Study of Program Synthesizers & Novice Programmers

Dhanya Jayagopal
Justin Lubin
Sarah Chasins

Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2022-59
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2022/EECS-2022-59.html

May 11, 2022

Acknowledgement

I would first like to thank my advisor, Professor Sarah Chasins whose expertise was invaluable in formulating the entirety of both projects covered in the paper. Your attention to detail, general insight about the field, and consistent encouragement pushed me to produce my best work.

I would also like to thank my PhD mentor, Justin Lubin whose dedication and commitment contributed so greatly to my process every single step of the way. Your willingness to always go above and beyond even when it meant helping me on top of your own work helped me feel so enthusiastic about my projects.

I would also like to thank my family and friends for their unwavering support.

Study of Program Synthesizers & Novice Programmers

by Dhanya Jayagopal

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

Committee:

Sarah E. Chasins

Professor Chasins
Research Advisor

May 13th, 2022

* * * * * * *

[illegible]

Professor Hartmann
Second Reader

May 13th, 2022

[illegible]

Professor Hearst
Second Reader

May 13th, 2022

[illegible]

Professor Parameswaran
Second Reader

May 13th, 2022

Study of Program Synthesizers & Novice Programmers

by

Dhanya Jayagopal

Submitted in partial fulfillment of the requirements for the degree of

Master of Science

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Berkeley, California
May 2022

Contents

Chapter 1

Introduction

Novice programmers face many challenges in introductory classes, which leads to high dropout rates. Meanwhile, modern program synthesizers have progressed a lot—they now deliver on their promise of lightening the burden of programming by automatically generating code. This has led many researchers to hypothesize synthesis may have a role to play in introductory classes.

However, there is little research on how to make these systems learnable not only by seasoned programmers, but also by novices. To explore the possibility of using program synthesizers in educational contexts, we must understand what makes program synthesizers learnable for novices in the first place. Furthermore, we must also understand how synthesizers affect student learning outcomes.

Both chapters in this thesis focus on the application of program synthesizers in educational contexts. In Chapter 2, we present an in-depth analysis of the learnability of different publicly available program synthesizers by novice programmers with the goal of understanding how different design dimensions affect the learnability of such tools. In Chapter 3, we provide an overview of existing literature on the pedagogical impact of program synthesizers—or, rather, lack thereof. Also in Chapter 3, we contribute a user study done to evaluate the impact of our own custom program synthesizer on student learning outcomes and do not find any evidence of improved learning outcomes.

This project is joint work with Justin Lubin and Sarah Chasins, and each chapter lists my specific contributions at the start.

Chapter 2

The Learnability of Program Synthesizers by Novice Programmers

We will first discuss our qualitative study done on novice programmers which evaluates the learnability of different program synthesis tools.

Description of contributions. Dhanya Jayagopal designed and carried out the experiment and performed the qualitative coding. All authors discussed results and contributed equally to writing.

Abstract

Modern program synthesizers are increasingly delivering on their promise of lightening the burden of programming by automatically generating code, but little research has addressed how we can make such systems learnable to all. In this work, we ask: What aspects of program synthesizers contribute to and detract from their learnability by novice programmers? We conducted a thematic analysis of 22 observations of novice programmers, during which novices worked with existing program synthesizers, then participated in semi-structured interviews. Our findings shed light on how specific points in the synthesizer design space affect these tools' learnability by novice programmers, including the type of specification the synthesizer requires, the method of invoking synthesis and receiving feedback, and the size of the specification. We also describe common misconceptions about what constitutes meaningful progress and useful specifications for the synthesizers, as well as participants' common behaviors and strategies for using these tools. We derive a set of implications to inform the design of future program synthesizers that strive to be learnable by novice programmers. This work serves as a first step toward understanding how we can make program synthesizers more learnable by novices, which opens up the possibility of using program synthesizers in educational settings as well as developer tooling oriented toward easing novice programmer experiences.

2.1 Introduction

The promise of *program synthesis* is to lighten the burden of programming by automatically generating code that satisfies a user-given specification. However, little work has studied how novice programmers learn and use synthesis tools. Our work draws on observations of early-stage programmers and identifies synthesizer design dimensions that affect synthesizer learnability. The end goal is to inform design guidelines so that the community can make synthesizers more approachable and ultimately boost their impact on a broader class of users.

We observed 22 novice programmers using five existing program synthesis tools (Blue-Pencil [58], Copilot [28], Flash Fill [30], Regae [90], and SnipPy [21]) and followed each session with a semi-structured interview.

We identified a number of influential design dimensions. One such dimension is that synthesizers can (i) require users to engage in a separate synthesis-specific specification mode or (ii) derive a specification as a byproduct of normal non-synthesis tool use. Another important dimension was whether users were in charge of triggering synthesis runs and the display of synthesis outputs or whether the tool was in charge. The size of the specification also mattered, but seemingly not as much other dimensions—a surprising finding in light of popular design guidelines from the synthesis algorithm literature, which strongly emphasizes specification size.

We also identified important user knowledge gaps and common strategies. Novices struggle with plan composition during synthesis in much the same way as during manual coding. Novice programmers struggle to figure out what kinds of specifications work well for a given synthesis tool. For synthesis tools embedded in familiar environments, novice programmers may also borrow behaviors from their pre-synthesizer use. Finally, novice programmers may engage more deeply with synthesis-written programs relative to teacher-written programs provided as exercise solutions.

Based on our findings, we derive a set of implications to inform the design of future program synthesizers that aim to be learnable by novices.

No element of this chapter is intended as an evaluation of the tools used in the study. In particular, we note that none of the tools we used in this study are explicitly designed for learnability by novice programmers. Rather, we chose a stable of tools that exhibit different design choices across synthesis algorithms, interfaces, and user interaction models as a means to uncover patterns in how these design choices affect users.

Contributions. This chapter presents the following contributions:

- An **observational study of novice programmers using program synthesis tools** for the first time.
- An analysis that identifies the **key synthesizer design dimensions that affect learnability**; a preliminary analysis of particular design decisions that improve or reduce synthesizer learnability.
- A set of **implications for synthesizer designers**, to guide designers towards choices that make synthesis tools learnable by novice programmers.

