takes a snapshots of a file. This allows user to undo changes without having to rely on any complicated version control system, like `git` or `svn`. Internally, we also replicate user files to backup and take snapshots; the details are beyond the scope of this chapter.

Docker provides several benefits. First, it provides security since all user actions are contained in Linux containers. Second, Linux containers allow for faster execution and response time since they have lower overhead than virtual machines. Third, it provides a easy and quick method to program in multiple programming languages. For example, to switch from Lisp to Prolog, a student simply needs to switch to the provided `Prolog` environment. No installation or configuration is required. Fourth, it provides a uniform execution environment. This means execution will be consistent in the same environment. This can help students collaborate, and can make it easier for teaching assistants and instructors to help students debug.

### 2.3.3 Deployment and Adoption

To deploy KODETHON to students, we asked instructors who were teaching programming-intensive courses for the opportunity to give a live demonstration of KODETHON during one of their lectures. We gave our first live demo to students taking "Introduction to Programming", which uses Python programming language, in the Spring 2015. Since then, we have given more than ten live demos. Professors usually allot 5 to 15 minutes at the beginning of a lecture in the first or second week of the academic quarter. We usually demonstrate 1) how to create and edit programs, 2) how to execute programs with the smart button, 3) how to use the CDE shell, 4) how to collaborate in real-time, and 5) how to access the more advanced but traditional Unix terminal. The programming language we used in the demo depends on the main programming language of the class. For a class like "Programming Languages", where students have to code in Java, Lisp, and Prolog, we emphasized the ease with which students can switch to and between the appropriate programming environments.

With each new wave of users, we sought and received user feedback. Students provided feedback in-person, on class forums (e.g., Piazza), indirectly to teaching assistants and professors, and via email. Based on this feedback, we found and fixed bugs, and added new features.

To date, more than $3,000$ students have signed up to use KODETHON in at least 15 different courses at University of California, Davis. Table 2.1 lists the courses where students have reported using KODETHON, where we have given live demonstrations, and where we distributed our user survey (Section 2.4). We gave live demonstrations in lower division programming courses, and upper division "Programming Languages" – where students are expected to program in Java, Lisp, and Prolog in a ten-week quarter. Student also reported using KODETHON in courses that we never gave demos too, *e.g.*, "Software Engineering", but several students reported using it there. which indicates that KODETHON has been useful across our CS curriculum.

Figure 2.3 shows how many users have signed up each month. The spikes corre-

Table 2.1. Responses to the survey question: *"I use or have used* KODETHON *to do coursework in the following courses."*

| # | Title (Intervention(s)) | Programming Language(s) |
|---|---|---|
| 1 | Introduction to Programming (LD) | Python |
| 2 | Introduction to Programming (LD) | C |
| 3 | Software Development and Object-Oriented Programming **(LD, S)** | C++, Rust |
| 4 | Computer Organization and Machine-Dependent Programming **(LD, S)** | Assembly, C++ |
| 5 | Data Structures and Programming (LD) | C, C++ |
| 6 | Theory of Computation | N/A |
| 7 | Algorithm Design & Analysis | C, C++ |
| 8 | Probability and Statistical Modeling for Computer Science | Python, R |
| 9 | Programming Languages **(LD, S)** | Java, Lisp, Prolog |
| 10 | Scripting Languages and Their Applications | Python, R |
| 11 | Parallel Architecture | C |
| 12 | Software Engineering | Varies |
| 13 | Web Programming | Javascript, HTML, CSS |
| 14 | Introduction to Artificial Intelligence | Varies |
| 15 | Computer Graphics | Varies |

LD = Live Demo, S = Survey

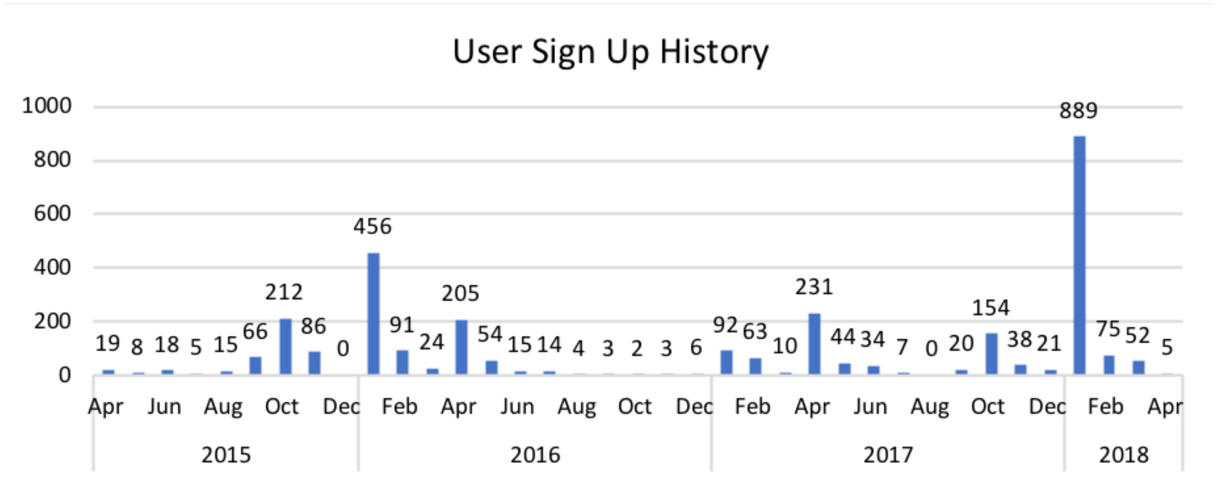


Figure 2.3. User sign up history. As of April 2018, 3,041 users have signed up to use KODETHON.

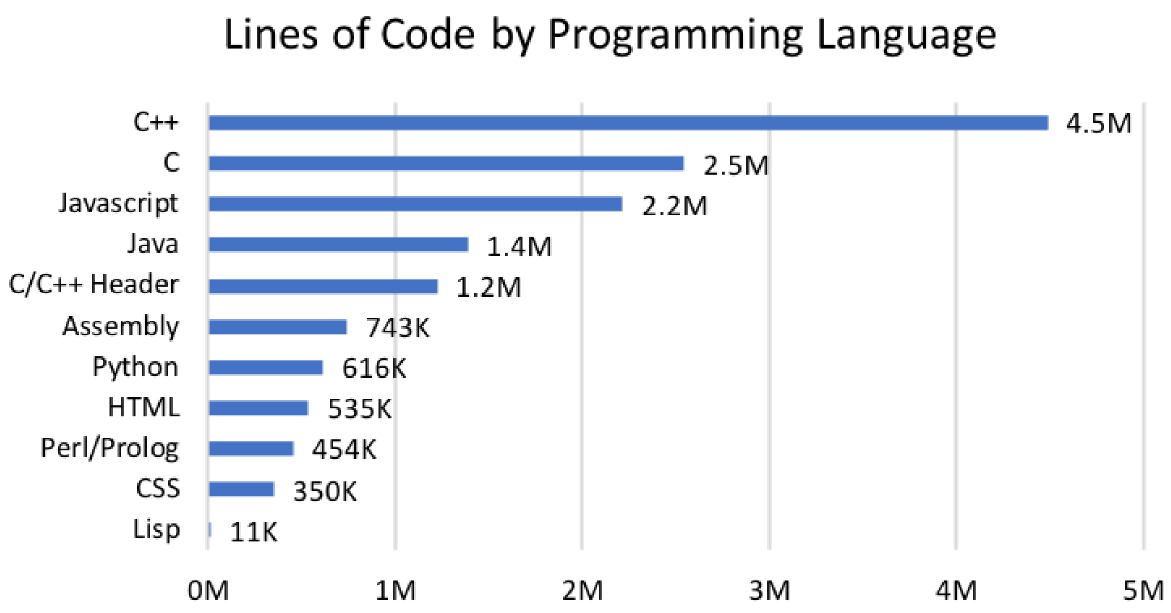


Figure 2.4. The lines of code students have written using KODETHON by programming language. This chart shows the top-10 languages and Lisp which is ranked 25 but is teaching language used at University of California, Davis.

spond to the month we gave live demonstrations to classes. The magnitude of the spike depends on the size of the class. The largest spike occurred during January 2018 which is when we introduced the LMS feature and the professors asked students to submit their assignments via KODETHON.

To measure how much code students have using KODETHON to write code, we ran cloc [2] over 7,643 private and public projects to count the number of lines of codes in their file directories. To date, students have used KODETHON to write over 15.1 million lines of code. This is an under-approximation in that we do not count all of the intermediate code students wrote and delete/overwrote. The top-5 languages were C++ (4.5M), C (2.5M), JavaScript (2.2M), Java (1.4M), and C/C++ Header (1.2M). C++ and C are used in large classes of about 300 students where students build projects that are hundreds of lines long. This is an over-approximation in that we count code not authored by students, like downloaded code.

## 2.4 User Survey

We conducted a multi-year, multi-course empirical study at University of California, Davis. The first phase began with the development of our main apparatus, KODETHON, in the summer of 2014, prior to the 2014-15 academic year. The second phase began with the deployment of the first prototype to students in an introductory programming course later that academic year, in spring 2015. The third phase began with the design and deployment of our first formal user survey in the academic year 2017-18. We performed quantitative analysis of student self-reported perceptions of KODETHON.

We developed a questionnaire consisting of 40 closed and open-ended questions, and scale items. Participants took 10 minutes, on average, to answer it completely. We asked about demographics, education, programming background [Siegmund et al., 2014], and general KODETHON usage. We also asked students about their perceptions of KODETHON using the measures defined in the USE Questionnaire [Lund, 2001]. Each items was scored on a 5-point Likert scale ranging from Strongly Disagree to Strongly Agree. We asked students perceptions about the impact of KODETHON in satisfying their learning objectives and their future career (Table 2.5). To learn about students' digital habits and their potential relationship with adoption of KODETHON, we asked students two questions: (1) "How often do you use Google Docs or Office 365[3]?" , and (2) How often do you use stand-alone editors and IDEs such as Sublime, Atom, and Eclipse?

[2] `https://github.com/AlDanial/cloc`
[3] Google applications and Office 356 are freely available to students.

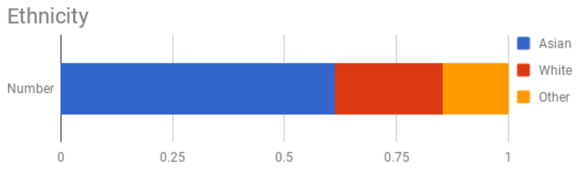


Figure 2.5. Ethnicity Distribution

We recruited participants by posting in Piazza class forums for two ongoing courses, and by emailing students from a third course from the previous academic quarter (Fall 2017) (see Table 2.1). We offered participants an entry in a drawing for ten $20 Amazon gift cards. Based on size of the classes, we estimate that we reached out to approximately 850 students in total from which 140 students chose to participate in our survey. The demographics of our participants were:

- Gender: Male (87), Female (51), Not identified (2).
- Ethnicity: Asian (90), White (35), Other (17), American Indian or Alaska Native (2), African American (1), Native Hawaiian or Pacific Islander (1).
- College: Letters and Science (91), Engineering (38), Agricultural and Environmental Sciences (8), Biological Sciences (2), Other (1).
- University Standing: Freshman (39), Sophomore (40), Junior (38), Senior (23).
- Years of Programming Experience: 0-1 (32), 1-2 (48), 2-3 (27), 3-4 (16), 4-8 (15), 8 or more (2).

Figure 2.5 shows the distribution of ethnicity in the our survey responses. Table 2.2 shows the distribution of college and status. Most participants were from College of Engineering (CE) and College of Letters and Science (CLS) because they host computer engineering, and computer science majors respectively.

## 2.5 Results

We were interested in gaining insights into student adoption, perceptions, and characteristics of adopters. In summary, we found that 48% of survey participants use KODETHON often. About a third, 33%, find it useful. And what they find most useful about

Table 2.2. Student Programming Experience by College. CAES = College of Agricultural and Environmental Sciences, CBS = College of Biological Sciences, CE = College of Engineering, CLS = College of Letters and Science

| Programming Experience | CAES | CBS | CE | CLS | Other |
|---|---|---|---|---|---|
| 0 to 1 years | 5 | 1 | 6 | 20 | 0 |
| 1 to 2 years | 2 | 0 | 13 | 32 | 1 |
| 2 to 3 years | 0 | 1 | 9 | 16 | 0 |
| 3 to 4 years | 0 | 0 | 6 | 9 | 0 |
| 4 to 8 years | 1 | 0 | 1 | 12 | 0 |
| 8 or more years | 0 | 0 | 1 | 1 | 0 |

Table 2.3. "I use KODETHON to." Multiple choices were allowed.

| Activity | Participants ↓ | Percentage of Participants ↓ |
|---|---|---|
| Submit Assignments | 117 | 84% |
| Test Programs | 63 | 45% |
| Write Programs | 46 | 33% |
| Collaborate | 36 | 26% |
| Share Programs | 18 | 13% |
| Other | 1 | 1% |

KODETHON is that it is a low-threshold, ready-to-use programming environment. We also discovered that the likelihood of adoption decreases with university standing.

### 2.5.1 Usage

Almost half of survey participants, 48%, reported using KODETHON often: Very Often (26, 19%), Often (41, 29%), Sometimes (23, 16%), Rarely (50, 36%), and Never (0, 0%). Table 2.3 shows that a large majority of participants, 83.6%, use KODETHON to submit assignments. It also shows that about a third of participants, 33%, use KODETHON to write their programs. This is consistent with our expectation that although not all students will opt to use KODETHON, a significant proportion will elect to use it. This is also consistent with our direct observations from previous courses where we observed about a third of all students opting to use KODETHON to complete the programming assignments. Most students tend to use the primary editor or IDE recommended by the instructor, for example, CLion or Eclipse.

Participants reported using KODETHON on a variety of devices. A great majority (95%) reported using KODETHON on their personal laptops. Interestingly, 8% use KODETHON on campus desktops. Indeed, we have directly observed students using KODETHON on campus desktops provided by the CS department which is interesting because those desktops already have ready-to-use programming environments. We hypothesize students find it more convenient to store their files on KODETHON where they can access their files from other devices. A large portion, almost half, of students used KODETHON in Windows. Students reported negligible use on other devices, *i.e.*, smart phones, tables, and other machines. Anecdotally, we have observed one student in complete all programming assignments for "Programming Languages" using only KODETHON on an iPad.

### 2.5.2 Perceptions

About a third of participants, 32%, perceived KODETHON to be useful. To measure this attitude, we used the Usefulness measure defined by Lund in the USE questionnaire [Lund, 2001]. We summarize the responses in Table 2.4. Considering all items, out of 1,082 responses, 32% (348) agreed that KODETHON is useful, 30% (325) were neutral, and 38% (409) disagreed. One of the most interesting items was the direct statement, "It is useful.". 55% of participants agreed. We conjecture that participants find different aspects of KODETHON useful and they converge under this broad statement. Another interesting item is "It does everything I would expect it to do." to which only 30% of participants agreed. This tells us that although we built many features into KODETHON, students still expect more from a web IDE. In future work, we plan to investigate what else students expect KODETHON to do.

We developed additional usefulness items to get a sense of students perceptions about KODETHON's usefulness beyond the classroom. Table 2.5 summarizes the re-

Table 2.4. Responses to the items in the Usefulness measure.

| # | Item | SA | A | N | D | SD | Count |
|---|---|---|---|---|---|---|---|
| 1 | It helps me be more effective. | 5% | 27% | 33% | 20% | 15% | 134 |
| 2 | It helps me be more productive. | 5% | 23% | 30% | 26% | 16% | 135 |
| 3 | It is useful. | 5% | 50% | 27% | 8% | 10% | 136 |
| 4 | It gives me more control over the activities in my life. | 5% | 16% | 38% | 22% | 19% | 129 |
| 5 | It makes the things I want to accomplish easier to get done. | 4% | 21% | 32% | 26% | 23% | 135 |
| 6 | It saves me time when I use it. | 4% | 24% | 25% | 29% | 18% | 138 |
| 7 | It meets my needs. | 4% | 33% | 32% | 18% | 13% | 136 |
| 8 | It does everything I would expect it to do. | 6% | 24% | 25% | 29% | 17% | 139 |
|  | Total | 5% | 27% | 30% | 22% | 16% | 1082 |

SA = Strongly Agree, A = Agree, N = Neither Agree nor Disagree, D = Disagree, SD = Strongly Disagree

sponses. The most surprising agreement rate was for "It helps me focus on the important aspects of programming." We expected higher agreement given that KODETHON allows students to simply write and run code without having to install anything. One explanation is that students may have varying opinions on what is "important" or even what is "programming." We were surprised that about a third of participants, 32%, feel that KODETHON can be useful for "real-world programming." However, it is in contrast to observations in [Benotti et al., 2018] that a vast majority of students perceived programming in a web IDE as real programming. Exploring these variances may also be interesting future work.

Lastly, for all items, a significant fraction of responses were neutral. We expected most students to agree or disagree. One possible explanation is that, as many participants reported, they do not use KODETHON often enough to form an opinion. An-

Table 2.5. Responses to additional Usefulness items.

| # | Item | SA | A | N | D | SD | Count |
|---|---|---|---|---|---|---|---|
| 1 | It helps me become a better programmer. | 4% | 16% | 43% | 22% | 14% | 137 |
| 2 | It is useful for real-world programming. | 3% | 29% | 36% | 19% | 13% | 136 |
| 3 | It helps me develop skills I will need in software engineer jobs. | 4% | 21% | 41% | 22% | 10% | 135 |
| 4 | It helps me focus on the important aspects of programming. | 5% | 20% | 38% | 24% | 12% | 138 |
|  | Total | 4% | 22% | 40% | 22% | 12% | 546 |

SA = Strongly Agree, A = Agree, N = Neither Agree nor Disagree, D = Disagree, SD = Strongly Disagree

other possible explanation is that students may feel that they have not been exposed to enough IDEs to form an opinion. We did not find that any participant simply replied the same to all items. It would be interesting to investigate this deeper and explore if there are missing features or usability traits that would cause participants to shift from neutral to agreement that KODETHON is useful.

We provided participants with a list of 18 features, and asked them to select which features they found most useful. Table 2.6 shows a summary of the responses. The most important finding is that what participants find useful corresponds with our hypothesis that many students would find a web IDE useful because it is a convenient, low-threshold, ready-to- use programming environment. Participants selected "Web-based" (65%) and "No Installation Required" (61%) as the top two features.

We asked participants to list the most positive aspects (up to 3) in their own words. We coded 262 responses into 16 categories, shown in Table 2.7. Participants mentioned many of the same features we provided earlier but in their words. However, here students mention the LMS (automatic grading and feedback) 55 times, and the real-time

Table 2.6. The top-10 responses to "What features of KODETHON do you find most useful?"

| Feature | Participants ↓ | Percentage of Participants ↓ |
|---|---|---|
| Web-based | 90 | 65% |
| No Installation Required | 85 | 61% |
| Assignment Grading | 73 | 52% |
| Works on Multiple Devices | 52 | 37% |
| Unix Terminal | 49 | 35% |
| Assignment Feedback | 48 | 35% |
| Real-time Collaboration | 46 | 33% |
| Programming Language Support | 35 | 25% |
| File Cloud Storage | 32 | 23% |
| Syntax Highlighting Code Editor | 32 | 23% |

collaboration features 54 times. Participants also commented on the usability of KODETHON saying it was easy to use 36 times and easy to learn 2 times. Here are some example responses:

- "I appreciate the cloud storage a lot. It saved my grade when my laptop broke."
- "Code accessible and runnable on any device."
- "I like how you don't need to install anything, for beginners this is helpful."

We also asked participants to list the most negative aspects (up to 3) in their own words. We coded 237 responses into 14 categories. 56 participants reported difficulties or issues or difficulties with the user interface. 37 participants mentioned experiencing issues with file saving. 34 participants responded simply "buggy" or "glitchy" without elaborating on which feature. A large fraction of the issues experienced by users have been due to heavy load on the system. As a result, we responded by increasing the nodes in our cluster, and making performance improvements.

Table 2.7. The top-10 categories of open-ended responses to "List the most positive aspect(s) of KODETHON."

| Feature | Responses ↓ | Percentage of Responses ↓ |
|---|---|---|
| Assignment Grading and Feedback | 55 | 21% |
| Real-time Collaboration and Chat | 54 | 21% |
| Easy to Use | 36 | 14% |
| Web-based | 18 | 7% |
| File Cloud Storage | 18 | 7% |
| Other | 16 | 6% |
| Unix Terminal & CDE Shell | 13 | 5% |
| Programming Language Support | 13 | 5% |
| No Installation Required | 3 | 9% |
| Smart Run Button | 3 | 7% |
| Total | 262 | 100% |

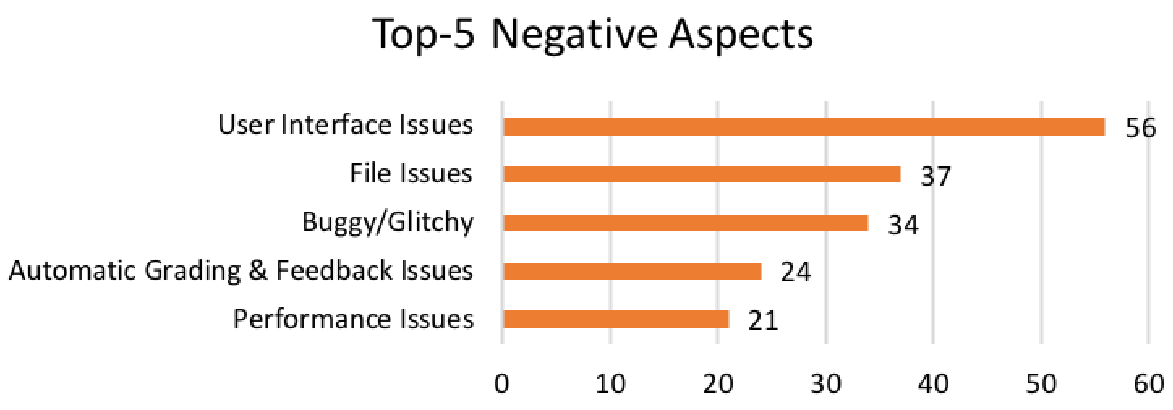


Figure 2.6. *"List the most negative aspect(s) of* KODETHON*:"*

### 2.5.3 Characteristics of Adopters

We classified students into two groups: 67 *adopters* and 73 *non-adopters*. Adopters are those who reported using KODETHON "Often" or "Very Often". The rest are non-adopters. We searched for individual factors that might correlate with adoption of KODETHON. We used Chi-square test for independence between adoption and 1) gender

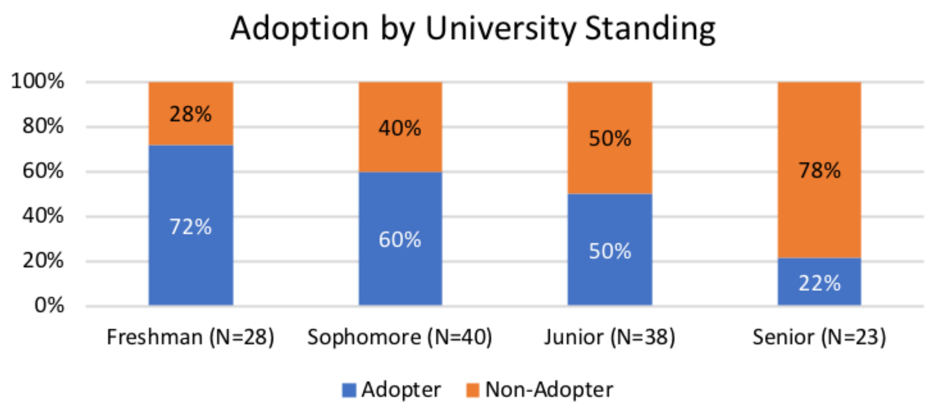


Figure 2.7. Adoption by students by university standing.

(p > 0.05), 2) university standing (p < 0.002), 3) programming experience (p > 0.05), and 4) university college (p > 0.05). We found that adoption correlates with university standing (p < 0.002). Students were less likely to adopt KODETHON as standing increased. We show the results in Figure 2.7. For digital habits, test of independence failed to find difference between the digital habits of adopters and non-adopters in using online editing tools such as Google Doc or other web IDEs ($p > 0.05$). However, Chi-square test suggested a difference in usage of stand-alone IDEs and editors between adopters and non-adopters ($p < 0.002$) which indicates that as students adopt KODETHON they tend to not use the alternative stand-alone tools and vice versa.

## 2.6 Broader Lessons

We discuss some broader lessons resulting from interactions with students and instructors and supported by our student adoption and perception findings.

**Some students prefer a web IDE:** Usually, instructors tend to select and present a single IDE, *e.g.*, CLion. Instead, instructors can provide students with multiple programming environment options, including a desktop and a web IDE, and encourage students to choose one, weighing the benefits and trade-offs. This is consistent with the principle that every student learns differently and with previous work [Helminen et al., 2013]. And as our results show, many students will voluntary opt to use a web IDE and adopters tend to not use stand-alone IDEs. Moreover, students adopted KODETHON not merely because it was novelty but because it provided tangible benefits to them, ease of use, convenience, and portability. As new generation of students use the web more often and longer [Matrix, 2014, Strasburger et al., 2013], a web IDE will be a familiar system for them to interact with. Moreover, cloud storage of files seems to assure students that their programs and homework will not be lost due unexpected hardware or software failure on their machines. KODETHON also provides them with an easy way of collaborating in real-time to pair program without having to resort to tools like `git` which often introduce more complexities [De Rosso and Jackson, 2016].

**A web IDE is not a silver bullet:** As our results show, a web IDE is not an ideal tool for every student. First, many students do not struggle with or do not mind installing programming tools. Second, web IDEs introduce different challenges. For example, the user interface is different and can take effort and time to learn. While many users found KODETHON easy to use, many others felt otherwise. Another example, because KODETHON is a web application, fluctuations in network speed can cause the user to experience latency or even downtime. Another example, even if a user enjoys the user interface and performance, network glitches can negatively impact their experience; an non-issue in desktop IDEs.

**Learning objectives need to be explicit and clear:** In discussions with instructors at University of California, Davis, we have learned that they often have hidden learning objectives, often unknowingly. One is to learn how to install and configure development tools. The rationale is that there is value in such a skill. A problem is that this learning objective is rarely made explicit or evaluated. On balance, some instructors have found, it is more important to give students a ready-to-use programming environment so they can spend more time learning to actually write code. Another problem with this hidden objective that it is too vague; it does not specify which tools (IDEs, text editors, compilers/interpreters, *etc.*) should be learned.