2.2 Background: Program Synthesis

Program synthesis is the automatic generation of code based on some *specification* that communicates user intent. This expression of user intent can take many forms, including, for example, a logical formula, input-output examples, or task demonstrations. The term specification is typically used to mean a *formal specification*—an objective, machine-checkable predicate that gives yes or no answers about whether a given program meets the specification. For instance, we can automatically check whether a given program meets the logical specification $output = 2 * input$ or the input-output example specification $[3 \rightarrow 6, 5 \rightarrow 10]$. For this thesis, we use a broader definition of specification that includes any expression of user intent to a synthesizer. In particular, the input to Copilot is a large portion of the content of an in-progress file in an IDE. Although this input does not offer a way of automatically and non-subjectively checking whether a synthesized program "meets" this specification, we will still use the term specification.

2.3 Related Work

2.3.1 Learnability & Usability of Synthesis Tools

Recent work in the programming languages community and a variety of other subfields has produced tremendous advances in program synthesis technology [63, 30, 43, 66, 76, 45, 22, 3, 31, 55, 73, 33]. Synthesis approaches are now sufficiently mature that we are seeing them adopted within HCI [41, 85, 50, 10, 80, 18, 57, 36] and integrated into mainstream products [72, 25, 88, 60]. Although the early wave of program synthesis works from the programming languages community largely did not assess synthesizers' learnability or usability, the more recent wave of synthesis-augmented interfaces within the HCI community has started to offer answers about how humans interact with synthesizers.

Existing user studies of program synthesis tools fall into a few categories. Many works compare a novel synthesis tool against a manual programming condition [26, 10, 85, 40, 21]. These studies typically find that participants are faster—sometimes much faster—with synthesis than with manual coding, that they make fewer errors, that they are more likely to complete tasks, or some subset of these. Others focus exclusively on one novel synthesizer, with no control condition, and assess whether users can complete tasks with the tool at all [51, 83, 20]. These studies typically find that participants can complete tasks with the synthesizer under test.

Comparison of synthesizer variants. More recently we have begun to see user studies that test a novel synthesizer against a tweaked versions of the same synthesizer. Such studies begin to answer a question similar to ours: what *aspects* of program synthesizers affect their usability? To date, we know of two such studies.

The first of these studies [65] compares how successful participants were using (i) a traditional programming-by-example synthesizer in which users provide input-output examples versus (ii) a variant of the synthesizer using a paradigm the authors dub "programming not only by example" that also allows users to select sub expressions of a candidate program to either keep or reject. The authors found that the subexpression selection mechanism was faster to use than standard input-output examples and strongly preferred to examples overall except for with expert users.

The second of these studies (which investigates one of the tools we use in our study, Regae) [90] compares how successful participants were using (i) the second condition of the above study versus (ii) a further enhancement on the programming-by-example paradigm that allows users to mark parts of their regex as either "general" or 'literal" (so-called *semantic augmentation*) as well as receive automatic example generation of additional input examples (so-called *data augmentation*). The authors found that participants were significantly faster in the second condition.

User studies of synthesizers in our study. We highlight in particular that three of the five tools in our protocol have already been studied in published user studies: Regae (as we discussed in previous subsection), SnipPy, and Blue-Pencil.

The authors of SnipPy ran their user study [21] as a controlled experiment between (i) a live programming environment with no synthesis features and (ii) the same live programming environment with SnipPy activated. Neither correctness nor speed differences were statistically significant between the two groups, but the authors did find that when the participants broke the task down in a way that the synthesizer understood, the synthesizer produced substantial and helpful parts of the overall solution.

The authors of Blue-Pencil did a field study [58] that included follow-up interviews to collect qualitative feedback of their tool in an uncontrolled environment. Their study revealed two key usability barriers: (i) low discoverability and (ii) frustration with lacking a mechanism to preview the effects of running the synthesized code before accepting or rejecting it.

Our approach. All the aforementioned studies of synthesizer usability and learnability are highly relevant to the questions we address in our work. However, in contrast to the existing foundational studies in this space—which explore either one synthesis tool at a time, one synthesis tool and one traditional programming language, or two very close variants of a single synthesizer—our work draws on observations of participants using five diverse synthesis tools. Comparing across a variety of divergent tools lets us explore a somewhat broader swath of the synthesis design space and situate our observations in the broader landscape of that design space.

2.3.2 Interactive Program Synthesis

Recent work in the programming languages and program synthesis community has begun to emphasize *interactive program synthesis*. Within the program synthesis community, interactive program synthesis usually refers to a synthesizer in which either (i) if the user updates an existing specification, the synthesis algorithm works differently than if the user had provided this specification initially, or (ii) the synthesis tool can guide the user in how to augment an ambiguous specification. For instance, tools may show partially completed programs at multiple stages during synthesis and request user feedback before completing more [4], or they may show users examples of program behaviors and ask if they are correct [27]. Some recent work even presents a framework for reasoning about how user inputs can be used to reduce specification ambiguity [64]. More than the first wave of program synthesis research, this recent wave is wrestling with the role of human users in the synthesis process and recognizing synthesis as an interactive human-machine collaboration. However, the work in this space still currently emphasizes advances in algorithm design or performance over advances in usability; user studies are not currently more common in the space of interactive program synthesis techniques than in the more traditional techniques.

2.3.3 Learnability & Usability of Novice-Focused Programming Environments

Research on programming environments has studied novice programmers to explore the learnability of a variety of tools from Codeacademy[1]-style tutorial systems with built-in programming environments [61, 74] to inquisitive IDEs that prompt users with questions about program behavior during development [37]. Because program synthesizers often display synthesized programs in a specialized or general-purpose programming environment, many of the key insights from existing work on programming environment usability can be directly applied to the programming environment elements of program synthesizers. In this study, we focus specifically on the aspects of the program synthesizers themselves that impact their learnability by novice programmers.