**Students confuse programming and system building:** Based on conversations with students, many do not consider programming in a web IDE to be "real programming". This negative view may stem from the fact that some students confuse computational thinking and programming the system building. In system building, developers must consider the production environment and compile, configure and tests their programs to meet its requirements, while computational thinking concerns with creating an appropriate abstraction of the problem domain and encoding the abstraction in a given programming language. A large portion of topics in computer science curricula, especially in the introductory courses, focus on the latter. Moreover, system building is still possible in a web IDE like KODETHON. In fact, there are a growing number of professional programmers working primarily on web IDEs, like Amazon Cloud9 [Inc., 2017b] and Eclipse Che [Foundation, 2017], developing on cloud instances.

**There is a need for a simple web IDE:** When we began developing KODETHON, there were a few existing web IDEs, namely Runnable [Runnable, 2017], Koding [Koding, 2017], Nitrous [Lardinois, 2016], and Cloud9 [Inc., 2017b]. None of them were designed for students and none of them were customizable to our needs at University of California, Davis. So we decided to build our own. We learned that building a web IDE that is reliable and scalable for *real use* is an expensive endeavor with a lot of technical challenges. With two engineers, it still has taken almost four years to reach our current state. Nonetheless, it was a good decision considering that Runnable and Koding have since changed focus and Nitrous has shut down. While others have surfaced like CodeAnyWhere [Inc., 2017a] and CodeEnvy [CodeEnvy, 2017], we have been able to leverage our architecture to introduce a learning management system to help instructors grade assignments automatically and provide students with instant feedback, a feature which 52% of participants found useful.

## 2.7 Threats to Validity

The data presented is based on quantitative anonymous survey study. The results are limited to responses to the questions. We did not analyze and could not analyze the system logs in the system to verify the accuracy of responses to questions. Students background can impact the generalization of the results beyond this study. We deployed the system in a public, selective university, where most students have some prior programming experience. Prior experience can influence their view of educational programming tools. For example, they may already know how to set up and configure the programming tools; in that case, a web IDE is of little value to them. However, in our survey, majority of students, even some non-adopters, indicated that a web IDE is a useful tool.

## 2.8 Conclusion and Future Work

We described a web IDE and its deployment in a large public university. To date, it has been used by more than 3,000 students in multiple programming courses. We used a quantitative survey study to evaluate students satisfaction and perception of the system.

In future, we plan to investigate the impact of instructors in the adoption of the tools and the impact of using a web IDE in the performance of students in courses. Instructors have a pivotal role in adoption of the system, and forming students' perceptions about web IDE. Our informal discussions with instructors showed that there are mixed views to adopting a web IDE by students: while some instructors are open to adoption of a web IDE by students, others consider challenges in the installation of the programming environment an important step that students should deal with them first. We plan to measure the extent of the impact of instructors view on the adoption of the web IDE by students.

# Chapter 3

# CompAssist: Synthesizing Minimal Compilation Repair Examples

Every programmer, from novices to professionals, makes compilation errors. Resolving compilation errors can be time-consuming, difficult, and frustrating. For decades, error messages have been identified as a source of this difficulty. A promising approach to help programmers is to augment error messages with *compilation repair examples*. The challenge is how to obtain and present these repair examples.

We present COMPASSIST, a system that generates and refines repair examples. Based on these repair examples, the system suggests possible patches to users when their program fails to compile. We evaluated COMPASSIST on a mainstream C++ compiler, and demonstrate that it can generate examples for more than half (867/1,686) of compiler errors. We also conducted a user study where participants found this synthetic repair examples to be helpful in a majority (5/9) of tasks involving real-world C++ compiler programs.

## 3.1 Introduction

Programmers of all levels struggle with compiler errors [Altadmri and Brown, 2015, Hristova et al., 2003, Jadud, 2006, Kummerfeld and Kay, 2003]. Compilers display error messages to help users locate and resolve code defects. However, these messages are notoriously difficult to understand, especially by beginners, and can even be misleading [Barik et al., 2017, Brown, 1983, Becker, 2015, Isa et al., 1983, Traver, 2010]. In addition to wasting time, programmers can become frustrated and beginners can lose interest and even quit learning. As programmers gain experience, they learn to recognize common messages, remember root causes and repairs, and eventually struggle less. But even as professionals, software engineers continue to make errors [Barik et al., 2017, Böhme et al., 2017, Seo et al., 2014]. For example, a recent case study at Google found that C++ and Java developers fail to build programs, on average, 37.4% and 29.7% of the time [Seo et al., 2014]. And it takes them a median of 5 and 12 minutes, respectively, to resolve each error.

One further complication is that the quality of error messages and tool support varies by programming language. As a result, as highlighted by the Google study, for example, the compiler errors made in C++ are different from those made in Java. One likely explanation is that Java developers (unlike C++ developers) rely on Eclipse's QuickFix which continuously suggests repairs for Java errors [Eclise, 2018]. However, many programming languages lack this tool support. Thus, how can we help programmers resolve compiler errors who are using an arbitrary programming language and compiler?

One natural step is to improve the messages emitted by compilers [Marceau et al., 2011a, Marceau et al., 2011b, Nienaltowski et al., 2008, Traver, 2010]. Almost a decade ago, GCC developers completely replaced the C parser with a top-down parser to provide better "diagnostic messages" [Free Software Foundation, 2004]. Clang, a newer compiler frontend, is well-known for focusing on "expressive diagnostic messages" from the start [Clang, 2018]. However, many compiler error messages remain cryptic across compilers, including Clang and GCC.

Another approach is to attempt direct repair. Recently, several data-driven automated repair techniques have been proposed. One group aims to learn shallow models of correct code and repairs [Campbell et al., 2014, Long and Rinard, 2016, Pu et al., 2016, Gupta et al., 2017, Ahmed et al., 2018]. The hope is that these models can be used to locate defects and suggest repairs. Another group aims to learn rules/transformations from repair examples [D'Antoni et al., 2017, Long et al., 2017, Rolim et al., 2017]. While promising, these techniques have relatively low accuracy and require data, repair examples. Often the data is mined from student homework submissions and thus can be a small subset of all possible errors and biased toward beginner errors.

A different and promising direction is to augment existing error messages with examples [Barik et al., 2014, Becker, 2016, Flowers et al., 2004, Hartmann et al., 2010]. A *compilation repair example* is a pair of programs that illustrates how to repair a compilation error. Seeing how a similar compiler error has been solved before may help a programmer resolve his/her own error. The challenge is: how to obtain and present these repair examples? Given that there many languages, many compilers, and that each error can be repaired in different ways, manually crafting examples [Kummerfeld and Kay, 2003] is not feasible. Crowd-sourcing [Hartmann et al., 2010, Mujumdar et al., 2011] is more feasible but is limited by bootstrapping, data bias, and privacy issues — see Section 3.8.

We present CompAssist, a system that synthesizes minimal compilation repair examples automatically via a novel Fuzz-and-Reduce technique. And, based on these examples, suggests possible patches for a compiler error. Since the primary intention of a repair example is to help programmers, examples should be *minimal* in the sense that they provide only sufficient code context for a programmer to understand it and decide if the patch is applicable. Irrelevant context can make a repair example difficult to comprehend and ultimately not helpful.

The *key insight* of our approach is that it is much easier to break compilable code than to repair uncompilable code. At a high-level, we start with a compilable program $c$, and randomly mutate it. If the mutated program $u$ is uncompilable, then we have a compilation repair example! The *intuition* is that if we do this over many different compilable programs with random mutations, we can get many repair examples and high coverage of the compiler error space.

We evaluated COMPASSIST on Clang++, a popular mainstream C++ compiler, with respect to coverage (breadth and depth), example simplicity, and helpfulness. Even with the simplest token-level fuzzing strategy (single-token mutations), our system is able to cover more than half (51.4%) all possible error messages in a mainstream C++ compiler, and can provide at least two distinct candidate patches for a large majority (79.8%) of error messages. A majority (59.6%) of the synthesized examples are "small" (a proxy for simplicity), containing five or fewer lines of code. Lastly, in a user study involving 14 participants, with a median 2.5 years of C++ experience, participants found the examples "helpful" in 5 out of 9 tasks involving real-world C++ compiler errors.

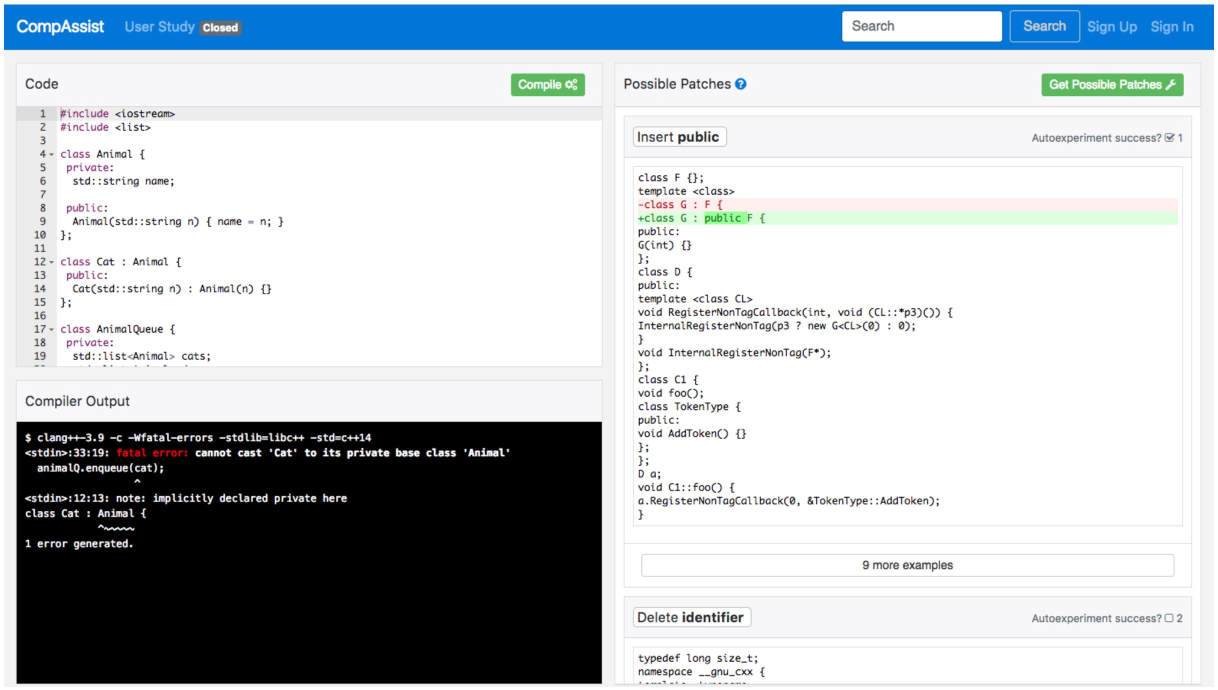
In summary, our contributions are as follows:

- FUZZ-AND-REDUCE, an offline technique to generate and refine compilation repair examples from a collection of compilable seed programs, and an online technique to search and present relevant candidate repairs with examples to users.

- COMPASSIST, a practical realization of these techniques, and a quantitative evaluation that shows that our approach can achieve high coverage of compiler errors, in terms of breadth and depth, and that it can generate simple examples.

- A user study that shows that *synthetic* repair examples are helpful to programmers in fixing compiler errors in real-world programs.

The remainder of the chapter is organized as follows: In Section 3.2, we provide an overview of COMPASSIST. In Section 3.3 and Section 3.4 , we describe the offline generation and online search components of our system. In Section 3.5, we present the results of our quantitative evaluation. In Section 3.6, we present our user study. In Section 3.8, we discuss related work. Lastly, we conclude.



```
#include <iostream>
#include <list>

class Animal {
 private:
  std::string name;

 public:
  Animal(std::string n) { name = n; }
};

class Cat : Animal {
 public:
  Cat(std::string n) : Animal(n) {}
};

class AnimalQueue {
 private:
  std::list<Animal> cats;
```

```
$ clang++-3.9 -c -Wfatal-errors -stdlib=libc++ -std=c++14
<stdin>:33:19: fatal error: cannot cast 'Cat' to its private base class 'Animal'
  animalQ.enqueue(cat);
                  ^
<stdin>:12:13: note: implicitly declared private here
class Cat : Animal {
            ^~~~~~
1 error generated.
```

```
class F {};
template <class>
-class G : F {
+class G : public F {
public:
G(int) {}
};
class D {
public:
template <class CL>
void RegisterNonTagCallback(int, void (CL::*p3)()) {
InternalRegisterNonTag(p3 ? new G<CL>(0) : 0);
}
void InternalRegisterNonTag(F*);
};
class C1 {
void foo();
class TokenType {
public:
void AddToken() {}
};
};
D a;
void C1::foo() {
a.RegisterNonTagCallback(0, &TokenType::AddToken);
}
```

```
typedef long size_t;
namespace __gnu_cxx {
```

Figure 3.1. CompAssist User Interface: A user types a C++ program, and compiles it with the "Compile" button. The system shows colorized compiler output. If the program triggers a compiler error, CompAssist suggests repair patches with minimal examples to help a user understand the suggested patch. This C++ program was posted to StackOverflow by a person seeking help understanding and repairing the compiler error. We label this program T10 in our user study.

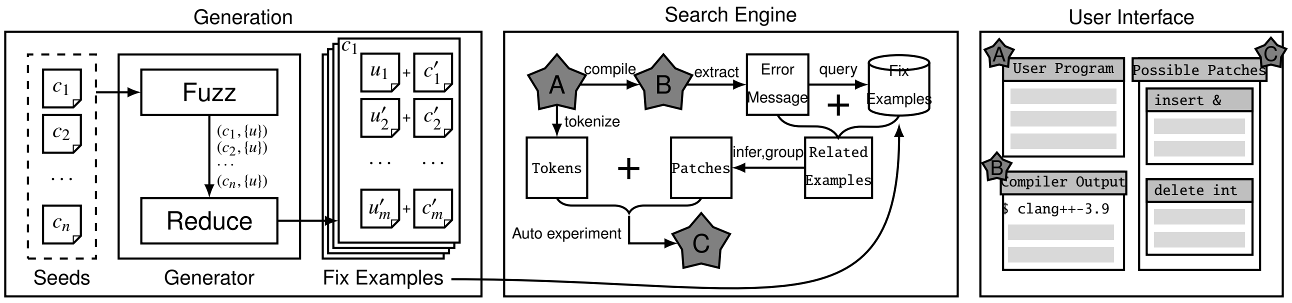


Figure 3.2. CompAssist Architecture: The generator synthesizes example fixes from a collection of compilable seed programs. The search engine retrieves and ranks example fixes given a user program as a query. The web user interface presents example fixes and suggest possible patches when the user program fails to compile.

## 3.2 Overview

CompAssist's user interface, shown in Figure 3.1, is a web application that compiles a program, extracts the compiler error message from the compiler output, retrieves related repair examples, and suggests a list of possible patches. It ranks the patches to show those are most likely to fix the user program. For each patch, it provides repair examples which may help the programmer understand the suggested patch, and determine if it is applicable to his/her particular error. It ranks the repair examples to show those that have more in common with the user program first.

To illustrate how a programmer benefits from CompAssist, let us discuss a real-world programmer, whom we shall call Mary. Mary wrote the small `C++` program shown in Figure 3.1; she may be just learning inheritance, an important concept in `C++`. Her program fails to compile due to a defect involving inheritance. The actual defect is that the `CAT` class inherits through private inheritance, which is the default behavior. Instead, it should inherit through public inheritance, as Mary confirms in her answer. In other words, she needed to insert `public` at line 12 before `Animal`.

Like many programmers, she posts her code and the compiler error to StackOverflow, and asks for help.[1] Other users try but fail to help her. One offers help about casting. Another asks for additional information, which is probably unnecessary based on her post. Eventually, Mary posts that she was able to resolve the compiler error on her own and provides her solution.

Now suppose that instead she uses CompAssist for help. She copy-pastes her program into the code editor. Then, she clicks "Compile". The first patch suggestion is the correct patch, "Insert `public`". Mary resolves her compiler error in a matter of seconds instead of minutes, based from the time difference between her question and answer. We expect CompAssist to be helpful to a programmer in scenarios like this where the programmer reads the compiler output but still fails to fix the defect. In these cases, the suggested patches and repair examples would simply be additional information.

[1]https://stackoverflow.com/questions/27594593/cannot-cast-subclass-to-its-private-base-class/27594850

```
1 //Error Message: invalid operands to binary expression ('int' and
     'int (*)(const Foo &)' (aka int (*)(const int &)'))
2 constexpr int Apply(const int in, int (*f)(const int&)) { return
     f(in); }
3 using Foo = int;
4 static constexpr int id(const Foo& i) { return i; }
5 - static constexpr int results1 = Apply(0, &id);
6 + static constexpr int results1 = Apply(0 &id); //Delete comma
```

Figure 3.3. This is a compilation repair example produced by our FUZZ implementation. Deleting a comma from line 4 transforms this compilable program $c$ into an uncompilable program $u$. The seed is the GCC Test Suite test case “constexpr-60245.C”.

```
1 //Error Message: invalid operands to binary expression ('int' and
     'int (*)(const Foo &)' (aka int (*)(const int &)'))
2 int Apply(int, int(const int&)) {}
3 using Foo = int;
4 int id(const Foo&) {}
5 - int results1 = Apply(0 &id);
6 + int results1 = Apply(0, &id); // Insert comma
```

Figure 3.4. This is a *reduced* compilation repair example produced by our REDUCE algorithm. Compare this with the repair example shown in Figure 3.3. Both trigger the same error message and both are fixed with the same patch. However, this repair example is smaller.

Figure 3.2 shows COMPASSIST’s architecture. The frontend is supported by two backend components 1) an offline generator that synthesizes repair examples from a corpus of compilable programs and stores them in a database, and 2) an online search engine that retrieves and ranks repair examples in response to user queries.

## 3.3 Offline Generation

The offline generation component, outlined in Figure 3.2, generates *compilation repair examples* via a novel FUZZ-AND-REDUCE approach. The example shown in Figure 3.3 was generated by this component.

**Algorithm 1:** Generate compilation repair examples via the FUZZ-AND-REDUCE approach.

```
Data: $\mathcal{P}$ is a corpus of compilable programs.
Function Generate($\mathcal{P}$)
    $E \leftarrow \{\}$                                  // Repair Examples
    foreach $c \in \mathcal{P}$ do
        $u \leftarrow$ Fuzz($c$)                          // Fuzz seed
        $(exit\_status, output) \leftarrow$ Compile($u$)
        if $exit\_status == ERROR$ then
            $(u',c') \leftarrow$ REDUCE($u,c$)
            $E \leftarrow E \cup \{(u', c')\}$
    return $E$
```

**Definition 1 (Compilation Repair Example)** *For a specific compiler, (u, c) is a* compilation repair example *for error message e, if u is an uncompilable program that triggers e, and c is a compilable program. The difference between u and c is a repair patch (diff) $\Delta$.*

**Remark 1 (Program Snapshot Pair)** *A compiler repair example is a specific case of a bug fix, $(buggy, fixed)$. These are often collected implicitly, as in HelpMeOut [Hartmann et al., 2010] and Tracer [Ahmed et al., 2018], or explicitly, as in NoFAQ [D'Antoni et al., 2017].*

In general, compilers emit different kinds of error messages and there is usually more than one way to repair a single kind of compiler error. How can we generate compilation repair examples for *all* possible kinds of errors, and generate diverse repair examples for *each* kind of error? In other words, how can we get high coverage of the compiler error space, in terms of *breadth* and *depth*? The size of the repair examples also matters; large repair examples will be likely difficult to comprehend. How can we minimize repair examples?

A natural strategy for these problem would be to collect uncompilable programs, and repair them. Tools like HelpMeOut [Hartmann et al., 2010], Crowd::Debug [Mujumdar et al., 2011], NoFAQ [D'Antoni et al., 2017] crowd-source this work to their users. Other tools like sk_p [Pu et al., 2016], Prophet [Long and Rinard, 2016], DeepFix [Gupta et al., 2017], Genesis [Long et al., 2017] and TRACER [Ahmed et al., 2018] simply mine existing repair examples from software repositories.

We define a generation function GENERATE that employs a novel FUZZ-AND-REDUCE approach, shown in Algorithm 1. For a given compiler, our approach requires a corpus of compilable programs $\mathcal{P}$. We process each seed individually (line 3). In each iteration, we fuzz $c$ into $u$ (line 4). We attempt to compile $u$ (line 5). If compilation failed (line 6), then $(u, c)$ is a repair example. We attempt to reduce it from $(u, c)$ to $(u', c')$ (line 7). We add the reduced example fix $(u', c')$ to our set of examples $E$ (line 8). We can call GENERATE indefinitely to attempt to generate more examples from each seed. In our implementation, we store the examples in a PostgreSQL 9.3 database. Figure 3.4 shows an example generated by Algorithm 1.

Exactly how many examples are generated (the size of $E$) and how many distinct error messages are covered will depend on many factors, including the seed programs $\mathcal{P}$ and the implementation of FUZZ, which we we describe in the next section.

### 3.3.1 Fuzzing a Compilable Program

Our approach does not assume or require a specific method of fuzzing seed programs. Fuzzing the code randomly at the character level will *likely* generate repair examples. For example, rewriting C++ programs into Shakespeare excerpts will *break* them, and the result will be a compilation repair example, by definition. However, the resulting patches will likely not generalize to real-world programs. More specifically, it is likely that few of those patches will be "acceptable" to users — see [Monperrus, 2014] for a discussion on "fix acceptability".

We implemented FUZZ as a token-level fuzzer. First, it tokenizes the source code. Then, it mutates the token sequence. We mutate it by deleting a random token or by choosing a random token and inserting it at a random position. Finally, it translates the token sequence back to source code. Figure 3.3 shows a repair example produced by our FUZZ. We implemented *single-token* mutations. So patches have high-level interpretations like “Insert TOKEN” or “Delete TOKEN” where TOKEN is one of the many different token types by Clang.[2] Many token types precisely identify the corresponding lexeme. For example, “comma” is just “,” and “ampamp” is “&&”. Some of the token types are abstract classes, for example, “identifier”, which can represent variable names, functions names, and type names.

Our fuzzer is 1) black-box, 2) mutation-based, and 3) smart. It is black-box in the sense that it does not require access to the compiler source code. It mutates existing compilable programs as opposed to generating seeds from scratch. And it is smart in the sense that it has knowledge of the token-structure of its inputs (source code).

In a quantitative evaluation (Section 3.5), we show that this simple single-token fuzzer is effective at covering different error messages. This implementation also helps ensure that, by construction, *patches are small*. We leave for future work implementing fuzzers that perform more complex token-level mutations, and that mutate other structural representations of the source, *e.g.*, the abstract syntax tree.

### 3.3.2 Reducing a Compilation Repair Example

Given a compilation repair example $(u, c)$, let $e$ be the error message triggered by compiling $u$. Let $\Delta$ be the patch from $u$ to $c$. Let $\Delta^R$ be the (reverse) patch from $c$ to $u$. How do you reduce $c$ to $c'$ such that applying $\Delta^R$ to $c'$ yields a reduced $u'$ that fails to compile with the same error message $e$? The problem is stated this way to require that, though $u'$ and $c'$ may be smaller, they are still a repair example for the same error message $e$ and that the repair is, at a high-level, the same. In short, reduction should not change the essence of the repair example. The repair example shown in Figure 3.4 is a reduced version of the example shown in Figure 3.3. Note that the programs are smaller but the patch is the same, "Insert a comma between the arguments to the call to Apply.", with the difference of positions, of course, since the code has shifted.

A well-known general method of reducing a program is the minimizing Delta Debugging (DD) algorithm [Zeller and Hildebrandt, 2002]. Applying DD naively to $u$ would systematically delete chunks (*e.g.* characters) from $u$ until removing chunks no longer

[2]`https://code.woboq.org/llvm/clang/include/clang/Basic/TokenKinds.def.html`

**Algorithm 2:** Reduce a compilation error repair example.

```
Data: $u_0$ is the source code of a uncompilable program.
Data: $c_0$ is the source code of a compilable program.
1  Function Reduce($u_0$, $c_0$)
2      $u_1 \leftarrow$ Format($u_0$)                    // 1 token per line.
3      $c_1 \leftarrow$ Format($c_0$)                    // 1 token per line.
4      ($u_2$, $c_2$) $\leftarrow$ Align($u_1$, $c_1$)    // See description.
5      $\Delta_0^R \leftarrow$ Diff($c_2$, $u_2$)         // Breaking mutation.
6      $c_3 \leftarrow$ DeltaDebug($c_2$, $\Delta_0^R$)
7      $u_3 \leftarrow$ Patch($c_3$, $\Delta_0^R$)
8      $u_4 \leftarrow$ PrettyPrint($u_3$)
9      $c_4 \leftarrow$ PrettyPrint($c_3$)
10     return ($u_4$, $c_4$)                         // Reduced repair example.
```

triggers the same error message $e$. This could result in a smaller $u$. A problem with this approach is that this does not simultaneously reduce $c$. Also, the new diff $\Delta'$, the difference between the reduced $u'$ and the original $c$, $\Delta'$ may represent a drastically different repair than the original $\Delta$. There are other well-known methods for reducing programs. Hierarchical Delta Debugging [Misherghi and Su, 2006] is a faster reduction algorithm for dealing with tree-structure inputs like XML and program parse trees. CReduce [Regehr et al., 2012] is a tool specialized for C and C++ programs.

To reduce a compilation repair example, we introduce a novel reduction algorithm outlined in Algorithm 2. The key ideas are to align the pair of programs first, and to use a specialized version of DD. We specialize DD in ways. First, we define the chunk-level to be tokens not characters. Second, we define a different testing function which DD will use to test if a chunk can be removed. With respect to a compilation repair example $(u, c)$, let $c'$ be the reduced version of $c$ in each iteration of DD. The test succeeds if and only if:

- $c'$ compiles successfully, and

- $u'$, the result of applying the (reverse) patch $\Delta^R$ to $c'$, fails to compile with the same error message $e$ as $u$.