2.3.4 Learnability & Usability of AI Tools

With at least two decades of work in the space of human-AI interaction, there has been substantial work that addresses the learnability and usability of AI tools. Some works offer guidelines for improving human-AI collaboration [2, 38, 39, 62, 46, 5, 52]; others detail the usability or learnability of particular classes of AI tools [54, 69, 89, 23, 59]. We are not aware of any usability studies to date from the human-AI interaction community that investigate the usability of AI tools that produce computer code.

[1]`https://www.codecademy.com/`

2.3.5 Think-Aloud and Observational Studies with Novices

The observational think-aloud design used in our work connects closely with prior studies [49, 86, 9, 56] that use observations of novice programmers to uncover patterns in a variety of processes, from novices' plan composition strategies to how they track program state. Although existing observational studies of novice programmers have yielded important insights in CS education [49, 86, 9] as well as language and IDE design [56], this approach has not yet been applied to synthesis-augmented programming environments.

2.3.6 Connections to General Interface Design Principles

Several of the themes that rose to the surface in our analysis echo existing design guidelines from the broader space of interface design in general. For instance, we observed that participants struggled with synthesis interaction models that demanded mode switching, which echoes existing findings across many kinds of user interfaces [17]. For connections like this, to existing work that relates to a specific takeaway from our findings, we describe relevant works directly in the analysis section.

2.4 Method

Participants and Recruitment. We conducted virtual study sessions with 22 participants (with identifiers P0–P21), all of whom were enrolled in a second-semester computer science course on data structures at an R1 university in the United States. We screened participants (recruited from Piazza, Facebook groups, and Slack workspaces) via a survey for little to no external programming experience beyond their first semester of computer science.

Study Protocol. Study sessions consisted of one recorded Zoom call between the author of this thesis and the participant. These calls lasted approximately 45 minutes to 1 hour and were divided into two sections. In the first section, participants narrated their thought processes aloud as they used various program synthesizers to complete tasks that we assigned. The tasks that we assigned depended on the program synthesis tools that the participants used; we describe both the program synthesis tools and the tasks that we assigned with them in Section 2.5. We selected tools to be shown to each participant at random such that an equal number of participants were spread out across the five different tools. In the second section of the study, the investigator asked participants in a semi-structured interview to talk about their experience with the program synthesis tools that they used, and to elaborate on certain topics that came up during their use of the tools.

Analysis. We used reflexive thematic analysis [7] to analyze the recorded sessions of novice programmers working with program synthesizers. The first author reviewed each recorded session and tagged any relevant participant actions or narration with a low-level textual summary. After accumulating many such summary tags, the first author grouped these tags into loose categorizations (the beginnings of the themes in this chapter), and re-watched existing footage to annotate any missing occurrences of these categorizations. This inductive, open coding process continued, with the themes becoming more concrete and robust over time via discussion with the other collaborators on this work until arriving at the set of themes that form the subsection headers in Section 2.6. Finally, all authors collaborated to explore and refine these themes even further while the first author reviewed the recorded session for any additional evidence and nuance for the final themes.

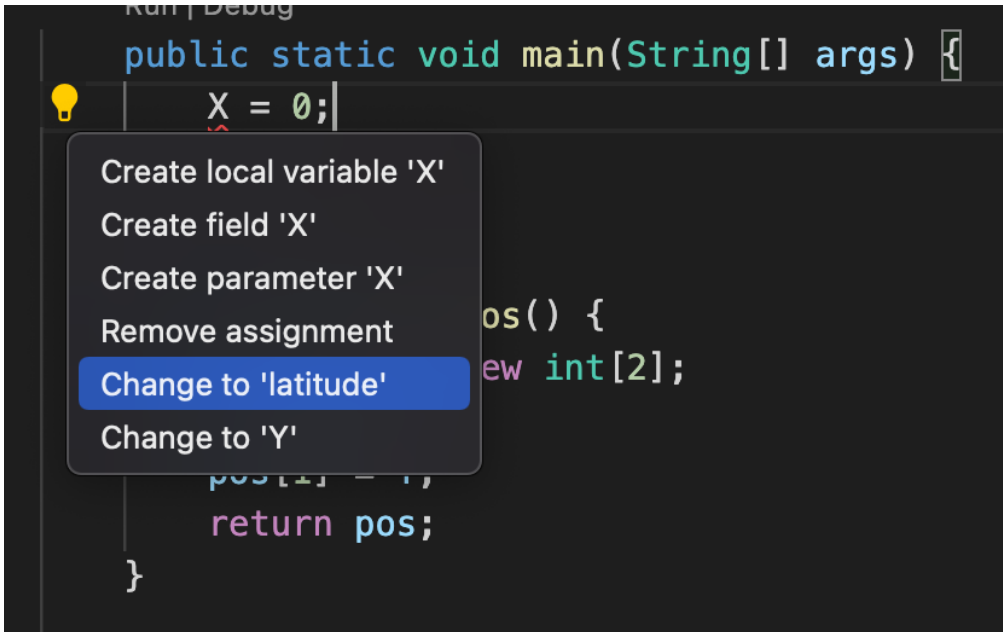
2.5 Tools and Tasks

Table 2.1: *Synthesis tasks for the study sessions.* For a given synthesizer, we assigned each task sequentially according to task number, and each subsequent task is approximately more difficult than the previous. In the rest of this chapter, we refer to each of the tasks in this table by the identifier SYNTHESIZER-NAME-#.