Algorithm 2 takes a compilation repair example as input (line 1). It formats each program so that each program token is on one line (line 2-3). Then, it aligns the programs (line 4). The basic idea of alignment is that if $u_1$ is shorter than $c_1$ than you need to insert padding space into $u_1$ that will make it as long as $c_1$. And if $u_1$ is longer than $c_1$, than you need to insert padding space into $c_1$ to make it as long as $u_1$. Once aligned, we can compute a patch (diff) $\Delta_0$ from $u_2$ to $c_2$ (line 5). We compute a reverse patch $\Delta^R$ (line 6). We apply the specialized DD algorithm (line 7), which we described above. We apply $\Delta^R$ to the reduced compilable program $c'$ to get a smaller uncompilable program $u'$ (line 8), which, by definition, of the specialized DD should still trigger the same error message $e$. Lastly, we pretty print in a standard style so that users can read it (lines 9-10).

The example shown in Figure 3.4 is a reduced version of the example shown in Figure 3.3. The new patch $\Delta'$ is the same (except for shifted column position), but, clearly, it is much easier to understand the surrounding context.

## 3.4 Online Search

By sampling the repair search space ahead of time, the list of candidate repairs has been reduced. However, a single error message may contain tens, hundreds, even thousands of repair examples. This leads to the *information retrieval problem* of retrieving and ranking compilation repair examples. In this section, we describe CompAssist's search engine, outlined in Figure 3.2. As a *query*, the search engine is given 1) an arbitrary user program, and 2) a compiler name. The search engine compiles the program using the specified compiler. If compilation succeeds, the search engine returns an empty result. Otherwise, it proceeds to retrieve and rank related example fixes.

**Retrieval:** For an uncompilable program that triggers some error message $e$, we say that *related compiler repair examples* are those that trigger the same error message $e$ or the same *kind of error*. The complete set of possible kinds of errors and error message templates can usually be found in the source code of the compiler. In practice, the set can be inferred and approximated from observed error messages. Given an error message $e$, we map it to its kind, and we retrieve all repair examples of the same kind. For example, if $e$ is "`use of undeclared 'x'`", then it matches "`use of undeclared %0`" so we retrieve all example fixes for `err_undeclared_var_use` which includes examples for "`use of undeclared 'y'`".

**Patch Inference:** In general, a patch is just the difference between two texts. And it is difficult to infer the high-level syntactic transformations from a low-level patch [D'Antoni et al., 2017, Long and Rinard, 2016, Long et al., 2017, Rolim et al., 2017]. Since we synthesize the repair examples, we obviate this problem. For each repair example, we know the exact high-level patch (or transformation). This is due to the fact that, in generating example fixes, we fuzz at a "high-level" representation of the source code. For example, in Figure 3.1, we show example fixes grouped by patch. The first group of example fixes contains example fixes where the patch is "Insert public". To support user-contributed example fixes, we would have to implement patch inference.

**Ranking:** We aim 1) to present plausible and "acceptable" possible patches first, and 2) to present repair examples most similar to the user program first in each patch group. First, we score each of the related repair examples. With input user program $u_{user}$, to score a repair example $(u, c)$, we compute the overlap coefficient [Manning and Schütze, 1999] and subtract the Levenshtein distance [Jurafsky and Martin, 2009] between $u_{user}$ and $u$.

$$score(u, u_{user}) = overlap(u, u_{user}) - lev\_dist(u, u_{user}) \tag{3.1}$$

Second, we assign a preliminary score to each patch. Let $RE_p$ denote a group of repair examples that exhibit patch $p$. To score a patch $p$ with respect to a user program $u_{user}$, we assign it the maximum score of its corresponding examples $RE_p$.

$$score(p, u_{user}) = max(\{score(u, u_{user}) \mid (u, c) \in RE_p\}) \tag{3.2}$$

At this stage, the search engine returns a ranked list of patches where each patch has a ranked list of example fixes. This helps ensure that the time-to-interaction (TTI) is low.

**Auto-experimentation:** We test each patch on the user program using brute-force. It updates the score of each patch if any experiments resulted in a successful compilation. Based on pilot studies, auto-experimentation improves the ranking of the patch suggestions. For example, suppose a possible patch is "Insert comma" (ignoring location). Then, we apply this patch to every possible location of the tokenized user program checking if the patched program compiles. We record experiments that result in successful compilations. "Delete TOKEN" transformations are easier to experiment with because we only have to delete the instances of the token type indicated by the patch. For example, if a patch is "Delete amp", then we only have to delete the "&" instances of the user program. If the user program does not have any "&", then no experimentation is needed and the score remains the same.

## 3.5 Evaluation

We evaluated CompAssist in terms of compiler error message coverage, and repair example simplicity. As a proxy for simplicity, we used program size. Specifically, we investigated the following research questions:

- **RQ1:** What proportion of compiler errors are covered by the repair examples?
- **RQ2:** Do we generate a diverse collection of repair examples for each compiler error?
- **RQ3:** Are the generated compiler repair examples simple?

### 3.5.1 Experimental Setup

We focused on a mainstream C++ compiler (`clang++-3.9`). As the corpus of seed programs $\mathcal{P}$, we used the GNU GCC test suite. The programs in this corpus are designed to test how the compiler handles all sorts of programming language features. We used a subset of $25,240$ compilable programs for C and C++. We ran Algorithm 1 for 10 days. We executed the compiler, with a timeout of 30 seconds, using the following command:

```
$ clang++-3.9 -c -Wfatal-errors -stdlib=libc++ -std=c++14
```

Table 3.1. Error Message Coverage.

| Component | Repair Examples | Covered Errors | Total Errors | Coverage |
|---|---|---|---|---|
| Lex | 925 | 4 | 105 | 3.9% |
| Parse | 27,505 | 140 | 216 | 64.8% |
| Sema | 90,955 | 734 | 1374 | 53.4% |
| CodeGen | 5 | 2 | 12 | 16.7% |
| All | 116,019 | 867 | 1686 | 51.4% |

We inspected Clang's source code to obtain a complete list of kinds of errors and their corresponding templates. We counted at least $4,092$ distinct diagnostic messages, including errors, warnings, notes, remarks, and other type of messages. From these, we identified $1,686$ kinds of error that could potentially be triggered by compiling `C` and `C++` programs.

### 3.5.2 Error Message Coverage (Breadth)

We generated $116,019$ repair examples which, combined, triggered a total of $39,835$ distinct error messages. We classified each error message by its corresponding kind of error. Table 3.1 shows that the repair examples cover 867 (51.4%) out of the $1,686$ different kinds of `C++` error messages in Clang. Table 3.1 also shows coverage of Clang's four major compilation components. The repair examples trigger 53% and 65% of the possible error messages in Parse and Sema, respectively. However, the coverage is much lower for Lex and CodeGen, 4% and 17%, respectively.

*Lex* handles preprocessing and tokenization of the input source file. *Parse* and *Sema* handle parsing, translating preprocessor tokens into a parse tree, and semantic analysis of the parse tree. They determine if the code is well-formed. Most compiler error messages are triggered from these two components. *CodeGen* translates the abstract syntax tree (AST) built by Parse and Sema into LLVM IR, optimizes the IR, and generates assembly code. More details can be found in Clang's documentation.[3]

[3]`https://clang.llvm.org/docs/`

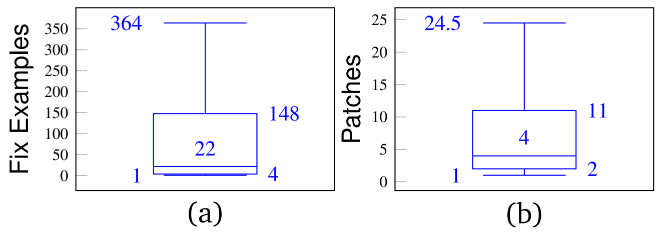


Figure 3.5. Error Message Coverage (Depth): (a) The distribution of repair examples shows that we generate multiple distinct examples for each error. (b) Similarly, the distribution of patches shows that we generate multiple distinct patches for each error.

To categorize errors by component, we searched the `.cpp` files of each component for instances of the error kind. For example, we searched all files for instances of the `err_undeclared_var_use` and found that it occurs in 5 files of the *Sema* component. We show how many error kinds are potentially triggered in each component in the "Total Errors" column of Table 3.1.

**Answer to RQ1:** The repair examples generated by CompAssist cover 867 out of 1,686 (51.4%) different kinds of C++ error messages in Clang.

## 3.5.3 Error Message Coverage (Depth)

When we imported repair examples to our PostgreSQL database, we identified and filtered *duplicate* examples. Since we are using a token-level fuzzer, we used the following definition:

**Definition 2 (Compilation Repair Example Equivalence)** *Two repair examples,* $(u_1, c_1)$ *and* $(u_2, c_2)$ *are equivalent if* $u_1$ *and* $u_2$ *trigger the same error message, the token sequences are identical for* $u_1$ *and* $u_2$*, and the token sequences are identical for* $c_1$ *and* $c_2$*.*

Figure 3.5 shows the distribution of repair examples for each kind of error. The median number is 22. We generated at least two examples for 777 out of 867 (89.6%) Table 3.2 shows the ten errors with the most examples. `err_init_conversion_failed` is the error with the most examples, 3,631. Interestingly, all of the top ten errors belong in the $Sema$ component.

For each error, we also counted the number of distinct patches illustrated by its re-

Table 3.2. Errors with the most example fixes.

| Error | Component(s) | Repair Examples |
|---|---|---|
| err_init_conversion_failed | Sema | 3631 |
| err_typecheck_nonviable_condition | Sema | 1644 |
| err_no_member_suggest | Sema | 1449 |
| err_bound_member_function | Sema | 1408 |
| err_member_decl_does_not_match | Sema | 1250 |
| err_typecheck_convert_incompatible | Sema | 1241 |
| err_template_arg_list_different_arity | Sema | 1076 |
| err_template_decl_ref | Sema | 1068 |
| err_unknown_type_or_class_name_suggest | Sema | 967 |
| err_no_member | Sema | 967 |
| err_ovl_no_viable_function_in_init | Sema | 962 |

pair examples. Figure 3.5 shows the distribution of patch counts. The median number of patches is 4. We generated at least two distinct patches for 692 out of 867 (79.8%). Table 3.3 shows the top ten errors ranked by number of patches. The error with the highest number of patches is `err_init_conversion_failed` with 101 distinct patches. In this case, the top ten errors are not exclusively from *Sema* but also from *Parse*. `err_expected` is also potentially triggered by the *Lex* component.

**Answer to RQ2:** For each error, CompAssist generates a diverse collection of repair examples. It generated at least 2 different repair examples for 89.6% of errors. Also, for 79.8% of errors, it generated at least 2 distinct patches.

## 3.5.4 Repair Example Simplicity

Recall that seed programs can be of arbitrary size and complexity but the repair examples should be simple for users to comprehend. As a proxy for simplicity (or complexity), we use the size (or length) of the uncompilable program, $|u|$, of the repair example. Cyclomatic complexity [McCabe, 1976] and similar metrics [Kasto and Whalley, 2013] are

Table 3.3. Errors with the most patches.

| Error | Component(s) | Patches |
|---|---|---|
| err_init_conversion_failed | Sema | 101 |
| err_expected_unqualified_id | Parse | 96 |
| err_expected_expression | Parse | 87 |
| err_typecheck_invalid_operands | Sema | 74 |
| err_expected | Parse, Lex | 74 |
| err_typecheck_nonviable_condition | Sema | 73 |
| err_bad_variable_name | Sema | 70 |
| err_expected_semi_after_stmt | Parse | 68 |
| err_expected_either | Parse, Sema | 67 |
| err_expected_semi_declaration | Parse | 63 |

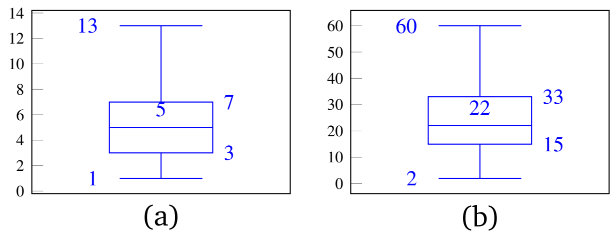


Figure 3.6. Repair Example Sizes (a) The median LOC length of repair examples is 5. (b) The median token length is 4. These distributions tell us that the generated repair examples are small, and by proxy, simple. Note: Outliers are not shown.

more precise measures but size tends to be a good proxy.

We formatted the repair examples using `clang-format-3.9` and the Chromium style.[4] Then we counted the lines of code (LOC), and measured the token sequence length, tokenized using:

```
$ clang++-3.9 -cc1 -dump-tokens -stdlib=libc++ -std=c++14
    -Wfatal-errors
```

Figure 3.6 shows the distribution of sizes by LOC and token sequence length. In summary, 59.6% (69,418) repair examples have five or fewer LOC. While 6.3% (7,302)

[4]https://clang.llvm.org/docs/ClangFormat.html

examples consists of just one LOC. We counted 668, 604 LOC and 3, 257, 307 tokens in total. This means that, on average, each LOC has 4.87 tokens. A majority of examples, 60.75%, have 25 or fewer tokens. There are a few outliers with as many as 209 LOC and 1, 041 tokens.

**Answer to RQ3:** COMPASSIST generates small example fixes. 59.6% have five or fewer lines of code.

## 3.6 User Study

In designing this study, we follow the advice by Ko *et al.* [Ko et al., 2015] and exemplified well by Böhme *et al.* [Böhme et al., 2017]. We recruited participants and asked them to resolve compiler errors of 9 C++ programs using the COMPASSIST web application. Specifically, we investigate the following research question:

- **RQ4:** Do programmers find the generated compilation repair examples helpful in resolving compiler errors of real-world C++ programs?

### 3.6.1 Pilot Studies

We conducted pilot studies to test the study design and tool implementation. We recruited 6 undergraduate students from a *Data Structures* university course where the instruction programming language is C++. We also posted flyers in our department building. We compensated each participant with a $15 Amazon gift card. We learned the following lessons:

- Limit Task Scope: Since our implementation generates repair examples illustrating single-operation, single-token patches, we limit tasks to C++ snippets that require similar patches.

- Clarify Training: Initially, students felt obligated to read and understand the presented example fixes. We clarified examples fixes are additional help.

- Infrastructure: Pilot testers suggested improvements to the interface. We decided

Table 3.4. C++ Programs Used in Debugging Tasks.

| Program | Error Message |
|---|---|
| T1 | expected ';' after struct |
| T3 | use of undeclared identifier 'height'; did you mean 'heigh'? |
| T4 | static_cast from 'unsigned int *' to 'int' is not allowed |
| T5 | expected unqualified-id |
| T6 | type name does not allow constexpr specifier to be specified |
| T7 | indirection requires pointer operand ('size_type[TRUNCATED] |
| T8 | typedef redefinition with different types ('std::[TRUNCATED] |
| T9 | constexpr variable 'res_foo' must be initialized[TRUNCATED] |
| T10 | cannot cast 'Cat' to its private base class 'Animal' |

to not record the user session because the video did not provide much additional information.

## 3.6.2 Design

### 3.6.2.1 Objects and Infrastructure

We searched for real-world examples of C++ programs where users asked online for help with compiler errors. To search, we randomly sampled error messages triggered by our example fixes. For each message, we copy-pasted the message into Google. We reviewed each web page and collected a snippet that triggered that specific diagnostic message. Though our approach should generalize, we focused mainly on those snippets that can be fixed with single-operation, single-token patches due to our current implementation. Like Barik *et al.* [Barik et al., 2017], we aimed to constraint the study to one hour or less. We selected 9 programs, listed in Table 3.4.

To conduct the study remotely, we built a mini application in COMPASSIST that presents 1) the consent form, 2) the demographics questionnaire, 3) a tutorial video, 4) the list of tasks and task instructions, 5) individual tasks, 6) task questionnaires, and a 7) final questionnaire.

#### 3.6.2.2 Participants and Training

To recruit participants, we distributed flyers in a university algorithms course, via email to CS students, and on the Facebook page of the university's CS club. In one week, we received over 20 responses. We selected 14 participants who had some C++ experience, *e.g.*, one of the two C++ courses taught at our university. Out of the 14, 8 were undergraduate students; 3 had a Bachelor's degree; 2 were graduate students; 1 had a Ph.D.; 2 self-identified as female. Participants reported a median 2.5 years of C++ experience, ranging from 0 to 8. All participants read and signed an online consent form which provided an overview of the study along with basic instructions.

Participants were asked to watch a tutorial video, 2 min 30 secs long, that described COMPASSIST's user interface: the code editor, compiler output, and list of possible patches.

#### 3.6.2.3 Tasks

We instructed participants to 1) find and fix the code defect, and 2) submit their solution for each of the 10 C++ programs. As constraints, we allowed 5 minutes and disallowed internet or external help, since the programs and solutions were found online. Submissions after 5 were marked as *timeout*. After each task, we asked participants:

*Q1) How helpful was the compiler error message?*

*Q2) How helpful were the repair examples?*

We defined the following 4-point Likert-type scale:

| | | | |
|---|---|---|---|
| 0. Not at all helpful | 1. Somewhat helpful | 2. Helpful | 3. Very helpful |

#### 3.6.2.4 Debriefing

Since we performed the study online, we debriefed via email. We identified the defect in each program, and explained the solution.

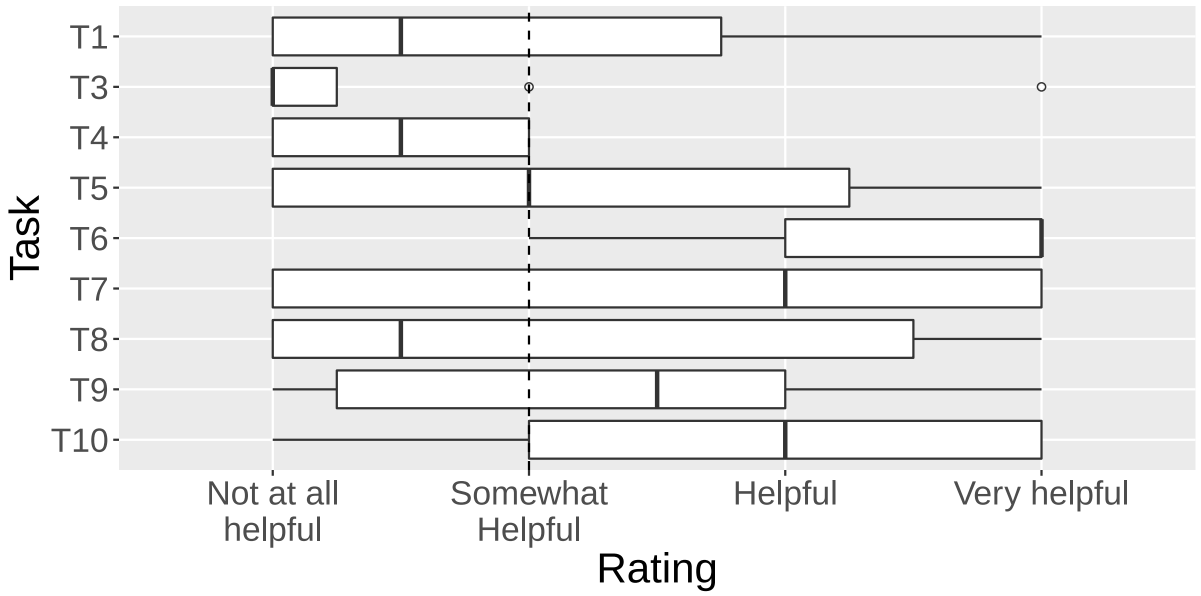


Figure 3.7. After each task, participants rated the helpfulness of repair examples. Participants rated examples as helpful (“Somewhat helpful” or above) in 5 out of 9 tasks.

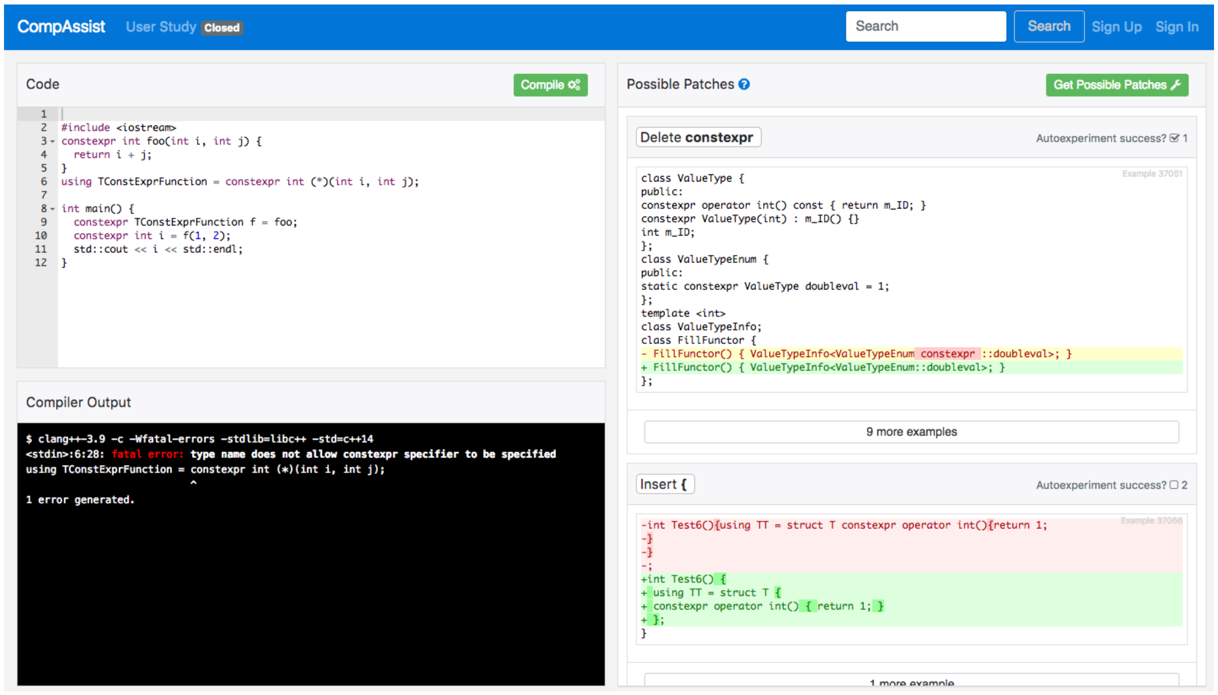
### 3.6.3 Results

Figure 3.7 shows a summary of the helpfulness ratings. Based on the median rating, participants found the example fixes “Somewhat helpful” or better for 5 out of the 9 tasks , T5, T6, T7, T9, and T10.

As expected, we observed a negative correlation between the helpfulness of compiler error messages and repair examples. When participants rated compiler error messages as “Very helpful”, they rated repair examples as less than “Somewhat helpful”. In contrast, when they found example fixes to be “Very helpful”, they rated error messages to be least helpful, between “Somewhat helpful” and “Helpful”.

Participants rated the repair examples as “Very Helpful” for T6, shown in Figure 3.8, giving it a median “Very helpful” rating. In contrast, participants rated the error message as “Somewhat helpful”. In the source web document, the patch suggested to resolve the compilation error is “Delete constexpr from line 5”.[5] Our tool shows the same patch (minus the location) as the first suggestion, and shows that auto-experimentation was

[5]https://stackoverflow.com/questions/37993732/is-it-possible-to-define-type-alias-to-constexpr-function



```

#include <iostream>
constexpr int foo(int i, int j) {
  return i + j;
}
using TConstExprFunction = constexpr int (*)(int i, int j);

int main() {
  constexpr TConstExprFunction f = foo;
  constexpr int i = f(1, 2);
  std::cout << i << std::endl;
}
```

```
$ clang++-3.9 -c -Wfatal-errors -stdlib=libc++ -std=c++14
<stdin>:6:28: fatal error: type name does not allow constexpr specifier to be specified
using TConstExprFunction = constexpr int (*)(int i, int j);
                           ^

1 error generated.
```

```
class ValueType {
public:
constexpr operator int() const { return m_ID; }
constexpr ValueType(int) : m_ID() {}
int m_ID;
};
class ValueTypeEnum {
public:
static constexpr ValueType doubleval = 1;
};
template <int>
class ValueTypeInfo;
class FillFunctor {
- FillFunctor() { ValueTypeInfo<ValueTypeEnum constexpr ::doubleval>; }
+ FillFunctor() { ValueTypeInfo<ValueTypeEnum::doubleval>; }
};
```

```
-int Test6(){using TT = struct T constexpr operator int(){return 1;
-}
-}
-;
+int Test6() {
+ using TT = struct T {
+ constexpr operator int() { return 1; }
+ };
}
```

Figure 3.8. This is C++ program T6. Participants rated the repair examples show for this program as “Very helpful”.

successful. All participants, except one, applied this patch.

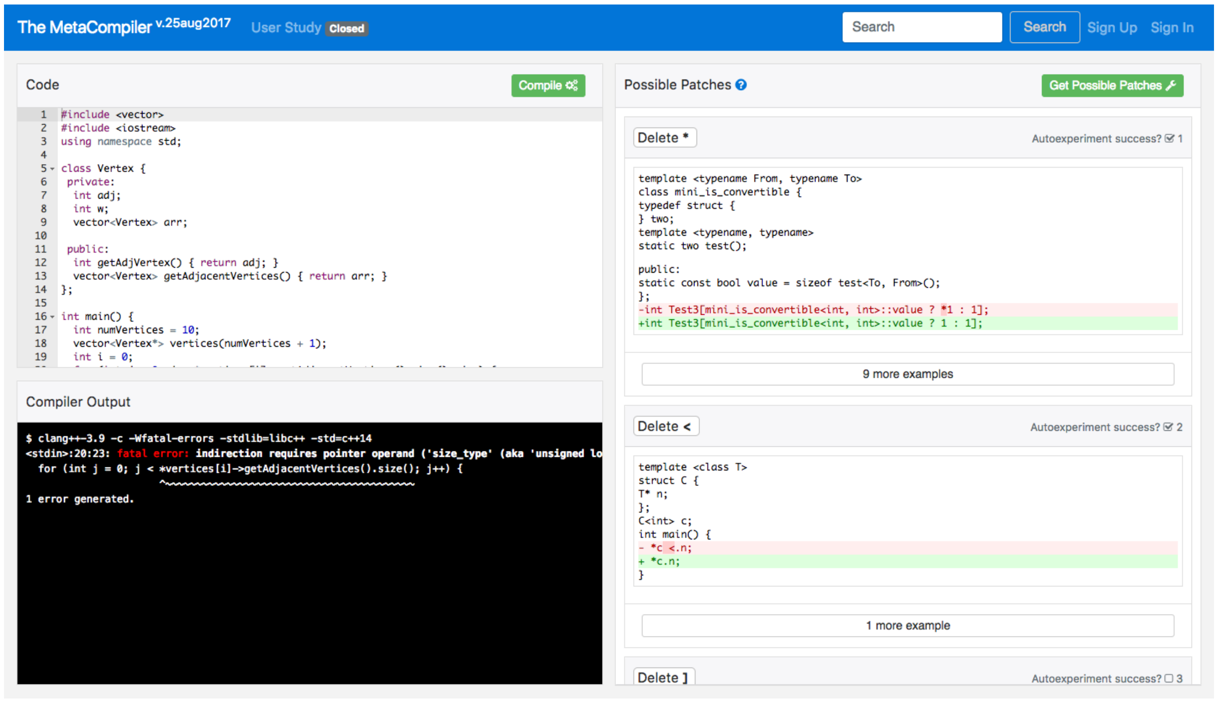
Participants rated the repair examples as “Helpful” for T10, shown in Figure 3.1. As we discussed in Section 3.2, our tool also suggests a patch, “Insert public”, that resolves the compiler error.