SYNTHESIZER	NAME	#	DESCRIPTION
BLUE-PENCIL	Point	1	Change the program to use `Point` objects to represent position rather than a pair of integers `(x, y)`.
BLUE-PENCIL	Rename	2	Change the name of variable `X` to `latitude`. Change the name of variable `Y` to `longitude`.
BLUE-PENCIL	LinkedList	3	Change a list to be a `LinkedList` (built-in Java class) instead of a primitive array.
COPILOT	Abbreviate	1	Write a program to return the abbreviation of a given name.
COPILOT	Occurrences	2	Write a program to turn a list into a dictionary that counts the number of occurrences of each character.
COPILOT	Subsequence	3	Write a program to find the length of the longest subsequence of a given sequence such that all elements of the subsequence are sorted in increasing order. For example, the output for `[10, 22, 9, 33, 21, 50, 41, 60, 80]` should be `6` because the longest sorted subsequence is `[10, 22, 33, 50, 60, 80]`.
FLASH FILL	Names	1	Using the data from Column A, populate Column B with `<Last Name>, <First Name>` and populate Column C with `<First Initial><Last Name>` in lower case.
FLASH FILL	Emails	2	Populate Column B with the prefixes of the email addresses in Column A.
FLASH FILL	Characters	3	Populate Column B with all of the upper case letters from Column A. Populate Column C with all of the lower case letters from Column A. Populate Column D with all of the numbers from Column A.
REGAE	Plus	1	Write a regular expression that accepts strings that contain `+` or digits but no `++`.
REGAE	ABC	2	Write a regular expression that accepts strings that only have `A`, `B`, `C`, or any combinations of them.
REGAE	Phone	3	Write a regular expression that accepts phone numbers that start with one optional `+` symbol and follow with a sequence of digits. For example, `+91` and `91`, but not `91+`.
SNIPPY	Abbreviate	1	Write a program to return the abbreviation of a given name.
SNIPPY	Reverse	2	Write a program to reverse a given string.
SNIPPY	Filter	3	Write a program to return a given string without a specified letter.

We selected a range of program synthesis tools for our participants to use based on two main criteria: (i) the tool could plausibly support novice programmers, and (ii) the tool is publicly available and functional in their publicly available form. We selected five such tools based on the additional goal of exploring a diversity of program synthesis algorithms and interfaces. In this section, we provide details about the five tools that we selected: FLASH FILL, BLUE-PENCIL, REGAE, SNIPPY, and COPILOT.

We also provide details about the tasks that we assigned participants who used each tool in Table 2.1. In most cases, we designed the tasks to be similar to the tasks that were used as examples in the associated publications for each of the tools.



```
    public static void main(String[] args) {
        X = 0;
                                   os() {
                                   ew int[2];

        return pos;
    }
```

Figure 2.1: *Visual Studio Code with the* Blue-Pencil *plugin enabled.* The light bulb on the left appears when the plugin detects repetitive edits; when clicked, Blue-Pencil provides the user with suggestions to automate such edits.

2.5.1 Blue-Pencil

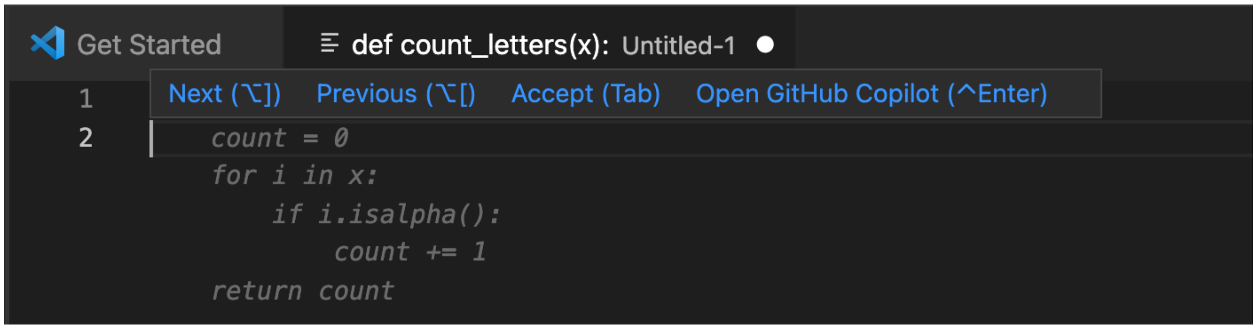
Blue-Pencil [58] is a plugin for Microsoft's Visual Studio Code that aims to automate repetitive code edits, such as changing the name of a variable. The Blue-Pencil team describes the synthesizer as *modeless*, meaning that users do not explicitly enter a distinct mode to author a specification for the synthesize.

- **Programming Domain**: Code transformations.
- **Synthesizer Input**: A trace of the user's code edit actions.
- **Synthesizer Output**: A program for automating the detected repetitive program edits.
- **Communicating the Input to the Tool**: The user uses Visual Studio Code as normal to edit their code. The tool collects the trace of the user's actions transparently and automatically, running synthesis in the background.
- **Communicating the Output to the User**: When the synthesizer has identified a repetitive edit, the tool adds a light bulb icon to the sidebar. If the user clicks the icon, they see a drop-down menu of suggested changes.
- **Communicating Failure to User**: The synthesizer does not display anything if it has no suggestion to provide.

We informed Blue-Pencil participants that Visual Studio Code might display a light bulb icon as they wrote code and explained that clicking the icon would give access to Blue-Pencil's synthesized code. Figure 2.1 shows an example of the Blue-Pencil interface, including its light bulb indicator.

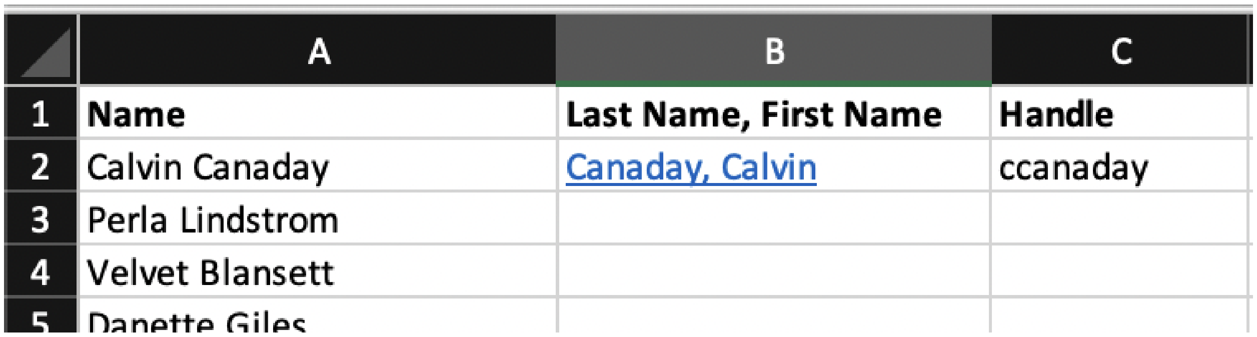
2.5.2 Copilot

GitHub Copilot [28] is an editor plugin powered by OpenAI's Codex model [11] that automatically synthesizes code on the fly based on the code and comments the user has already written. We had participants use Copilot within the Visual Studio Code editor.


Get Started
def count_letters(x): Untitled-1
Next (⌥])
Previous (⌥[)
Accept (Tab)
Open GitHub Copilot (^Enter)

```
count = 0
for i in x:
    if i.isalpha():
        count += 1
return count
```

Figure 2.2: *The* Copilot *autosuggestion interface.* The most likely—as determined by Copilot—block of code that the user might want is shown in gray. The user can see additional suggestions one at a time by pressing the tab key or opening the Copilot tab to see them all at once.

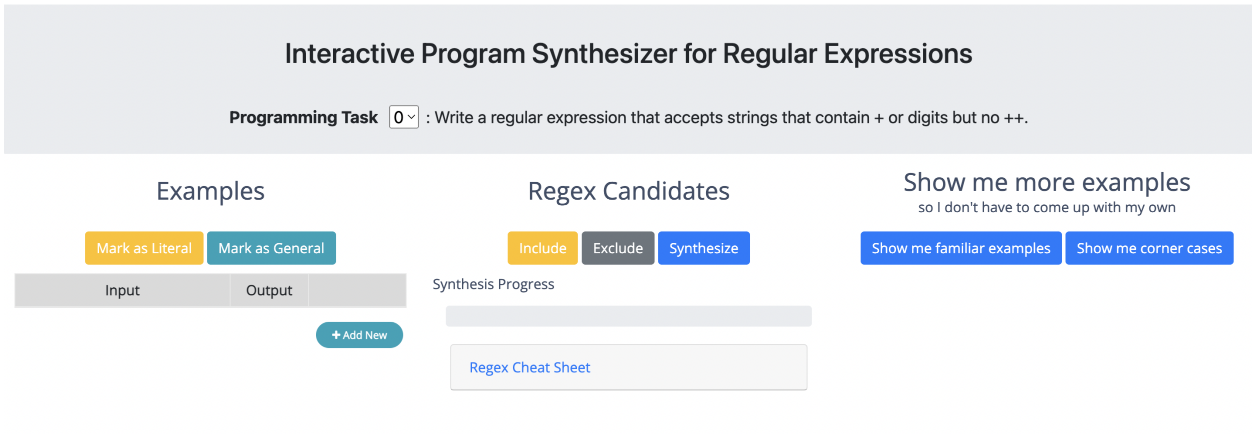


	A	B	C
1	Name	Last Name, First Name	Handle
2	Calvin Canaday	Canaday, Calvin	ccanaday
3	Perla Lindstrom		
4	Velvet Blansett		
5	Danette Giles		

Figure 2.3: *The first* Flash Fill *task,* FlashFill-Names-1*.* We asked participants to fill Column B with the last and first names of the entries in the Column A separated by a comma and a space. We additionally asked participants to fill Column C with the first initial and last name of the entries in Column A (all in lowercase).

- **Programming Domain**: Arbitrary programs.
- **Synthesizer Input**: The text of a document; for example, part of a program that the user has written so far.
- **Synthesizer Output**: A program.
- **Communicating the Input to the Tool**: The user writes code normally using their editor. The tool collects the document's content transparently and automatically, running synthesis in the background.
- **Communicating the Output to the User**: When the synthesizer has identified a plausible program, the tool presents the program as a suggestion to the user in grayed-out text as a form of autocomplete. The user can either accept the suggestion, request another suggestion, or open a separate pane to view multiple suggestions at once.
- **Communicating Failure to the User**: The synthesizer does not display anything if it has no suggestion to provide.

When introducing Copilot to participants, we did not describe or demonstrate the tool. Participants were told that there was a synthesis tool available that they could use for the tasks.


Interactive Program Synthesizer for Regular Expressions

Programming Task 0 : Write a regular expression that accepts strings that contain + or digits but no ++.

Figure 2.4: *The web interface for Regae.* The left column is where the user can add and annotate regex examples. The middle column is where the user can trigger the synthesizer as well as where the tool displays its results. The right column is where the tool shows additional examples to the user.

2.5.3 Flash Fill

Flash Fill [30] is a feature of Microsoft Excel that uses programming-by-example to automatically populate data cells based on existing patterns in other data cells. For example, Flash Fill can separate first and last names from a single column into two or vice versa.