We reviewed the reasons provided by participants as to why the found repair examples and the suggested patches helpful. We categorized their responses into three major reasons:

1. *Repair examples illustrated patches that resolved the compiler error.* As P12 put it, “the ones they got right they were perfectly good”

2. *Repair examples helped with unfamiliar C++ concepts.* P1 said, “Suggested patches was helpful for error messages to concepts that I am unfamiliar with.” P8 said, “The suggested patches and examples were a good refresher and helped save me a lot of time.”

3. *Repair examples suggested possible actions.* P2 said, “repair examples some possible



```
#include <vector>
#include <iostream>
using namespace std;

class Vertex {
 private:
  int adj;
  int w;
  vector<Vertex> arr;

 public:
  int getAdjVertex() { return adj; }
  vector<Vertex> getAdjacentVertices() { return arr; }
};

int main() {
  int numVertices = 10;
  vector<Vertex*> vertices(numVertices + 1);
  int i = 0;
```

```
$ clang++-3.9 -c -Wfatal-errors -stdlib=libc++ -std=c++14
<stdin>:20:23: fatal error: indirection requires pointer operand ('size_type' (aka 'unsigned lo
  for (int j = 0; j < *vertices[i]->getAdjacentVertices().size(); j++) {
                      ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

1 error generated.
```

```
template <typename From, typename To>
class mini_is_convertible {
typedef struct {
} two;
template <typename, typename>
static two test();

public:
static const bool value = sizeof test<To, From>();
};
-int Test3[mini_is_convertible<int, int>::value ? *1 : 1];
+int Test3[mini_is_convertible<int, int>::value ? 1 : 1];
```

```
template <class T>
struct C {
T* n;
};
C<int> c;
int main() {
- *c <.n;
+ *c.n;
}
```

Figure 3.9. This is C++ program T7. Participants rated the repair examples show for this program as "Helpful".

moves you can make from the current state of the program." P11 said, "repair examples told me exactly what to do to fix the compiler error even though I had no idea what was going on."

Repair examples are intended to complement the compiler output not substitute it. In this study, we either presented the compiler output and example fixes, or just the compiler output. Ideally, example fixes would be shown to a user after he/she 1) reads the error message(s), and 2) fails to resolve the error. In such a scenario, we believe example fixes can only help a programmer. We need to collect more data to investigate this effect.

#### 3.6.3.1 User Performance

We graded submissions as correct, incorrect, or timeout. Table 3.5 shows an overview of task performance with respect to correctness. Overall, participants performed better with repair examples, 93%, than without, 84%; a difference of 11%. In 4 out of 9 tasks, T1, T4, T9, and T10, participants performed better with repair examples. However, in 3 tasks, T5, T6, and T8, participants performed better without repair examples. In 2 out

Table 3.5. Overview of Task Performance.

| Task | $n_c + n_e$ | Control |  |  |  |  | Experimental |  |  |  |  |
|---|---|---|---|---|---|---|---|---|---|---|---|
|  |  | $n_c$ | C | C% | I | T | $n_e$ | C | C% | I | T |
| T1 | 14 | 4 | 3 | 75% | 0 | 1 | 10 | 10 | 100% | 0 | 0 |
| T3 | 14 | 4 | 4 | 100% | 0 | 0 | 10 | 10 | 100% | 0 | 0 |
| T4 | 14 | 10 | 6 | 60% | 2 | 2 | 4 | 3 | 75% | 0 | 1 |
| T5 | 14 | 6 | 6 | 100% | 0 | 0 | 8 | 7 | 88% | 0 | 1 |
| T6 | 14 | 4 | 4 | 100% | 0 | 0 | 10 | 9 | 90% | 1 | 0 |
| T7 | 14 | 7 | 7 | 100% | 0 | 0 | 7 | 7 | 100% | 0 | 0 |
| T8 | 14 | 8 | 8 | 100% | 0 | 0 | 6 | 5 | 83% | 1 | 0 |
| T9 | 14 | 8 | 6 | 75% | 0 | 2 | 6 | 6 | 100% | 0 | 0 |
| T10 | 14 | 5 | 3 | 60% | 1 | 1 | 9 | 8 | 89% | 0 | 1 |
| All | 140 | 64 | 54 | 84% | 4 | 6 | 76 | 71 | 93% | 2 | 3 |

C = Correct, I = Incorrect, T = Timeout

of 9, T3 and T7, there was no difference.

We also recorded the start time and submission time for each task. Figure 3.10 shows the distribution of time-to-submission. Participants were not instructed to submit as early as possible. The variance across tasks is clearly visible. Overall, participants submitted a solution faster when shown repair examples; median of 48 secs compared to a median of 64 secs.

**Answer to RQ4:** Participants found repair examples helpful ("Somewhat helpful" or better) in resolving compiler errors of 5 out of 9 real-world C++ programs. Participants also performed better with repair examples with respect to correctness and time-to-submission.

### 3.6.4 Additional Feedback

Participants suggested two major improvements.

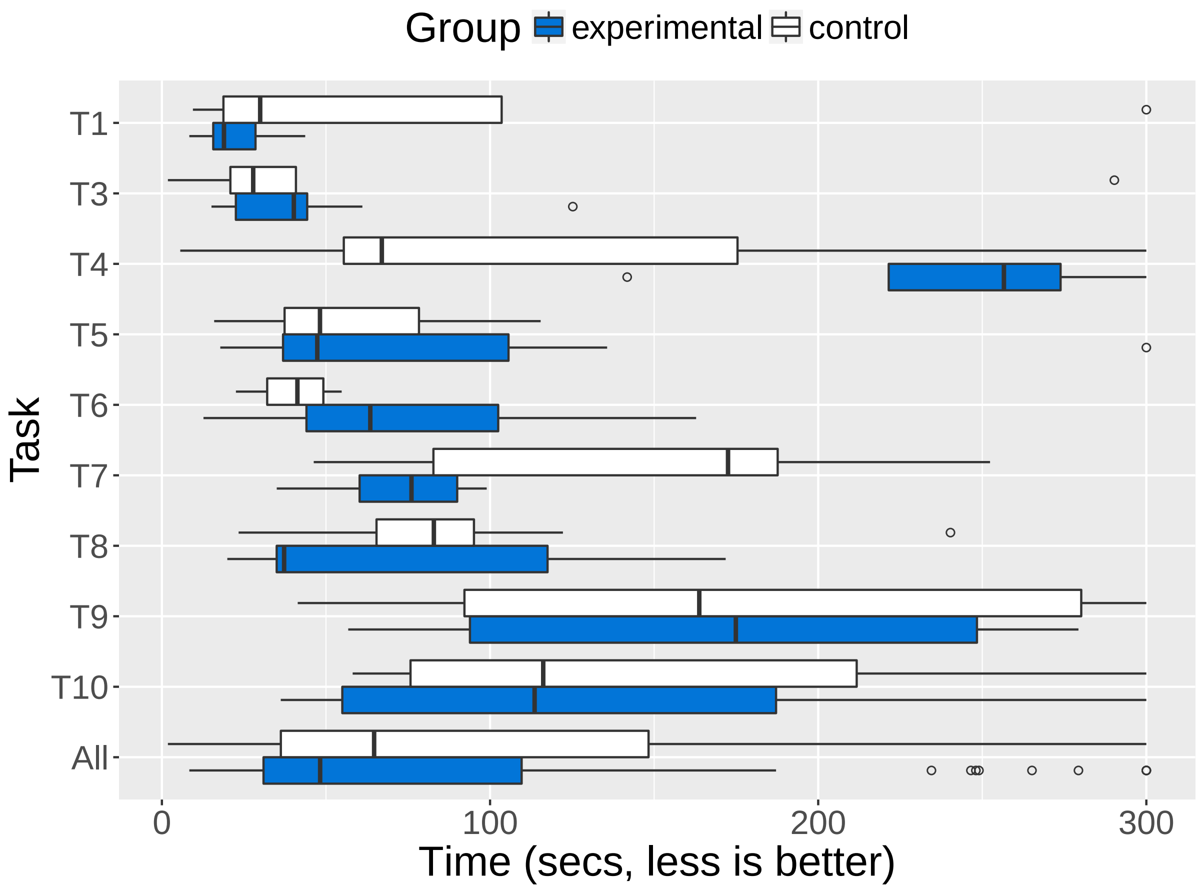


Figure 3.10. Time to Submission: Overall, participants submitted a solution faster when shown repair examples; median of 48 secs compared to a median of 64 secs.

1. Automatically apply patches where auto-experimentation succeeded to the user program. As P7 put it, "the tool should demonstrate the fix on the code snippet itself". Since this study, we have implemented this change.

2. The tool should provide explanations. As P11 put it, "the tool should provide an explanation as to why a change would fix the error." P12 said, "give better patch explanations". We plan on implementing a feature to allow users to provide English explanations

## 3.7 Discussion

**Generalizability:** Based on our quantitative and qualitative results, we believe our approach is effective and can be adapted to any mainstream compiler, *e.g.*, `javac` and gcc. Mainstream compilers emit a diverse set of error messages that are meaningfully partitioned. Our approach should still work for a hypothetical compiler that only emits "Compiler Error". Though, the generated repair examples will not be partitioned.

**Coverage:** Based on achieved coverage, it is fair to say that our implementation needs improvement. Fortunately, there are clear ways in which we can improve it. First, we can run the generation (precomputation) step for a longer time; we only ran it for 10 days. Second, we can use seeds from different sources, not just from the GCC test suite. This may increase the diversity of language features and idioms. Fourth, we can add more diverse mutations to our fuzzer, *e.g.*, AST mutations. One uncontrollable factor that may be affecting our measurements is the feasibility of triggering errors. We identified where error messages may be triggered but we did not determine if it is possible to trigger them.

**Fuzzer:** Based on our results, our fuzzer which only performs single-token, single-operation mutations is simple but profitable. Its simplicity reflects our intuition, likely shared by others, that compiler repairs are small (and simple). In conducting our user study, it was relatively easy to find real-world uncompilable programs which required single-token repairs. Our fuzzer also reflects our priority to explore the compiler error space, not the space of all possible mutations.

There are clear ways in which we can improve our fuzzer. First, we can support multi-token fuzzing. Theoretically, we should be able to trigger any error in this manner except lexer errors, though there are relatively few lexer errors (see Table 3.1). Perhaps more profitable, we can implement more complex mutations, like AST operations. For example, we can support randomly inserting or deleting entire program statements (or expressions) — consider the common case where you forget to declare a variable and have to go up in the code and add a declaration statement. Third, given a distribution of actual repairs, we can sample mutations non-uniformly. This could help ensure that our repairs are more natural.

**Bias:** A goal of our approach is to generate repair examples for *all* compiler errors. However, it is possible that the errors we trigger are those that people do not care about (unimportant). Search engine result count can be a good proxy for importance since when someone has trouble with an error message, he/she may post about it online. We re-scored search result counts from Bing. We observed no difference between the distributions of covered error messages and all error messages. Hence, we conclude that there is no apparent bias.

**Auto-experimentation:** In our current implementation, we use brute-force to test candidate repairs. Auto-experimentation depends on the number of candidate repairs and the size of the user program. Can we auto-experiment efficiently on an arbitrary program? We believe we can since 1) the number of possible repairs is, on average, small, and 2) the size of the user program can become irrelevant if restrict to a small region.

**Implicit Dialogue:** There is an implicit dialogue between COMPASSIST and the user. The user says, "This is my program and this is the error message." COMPASSIST says, "Here are some candidate repairs with examples of how you can apply it." The user reviews the suggested repairs and decides which one, if any, is applicable. With auto-experimentation, COMPASSIST also tells the user, "I have experimented with these patches and these repair your program." We can improve this dialogue by automatically aligning the user program to the repair examples. One potential method is to perform alpha-renaming of variable names, and perform approximate string matching.

## 3.8 Related Work

Numerous tools, techniques, and even programming languages have been developed to help programmers with compiler errors [McIver, 2000, Omar et al., 2017]. To the best of our knowledge, this is the first approach that pre-computes compilation repairs with minimal examples, and presents them to the user as candidate patches.

### 3.8.1 Compiler Error Augmentation

Early, compilation repair examples were created *manually*. Over a decade ago, Kummerfield *et al.* developed a "web-based reference guide" where "each error message is explained with examples highlighting the problem and at least one possible correction" [Kummerfeld and Kay, 2003]. They gave students 8 uncompilable programs and asked them to fix them using their tool. They found that the tool helped novice students perform similarly to expert students. Toomey [Toomey, 2011] developed a modified version of the BlueJ IDE that provided English advice on how to fix common errors which sometimes included snippets.

Recently, repair examples have been *crowd-sourced*. Hartmann *et al.* developed HelpMeOut, a plugin to a Java IDE, that crowd-sources the creation of "solutions" (compilation repair examples) to its users [Hartmann et al., 2010]. It is rather comprehensive in that it explores ranking repair examples, system that collects "successful solutions" from users, supports manually adding English explanations, and even voting. Mujumdar *et al.* presented Crowd::Debug, a tool that aimed to apply HelpMeOut's approach to dynamic programming languages, namely Ruby [Mujumdar et al., 2011].

The strength of crowd-sourcing is that the collected repair examples are "natural". Though, our user study suggests that *synthetic* examples are helpful as well. As an approach, crowd-sourcing has serious limitations. The collection needs to be bootstrapped to be useful to initial users. The collected examples may not cover the entire compiler space and may saturate certain errors. User adoption may also be low because people usually do not want to or are not allowed to share their code.

COMPASSIST differs in that repair examples are synthesized completely automatically via our FUZZ-AND-REDUCE approach. This allows it to cover a greater portion of the compiler error space, and to provide multiple examples for each compiler error. Since we create high-level patches, this allows us to not only augment compiler errors but to attempt automatic repair!

Our user study and the results of related tools support the hypothesis that good examples can help programmers. However, there is still ongoing debate regarding the effectiveness of augmenting compiler errors with repair examples [Denny et al., 2014, Pettit et al., 2017].

### 3.8.2 Automated Program Repair

Most well-known program repair techniques focus on logic errors while CompAssist deals with compilation errors. For example, recently, Yi *et al.* studied the feasibility of using automated program repair of in introductory programming assignments but focused solely on logic errors [Yi et al., 2017].

**Search-based:** Traditional search-based automated program repair techniques, like GenProg [Le Goues et al., 2012, Weimer et al., 2009] and Par [Kim et al., 2013], focus on repairing logic errors. They depend on a fitness function and a test suite to guide their search for a correct program. The intuition is that if a patch leads to a mutant that passes more test cases, then that direction should be further pursued. We are not aware of any extension of these techniques that successfully constructs a fitness function based primarily on compiler output and independent of a test suite. We believe the difficulty lies in the coarseness of the compiler output; a program either compiles or not. Worst, if a patch triggers a different error, then it is unclear if the new program is closer to being compilable.

**Model-based:** Learned models can help identify and fix compilation and logic defects. Campbell *et al.* trained an *n*-gram model over compilable Java programs, and evaluated it by locating errors in a synthetic corpus of uncompilable code [Campbell et al., 2014]. Long *et al.* developed Phropet, a system that learns a probabilistic model from successful patches. It uses this model to assign probabilities to candidate patches [Long and Rinard, 2016]. Pu *et al.* designed and trained seq2seq neural network model sequence on correct student programs and used it to correct syntactic and logic errors of incorrect programs [Pu et al., 2016]. Similarly, Gupta *et al.* trained a seq2seq model with attention with the design objective of being task-independent. TRACER is a recent effort that combines program analysis and deep learning to fix compiler errors in student programs [Ahmed et al., 2018].

**Rule-based:** Other approaches learn rules (or transformations) from repair examples. NoFAQ learns transformations rules from crowdsources examples of “buggy and repaired [shell] commands”. With these rules, it can repair “buggy [shell] commands" [D'Antoni et al., 2017]. Refazer is a similar technique that learns syntactic transformations from "input-output examples" [Rolim et al., 2017]. The learned transformations can be useful in debugging and refactoring tasks. Singh *et al.* introduced an approach that learns transformations expressed in a error modeling language [Singh et al., 2013].

CompAssist differs from most of the approaches described above in that it is not data-driven; all repair examples are synthesized via fuzzing. This leads to a different search strategy because the precomputation step vastly reduces the patch search space and enables quick auto-experimentation. In our current implementation, this space reduction means that we cannot always suggest a patch acceptable to the user. But we can always show possible patches with minimal examples, something that users found to be a helpful fallback.

## 3.9 Conclusion and Future Work

This chapter presented CompAssist, a system that generates example fixes for compiler errors offline, and, based on these example fixes, suggest possible patches to users when their program fails to compile. We also presented the Fuzz-and-Reduce technique underlying this system. More importantly, we have shown that is possible to automatically generate example fixes that can be used to augment compiler error messages and potentially for other applications.

As future work, we plan to support user-contributed fix examples. The challenge we foresee is how to infer high-level syntactic transformations [Rolim et al., 2017]. We also plan to investigate the possibility of automatically generating English explanations of the fix examples. Lastly, we plan to investigate the effectiveness of this approach on compilers and interpreters for other programming languages.

# Chapter 4

# On the Lexical Distinguishability of Source Code

Natural language is robust against noise. The meaning of many sentences survives the loss of words, sometimes many of them. Some words in a sentence, however, cannot be lost without changing the meaning of the sentence. We call these words "wheat" and the rest "chaff". The word "not" in the sentence "I do not like rain" is wheat and "do" is chaff. For human understanding of the purpose and behavior of *source code*, we hypothesize that the same holds. To quantify the extent to which we can separate code into "wheat" and "chaff", we study a large (100M LOC), diverse corpus of real-world projects in Java. Since methods represent natural, likely distinct units of code, we use the ~9M Java methods in the corpus to approximate a universe of "sentences." We extract their wheat by computing the function's *minimal distinguishing subset (*MINSET*)*. Our results confirm that functions contain much chaff. On average, MINSETS have 1.56 words (none exceeds 6) and comprise 4% of their methods. Beyond its intrinsic scientific interest, our work offers the first quantitative evidence for recent promising work on keyword-based programming and insight into how to develop a powerful, alternative programming model.

## 4.1 Introduction

A basic but strong assumption underlies many research and engineering efforts like code search, code completion, keyword programming, and natural programming: From a "small" subset of words, a system can find or generate a larger, executable piece of code.

This assumption is crucial in code search work. The body of work breaks the search problem into three sub-problems 1) how to store and index code [Bajracharya et al., 2006, McMillan et al., 2011], 2) what queries (and results) to support [Reiss, 2009a, Reiss, 2009b], and 3) how to filter and rank the results [Bajracharya et al., 2006, Mandelin et al., 2005, McMillan et al., 2012]. The person doing the search only has one concern: "*What should I type to find the code I want?*". Efforts focus on building better search engines not on determining to what extent this assumption holds.

This assumption is also critical in keyword and natural programming implementations [Little and Miller, 2007, Little et al., 2010, Le et al., 2013, Miller et al., 2008]. Almost a decade ago, Little *et al.* devised a keyword programming technique to translate keyword queries into valid Java expressions [Little and Miller, 2007]. Several tools and tools and techniques grouped under the general term of Sloppy Programming followed [Little et al., 2010, Miller et al., 2008]. These tools interpret keyword queries directly by first translating them into source code. SmartSynth [Le et al., 2013] is a much more recent incarnation. It generates automation scripts for smartphones from natural language queries. First, it uses natural language processing techniques to parse the queries. Then it applies program synthesis techniques to the parsing result to construct the scripts.

Our *vision* is to *generalize* current keyword programming systems into a new programming model where users "program" using a minimalist, universal programming language. The programmer should be to write down thoughts and not worry about syntax details.

Our idea to advance this vision is inspired by the observation that natural language is robust against noise. The meaning of many sentences survives the loss of words, sometimes many of them. In other words, the sentence or one similar can often be reconstructed given a few key words. We call these words "wheat" and the rest "chaff". We hypothesize that this intuitive observation about natural language also holds for programming languages:

**Wheat and Chaff Hypothesis:** Units of code consists of 1) "wheat", important lexical features that preserve meaning, and "chaff", and 2) the "wheat" is small compared to "chaff."

If we can *distill* source code into "wheat", perhaps, we can gain insights into how to *expand* "wheat" into source code and, thus, take a step toward realizing the new programming model we envision. In these terms, the programmer would write the "wheat" and the system would fill in the "chaff."

We call the phenomenon of distilling source code into a subset of lexical features that uniquely identifies it, *lexical distinguishability*. By studying lexical distinguishability, we are the first to provide quantitative and qualitative evidence that the Wheat and Chaff Hypothesis holds. The benefit of our approach is that we establish the existence of a "small" subset of words that uniquely *maps* to a larger, executable piece of code; thus, we provide evidence supporting the assumption underlying much work. The main limitation of our approach is that the "wheat" is artificial; it may not be what a human would use in applications like code search or keyword programming. We attempt to overcome this limitation.

We focus our study on a diverse corpus of real-world Java projects with 100M lines of code. The approximately 9M Java methods in the corpus form our universe of discourse as methods capture natural, likely distinct units of source code. Against this corpus, we compute a *minimal distinguishing subset (*MINSET*)* for each method. This MINSET is the wheat of the method and the rest is chaff. We represent each method as a bag-of-words. We develop an algorithm to compute their MINSETS. A lexicon is a set of words. Like web search queries, MINSETS are built from words in a lexicon. We run our algorithms over different lexicons, ranging from raw, unprocessed source tokens to various abstractions of those tokens, all in a quest to find a natural, expressive and meaningful lexicon that culminated in the discovery of a natural lexicon to use for queries (Section 4.4.3).

Our results show programs do indeed contain a great deal of chaff. Using the most concrete lexicon, formed over raw lexemes, MINSETS compose only 4% of their methods on average. This means that about 96% of code is chaff. While the ratios vary and can be large, MINSETS are always small, containing, on average, 1.56 words, and none exceeds 6. We observed the same trend over other lexicons. Detailed results are in Section 4.4. Section 4.6 also discusses existing and preliminary applications of our work. Our project web site (`http://jarvis.cs.ucdavis.edu/code_essence`) also contains more information on this work, and interested readers are invited to explore it.

Our main contributions follow:

- We define and formalize the MINSET problem for rigorously testing the Wheat and Chaff hypothesis (Section 4.2.4);
- We prove that MINSET is *NP-hard* and provide a greedy algorithm to solve it (Section 4.2.5);
- We validate our central hypothesis — *source code contains much chaff* — against a large (100M LOC), diverse corpus of real-world Java programs (Section 4.4); and
- We design and compare various lexicons to find one that is natural, expressive, and understandable (Section 4.4.3).

The rest of this chapter is organized as follows. In Section 4.2 we define lexical distinguishability of source code and explain how we study it. Section 4.3 describes our Java corpus, and implementations of the feature extractor and the MINSET algorithm. Section 4.4 presents our detailed quantitative and qualitative results. Section 4.5 analyzes our results and their implications. Section 4.7 places our work into the context of related work, and Section 4.8 concludes.

## 4.2 Problem Formulation

In this section, we describe how we determine if a piece of code is lexically distinguishable. We explain our representation of code. We also introduce several definitions including *distinguishing subset*, MINSET, and the MINSET problem. Finally, we present and discuss our MINSET algorithm.

### 4.2.1 Bag-of-Words Model

The first step in our formulation is to define the unit of code. One could choose units like individual statements, blocks, functions, or classes. In this study, we view functions as the units of code. This granularity seems adequate. Functions are natural, likely distinct, pieces of code and functionality. Functions are also reusable building blocks of more complex components.

We represent a unit of code, function, as a set of lexical features or bag-of-words. We disregard syntactic structure, order, and multiplicity. First, we parse each function to get its set of lexemes. A lexeme is a delimited string of characters in code, where space and punctuation are typical delimiters; it is an atomic syntactic unit in a programming language.[1] Then, we map each lexeme to a *word*.

***What is a "word"?*** A *word* is a lexeme, or some abstract or refined form of it. A *lexicon* is a set of words. For example, a natural, basic lexicon is the set of raw lexemes. Using this lexicon, the mapping of lexemes-to-words would be simple. Each lexeme would map to itself. The bag-of-words for each function would be its set of lexemes.

### 4.2.2 Lexicons

What is a word depends on the choice of the lexicon. The freedom to define the lexicon allows us to sharpen, blur, or even disregard certain lexical features.

New lexicons can be formed by abstraction over lexemes. In natural languages, for example, the words in a sentence can be replaced by their part of speech, like NOUN, VERB, or ADJECTIVE, to highlight phrase structure. Similarly, code parsers tag each lexeme with one of a set of token types. For example, the javac lexer defines 101 token types, for example, IDENTIFIER and INTLIT [Oracle, 2012]. This set of token types is another natural but clearly more abstract lexicon. Using this lexicon, we would map each lexeme to its token type. For example, "`3.14`" would map to INTLIT. A word would

[1]Linguistics defines a lexeme differently. A lexeme is the *set* of forms a single word can take. For example, 'run', 'runs', 'running' are all forms of the same lexeme identified by the word 'run'.

be one of the 101 token types. The bag-of-words for each function would be a subset of these 101 token types.

New lexicons can also be defined by filtering specific lexemes. For example, we can define a lexicon consisting of all lexemes except separators, like "`(`" and "`)`". Using this lexicon, we would map each lexeme to itself except separators. Separator lexemes would map to nothing. The bag-of-words for each function would be it's set of lexemes minus the separator lexemes.