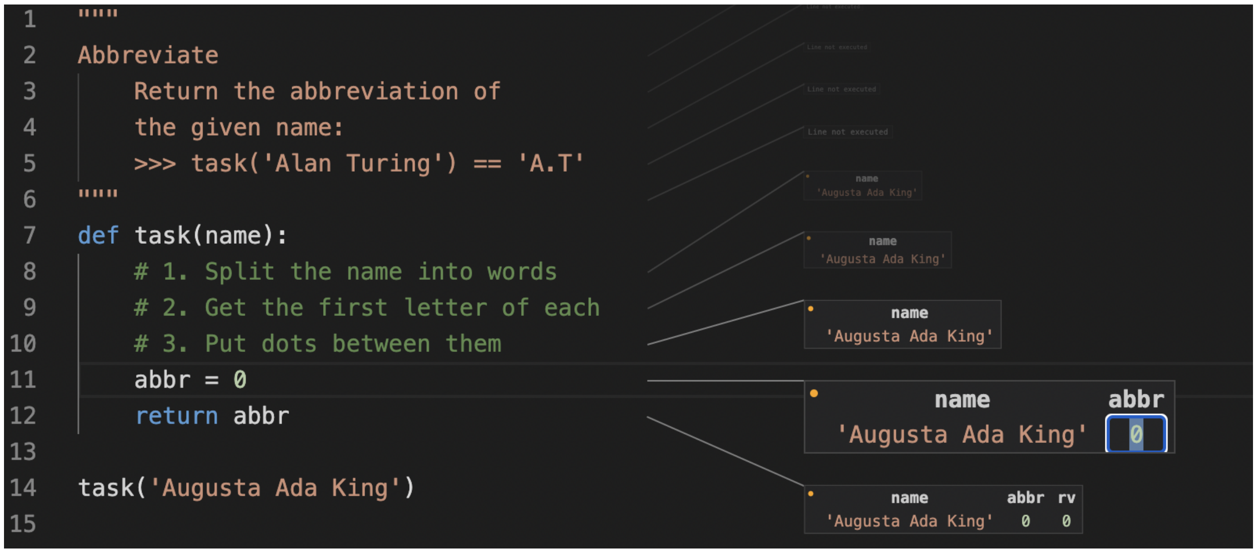
- **Programming Domain**: String transformations.
- **Synthesizer Input:** Input-output pairs of strings.
- **Synthesizer Output:** A string transformation program that transforms each example input into its output.
- **Communicating the Input to the Tool:** Given a column of "inputs," the user begins to fill an adjacent empty column that serves as the "outputs" for the input-output pairs, then clicks the Flash Fill button.
- **Communicating the Output to the User:** The tool fills the remaining cells of the output column by running the string transformation program synthesized from the user-provided input-output pairs.
- **Communicating Failure to the User**: The synthesizer displays the following pop-up message: "We looked at all the data next to your selection and didn't see a pattern for filling in values for you. To use Flash Fill, enter a couple of examples of the output you'd like to see, keep the active cell in the column you want filled in and click the Flash Fill button again."

When introducing Flash Fill to participants, we directed them only to the location of the Flash Fill button; participants did not receive any further instructions or demos. Figure 2.3 shows part of the spreadsheet that we gave the participants for the first task; the remaining tasks had similar starting spreadsheets.

2.5.4 Regae

Regae [90] is a standalone tool that implements programming-by-example synthesis for regular expressions (regexes).

- **Programming Domain**: Regular expressions.



```
"""
Abbreviate
    Return the abbreviation of
    the given name:
    >>> task('Alan Turing') == 'A.T'
"""
def task(name):
    # 1. Split the name into words
    # 2. Get the first letter of each
    # 3. Put dots between them
    abbr = 0
    return abbr

task('Augusta Ada King')
```

```
Line not executed
Line not executed
Line not executed
Line not executed
name
'Augusta Ada King'
name
'Augusta Ada King'
name
'Augusta Ada King'
name                abbr
'Augusta Ada King'  0
name                abbr  rv
'Augusta Ada King'  0     0
```

Figure 2.5: *The SnipPy interface with visible projection boxes.* The pop-up projection boxes dynamically show the values of variables at points in the program and let the user provide corresponding outputs to form input-output pairs.

- **Synthesizer Input**: (i) A set of strings that should be accepted by the regular expression. (ii) A set of strings that should *not* be accepted by the regular expression. (iii) Labels applied to substrings of the accepted strings, marking whether the substring should be interpreted as a literal or as something generalizable. (iv) A set of regular expression subexpressions labeled as being desirable or undesirable for inclusion in the overall synthesis.
- **Synthesizer Output**: A regular expression satisfying the user-provided examples.
- **Communicating the Input to the Tool**: In a custom REGAE-specific user interface, the user enters strings that should be accepted or rejected; in this same pane, the user can mark substrings that should be literal or general. In a second pane, the tool displays any candidate regexes, and the user can mark sub-expressions that are desirable or undesirable. In a third pane, the user can indicate whether a set of tool-suggested strings should be accepted or rejected.
- **Communicating the Output to the User**: After an initial round of input, the tool displays a set of candidate regexes.
- **Communicating Failure to the User**: The synthesizer progress bar turns red and displays "Synthesis Failed."

When introducing REGAE to participants, we informed them that they should add input-output string examples and press the synthesize button. We additionally informed them that they may use all other buttons in the user interface, and that a description of each button's functionality would pop up if they hovered over it.

2.5.5 SnipPy

SNIPPY [21] is a fork of Visual Studio Code that supports "small-step live programming by example" for Python. It lets users supply input-output examples via *projection boxes* [48] that dynamically show the values of variables at different points in the program.

- **Programming Domain**: Python programs.

- **Synthesizer Input**: A Python program with a hole (??) and a set of input-output examples.
- **Synthesizer Output**: A Python program to fill the hole.
- **Communicating the Input to the Tool** (i) The user types a hole (??) in the program text. (ii) A projection box displays the values of variables in scope, which serve as the inputs for the set of input-output examples. (iii) The user types what the program hole should evaluate to for each set of inputs.
- **Communicating the Output to the User**: When the user indicates they have finished entering input-output examples by pressing the enter key, the synthesizer replaces the hole in the original program with a synthesized program that satisfies the user-provided input-output examples.
- **Communicating Failure to the User**: The synthesizer inserts a comment that says "Synthesis Failed" where the hole was.

When introducing SnipPy to participants, we gave a short demo that followed the example shown in the paper that accompanied the launch of the tool. Specifically, we showed participants how to instantiate the synthesis pop-up by typing ??, adding 2 output examples to the projection box, then pressing the enter key, allowing the synthesizer to populate fill the hole with synthesized code.

2.6 Results and Implications

Our analysis of novice programmers using program synthesizers revealed seven main themes:

(§2.6.1) Synthesizers can either (i) require users to engage in a separate process to produce a specification or (ii) derive a specification as a byproduct of normal non-synthesis tool use; the latter is more learnable for novice programmers.

(§2.6.2) Synthesizers give users varying amounts of control over *when* to trigger synthesis-related activities; in general, synthesizers that exposed less control were more learnable to novice programmers, although this choice had drawbacks too.