***Homonyms*** Functions may contain, to adapt a word from linguistics, *homonyms*: identical lexemes with distinct effects on behavior. For example, in Java, the lexeme "`get`" could be a method call of "`java.util.Map.get()`" or "`java.util.List.get()`". In Java, we can fully qualify homonyms to distinguish them.

***Synonyms*** We can preserve lexical differences that we suspect capture differences in the behavior of a method by ensuring that different lexemes map to distinct words. We can also blur lexical differences by abstracting distinct lexemes we suspect have the same effect on behavior, *i.e. synonyms*, to the same word. For example, variable identifiers can be replaced with their type under a language's type system. In the top method shown in Figure 4.1, the parameter "`array`" could just as well have been named "`values`".

In general, a lexicon that is fine-grained and concrete may exaggerate unimportant differences between functions, while one that is coarse and abstract may blur important differences. Varying the lexicon allows us to explore programming language-specific information. The lexicon consisting of all lexemes probably includes many elements that distinguish but probably have little to do with the behavior of functions, *i.e.*, delimiters and string literals like `"Joe"`. We can filter those lexemes. We can also filter other lexemes, like the type annotation "`int`" in "`int cars = 0;`", to explore how distinguishing they are.

## 4.2.3 Illustration of the Bag-of-Words Model

Figure 4.1 shows two Java methods found in real-world projects, Apache Log4j and JMRI (A Java Model Railroad Interface), respectively. The first method sorts an array of

```java
private static void bubbleSort(int array[]) {
  int length = array.length;
  for (int i = 0; i < length; i++) {
    for (int j = 1; j > length - i; j++) {
      if (array[j-1] > array[j]) {
        int temp = array[j-1];
        array[j-1] = array[j];
        array[j] = temp;
      }
    }
  }
}

static void bubblesort(String[] values) {
  for (int i=0; i<=values.length-2; i++) {
    for (int j=values.length-2; j>=i; j--) {
      if (0 < values[j].compareTo(values[j+1])) {
        String temp = values[j];
        values[j] = values[j+1];
        values[j+1] = temp;
      }
    }
  }
}
```

Figure 4.1. This listing shows two Java methods. Both implement the BubbleSort algorithm. (top) Sorts an array of integers. (bottom) Sorts an array of strings.

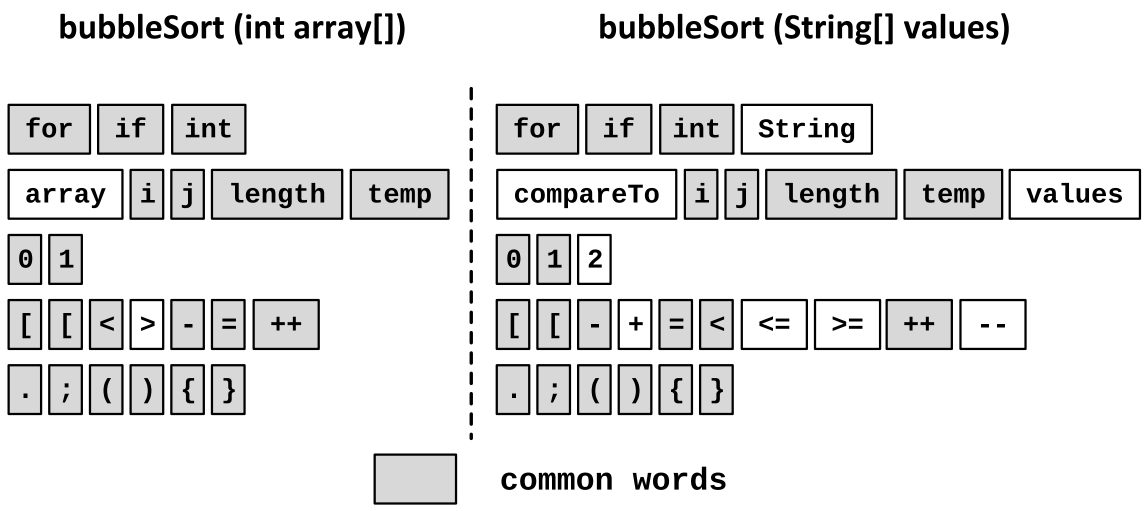


Figure 4.2. (left) This is the simplified bag-of-words representation of the method that sorts an array of integers using Bubble Sort. (right) This is the simplified bag-of-words representation of the method that sorts an array of Strings using Bubble Sort. Note: The words in common are shaded.

integers. The second one sorts an array of Strings. They both sort using the Bubble Sort algorithm.

Figure 4.2 shows their simplified representation as a bag-of-words. For this example, we have defined the lexicon to be all lexemes. Thus, the words are simply the raw lexemes. To help visualize the similarity between these two methods, we have shaded the words in common; there have 21 words in common. This should not be surprising. Both methods implement the same functionality. The main difference is that they operate over elements of different types, "`int`" and "`String`". By shading the common words, we also highlight the differences between these two methods. For example, the second method uses the "`--`" (decrement) operator in the second loop to iterate backwards.

## 4.2.4 Distinguishable Code

We simplified the representation of a function by mapping its source code to a set of lexical features, *bag-of-words*. Finding what distinguishes a function lexically is thus reduced to finding a unique subset of code features or words. This unique subset distinguishes each function from all other functions (when each functions is represented as a bag-of-words). We call any such subset a *distinguishing subset*, and define it precisely in Definition 4.2.1. A function may not have a distinguishing subset. We call those that do, *distinguishable* (Definition 4.2.2).

**Definition 4.2.1** *Given a finite set $S$, and a finite collection of finite sets $C$, $S^*$ is a* distinguishing subset *of $S$ if and only if:*

**P1** $S^* \subseteq S$, *$S^*$ is a subset of $S$*

**P2** $\forall C \in \mathcal{C},\ S^* \nsubseteq C$, *$S^*$ is* only *a subset of $S$*

**Definition 4.2.2** *A unit of code is* lexically distinguishable *if it has a distinguishing subset.*

***The* MINSET *problem*** A unit of code may have more than one distinguishing subset. To determine if it is distinguishable, we simply need to find one. We focus on finding a *minimum distinguishing subset* (MINSET). We call this The MINSET Problem (Definition 4.2.3). It is the *core* computational problem that we study.

**Definition 4.2.3 (The MINSET Problem)** *Given a finite set $S$, and a finite collection of finite sets $C$, find a* minimum *distinguishing subset (*minset*) $S^*$ of $S$.*

A MINSET identifies a piece of code. It consists of lexically distinguishing features. Some features may crucially differentiate its behavior from similar functions. Some may not. A MINSET, however, is not itself executable. It depends on its surrounding context to execute and provide functionality.

In the keyword-query sense, a MINSET is the smallest query that will uniquely identify and recall a piece of code. It may not be what humans would actually attempt to use. That is a separate challenge. In this study, we focus on finding and studying minsets.

**Theorem 4.2.1** MINSET *is NP-hard.*

**Proof 1** *We reduce* HITTING-SET *to* MINSET. *First, let us recall the* HITTING-SET *problem: Given a set $A$, a collection of subsets of $A$, $\{B_1, \ldots, B_m\}$, and a number $k$, does there exist a set $H \subseteq A$ of size $k$ such that $H \cap B_i \neq \emptyset$ for $1 \leq i \leq m$?*

*Now, let us define* MINSET *as a decision problem. Given a finite set of finite sets $C$, a finite target set $S$, and a number $p$, does there exist a set $S^* \subseteq S$ of size $p$ such that $S^*$ is not contained in any of the sets in $C$?*

*Consider an arbitrary instance $hs = \langle A, \{B_1, \ldots, B_m\}, k\rangle$ of the* HITTING-SET *problem. We construct an instance $ms$ of the* MINSET *problem with the following equations:*

1. $C = \{A \setminus B_1, \ldots A \setminus B_m\}$;
2. $S = A$: *and*
3. $p = k$.

*By definition, $hs$ is a "Yes" instance of the* HITTING-SET *problem if and only if there exists $H \subseteq A$ of size $k$ such that $H \cap B_i \neq \emptyset$, $1 \leq i \leq m$. Since $H \cap B_i \neq \emptyset$ if and only $H \nsubseteq A \setminus B_i$, when $H \subseteq A$ and $B_i \subseteq A$, a hitting set $H$ exists if and only if there exists $H \subseteq A$ of size $k$ such that $H \nsubseteq A \setminus B_i$, $i \leq i \leq m$. By definition, such a* MINSET *$H$ exists if and only if $ms$ is a "Yes" instance of the* MINSET *problem.* ■

### 4.2.5 The MINSET Algorithm

Since the MINSET problem is NP-hard, we present Algorithm 11, a greedy (approximation) algorithm that finds the locally minimal distinguishing subset of a set $S$. Given inputs $S$, the target set to be minimized, and $C$, a collection of sets against which $S$ is minimized, the MINSET algorithm computes $S^*$, and $C'$. $C'$ is the subset of $C$ whose sets contain $S$ so $C \setminus C'$ contains those sets in $C$ that do not contain $S$. When $C' = \emptyset$, $S^*$ is a subset of $S$ that distinguishes $S$ from all sets in $C$. The core of the algorithm is Line 4. Equality is needed in the cardinality test for cases like $S = \{a, b\}, C = \{\{a, x\}, \{a, y\}, \{b, x\}, \{b, y\}\}$, where all the elements in $S$ differentiate $S$ from the same number of sets in $C$. Equality also means that $C_x$ can be empty, as for $S = \{a\}$ and $C = \{\{x\}, \{y\}\}$, since $|C_a| \leq |C_a| = 0$, and $C_x$ can also be $C$ again, when $S \subseteq C, \forall C \in C$, as in $S = \{a\}$ and $C = \{\{a\}, \{a, b\}, \{a, b, c\}\}$.

Figure 4.3 shows a sample run of the algorithm. It ends in two iterations. It finds a MINSET of $\{a, b, e\}$ with respect to the collection $C = \{\{a, c\}, \{b, c, d\}, \{a, d, e\}\}$. The MINSET is $\{b, e\}$. None of the sets in $C$ contain this MINSET.

**Algorithm 3:** Given the universe $U$, the finite set $S$, and the finite set of finite sets $C$, MINSET has type $2^U \times 2^{2^U} \rightarrow 2^U \times 2^{2^U}$ and its application MINSET$(S, C)$ computes 1) $S^* \subset S$, a subset that distinguishes $S$ from sets in $C$, and 2) $C'$, a "remainder", *i.e.* a subset of $C$ whose sets contain $S$ and therefore from which $S$ could not be distinguished; when $C' = \emptyset$, $S^*$ distinguishes $S$ from all the sets in $C$'; when $C' = C$, $S^* = \emptyset$.

```
   Input: $S$, the set to minimize.
   Input: $C$, the collection of sets against which $S$ is minimized.
1  $C_e \leftarrow \{C \mid C \in C \wedge e \in C\}$ are those sets in $C$ that contain $e$.
2  $S^* \leftarrow \emptyset$
3  while $S \neq \emptyset \wedge C \neq \emptyset$ do
               // Greedily pick an element that most differentiates $S$.
4      $e :=$ CHOOSE$(\{x \in S \mid |C_x| \leq |C_y|, \forall y \in S\})$
5      if $C_e = \emptyset \vee C_e = C$ then
6          $C_e = \emptyset \vee C_e = C$
7          break
8      $S^* := S^* \cup \{e\}$
9      $S := S \setminus \{e\}$
10     $C := C_e$
11 return $S^*, C$
```

**Theorem 4.2.2** *Consider* MINSET$(S, C) = S^*, C'$. *The $S^*$ that Algorithm 11 computes distinguishes $S$ from a subset of $C$; when $C' = \emptyset$, $S^*$ is a locally minimal distinguishing subset of $S$.*

**Proof 2** *By induction on $S^*$.*

The worst case time complexity of MINSET$(S, C)$ is $O(|S|^2|C|)$. First, there are $|S|$ iterations and, in each call, for each element $x \in S$, we need to, 1) compute $C_x$, each at a cost of $|C|$, for a total cost of $O(|S||C|)$, then 2) then find the minimum $|C_x|$ at a cost of $O(|S|)$. Of course, $S$ and $C$ are smaller in each iteration, but we ignore this and

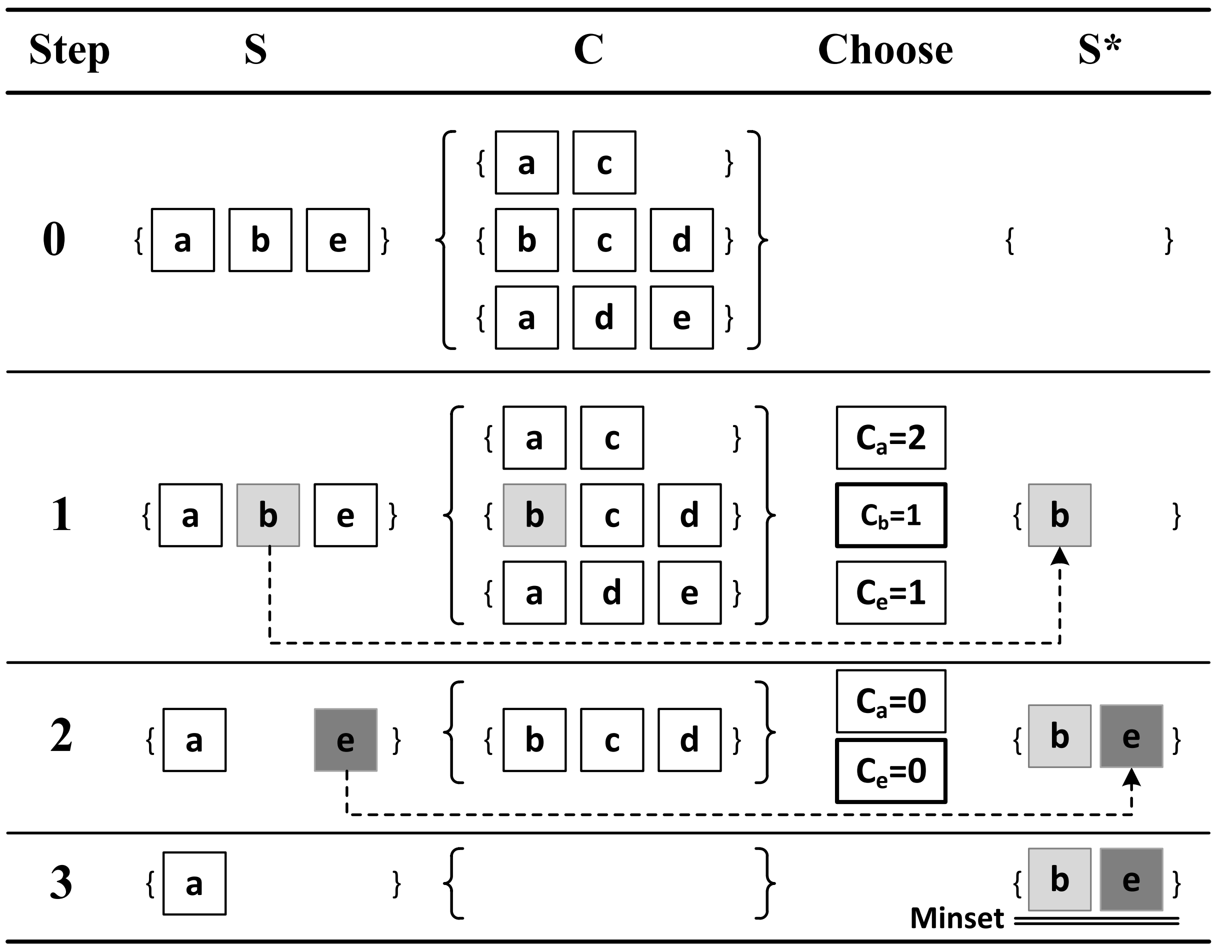


Figure 4.3. The execution of Algorithm 11 illustrated on the following problem instance: MINSET$(\{a, b, e\}, \{\{a, c\}, \{b, c, d\}, \{a, d, e\}\})$.

over-approximate. Thus, we have $O(|S|(|S||C| + |S|)) = O(|S|^2|C|)$.

As mentioned earlier, modeling functions as sets discards differences in methods due to multiplicity. We have also developed a multi-set version of the MINSET algorithm, which we omit due to lack of space.

## 4.3 Setup and Implementation

We selected a very popular, modern programming language, Java, and collected a large (100M lines of code), diverse corpus of real-world projects. Ignoring scaffolding and very simple methods, which we define as those containing fewer than 50 tokens, there are 1, 870, 905 distinct methods in our corpus. We selected a simple random sample of

Table 4.1. Corpus summary.

| Repository | Projects | Files | Lines of Code |
|---|---|---|---|
| Apache | 103 | 101,480 | 10,891,228 |
| Eclipse | 102 | 287,669 | 32,770,246 |
| Github | 170 | 133,793 | 13,752,295 |
| Sourceforge | 533 | 373,556 | 42,434,029 |
| **Total** | 908 | 896,498 | 99,847,798 |

$10,000$ methods[2]. Our software and data is available online[3].

### 4.3.1 Code Corpus

We downloaded almost one thousand of the most popular projects from four widely-used open source code repositories: Apache, Eclipse, Github, and Sourceforge.

***Curation*** Since some projects in our corpus are hosted in multiple code repositories, we removed all but the most recent copy of each project. Also, since many project folders contained earlier or alternative versions of the same project, and even other projects, where we could, we identified the main project and kept only its most current version. Table 4.1 summarizes our curated corpus. After curation, clones still existed in the corpus, for example, within projects. A search program we wrote helps us find clones. When we compute minsets, we assume no clones remain. Our results in Section 4.4.1 give us confidence that this is the case.

***Filtering Scaffolding Methods*** Java, in particular, requires that a programmer write many short scaffolding methods, for example, getters and setters. Many languages, like Ruby and Python, eliminate the need for such scaffolding code. After manual inspection, we found that such methods usually contain less than 50 tokens, or about 5 lines of code. This is consistent with other research [Li et al., 2004, Basit and Jarzabek, 2007] that also ignores shorter methods. At this size, we also filter methods with very simple

[2]Given the population size, this sample size gives us a confidence level of 95%, and a margin of error of ±1% in our measures.

[3]`https://bitbucket.org/martinvelez/code_essence_dev/downloads`.

Table 4.2. Method counts.

| Methods | Count |
|---|---|
| Total (in corpus) | 8,918,575 |
| Unique | 8,135,663 |
| Unique (50 or more tokens) | 1,870,905 |
| Unique (50 to 562 tokens) | 1,801,370 |

functionality. After filtering, 905 out of 908 projects are still represented. Table 4.2 shows the method counts.

### 4.3.2 The Feature Extractor

We developed a tool, which we call JavaFE, that processes all the functions in our corpus. JavaFE leverages the Eclipse JDT parser which parses Java code and builds the syntax tree[4]. JavaFE can take as input `.java`, `.class`, and `.jar` files. Projects can contain these and other types of files. The tool builds a list of tokens for each method. It collects the lexeme of each token and additional information as it traverses the syntax tree.

To address the *homonym* problem, JavaFE collects the fully qualified method name (FQMN) for method name lexemes, and the fully qualified type name (FQTN) for variable identifiers and type identifiers. Collecting this information allows us later to classify methods and types based whether they are part of the Java SDK library or if they are local to specific projects. When projects are missing dependencies, resolving names to either FQMN or FQTN may not be possible. In our corpus, we encountered this problem with 0.03% of the tokens. JavaFE can also collect more abstract information like lexer token types as defined in the `javac` implementation of OpenJDK, an open-source Java platform [Oracle, 2012].

[4]`http://www.eclipse.org/jdt/`.

Table 4.3. Lexicons.

| Name | Description | Size (words) |
|---|---|---|
| LEX | All (raw) lexemes | 5,611,561 |
| LTT | All lexer token types | 101 |
| MIN1 | Fully qualified standard library method names | 55,543 |
| | and basic operators | 55,543 |
| MIN2 | MIN1 plus control keywords | 55,556 |
| MIN3 | MIN2 plus fully qualified public type names | 91,816 |
| MIN4 | MIN3 plus additional keyword and token types | 91,829 |

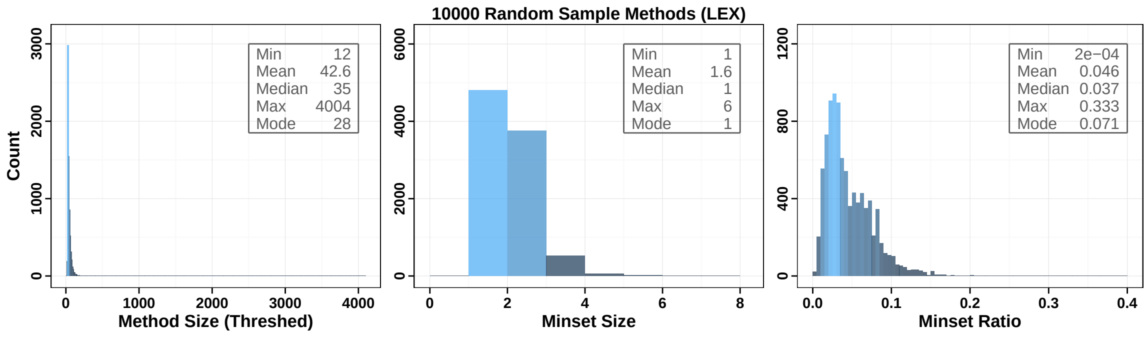


Figure 4.4. The histogram of minset sizes tells us that minsets are small. Comparing minset sizes with method sizes shows that minsets are also relatively small. The minset ratio histogram confirms this.

### 4.3.3 The MINSET Algorithm Implementation

All the information collected by JavaFE is stored in a PostgreSQL database. We developed a Ruby program that runs the MINSET algorithm for each method and stores the result in the same database. If a method does not have a minset, it stores a list of its strict supersets, and a list of methods that are duplicates when represented as a bag-of-words.

## 4.4 Results and Analysis

We provide quantitative and qualitative results for the following questions:

1. How many units of code are lexically distinguishable?

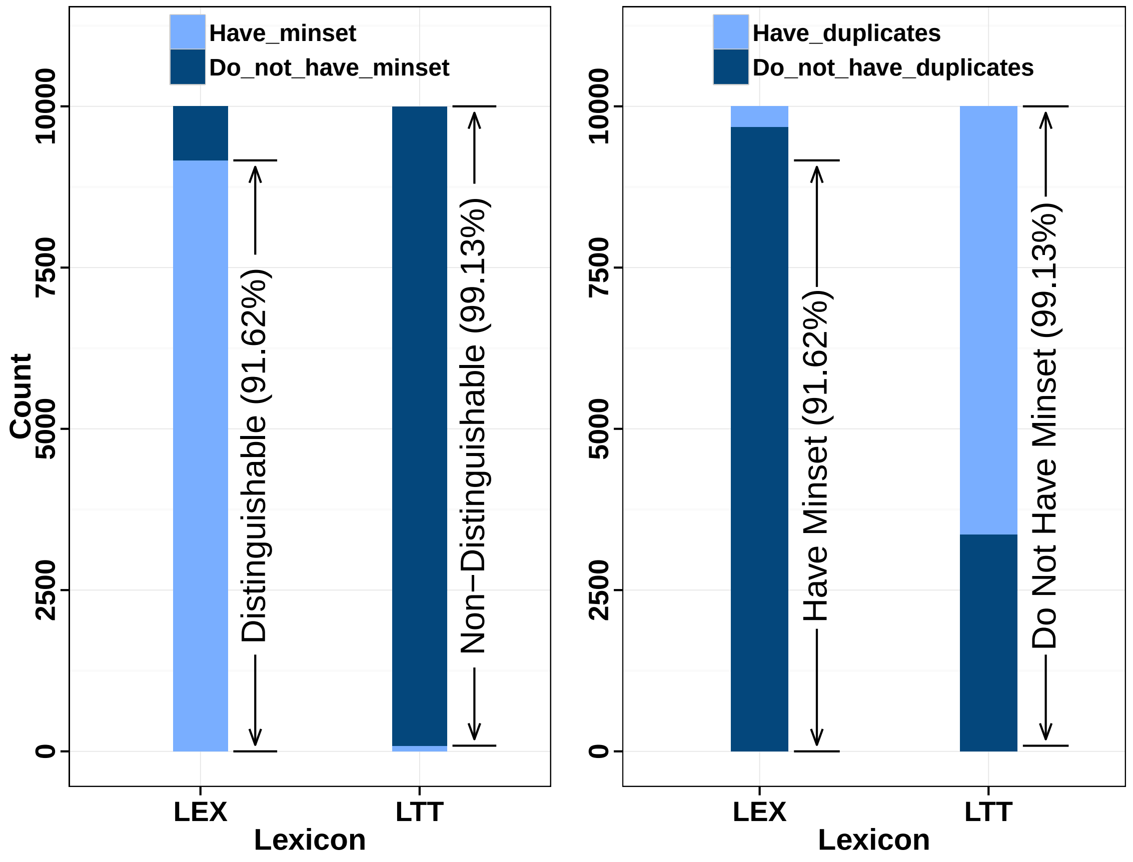


Figure 4.5. Random Sample of 10,000 Methods: (left) *Proportion of Methods with Minsets:* There is a stark difference in that proportion between LEX and LTT. (right) *Proportion of Methods with Duplicates:* LEX induces very few duplicates compared to LTT. LTT maps almost three quarters of the methods to the same set as another. It is too coarse, and does not model methods well.

2. How much of code is needed to distinguish it?

3. To what extent do minsets also capture code behavior and behavior differences?

4. What is a natural, minimal lexicon?

***Measures***[2] We define the *yield* of a lexicon to be the percentage of distinguishable methods in our corpus. The second question can be addressed in terms of absolute *minset size*, or in terms of *minset ratio*, minset size to threshed method size. While minset sizes and minset ratios will almost undoubtedly vary across functions, we hypothesize that the *mean* minset size and the *mean* minset ratio are small.