(§2.6.3) In general, synthesizers that required smaller specifications were more learnable to novice programmers, but this pattern was not as influential as the others we identified.

(§2.6.4) Novice programmers may not know what constitutes meaningful progress in the context of a given synthesizer.

(§2.6.5) Novice programmers may not know what constitutes a good specification for a given synthesizer.

(§2.6.6) In familiar programming environments, novice programmers may borrow behavior from their pre-synthesizer use of the environment.

(§2.6.7) Novice programmers may engage with synthesis-written programs in ways that they would not engage with teacher-written programs provided as exercise solutions.

Based on these results, we derive a set of implications that that we place in outlined, yellow boxes throughout this section; each implication appears immediately after the relevant findings.

Table 2.2: *Voluntary and Incidental Specification.* We categorized the five program synthesizers that participants used into two categories based on how the user communicated a specification to the tool. The first of these categories comprises the tools that rely on *voluntary specifications*, in which the user designs a specification separately from the primary artifact they are developing. The second category comprises the tools that rely on *incidental specifications*, in which the user produces a specification as a byproduct of proceeding toward an existing goal.

Voluntary Specification	Incidental Specification
Regae, SnipPy	Blue-Pencil, Copilot, Flash Fill

2.6.1 Voluntary and Incidental Specifications

Synthesizers vary in how they have the user communicate a specification, and we observed that participants faced more learnability barriers when they had to use an entirely new process to craft the specification separately from the primary artifact they were developing. We call these separate, standalone specifications *voluntary specifications*. On the other hand, we observed that synthesizers were notably more learnable to our participants when our participants' existing, pre-synthesizer behaviors created the specification for the synthesizer. In such cases, when a specification is a byproduct of proceeding toward an existing goal, we call the specification an *incidental specification*.

Voluntary Specifications: Specifications created only as input to a synthesizer.

Two of the synthesizers in our study—SnipPy and Regae—require participants to craft specifications beyond what they would have produced during their normal programming process. All participants who used tools that required such a specification—what we call a *voluntary specification*—asked for some form of clarification or assistance before successfully eliciting a synthesis output.

SnipPy offers a traditional programming environment in which users *could* author code manually, but in which manual coding does *not* produce a specification. Instead, participants use a special synthesis mode if they want to take advantage of the synthesizer.

Four of the six SnipPy participants attempted to write code manually to complete SnipPy-Reverse-2 instead of entering input-output examples to make a specification for the synthesizer. P6 jumped to thinking about the structure of the code rather than input-output examples, remarking, ***"I was thinking I could use a for loop on the input. Would that make sense?"*** Similarly, P5 was surprised that they were not supposed to manually write the code, remarking,

> So I'm not allowed to type my own code, I have to do it this way?

In contrast to SnipPy, Regae offers an entirely new programming environment which does not resemble the programming environments that study participants had seen before. Regae gives users functionality for entering (i) strings that their target regular expression should reject or accept and (ii) hints for shaping how the synthesizer searches the space of programs; notably, it does *not* allow users to author regular expressions manually. However, despite lacking visual similarities with traditional programming environments (see Figure 2.4), participants still attempted to use the environment as a site to write programs manually;

four of the six Regae participants all tried to write partial regular expressions rather than input-output examples. Looking at a synthesized (but incorrect) regular expression, P2 wondered aloud, ***"Is there anywhere I can write the regex directly then? I can't find the expressions I want in [these suggestions]."***

Even before looking at Regae's synthesis results, participants were still inclined to write code rather than examples. P15's immediate reaction to Regae was to inspect the interface for about 15 seconds and subsequently ask, ***"So where should I write the code? I don't see anywhere for me to write the expressions."*** Similarly, while using

REGAE, P1 explained, ***"Basically I am trying to exclude two plus signs from being next to each other, and the regex guide said that the 'not' expression would work for that."*** After struggling for about 90 seconds to enter a regex directly rather than an input-output example, we informed P1 that the interface expected an input-output example, to which they responded: ***"Oh! I'm sorry. I thought I had to write the code directly in the box. Let me try entering an example instead."***

Implication. Novice programmers may prefer synthesis tools where they can *also* complete the task using non-synthesis strategies or pre-existing expertise, such as by writing code manually.

Although voluntary specifications proved to be a barrier to initial learnability, participants did not necessarily view them as a long-term barrier to usability. P5 later remarked,

> I'm so used to writing code—not writing output of it—but now that I got the hang of it … if I hadn't done an intro to Python course, I think it would actually be really helpful.

P4 commented,

> When you first showed me how to use [it], I was still pretty confused because I have never used anything like that pop-up before. Once I saw it a few times though, I was able to get used to it.

Incidental Specifications: Specification as a byproduct of normal tool use

The remaining three synthesizers in our study elicit specifications as a byproduct of normal tool use without requiring synthesizer-specific specification input. We refer to these specifications as *incidental specifications.*

- In Excel, users build FLASH FILL inputs by filling a subset of cells in a column. This is the same first step that a user would take to fill a whole column manually.
- In Visual Studio Code, users build COPILOT inputs by writing part of a program. This is the same first step that a user would take to write a program manually.
- In Visual Studio Code, users build BLUE-PENCIL inputs by editing a part of a program. This is the same first step that a user would take to perform the corresponding edits across the entire program manually.

Incidental specifications generally appeared to be the more exciting and less confusing for participants. For instance, after P19 was introduced to COPILOT, they responded ***"Oh, cool! Okay, so I just have to [write code] normally."*** Similarly, for FLASH FILL, P5 read the FLASH FILL instructions, then began to complete the first FLASH FILL task by filling in two of the cells manually. They clicked the FLASH FILL button and, upon seeing that it completed the rest of the task automatically, exclaimed, ***"Oh that is super useful! I didn't realize it was going to do [that]."*** P4 was similarly delighted with the incidental of nature of FLASH FILL's specification, commenting ***"Oh! That is very convenient. I didn't have to type anything except for the first cell."***

Incidental specifications also enabled participants to ignore synthesis features when they did not feel compelled to rely on them. For example, P9 started BLUEPENCIL-RENAME-2 by manually editing code. Then, while they were doing so, the contextual button to run BLUE-PENCIL's synthesis appeared three separate times—all while P9 was writing the first line of code alone. Although this contextual button is arguably not very *discoverable* (P14, for example, mentioned that ***"[they] barely noticed the light bulb when it showed up"***), the incidental nature of BLUE-PENCIL's specification (simply writing code as usual) did not *diminish* P9's pre-existing ability to complete the task.