***Lexicons*** We provide results over 6 different lexicons, listed in Table 4.3. LEX and LTT

Table 4.4. Types of lexemes (or words) in the minsets we computed over the lexicon LEX.

| Grain Type | Count | Examples |
|---|---|---|
| Variable Identifier (of Public Type) | 3235 | abilityType (java.lang.StringBuffer), defaultValue (int), lostCandidate (boolean), twinsItem (java.util.List) |
| String and Character Literal | 3202 | '\u203F', '&', "192.168.1.36", "audit.pdf", "Error: 3", "Joda", "Record Found", "secret4" |
| Method Call (Local) | 2942 | classNameForCode, getInstanceProperty, isUserDefaultAdmin, makeDir, shouldAutoComplete |
| Variable Identifier (of Local Type) | 1574 | arcTgt, component, iVRPlayPropertiesTab, nestedException, this_TemplateCS_1, wordFSA |
| Type Identifier (a Local Type) | 1413 | ErrorApplication, IWorkspaceRoot, Literals, NNSingleElectron, PickObject, TrainingComparator |
| Method Call (a Public Method) | 508 | currentTimeMillis (java.lang.System.currentTimeMillis()), replace (java.lang.String.replace(char,char)) |
| Number Literal (integer, float, etc.) | 310 | 0, 1, 3, 150, 2010, 0xD0, 0x017E, 0x7bcdef42, 255.0f, 0x1000000000041L, 46.666667 |
| Type Identifier (a Public Type) | 265 | int, ArrayList, Collection, IllegalArgumentException, PropertyChangeSupport, SimpleDateFormat |
| Operator | 260 | ^=, <, <<=, <=, =, ==, >, >=, >>, >>=, >>>=, \|, \|=, \|\|, -, -=, –, !, !=, ?, /, /=, @, *, &, &&, +, +=, ++ |
| Keyword (Except Types) | 196 | break, catch, do, else, extends, final, finally, for, instanceof, new, return, super, synchronized, this, try, while |
| Separator | 148 | <, >, ",", ., ] |
| Reserved Words (Literals) | 104 | false, null, true |
| Other | 112 | COLUMNNAME_PostingType, E, ec2, element, ModelType, org, T, TC |

are the lexicons we discussed in Section 4.2.2. MIN1- MIN4 are lexicons we explore in the search for a natural, minimal lexicon.

***Summary*** Code is lexically distinguishable. Perhaps just as importantly, only 1.56 words, on average, or just 4%, are needed to distinguish a unit of code from all others in the corpus over LEX. The problem with minsets over LEX is that they do no capture behavior and behavior differences well. Over MIN4, on the other hand, minsets are still small but reveal much more about the behavior of the code because we intentionally blurred lexical differences which we suspect do not distinguish behavior. We elaborate on this point in Section 4.4.3 and Section 4.4.5.

All of our data and data processing code can be downloaded from Bitbucket.$^3$

## 4.4.1 Lexical Distinguishability of Source Code

The question "How much of a piece of code is needed to distinguish it from others?" can be answered in two ways: in terms of *minset size* and *minset ratio*. We report both.

There are two natural views we can take of code: the raw sequence of lexemes the programmer sees when writing and reading code, and the abstract sequence of tokens the compiler sees in parsing code. We want to explore those two views, and capture each one as a lexicon, a set of words. LEX is the set of all lexemes found in code (5,611,561 words). LTT is the set of lexer token types defined by the compiler (101 words). Each word in LTT is an abstraction of a lexeme, like `3` into `INTLIT`.

**LEX** LEX is the primordial lexicon; all others are abstractions of its words. Unfortunately, it is noisy: it is sensitive to any syntactic differences, including typos or use of synonyms, so it tends to overstate the number of minsets and understate their sizes; spurious homonyms can have the opposite effect, but are unlikely in Java when one can employ fully qualified names. LTT is the minimal lexicon a parser needs to determine whether or not a string is in a language. We computed minsets of all the methods in our random sample of $10,000$ using each lexicon, and display a summary of our results in Figure 4.4 and Figure 4.5.

Using LEX, a tiny proportion of code is needed to distinguish it. The minset of a method, on average, contains 4.57% of the unique lexemes in a method which means that methods in Java contain a significant amount of chaff, 95.43% on average. More surprisingly, the number of lexemes in a minset is also just plain small. The mean minset size is 1.55. The minset sizes also do not vary much. In 85.62% of the methods, one or two unique lexemes suffices to distinguish the code from all others. The largest minset consists of only 6 lexemes. Minset ratios also do not vary much. 75% of all methods have a minset ratio of 6.35% or smaller. While the ratios are sometimes large, the absolute sizes never are. The method with the largest minset ratio, 33.3%, for example, consists of 18 unique lexemes but has a minset size of 6. The method with the second largest minset ratio, 29.41%, another example, consists of 17 unique lexemes and has a minset size of 5.

***Minset Sizes of Large Methods*** Minsets are surprisingly small; especially surprising is that the maximum size is small. One reason might be the compression inherent to representing functions as sets. We address this later when we experiment with multisets. To test the robustness of our results, we also focused our investigation on larger methods because they may encode more behavior and therefore have more information. Hence, they may have larger minsets. Selected uniformly at random, our sample set does not include many of the largest methods: the largest method in our random sample has 2025 lines of code while the largest one in our corpus contains $4,606$ lines of code. To answer this question about minset properties conditioned on large methods, we selected the 1,000 largest methods, by lines of source code, and computed their minsets. The mean and maximum minset sizes of the largest methods are slightly lower but similar to the previous sample, 1.12 and 4, respectively. This shows that minsets are small and potentially effective indices of unique information even for abnormally large methods.

**LTT** Using LTT, the proportion of words needed to distinguish code is larger but still small. The minset of a method, on average, contains 18.45% of the unique token types in a method. We observe again that sometimes minset ratios can be large but the absolute minsets sizes never are. It is not surprising that the minset ratio is larger. Information is lost in mapping millions of distinct lexemes to only 101 distinct lexer token types. Information is also lost as method sizes decrease from 42.7 using LEX to 18.2 using LTT.

These results show that few words are needed to distinguish code, in relative and absolute terms. Given that we preserve a lot of information with LEX, we claim that the mean minset size, and mean minset ratios we found are approximate lower bounds. In essence, we define a lexicon spectrum where LEX is one of the poles, and LTT is a more abstract point on the lexicon spectrum.

***Yield*** The *yield* of a lexicon is the percentage of distinguishable methods. Our exploration shows that the yield decreases as the lexicon becomes coarser, measured roughly by the number of words in the lexicon. Our coarsest lexicon, LTT, blurs lexical differences too much. Over LTT, only 87 out of 10,000, 0.87%, methods have a minset. This is in great part due to the fact that LTT induces many duplicates. Over LTT, 6,640 out of 10,000 are modeled to the same bag-of-words as another method in the corpus. Recall that all of these methods are unique at the source code level. In contrast, LEX appears to preserve sufficient lexical differences so that 9,087 out of 10,000 methods have a minset.

### 4.4.2 Minsets over LEX

Since there are thousands of minsets, we take a broad view of minsets. For all minsets, we partitioned lexemes by type, leveraging information collected JavaFE; the types we defined are similar to lexer token types but broader in some cases and narrower in others. We provide a list of the lexeme types we defined, along with the counts of lexemes belonging to that type in Table 4.4[5].

Public type variable identifiers, and string and character literals dominate minsets. String literals are constant string values like `"Joda"`. The strings can represent error or information messages, IP addresses, names, pretty much anything. Perhaps this is why are at the top of the list: they can be unique or very rare. We divide certain classes of words depending if they are public or local — method invocations, type identifiers, and variable names. Public words are more standard and common whereas local words are more specialized and rare. Not surprisingly then, we observe that standard language features, like keywords and operators, and public types and methods are less common in minsets. The only exception is variable identifiers of types local to their respective project. Their distinctiveness is due in part to synonyms and homonyms. A programmer has great freedom in creating them. For example, `dir` appears 8017 times, as a variable name in methods, while `directory` appears only 2774 times. Another reason is that variable identifiers are more prevalent than other type of identifiers, like types and method calls.

### 4.4.3 What is a Natural, Minimal Lexicon?

We have shown that a method can be uniquely reduced to and thus uniquely identified by a small minset over LEX and LTT. However, LTT is too coarse. Token types as are too abstract. LEX preserves lexical differences. The existence of minsets over LEX can be of practical use in applications like Code Search. The problem with minsets over LEX is that they do not capture behavior and behavior differences well. Raw lexemes are too specific and cryptic.

***Goal*** We set the goal of finding a lexicon that is *minimal*, as small as possible, and *natural*, consisting of words that a human would know and use in applications like code search and code synthesis. The words in this lexicon should be *meaningful*, in the sense that they reveal information about the behavior of the code to us, humans. Since a

[5]A caveat: Algorithm 11 at line 4 picks arbitrarily between two equally rare words. Thus, these counts could differ.

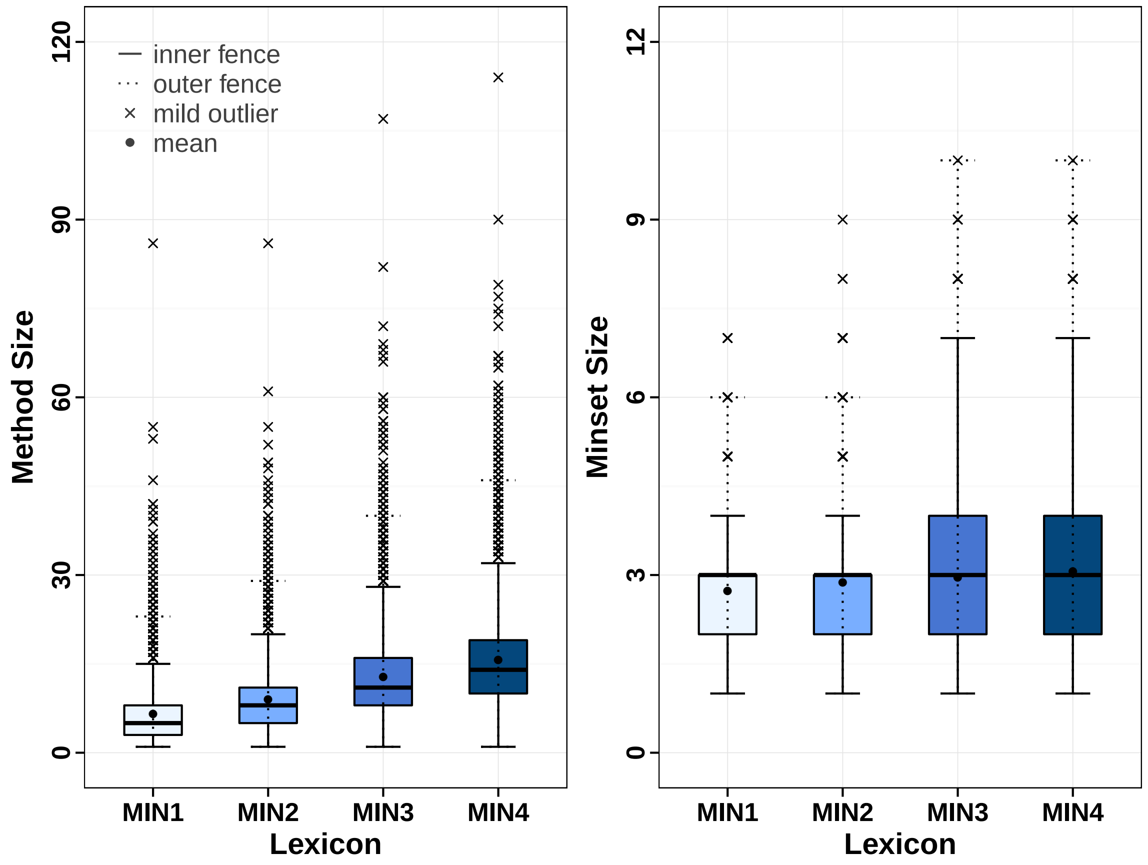


Figure 4.6. (left) As the lexicon grows from MIN1 to MIN4, the average size of the threshed methods also grows. (right) As the lexicon grows, the average minset size hardly changes. At least three quarters of the methods have a minset smaller than 4. Even as the lexicon grows, the maximum minset size is never more than 10.

MINSET is, by definition, distinguishing, we then expect minsets to capture behavior and behavior differences of a piece of code.

***Strategy*** We search by exploring the lexicon spectrum toward more abstract views of code. We additively construct a bag of words that approximates what a programmer might naturally use in applications like code search and code synthesis.

***Challenges*** Two issues confounds this search: lexicon specialization can *overfit* while lexicon abstraction introduces *imprecision*. To ameliorate overfitting, we restricted our search to natural lexicons. By natural, we mean simple and intuitive. We pursue natural abstractions to avoid unnatural abstractions that overfit our corpus, like one that maps every function in our corpus to a unique meaningless word. In our context, imprecision

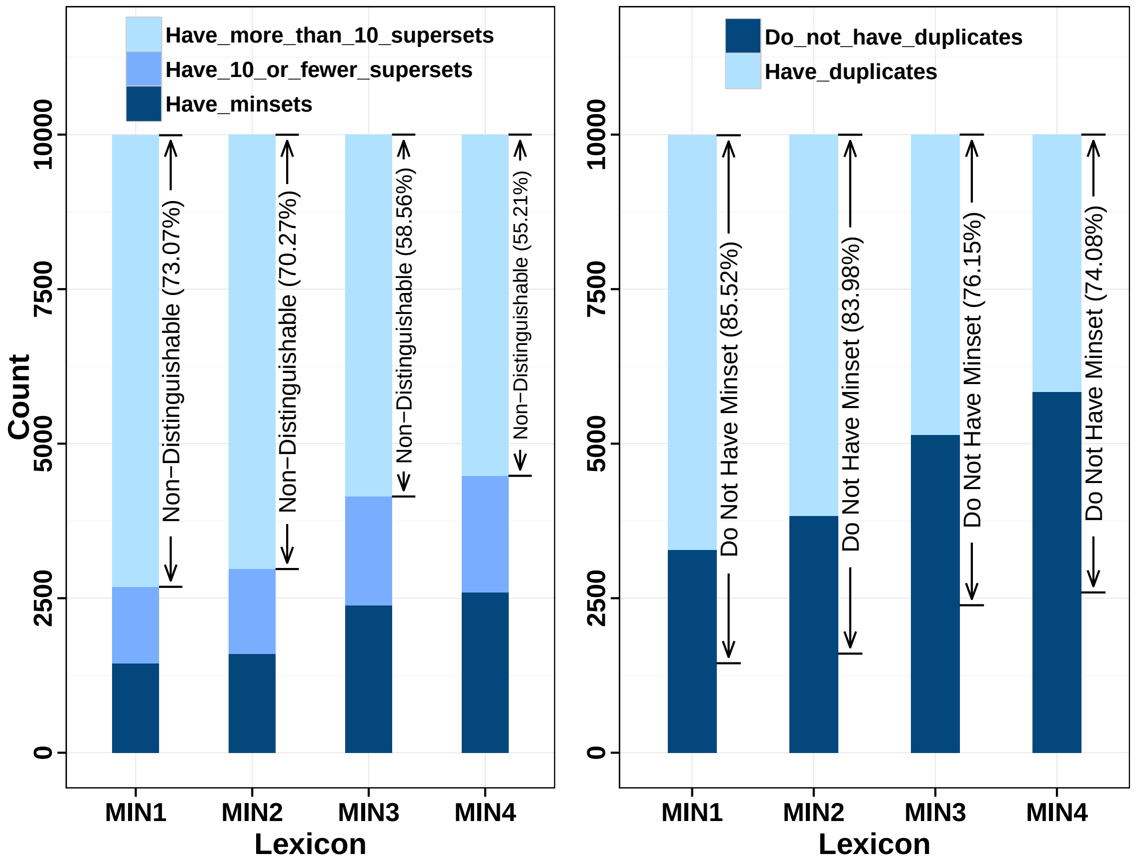


Figure 4.7. (left) *Yield:* The yield clearly improves with each change. At MIN4, the yield is 44.79%. (right) *Proportion of Methods With Duplicates:* Using this proportion as a rough gauge of threshing precision, there is a substantial improvement in threshing precision with each lexicon — fewer methods have duplicates. MIN4 pushes that precision past 50%.

leads spurious homonyms which reduces yield[6]. To handle this problem, we relax the definition of distinguishability (Definition 4.4.1). Henceforth, when we say distinguishable we mean 10-distinguishable. We chose 10 because that is consistent with what humans can process in a glance or two. Humans can rapidly process short lists [Miller, 1956].

**Definition 4.4.1** *A unit of code is* lexically k-distinguishable *if it does not have distinguishing subset but has* 10 *or fewer supersets.*

We considered four candidates, lexicons. We listed and introduced them briefly in

[6]Although LEX is rife with synonyms, our candidate lexicons have almost none.

Table 4.3. Our results appear in Figure 4.6 and Figure 4.7. We report absolute minset sizes. In searching or synthesizing code using minsets, the minset size is likely more important to the programmer than the minset ratio. We also report *yield*, the proportion of distinguishable methods. The yield approximates the likelihood of success for the programmer given that lexicon in the context of some code search or code synthesis application, Broadly, it gives us a sense of the potential practical usefulness of a lexicon.

**MIN1** First, we considered MIN1, a lexicon including only method names and operators. For public API methods, we used fully qualified method names to prevent the spurious creation of homonyms. For local methods, we abstracted all names to a single abstract word to capture their presence. Local methods tend to implement project-specific functionality not provided by the public API, and are not generally aimed for general use. The intuition in including method names is that a lot of the semantics is captured in method calls. They are the verbs or action words of program sentences. Our intuition is further supported by the effectiveness of API birthmarking [Schuler et al., 2007]. We also included operators because all primitive program semantics are applications of operators. Using this lexicon, the mean and maximum minset sizes are small, 2.73 and 7, respectively. The imprecision of MIN1 manifests itself in the low yield of 26.86%.

**MIN2** To try to improve yield, we created lexicon MIN2 by including control flow keywords as well; there are 13 in Java. From the programmer's perspective, these words reveal a great deal about the structure of a method that is critical to semantics. For example, the word `for` alone immediately tells us that some behavior is repeated. Using this lexicon, the mean and maximum minset sizes are still small, 2.88 and 9, respectively. The yield does not increase much. Only an additional 288 methods become threshable. The likeliest and simplest explanation for the small change is that these words are very common; at least one of them is present in 83.26% of the methods. It is more difficult to interpret this change. On the one hand, it is small. On the other hand, it is the result of adding only 13 new, semantically-rich words. In balancing the size of lexicon with the interpretability of minsets, this appears to be a good trade-off.

**MIN3** In our quest to improve yield, we defined MIN3 to include the types of variable identifiers (names). Those of a public type were mapped to their fully qualified type name. Those of a locally-defined type were mapped to a single abstract word to signal their presence. Locally-defined types, like local methods, tend to be project-specific and not of general use. Our reason for focusing on types is that they tell the programmer the kind of data on which methods and operators act. It is also a simple way of considering variable identifiers. Again, the mean and maximum minset size are small, 2.96 and 9, respectively. There is a notable increase in the yield, from 29.72% to 41.44%. It is now close to what we would imagine might be practical. In a MINSET-based application, a programmer would succeed 4 out of 10 times. Though, the lexicon grew substantially by 36,260 words. This trade-off appears reasonable considering as well that it is natural to supply the programmer with the convenience of a variety of primitive and composite types.

**MIN4** We defined a final lexicon, MIN4, which includes `false`, `true`, and `null`, object reference keywords, like `this` and `new`, and the token types of constant values, such as the token type `Character-Literal` for 'Z' or, for `5`, `Integer-Literal`. In total, we added 13 new words. Our intuition is that the *use* of hard-coded strings and numbers is connected to behavior. Certainly, knowing that hard-coded values are used can be informative. Also, in an application, a programmer may need to indicate that some constant string or number will be used. For example, if the programmer wishes to find a method that calculates the area of a circle, then it would be natural to indicate that target method likely contains a float literal like `3.14`. After including these words, the mean and maximum minset size remain small, 3.06 and 10, respectively. The yield increased from 41.44% to 44.79%. Adding this small number of semantically-rich words to the lexicon seems to be another reasonable exchange for a noticeable gain in yield: under this lexicon, the words are easier to interpret (see Section 4.4.5 for our analysis of the interpretability of minsets built from these words) while remaining small enough for humans to work with, *e.g.* a human could potentially write a minset from scratch while programming using key words [Little and Miller, 2007].

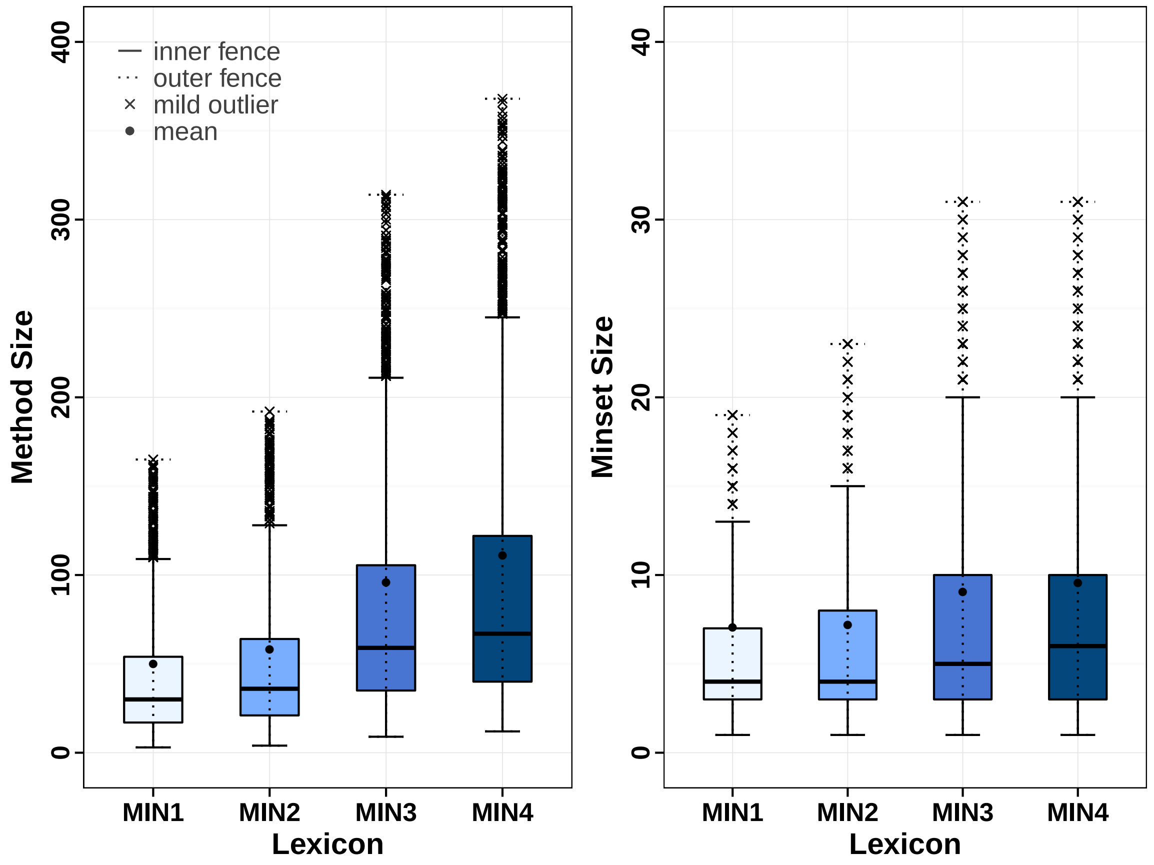


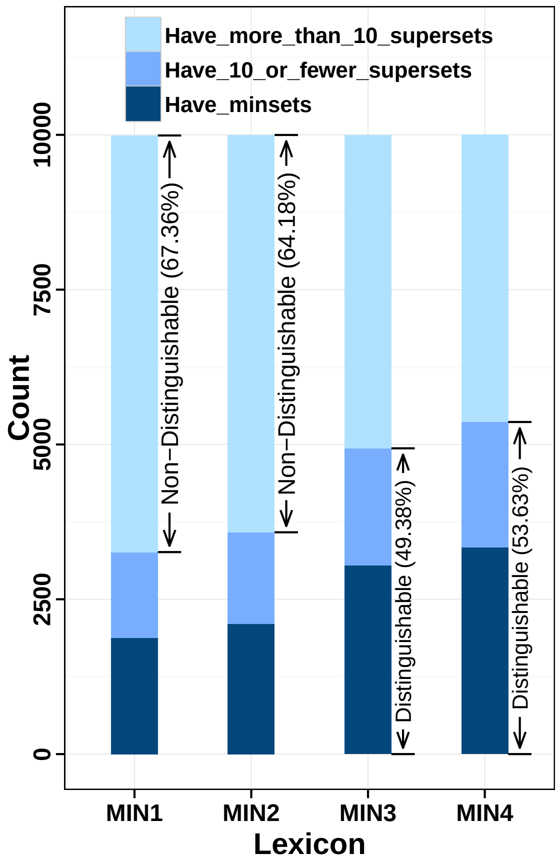
Figure 4.8. Multiplicity: (left) Like in Figure 4.6, as the lexicon grows, so does the threshed method size. In this case, methods are much larger because repetition is allowed. (right) The minset sizes, allowing repetition, are evidently larger. However, on average, they are still small across all lexicons. (To visualize both distributions, we omitted extreme outliers.[8])

## 4.4.4 The Effect of Multiplicity and Abnormally Large Methods on Distinguishability

Instead of continuing our search for lexicons generated from ever more complex abstractions over lexemes, we reconsidered *multiplicity*, the number of copies of a word in a method. We hypothesized that modeling methods as multi-sets would recapture some lexical differences, and thereby increase the yield of the lexicons MIN1 through MIN4. We used the multi-set version of Algorithm 11 to recompute minsets, and show our results in Figure 4.8.

Multiplicity improved yield at the cost of larger absolute minset sizes. The yield

[8]A point is an extreme outlier if it lies beyond $Q3 + 3 * IQ$ or below $Q1 - 3 * IQ$, where $IQ = Q3 - Q1$.



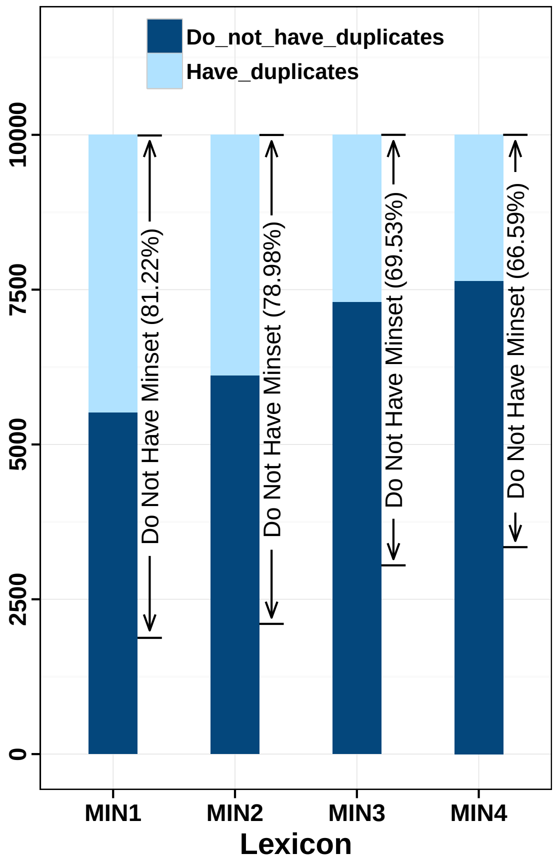


Figure 4.9. Multiplicity: (left) *Yield:* Multiplicity improves the yield of all lexicons. The yield of MIN4 now exceeds 50%. (right) *Proportion of Methods With Duplicates:* Using this proportion as a rough measure of threshing, multiplicity also improves the threshing precision of each lexicon. Less than 25% of the methods have duplicates using MIN4. (Note: Compare with Figure 4.7.)

increased for all lexicons. The new yields ranged from 32.64%–53.63%. The smallest increase in yield was using MIN1 (3.18%) and the largest was using MIN4 (8.84%). More concretely, using MIN4, the number of distinguishable methods increased by 884. Multiplicity also improved the minset ratios over all lexicons. For example, using MIN4, the mean minset ratio decreased from 15.47% to 5.35%. The cost of considering multiplicity, however, was an overall increase in minset sizes; the range of mean minset sizes shifted, 2.73–3.06, shifted and got a bit wider, 7.06–9.56. The outliers of minset sizes moved farther to the right. Previously, they ranged from 7–10 and now they range from 258–438. The right tails have grown longer. For example, using MIN4, 75.67% of the minsets have fewer than 10 words. Another cost of the gain in yield was in minset computation where we observed an approximate slowdown factor ranging from 4 to 7. For example, computing multi-set minsets using MIN1 took 44 hours instead of 6. In practice, the slowdown is much better than Algorithm 11's complexity implies. Overall, despite its cost, modeling methods as multi-sets over MIN4 produces a yield with practical value: it easily distinguishes more than half of the methods in our sample set.

Multiplicity appears to also improve the how well the bag-of-words preserves lexical differences. Modeling a method as a bag-of-words can map two unique methods to the same set or multi-set. When this happens, the MINSET algorithm cannot distinguish them. We can use the proportion of methods with duplicates to gauge the precision of the bag-of-words model. LEX gave us a baseline of 3.20%. When we experimented with lexicons MIN1 through MIN4 and no multiplicity, we observed the portion improved from 66.4% using MIN1 down to 41.64% using MIN4 (Figure 4.9). Multiplicity cut those figures nearly in half. For example, using MIN4, the proportion of methods with duplicates is only 23.59%.

The remaining portion of non-distinguishable methods is still intriguing. There are still 46.37% non-distinguishable methods, entirely subsumed by more than 10 other methods. We certainly expected some methods to subsume others because of their sheer size. We also expected families of methods with similar behavior where some subsume others. However, given that methods are not that small, containing, on average, 72.8 words over MIN4, and that the the portion of methods with duplicates is small, we suspected another reason. We hypothesized that there are abnormally large methods subsuming a great number of methods.

We conducted an experiment where we gradually filtered large methods to observe the effect on yield (Figure 4.10); we can perform this experiment without recomputing minsets. We initialized the filter size to $72,028$, the maximum method size (in tokens) in our corpus, and repeatedly halved it down to 70; the miminum size of a method is 50. Yield increases as the maximum method size filter is tuned down to 562. That appears to be the "sweet spot." If we filter methods with more than 562 tokens, or about 56 lines of code, then the yield improves from 53.67% to 61.74%. In an application that implements

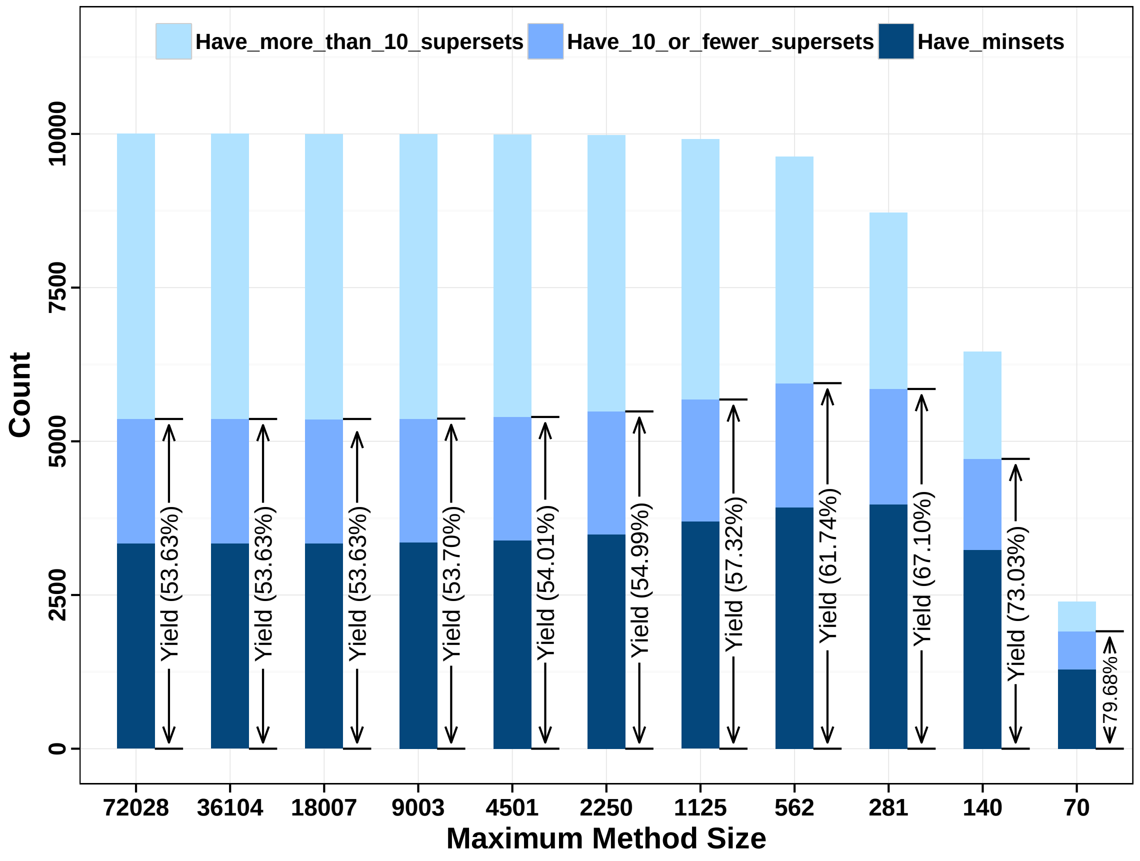


Figure 4.10. Yield, the percentage of distinguishable methods, increases as the maximum method size filter is tuned down to 562. From there, the number of methods and the number of threshable methods decreases substantially. Thus, setting the filter at 562 seems appropriate.

this filter would means that, a user would succeed 6 out of 10 times. For example, in a code search application, the likelihood of success of finding (recalling) a method would be improved if the application did not consider abnormally large methods. If we doubled the filter size to 1125, we would reconsider 55, 953 methods, and the yield would still be higher at 57.32% than without the filter. Since there is a relatively low number of these large methods, 69, 535 out of 1, 870, 905 (or 3.7%), the trade-off seems reasonable. A maximum size filter would clearly add practical value to a MINSET-based application.

MIN4 is a natural lexicon suited for code search, synthesis, and robust programming. We recomputed minsets using MIN4 considering multiplicity and a filter size 562. As we already mentioned, the yield is 61.74%. The mean minset size increases with the filter from 9.56 to 11.03. The minset sizes vary but have a clear positive skew where fewer than 25% contain more than 12 words. That right tail of the distribution is significantly shorter; the maximum size decreased from 689 to 173 because of the filter.

### 4.4.5 Minset Over MIN4

Minsets computed over LEX are small but do not capture behavior well. Minsets over MIN4 are still small; a few words are needed to distinguish a unit of code. *To what extent do minsets over* MIN4 *capture behavior and behavior differences amongst methods?* We provide a qualitative answer to this question via case studies: Minsets over MIN4 give insight into the behavior of a method.

We studied the minsets produced in our last experiment in Section 4.4.4. We selected nine minsets (Figure 4.11); we partitioned the methods into low, medium, and high minset ratios and picked three uniformly at random from each subset. For each minset, we tried to understand each element and what they revealed together about the behavior of a method. Then we inspected the method source code more carefully to assess how well the minsets capture method functionality. Due to lack of space, we discuss only three in detail.

*Low*: **L1** The method named `javax.xml.bind.Unmarshaller.unmarshal` from `(java.-xml.transform.Source)` de-serializes XML documents and returns a Java content tree object; `java.awt.Image` is an abstract classes that represents graphical images. From this minset, we infer that this method handles images and `XML` files. Since it reads the `XML` file, we also infer that it uses `XML` data in some manner. Perhaps the file contains a list of images, or the data in the file is used to create or alter an image. After inspecting the source code, we find that it is a method in the `LargeInlineBinaryTestCases` class of the Eclipse Link project, which manages `XML` files and other data stores. Our understanding was not far off: the method does read a binary `XML` file that contains images.

*Medium*: **M1** The `java.lang.Class.isInstance(java.lang.Object)` method checks if a given object is an object of type `Class` or assignment-compatible with its calling object.

| ID | MinSet (MIN4) | | | | Ratio |
|---|---|---|---|---|---|
| L1 | Java.awt.Image | javax.xml.bind.Unmarshaller.unmarshal(javax.xml.transform.Source) | | | 2.53% |
| L2 | javax.swing.DefaultBoundedRangeModel2Test.checkValues(javax.swing.BoundedRangeModel,int,int,int,int,boolean) | | | | 2.04% |
| L3 | / | java.text.Bidi.getRunLevel(int) | | | 4.55% |
| M1 | java.lang.Class.isInstance(java.lang.Object) | java.sql.Date.toString() | | | 12.5% |
| M2 | java.security.AccessController.<java.lang.Object>doPrivileged(java.security.PrivilegedAction<java.lang.Object>) | javax.security.auth.Policy.getPermissions(javax.security.auth.Subject,java.security.CodeSource) | | | 12.5% |
| M3 | @ | java.sql.PreparedStatement.setByte(int,byte) | | | 12.5% |
| H1 | = ×2 | java.lang.Exception | java.security.Security.addProvider(java.security.Provider) | super | 23.8% |
| H2 | boolean | java.lang.Object.equals(java.lang.Object) | org.eclipse.linuxtools.tmf.core.trace.TmfExperiment<LTYPE> ×3 | | 27.8% |
| H3 | com.sun.javadoc.ClassDoc ×3 | java.lang.String[] | java.lang.String.equals(java.lang.Object) | | 31.3% |

Figure 4.11. This shows the minsets of nine methods (MIN4). L1-L3 are minsets that have low minset ratios. M1-M3 have medium minset ratios. H1-H3 have high minset ratios. The minset elements are rich and reveal some information about the behavior of their respective methods.

The `java.sql.Date. toString()` method converts a `Date` object, which has been wrapped as an SQL date value, to a `String`. From this minset, we understand the type of a variable is checked. Perhaps, reflection is used on an object to ensure it is an instance of type `Date` before it converted to a string, for printing or storage. Inspecting the source code we find that this method resides in the `DateType` class of the Hibernate ORM project. Again, our understanding is very close to the behavior of the method. The method is passed an object, which it ensures is a `java.sql.Date` class object, and then returns the value as a string in the appropriate SQL dialect.

*High*: **H1** The `java.lang.Exception` object is thrown in Java to indicate abnormal flow or behavior. The = operator tells us that there is an assignment but is very common. The `java.security.Security.addProvider(java.security.Provider)` method adds a security service object, `Provider`, to a `Security` object. The `Security` object centralizes all the security properties in an application. The `super` keyword refers to the superclass. From this minset, we can infer that it describes a constructor that probably overrides a method in its superclass. We also infer that it catches an exception when adding the provider fails. In the source, we confirm that it is a constructor in the `HsqlSocketFactorySecure` class in the CloverETL project. It wraps code that instantiates a `Provider` class and adds it to the `Security` object in a `try` block. If adding the provider fails, it catches the exception, as we had inferred.

## 4.5 Discussion

Our results clearly support our Wheat and Chaff Hypothesis. We have shown, over a variety of lexicons, that functions are lexically distinguishable, and that the distinguishing subsets tend to be small. We defined and analyzed four lexicons in search for a natural, minimal lexicon that induces more meaningful minsets. We offered MIN4 as the promising candidate.

***Other Lexicons*** Our lexicon exploration avoided variable names because they are so unconstrained, noisy, and rife with homonyms and synonyms. Minsets over lexicons, like LEX, that incorporated them could include trivial, semantically insignificant differences, like "`user`" vs. "`usr`" in Unix. At the same time, variable names are an alluring source of signal. Intuitively, and in this corpus, they are the largest class of identifiers, which comprise 70% of source code [Deißenböck and Pizka, 2005], and connect a program's source to its problem domain [Binkley et al., 2013]. In future work, we plan to separate the "wheat from the chaff" in variable names.

***Alternative Units of Code*** We chose functions as our unit of code. However, we can apply the same methodology at other syntactic levels. One alternative is to study blocks of code. A single function can have many blocks. This could be very useful in an alternative programming model where the user seeks a common block of code but for which there is no individual function. Another alternative is to use abstract syntax trees (AST) to preserve some syntactic structure in the lexical features. We could also consider using n-grams to preserve some order in the features.

***Threats to Validity*** We identify two main threats. The first is that we only studied Java. However, we have no reason to believe that the "wheat and chaff" hypothesis does not hold for other programming languages. Java, though more modern, was designed to be very similar to C and C++ so that it could be adopted easily. The second threat comes from our corpus: size and diversity. We downloaded a very large corpus, by any standard. In fact, we downloaded all the Java projects listed as "Most Popular" in the four code repositories we crawled. Those code repositories are known primarily for hosting open-source projects. Thus, there is no indication that they are biased toward any specific types of projects. We plan to replicate this study on a larger Java corpus and with languages of different paradigms like Lisp and Prolog to help us understand to what extent the lexical distinguishability phenomenon varies and to what extent the Wheat and Chaff Hypothesis holds.

## 4.6 Applications

Though our study is primarily empirical, in this section, we describe pre-existing and new applications for minsets.

***SmartSynth (Existing)*** As mentioned earlier, the clearest and, perhaps, most promising application for minsets is in keyword-based programming. SmartSynth [Le et al., 2013] is a recent, modern incarnation. SmartSynth generates a smartphone script from a natural language description (query). "Speak weather in the morning" is an example of a successful query. SmartSynth uses NLP techniques to parse the query and map it to a set of "components" (words) in its underlying programming language. Combining a variety of techniques, it then infers relationships between the words to generate and rank candidate scripts. At its heart is the idea that usable code can be constructed from a small set of words. This subset is a minset or another distinguishing subset.

***Code Search Engine (New)*** A major problem of code search is ranking results [Bajracharya et al., 2006, Mandelin et al., 2005, McMillan et al., 2012]. We built a code search engine that uses a new ranking scheme[9]. Relevant methods are ranked by the similarity between their minsets and the user's query. For example, the query "sort array int" returns 135 methods. The top result, with minset "sort array parseInt 16", returns a sorted array of integers, if the 'sort' flag is set.

***Code Summarizer (New)*** From our case studies of MIN4 minsets, we realized that minsets can effectively summarize code. We built a code summary web application[9]. A user enters the source code of a method, our tool computes a minset, and presents it as a concise summary. Due to space constraints, we omit a full example and invite interested readers to explore our web application. Figure 4.11 shows examples of minsets summarizing methods.

**MINSET*-powered IDE (Concept)*** Our results offer insight into how to develop a more powerful, alternative programming system. Consider an integrated development environment (IDE), like Eclipse or IntelliJ, that can search a MINSET indexed database of code and requirements to 1) propose related code that may be adapted to purpose, 2) auto-complete whole code fragments as the programmer works, 3) speed concept location for navigation and debugging, and 4) support traceability by interconnecting requirements and code [Cleland-Huang et al., 2005].

## 4.7 Related Work

Although we are the first to study the phenomenon of lexical distinguishability of source code, and propose the Wheat and Chaff Hypothesis[10], a few strands of related work exist.

***Code Uniqueness*** At a basic level, our study is about uniqueness. *What lexical features distinguish or uniquely identify a unit of code?* Gabel and Su also studied uniqueness [Gabel and Su, 2010]. They found that software generally lacks uniqueness which they measure as the proportion of unique, fixed-length token sequences in a software project. We studied uniqueness differently. We capture uniqueness as the size or pro-

[9] `http://jarvis.cs.ucdavis.edu/code_essence.`

[10] Others have used the "wheat and chaff" analogy in the computing world but in different domains [Rivest, 1998, Schleimer et al., 2003].

portion of minsets. The elements in a MINSET may not be unique or even rare but together uniquely identify a piece of code. We keep in mind that syntactic differences do not always imply functional differences as Jiang and Su demonstrated [Jiang and Su, 2009]. Thus, in some cases the uniqueness may be accidental. Two minsets may, in fact, represent the same behavior at some higher, more abstract semantic level.

***Code Completion and Search*** Observations about natural language phenomenon provide a promising path toward making programming easier. Hindle *et al.* focused on the “naturalness” of software [Hindle et al., 2012]. They showed that actual code is “regular and predictable”, like natural language utterances. To do so, they trained an *n*-gram model on part of a corpus, and then tested it on the rest. They leveraged code predictability to enhance Eclipse’s code completion tool. Their work followed that of Gabel and Su who posited and gave supporting evidence that we are approaching a “singularity”, a point in time where all the small fragments of code we need to write will already exist [Gabel and Su, 2010]. When that happens, many programming tasks can be reduced to finding the desired code in a corpus. Our work suggests that small, natural set of words, *i.e.*, minsets, can index and retrieve code. As for code completion, a MINSET-based approach could exploit not just the previous $n-1$ tokens, but on all the previous tokens and complete not just the next token but whole pieces of code.

Sourcerer and Portolio, two modern code search engines, support basic term queries, in addition to more advanced queries [Bajracharya et al., 2006, McMillan et al., 2011]. Our research suggests that the natural and efficient term query is a MINSET. Search results may differ in granularity. Portfolio focuses on finding functions [McMillan et al., 2011] while Exemplar, another engine, finds whole applications [Grechanik et al., 2010], MINSET easily generalizes to arbitrary units of code. Finally, code search must also be “internet-scale” [Gallardo-Valencia and Elliott Sim, 2009]. With a modest computer, we can compute minsets for corpora of code of various languages, and update them regularly as new code is added.

Code completion tools suggest code a programmer *might* want to use. They infer relevant code and rank it. Many diverse, useful tools and strategies exist [Bruch et al., 2009, Nguyen et al., 2012, Nguyen et al., 2013, Zhang et al., 2012]. Our work suggests a different, complementary MINSET-based strategy: If what the programmer is coding contains the MINSET of some existing piece of code, suggest that.

***Genetics and Debugging*** At a high-level, Algorithm 11 isolates a minimal set of essential elements. Central to synthetic biology is the search for the 'minimal genome', the minimal set of genes essential to living organisms [Acevedo-Rocha et al., 2013] [Maniloff, 1996]. Delta debugging is very similar in that it finds a minimal set of lines of code that trigger a bug [Cleve and Zeller, 2000]. Both approaches rely on an oracle who defines what is "essential" whereas we define "essentialness" with respect to other sets.

## 4.8 Conclusion and Future Work

Humans do not read code like compilers, nor do they write code sequentially, lexeme by lexeme. Nonetheless, current programming paradigms force programmers through the eye of the needle of syntax. We introduce a novel view of code: discrete semantic units, like functions, can be separated into 'wheat' and 'chaff'. At a high level, our approach is a simple attempt to model the complex abstraction mechanism of the programmer's mind when reading or writing code.

We imagine that code, to the human mind, is amorphous, and ask: "If a programmer were reading this code, what features would be semantically important?" and "If a programmer were trying to write this piece of code, what key ideas would the programmer communicate?" A MINSET is our proposal of a useful, formal definition of these key ideas as 'wheat.' Our definition is constructive, so a computer can compute Minsets to generate or retrieve an intended piece of code.

We evaluated Minsets, over a large corpus of real-world Java programs, using various, natural lexicons: the computed minsets are sufficiently small and understandable for use in code search, code completion, and natural programming.

# Chapter 5

# Conclusion

In recent years, we have observed an explosion in demand for a computer science education, due in part to the increased demand for software engineers in the labor market. We are seeing more students enroll in traditional university courses. Many are signing up for massive open online courses (MOOCs). While others resort to attending coding boot camps. However, in addition to the known challenges of learning programming, these students are experiencing new challenges resulting from education at these scales. There is a need for novel tools and techniques that can help students overcome these challenges and succeed in learning to program. This dissertation identifies some of the technical barriers students face in learning to program and presents novel tools, techniques, and concepts to try to to address and minimize them.

First, we introduced Kodethon, a web integrated development environment. Over the last few years, thousands of students have used Kodethon to learn to program at a major U.S. public university. Through user surveys, we learned that Kodethon does lower the threshold for many students and lets them run and test code right away.

Second, we introduced COMPASSIST, a tool that aims to help programmers, especially beginners, resolve compilation errors. This is important because, in languages like C and C++, students cannot run code that does not compile. This can halt their learning progress completely until they receive peer or instructor aid. COMPASSIST employs a novel FUZZ-AND-REDUCE that automatically synthesizes repair examples, and, in the best case, can automatically repair the programmer's compiler error directly.

Lastly, we introduced the *Wheat and Chaff Hypothesis,* which states that source code is composed of wheat and chaff and most code is chaff. Informally, we define wheat as those lexical elements that are necessary for human understanding of a piece of code. Formally, we define wheat as the MINSET of a piece of code. We tested our hypothesis empirically over a large corpus of Java methods by analyzing *lexical distinguishability*. Our quantitative and qualitative analysis supports our hypothesis. In addition to being of scientific interest, our results validate recent efforts and encourage more efforts towards a minimal programming paradigm. By minimal, we mean one where a programmer writes "wheat" and the computer fills in the "chaff" Such a paradigm would help students code more easily and focus more on computation problems and solutions.

There are many opportunities for future work stemming from this dissertation. Work that is already in progress is extending Kodethon with a learning management system (LMS). In this extended Kodethon, students can seamlessly program in the Kodethon IDE and submit solutions via the Kodethon LMS. This provides students with a more seamless learning programming platform. The LMS also gives educators more opportunities to help students, especially those who struggle. For example, through the LMS, instructors can provide better feedback on programming assignment submissions. The LMS can visualize test suite results, and, in the future, may even provide suggestions on how to fix code errors to help students pass failing test cases.

There are also future work opportunities based on COMPASSIST. First, we can try to improve C++ compiler error coverage. We can also deploy the tool to the public. By making it public, we can collect broader feedback on the idea and implementation. A potential sign of success would be whether we can directly integrate COMPASSIST into a popular integrated development environment, like Visual Studio or Eclipse. Another potential sign of success would be whether we can contribute repair examples directly into the source code of compilers, like `clang++` and `g++`. Lastly, we should explore implementing this concept for compilers of other programming Languages, like Java.

We are living in a technological revolution where software is becoming an integral part of our lives. There is an increasing demand for more software and for people who can build it and maintain it. This revolution will fundamentally change our education systems and curriculum. It will require more people to be literate in writing programs. To get there, we need to rethink our teaching methods and tools. Thankfully, the research community has been very responsive to this phenomenon and we are seeing a lot of work in the area of computing education. The tools and techniques described in this dissertation are part of this broader research effort.

## References

[Academy, 2017] Academy, K. (2017). Khan academy.

[Acevedo-Rocha et al., 2013] Acevedo-Rocha, C. G., Fang, G., Schmidt, M., Ussery, D. W., and Danchin, A. (2013). From essential to persistent genes: a functional approach to constructing synthetic life. *Trends in Genetics*, 29(5):273–279.

[Adams, 2014] Adams, J. C. (2014). Computing is the safe stem career choice today.

[Adams, 2016] Adams, J. C. (2016). Us-bls: Computing employment outlook remains bright.

[Ahmed et al., 2018] Ahmed, U., Kumar, P., Karkare, A., Kar, P., and Gulwani, S. (2018). Compilation error repair: For the student programs, from the student programs.

[Altadmri and Brown, 2015] Altadmri, A. and Brown, N. C. (2015). 37 million compilations: Investigating novice programming mistakes in large-scale student data. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*, SIGCSE '15, pages 522–527, New York, NY, USA. ACM.

[Altadmri et al., 2015] Altadmri, A., Brown, N. C., and Kölling, M. (2015). Using bluej to code java on the raspberry pi. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*, SIGCSE '15, pages 178–178, New York, NY, USA. ACM.

[Bajracharya et al., 2006] Bajracharya, S., Ngo, T., Linstead, E., Dou, Y., Rigor, P., Baldi, P., and Lopes, C. (2006). Sourcerer: a search engine for open source code supporting structure-based search. In *Companion to the 21st ACM SIGPLAN Symposium on Object-Oriented Programming Systems, Languages, and Applications*, pages 681–682.

[Barik et al., 2017] Barik, T., Smith, J., Lubick, K., Holmes, E., Feng, J., Murphy-Hill, E., and Parnin, C. (2017). Do developers read compiler error messages? In *Proceedings of the 39th International Conference on Software Engineering*, ICSE '17, pages 575–585, Piscataway, NJ, USA. IEEE Press.

[Barik et al., 2014] Barik, T., Witschey, J., Johnson, B., and Murphy-Hill, E. (2014). Compiler error notifications revisited: An interaction-first approach for helping developers more effectively comprehend and resolve error notifications. In *Companion Proceedings of the 36th International Conference on Software Engineering*, ICSE Companion 2014, pages 536–539, New York, NY, USA. ACM.

[Barr and Trytten, 2016] Barr, V. and Trytten, D. (2016). Using turing's craft codelab to support cs1 students as they learn to program. *ACM Inroads*, 7(2):67–75.

[Basit and Jarzabek, 2007] Basit, H. A. and Jarzabek, S. (2007). Efficient token based clone detection with flexible tokenization. In *Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 513–516.

[Becker, 2015] Becker, B. A. (2015). An exploration of the effects of enhanced compiler error messages for computer programming novices. Master's thesis, Dublin Institute of Technology.

[Becker, 2016] Becker, B. A. (2016). An effective approach to enhancing compiler error messages. In *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*, SIGCSE '16, pages 126–131, New York, NY, USA. ACM.

[Benotti et al., 2018] Benotti, L., Aloi, F., Bulgarelli, F., and Gomez, M. J. (2018). The effect of a web-based coding tool with automatic feedback on students' performance and perceptions. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*, SIGCSE '18, pages 2–7, New York, NY, USA. ACM.

[Binkley et al., 2013] Binkley, D., Davis, M., Lawrie, D., Maletic, J. I., Morrell, C., and Sharif, B. (2013). The impact of identifier style on effort and comprehension. *Empirical Software Engineering*, 18(2):219–276.

[Böhme et al., 2017] Böhme, M., Soremekun, E. O., Chattopadhyay, S., Ugherughe, E., and Zeller, A. (2017). Where is the bug and how is it fixed? an experiment with practitioners. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, pages 117–128, New York, NY, USA. ACM.

[Boyd and Allevato, 2012] Boyd, E. and Allevato, A. (2012). Streamlining project setup in eclipse for both time-constrained and large-scale assignments. In *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education*, SIGCSE '12, pages 667–667, New York, NY, USA. ACM.

[Brown, 1983] Brown, P. J. (1983). Error messages: The neglected area of the man/-machine interface. *Commun. ACM*, 26(4):246–249.

[Bruch et al., 2009] Bruch, M., Monperrus, M., and Mezini, M. (2009). Learning from examples to improve code completion systems. In *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 213–222.

[Brune et al., 2014] Brune, P., Leiser, M., and Janke, E. (2014). Towards an easy-to-use web application server and cloud paas for web development education. *Proceedings - 16th IEEE International Conference on High Performance Computing and Communications, HPCC 2014, 11th IEEE International Conference on Embedded Software and Systems, ICESS 2014 and 6th International Symposium on Cyberspace Safety and Security*, pages 1113–1116.

[Bureau of Labor Statistics, 2018] Bureau of Labor Statistics, U. D. o. L. (2018). Occupational outlook handbook: Software developers.

[Campbell et al., 2014] Campbell, J. C., Hindle, A., and Amaral, J. N. (2014). Syntax errors just aren't natural: Improving error reporting with language models. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, pages 252–261, New York, NY, USA. ACM.

[Clang, 2018] Clang (2018). Expressive diagnostics.

[Cleland-Huang et al., 2005] Cleland-Huang, J., Settimi, R., BenKhadra, O., Berezhanskaya, E., and Christina, S. (2005). Goal-centric traceability for managing non-functional requirements. In *Proceedings of the International Conference on Software Engineering*, pages 362–371.

[Cleve and Zeller, 2000] Cleve, H. and Zeller, A. (2000). Finding failure causes through automated testing. In *Proceedings of the Fourth International Workshop on Automated Debugging*.

[CodeEnvy, 2017] CodeEnvy (2017). CodeEnvy.

[D'Antoni et al., 2017] D'Antoni, L., Singh, R., and Vaughn, M. (2017). Nofaq: Synthesizing command repairs from examples. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM.

[DataUSA, 2018] DataUSA (2018). Computer science: Stem major.

[De Rosso and Jackson, 2016] De Rosso, S. P. and Jackson, D. (2016). Purposes, concepts, misfits, and a redesign of git. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2016, pages 292–310, New York, NY, USA. ACM.

[Deißenböck and Pizka, 2005] Deißenböck, F. and Pizka, M. (2005). Concise and consistent naming. In *Proceedings of the 13th International Workshop on Program Comprehension*, pages 97–106.

[Denny et al., 2014] Denny, P., Luxton-Reilly, A., and Carpenter, D. (2014). Enhancing syntax error messages appears ineffectual. In *Proceedings of the 2014 Conference on Innovation &#38; Technology in Computer Science Education*, ITiCSE '14, pages 273–278, New York, NY, USA. ACM.

[Dyke, 2011] Dyke, G. (2011). Which aspects of novice programmers' usage of an ide predict learning outcomes. In *Proceedings of the 42Nd ACM Technical Symposium on Computer Science Education*, SIGCSE '11, pages 505–510, New York, NY, USA. ACM.

[Eclise, 2018] Eclise (2018). Quick fix and quick assist.

[Fisker et al., 2008] Fisker, K., McCall, D., Kölling, M., and Quig, B. (2008). Group work support for the bluej ide. In *Proceedings of the 13th Annual Conference on Innovation and Technology in Computer Science Education*, ITiCSE '08, pages 163–168, New York, NY, USA. ACM.

[Flowers et al., 2004] Flowers, T., Carver, C. A., and Jackson, J. (2004). Empowering students and building confidence in novice programmers through gauntlet. In *34th Annual Frontiers in Education, 2004. FIE 2004.*, pages T3H/10–T3H/13 Vol. 1.

[Ford and Staley, 2016] Ford, C. and Staley, C. (2016). Automated analysis of student programmer coding behavior patterns (abstract only). In *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*, SIGCSE '16, pages 688–688, New York, NY, USA. ACM.

[Foundation, 2017] Foundation, E. (2017). Eclipse CHE.

[Free Software Foundation, 2004] Free Software Foundation, I. (2004). New c parser.

[Gabel and Su, 2010] Gabel, M. and Su, Z. (2010). A study of the uniqueness of source code. In *Proceedings of the 18th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 147–156.

[Gallardo-Valencia and Elliott Sim, 2009] Gallardo-Valencia, R. E. and Elliott Sim, S. (2009). Internet-scale code search. In *Proceedings of the 2009 ICSE Workshop on Search-Driven Development-Users, Infrastructure, Tools and Evaluation*, pages 49–52.

[Goldman et al., 2011] Goldman, M., Little, G., and Miller, R. C. (2011). Real-time collaborative coding in a web ide. In *Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology*, UIST '11, pages 155–164, New York, NY, USA. ACM.

[Grechanik et al., 2010] Grechanik, M., Fu, C., Xie, Q., McMillan, C., Poshyvanyk, D., and Cumby, C. (2010). A search engine for finding highly relevant applications. In *Proceedings of the ACM/IEEE International Conference on Software Engineering*, pages 475–484.

[Guo, 2013] Guo, P. J. (2013). Online python tutor: Embeddable web-based program visualization for cs education. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education*, SIGCSE '13, pages 579–584, New York, NY, USA. ACM.

[Gupta et al., 2017] Gupta, R., Pal, S., Kanade, A., and Shevade, S. (2017). Deepfix: Fixing common c language errors by deep learning. In *AAAI*, pages 1345–1351.

[Hartmann et al., 2010] Hartmann, B., MacDougall, D., Brandt, J., and Klemmer, S. R. (2010). What would other programmers do: Suggesting solutions to error messages. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '10, pages 1019–1028, New York, NY, USA. ACM.

[Helminen et al., 2013] Helminen, J., Ihantola, P., and Karavirta, V. (2013). Recording and analyzing in-browser programming sessions. In *Proceedings of the 13th Koli Calling International Conference on Computing Education Research*, Koli Calling '13, pages 13–22, New York, NY, USA. ACM.

[Hindle et al., 2012] Hindle, A., Barr, E. T., Su, Z., Gabel, M., and Devanbu, P. (2012). On the naturalness of software. In *Proceedings of the International Conference on Software Engineering*, pages 837–847.

[Horton and Craig, 2015] Horton, D. and Craig, M. (2015). Drop, fail, pass, continue: Persistence in cs1 and beyond in traditional and inverted delivery. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*, SIGCSE '15, pages 235–240, New York, NY, USA. ACM.

[Hristova et al., 2003] Hristova, M., Misra, A., Rutter, M., and Mercuri, R. (2003). Identifying and correcting java programming errors for introductory computer science students. In *Proceedings of the 34th SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '03, pages 153–156, New York, NY, USA. ACM.

[Hüsing et al., 2013] Hüsing, T., Korte, W. B., Fonstad, N., Lanvin, B., Cattaneo, G., Kolding, M., Lifonti, R., and van Welsum, D. (2013). *e-Leadership. e-Skills for Competitiveness and Innovation Vision, Roadmap and Foresight Scenarios Final Report*.

[Inc., 2017a] Inc., C. (2017a). CodeAnywhere.

[Inc., 2017b] Inc., C. I. (2017b). Cloud9 IDE.

[Isa et al., 1983] Isa, B. S., Boyle, J. M., Neal, A. S., and Simons, R. M. (1983). A methodology for objectively evaluating error messages. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '83, pages 68–71, New York, NY, USA. ACM.

[Jadud, 2006] Jadud, M. C. (2006). Methods and tools for exploring novice compilation behaviour. In *Proceedings of the Second International Workshop on Computing Education Research*, ICER '06, pages 73–84, New York, NY, USA. ACM.

[Jenkins et al., 2010] Jenkins, J., Brannock, E., and Dekhane, S. (2010). Javawide: Innovation in an online ide: Tutorial presentation. *J. Comput. Sci. Coll.*, 25(5):102–104.

[Jiang and Su, 2009] Jiang, L. and Su, Z. (2009). Automatic mining of functionally equivalent code fragments via random testing. In *Proceedings of the 18th International Symposium on Software Testing and Analysis*, pages 81–92.

[Jurafsky and Martin, 2009] Jurafsky, D. and Martin, J. H. (2009). *Speech and Language Processing (2Nd Edition)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

[Kasto and Whalley, 2013] Kasto, N. and Whalley, J. (2013). Measuring the difficulty of code comprehension tasks using software metrics. In *Proceedings of the Fifteenth Australasian Computing Education Conference - Volume 136*, ACE '13, pages 59–65, Darlinghurst, Australia, Australia. Australian Computer Society, Inc.

[Kazerouni et al., 2017] Kazerouni, A. M., Edwards, S. H., Hall, T. S., and Shaffer, C. A. (2017). Deveventtracker: Tracking development events to assess incremental development and procrastination. In *Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education*, ITiCSE '17, pages 104–109, New York, NY, USA. ACM.

[Kelleher and Pausch, 2005] Kelleher, C. and Pausch, R. (2005). Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers. *ACM Comput. Surv.*, 37(2):83–137.

[Kim, 2017] Kim, A. (2017). Cs 61a course enrollment reaches new high, nears 2,000.

[Kim et al., 2013] Kim, D., Nam, J., Song, J., and Kim, S. (2013). Automatic patch generation learned from human-written patches. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 802–811, Piscataway, NJ, USA. IEEE Press.

[Ko et al., 2015] Ko, A. J., Latoza, T. D., and Burnett, M. M. (2015). A practical guide to controlled experiments of software engineering tools with human participants. *Empirical Softw. Engg.*, 20(1):110–141.

[Koding, 2017] Koding (2017). Koding.

[Kolling et al., 2003] Kolling, M., Quig, B., Patterson, A., and Rosenberg, J. (2003). The BlueJ System and its Pedagogy. *Computer Science Education*, 13(4):249–268.

[Kori et al., 2015] Kori, K., Pedaste, M., TÃţnisson, E., Palts, T., Altin, H., Rantsus, R., Sell, R., Murtazin, K., and RÃijÃijtmann, T. (2015). First-year dropout in ict studies. In *2015 IEEE Global Engineering Education Conference (EDUCON)*, pages 437–445.

[Kummerfeld and Kay, 2003] Kummerfeld, S. K. and Kay, J. (2003). The neglected battle fields of syntax errors. In *Proceedings of the Fifth Australasian Conference on Computing Education - Volume 20*, ACE '03, pages 105–111, Darlinghurst, Australia, Australia. Australian Computer Society, Inc.

[Lardinois, 2016] Lardinois, F. (2016). Cloud development platform nitrous.io shuts down.

[Lazowska et al., 2014] Lazowska, E., Roberts, E., and Kurose, J. (2014). Tsunami or sea change? responding to the explosion of student interest in computer science. `http://lazowska.cs.washington.edu/NCWIT.pdf`. Accessed: 2017-11-28.

[Le et al., 2013] Le, V., Gulwani, S., and Su, Z. (2013). SmartSynth: synthesizing smartphone automation scripts from natural language. In *Proceeding of the 11th Annual International Conference on Mobile Systems, Applications, and Services*, pages 193–206.

[Le Goues et al., 2012] Le Goues, C., Nguyen, T., Forrest, S., and Weimer, W. (2012). Genprog: A generic method for automatic software repair. *IEEE Trans. Softw. Eng.*, 38(1):54–72.

[Li et al., 2004] Li, Z., Lu, S., Myagmar, S., and Zhou, Y. (2004). CP-Miner: a tool for finding copy-paste and related bugs in operating system code. In *Proceedings of the Symposium on Operating Systems Design & Implementation*, pages 289–302.

[Little and Miller, 2007] Little, G. and Miller, R. C. (2007). Keyword programming in Java. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, pages 84–93.

[Little et al., 2010] Little, G., Miller, R. C., Chou, V. H., Bernstein, M., Lau, T., and Cypher, A. (2010). Sloppy programming. In Cypher, A., Dontcheva, M., Lau, T., and Nichols, J., editors, *No Code Required*, pages 289–307. Morgan Kaufmann.

[Long et al., 2017] Long, F., Amidon, P., and Rinard, M. (2017). Automatic inference of code transforms for patch generation. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, pages 727–739, New York, NY, USA. ACM.

[Long and Rinard, 2016] Long, F. and Rinard, M. (2016). Automatic patch generation by learning correct code. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, pages 298–312, New York, NY, USA. ACM.

[Lund, 2001] Lund, A. (2001). Measuring usability with the use questionnaire. 8.

[Mandelin et al., 2005] Mandelin, D., Xu, L., Bodík, R., and Kimelman, D. (2005). Jungloid mining: helping to navigate the API jungle. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 48–61.

[Maniloff, 1996] Maniloff, J. (1996). The minimal cell genome: "on being the right size". *Proceedings of the National Academy of Sciences*, 93(19):10004–10006.

[Manning and Schütze, 1999] Manning, C. D. and Schütze, H. (1999). *Foundations of Statistical Natural Language Processing*. MIT Press, Cambridge, MA, USA.

[Marceau et al., 2011a] Marceau, G., Fisler, K., and Krishnamurthi, S. (2011a). Measuring the effectiveness of error messages designed for novice programmers. In *Proceedings of the 42Nd ACM Technical Symposium on Computer Science Education*, SIGCSE '11, pages 499–504, New York, NY, USA. ACM.

[Marceau et al., 2011b] Marceau, G., Fisler, K., and Krishnamurthi, S. (2011b). Mind your language: On novices' interactions with error messages. In *Proceedings of the 10th SIGPLAN Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2011, pages 3–18, New York, NY, USA. ACM.

[Matrix, 2014] Matrix, S. (2014). The netflix effect: Teens, binge watching, and on-demand digital media trends. *Jeunesse: Young People, Texts, Cultures*, 6(1):119–138.

[McCabe, 1976] McCabe, T. J. (1976). A complexity measure. *IEEE Trans. Softw. Eng.*, 2(4):308–320.

[McIver, 2000] McIver, L. (2000). The effect of programming language on error rates of novice programmers.

[McMillan et al., 2011] McMillan, C., Grechanik, M., Poshyvanyk, D., Xie, Q., and Fu, C. (2011). Portfolio: finding relevant functions and their usage. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 111–120.

[McMillan et al., 2012] McMillan, C., Hariri, N., Poshyvanyk, D., Cleland-Huang, J., and Mobasher, B. (2012). Recommending source code for use in rapid software prototypes. In *Proceedings of the 34th International Conference on Software Engineering*, pages 848–858.

[Mechaber, 2014] Mechaber, E. (2014). President obama is the first president to write a line of code. `https://obamawhitehouse.archives.gov/blog/2014/12/10/president-obama-first-president-write-line-code`. Accessed: 2017-11-28.

[Miller, 1956] Miller, G. A. (1956). The magical number seven, plus or minus two: some limits on our capacity for processing information. *Psychological review*, 63(2):81.

[Miller et al., 2008] Miller, R. C., Chou, V. H., Bernstein, M., Little, G., Van Kleek, M., Karger, D., and schraefel, m. (2008). Inky: a sloppy command line for the web with rich visual feedback. In *Proceedings of the 21st Annual ACM Symposium on User Interface Software and Technology*, pages 131–140.

[Misherghi and Su, 2006] Misherghi, G. and Su, Z. (2006). Hdd: Hierarchical delta debugging. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pages 142–151, New York, NY, USA. ACM.

[Monperrus, 2014] Monperrus, M. (2014). A critical review of "automatic patch generation learned from human-written patches": Essay on the problem statement and the evaluation of automatic software repair. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 234–242, New York, NY, USA. ACM.

[Mujumdar et al., 2011] Mujumdar, D., Kallenbach, M., Liu, B., and Hartmann, B. (2011). Crowdsourcing suggestions to programming problems for dynamic web development languages. In *CHI '11 Extended Abstracts on Human Factors in Computing Systems*, CHI EA '11, pages 1525–1530, New York, NY, USA. ACM.

[Munson, 2017] Munson, J. P. (2017). Metrics for timely assessment of novice programmers. *J. Comput. Sci. Coll.*, 32(3):136–148.

[Nguyen et al., 2012] Nguyen, A. T., Nguyen, T. T., Nguyen, H. A., Tamrawi, A., Nguyen, H. V., Al-Kofahi, J., and Nguyen, T. N. (2012). Graph-based pattern-oriented, context-sensitive source code completion. In *Proceedings of the 34th International Conference on Software Engineering*, pages 69–79.

[Nguyen et al., 2013] Nguyen, T. T., Nguyen, A. T., Nguyen, H. A., and Nguyen, T. N. (2013). A statistical semantic language model for source code. In *Proceedings of the 9th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*.

[Nienaltowski et al., 2008] Nienaltowski, M.-H., Pedroni, M., and Meyer, B. (2008). Compiler error messages: What can help novices? In *Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '08, pages 168–172, New York, NY, USA. ACM.

[Omar et al., 2017] Omar, C., Voysey, I., Hilton, M., Aldrich, J., and Hammer, M. (2017). Hazelnut: A Bidirectionally Typed Structure Editor Calculus. In *44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2017)*.

[Oracle, 2012] Oracle (2012). Oracle openJDK. `http://openjdk.java.net/`.

[Pappas et al., 2016] Pappas, I. O., Giannakos, M. N., and Jaccheri, L. (2016). Investigating factors influencing students' intention to dropout computer science studies. In *Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education*, ITiCSE '16, pages 198–203, New York, NY, USA. ACM.

[Paterson and Haddow, 2007] Paterson, J. H. and Haddow, J. (2007). Tool Support for Implementation of Object-Oriented Class Relationships and Patterns. *Innovation in Teaching and Learning in Information and Computer Sciences*, 6(4):108–124.

[Paterson et al., 2005] Paterson, J. H., Haddow, J., Birch, M., and Monaghan, A. (2005). Using the bluej ide in a data structures course. In *Proceedings of the 10th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education*, ITiCSE '05, pages 349–349, New York, NY, USA. ACM.

[Pettit et al., 2017] Pettit, R. S., Homer, J., and Gee, R. (2017). Do enhanced compiler error messages help students?: Results inconclusive. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '17, pages 465–470, New York, NY, USA. ACM.

[Pu et al., 2016] Pu, Y., Narasimhan, K., Solar-Lezama, A., and Barzilay, R. (2016). sk_p: A neural program corrector for moocs. In *Companion Proceedings of the 2016 ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity*, SPLASH Companion 2016, pages 39–40, New York, NY, USA. ACM.

[Regehr et al., 2012] Regehr, J., Chen, Y., Cuoq, P., Eide, E., Ellison, C., and Yang, X. (2012). Test-case reduction for c compiler bugs. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 335–346, New York, NY, USA. ACM.

[Reiss, 2009a] Reiss, S. P. (2009a). Semantics-based code search. In *Proceedings of the 31st International Conference on Software Engineering*, pages 243–253.

[Reiss, 2009b] Reiss, S. P. (2009b). Specifying what to search for. In *Proceedings of the 2009 ICSE Workshop on Search-Driven Development-Users, Infrastructure, Tools and Evaluation*, pages 41–44.

[Rivest, 1998] Rivest, R. (1998). Chaffing and winnowing: Confidentiality without encryption. web page.

[Rolim et al., 2017] Rolim, R., Soares, G., D'Antoni, L., Polozov, O., Gulwani, S., Gheyi, R., Suzuki, R., and Hartmann, B. (2017). Learning syntactic program transformations from examples. In *Proceedings of the 39th International Conference on Software Engineering*, ICSE '17, pages 404–415, Piscataway, NJ, USA. IEEE Press.

[Runnable, 2017] Runnable (2017). Runnable.

[Schleimer et al., 2003] Schleimer, S., Wilkerson, D. S., and Aiken, A. (2003). Winnowing: Local algorithms for document fingerprinting. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, SIGMOD '03, pages 76–85, New York, NY, USA. ACM.

[Schuler et al., 2007] Schuler, D., Dallmeier, V., and Lindig, C. (2007). A dynamic birthmark for Java. In *Proceedings of the International Conference on Automated Software Engineering*, pages 274–283.

[Schulte and Bennedsen, 2006] Schulte, C. and Bennedsen, J. (2006). What do teachers teach in introductory programming? In *Proceedings of the Second International Workshop on Computing Education Research*, ICER '06, pages 17–28, New York, NY, USA. ACM.

[Seo et al., 2014] Seo, H., Sadowski, C., Elbaum, S., Aftandilian, E., and Bowdidge, R. (2014). Programmers' build errors: A case study (at google). In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 724–734, New York, NY, USA. ACM.

[Shah, 2014] Shah, D. (2014). Online courses raise their game: A review of mooc stats and trends in 2014. `https://www.class-central.com/report/moocs-stats-and-trends-2014/`. Accessed: 2017-11-28.

[Siegmund et al., 2014] Siegmund, J., Kästner, C., Liebig, J., Apel, S., and Hanenberg, S. (2014). Measuring and modeling programming experience. *Empirical Software Engineering*, 19(5):1299–1334.

[Singh et al., 2013] Singh, R., Gulwani, S., and Solar-Lezama, A. (2013). Automated feedback generation for introductory programming assignments. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 15–26, New York, NY, USA. ACM.

[Society, 2018] Society, T. A. (2018). Six-figure tech salaries: Creating the next developer workforce.

[Soper, 2014] Soper, T. (2014). Analysis: The exploding demand for computer science education, and why america needs to keep up. `https://www.geekwire.com/2014/analysis-examining-computer-science-education-explosion/`. Accessed: 2017-11-28.

[Strasburger et al., 2013] Strasburger, V. C., Hogan, M. J., Mulligan, D. A., Ameenuddin, N., Christakis, D. A., Cross, C., Fagbuyi, D. B., Hill, D. L., Levine, A. E., McCarthy, C., et al. (2013). Children, adolescents, and the media. *Pediatrics*, 132(5):958–961.

[Toomey, 2011] Toomey, W. (2011). Bluej with modified error subsystem. `http://minnie.tuhs.org/Programs/BlueJErrors/index.html`. Accessed: 2017-09-18.

[Torres, 2018] Torres, C. (2018). Demand for programmers hits full boil as u.s. job market simmers.

[Traver, 2010] Traver, V. J. (2010). On compiler error messages: What they *Say* and what they *Mean*. *Adv. Human-Computer Interaction*, 2010:602570:1–602570:26.

[van Deursen et al., 2010] van Deursen, A., Mesbah, A., Cornelissen, B., Zaidman, A., Pinzger, M., and Guzzi, A. (2010). Adinda: A knowledgeable, browser-based ide. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 2*, ICSE '10, pages 203–206, New York, NY, USA. ACM.

[van Tonder et al., 2008] van Tonder, M., Naude, K., and Cilliers, C. (2008). Jenuity: A lightweight development environment for intermediate level programming courses. In *Proceedings of the 13th Annual Conference on Innovation and Technology in Computer Science Education*, ITiCSE '08, pages 58–62, New York, NY, USA. ACM.

[Wang et al., 2017a] Wang, K., Lin, B., Rettig, B., Pardi, P., and Singh, R. (2017a). Data-driven feedback generator for online programing courses. In *Proceedings of the Fourth (2017) ACM Conference on Learning @ Scale*, L@S '17, pages 257–260, New York, NY, USA. ACM.

[Wang et al., 2018] Wang, K., Singh, R., and Su, Z. (2018). Search, align, and repair: Data-driven feedback generation for introductory programming exercises. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, New York, NY, USA. ACM.

[Wang et al., 2017b] Wang, L., Sy, A., Liu, L., and Piech, C. (2017b). Deep Knowledge Tracing On Programming Exercises. In *Proceedings of the Fourth (2017) ACM Conference on Learning @ Scale - L@S '17*, pages 201–204, New York, New York, USA. ACM Press.

[Watson and Li, 2014] Watson, C. and Li, F. W. (2014). Failure rates in introductory programming revisited. In *Proceedings of the 2014 Conference on Innovation &#38; Technology in Computer Science Education*, ITiCSE '14, pages 39–44, New York, NY, USA. ACM.

[Weimer et al., 2009] Weimer, W., Nguyen, T., Le Goues, C., and Forrest, S. (2009). Automatically finding patches using genetic programming. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 364–374, Washington, DC, USA. IEEE Computer Society.

[Whitney et al., 2015] Whitney, M., Lipford-Richter, H., Chu, B., and Zhu, J. (2015). Embedding secure coding instruction into the ide: A field study in an advanced cs course. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*, SIGCSE '15, pages 60–65, New York, NY, USA. ACM.

[Yi et al., 2017] Yi, J., Ahmed, U. Z., Karkare, A., Tan, S. H., and Roychoudhury, A. (2017). A feasibility study of using automated program repair for introductory programming assignments. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, pages 740–751, New York, NY, USA. ACM.

[Zeller and Hildebrandt, 2002] Zeller, A. and Hildebrandt, R. (2002). Simplifying and isolating failure-inducing input. *IEEE Trans. Softw. Eng.*, 28(2):183–200.

[Zhang et al., 2012] Zhang, C., Yang, J., Zhang, Y., Fan, J., Zhang, X., Zhao, J., and Ou, P. (2012). Automatic parameter recommendation for practical API usage. In *Proceedings of the 34th International Conference on Software Engineering*, pages 826–836.

[Zhu et al., 2013] Zhu, J., Lipford, H. R., and Chu, B. (2013). Interactive support for secure programming education. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education*, SIGCSE '13, pages 687–692, New York, NY, USA. ACM.