Table 2.3: *Triggerless and User-Triggered Synthesis Initiation and Feedback.* We categorized the five program synthesizers that participants used into the quadrants of this table based on whether the user ("user-triggered") or the synthesizer ("triggerless") decides when to initiate synthesis and when to communicate the results of synthesis. User-triggered initiation implies user-triggered result communication, so that quadrant of the table is necessarily empty.

	Triggerless Result Communication	User-Triggered Result Communication
Triggerless Initiation	Copilot	Blue-Pencil
User-Triggered Initiation	–	Flash Fill, Regae, SnipPy

Moreover, while SnipPy and Regae users were reluctant to shift from authoring code manually to providing voluntary specifications built up by alternative interactions, Flash Fill users were comfortable providing a specification by filling spreadsheet cells rather than by writing cell formulas manually, which suggests that a key learnability stumbling block is *not necessarily* asking for non-program specifications, but rather asking users to take actions that do not visibly move the user towards the target output. These findings resonate with broader research on the relatively high cognitive costs of task-switching [1, 70, 42] and the usability issues around modes [17, 68].

Implication. Incidental specifications are a promising avenue for on-ramping novice programmers to using program synthesizers.

2.6.2 Triggerless and User-Triggered Initiation and Result Communication

We identified two important design decisions for program synthesizers regarding the mechanism and timing for synthesis initiation and result communication:

- Does the tool decide when it should run synthesis, or does the user decide?
- Does the tool decide when it should communicate synthesis results, or does the user decide?

We use the terms *triggerless initiation* and *triggerless result communication* to describe when the tool automatically makes these decisions; conversely, we use the terms *user-triggered initiation* and *user-triggered result communication* when the user makes these decisions. We categorize the program synthesizers that participants used into quadrants in Table 2.3 representing all possible combinations of the aforementioned design choices. We note, however, that only three of the four quadrants are inhabitable by tools: because synthesizer outputs can only be displayed after the synthesizer runs, a tool that relies on user-triggered synthesis necessarily also relies on user-triggered result communication.

Triggerless Initiation + Triggerless Result Communication

Copilot uses triggerless initiation and triggerless result communication: it proactively provides code suggestions as the user types normally. Although participants who used Copilot did not always accept the synthesis suggestions without modification (as we discuss later in Section 2.6.7), these participants all initiated synthesis rapidly and were able to see synthesis results immediately. For example, P20 started Copilot-Abbreviate-1 by writing `def abbrev(name):`, to which Copilot quickly showed a synthesis suggestion. P20 then used their mouse to hover over the suggested line of code and asked, ***"So should I just accept this suggestion? It looks correct to me."*** P18 had a similar experience, and simply pressed the tab key to accept the synthesis suggestion when presented with it. P19 summarized their experience by remarking, ***"I really liked how I could see the suggestion without having to do anything."*** This behavior may violate a maxim of *polite software* [87]—in particular, that "polite software respects, and does not preempt, rightful user choices." However, these findings findings resonate with the polite software observation that '[impolite software] may help novices … who may trade politeness for usefulness' [87].

On the other hand, a downside to an entirely triggerless experience was that, after relying on synthesis for parts of a task, participants sometimes wanted to continue to rely on the synthesizer by *manually* triggering it. For example, when using Copilot to synthesize a solution for Copilot-Subsequence-3, P20 noticed a bug in the synthesized code. When starting to debug, P20 wondered aloud, ***"How do I get the suggestions again? Can I use the tool to help find the error?"*** Ultimately, P20 could not figure out how to manually trigger Copilot suggestions for what they wanted to do, so they finished the task by fixing the bugs manually.

Triggerless Initiation + User-Triggered Result Communication

Unlike Copilot, Blue-Pencil has triggerless initiation but relies on user-triggered result communication. It identifies repetitive edit actions as the user types normally, and it automatically synthesizes program transformation scripts to automate the edits. However, the user must go out of their way to request to view synthesis outputs by clicking on a contextual light bulb icon that appears when synthesis has completed.

Overall, participants were confused about *when* they should click on this contextual light bulb. As P14 described, ***"I didn't really know when to click on it, because I didn't know how it could help."*** In some cases, Blue-Pencil's contextual light bulb appeared during a time when the participant struggled to complete an aspect of a task we assigned them, and if the participant had clicked on the light bulb, they would have seen a suggestion from Blue-Pencil that would have automatically completed the task successfully. For example, P9 tried to manually make modifications to the starter code in BluePencil-Point-1, but their edits resulted in errors because P9 did not initialize a `Point` object correctly. (Blue-Pencil would have initialized the `Point` correctly.) Similarly, P10 mentioned that they were confused about a syntactic construct in Java that was relevant to the edit they needed to make in BluePencil-Point-1, and therefore could not initially complete one of the repetitive edit tasks. (Blue-Pencil would have completed the edit manually with the correct syntax.)

Implication. Synthesis with triggerless initiation appears to be more learnable to novice programmers, but to reap the full benefits of triggerlessness, synthesis tools will likely need to support *both* triggerless initiation as well as triggerless result communication.

User-Triggered Initiation + User-Triggered Result Communication

Regae, Flash Fill, and SnipPy all rely on user-triggered initiation: users must manually decide when they have built a sufficient synthesizer input, then trigger synthesis themselves. As a result, these tools necessarily require a user trigger for the synthesis to communicate output results, which places them all in the bottom-right quadrant of Table 2.3: user-triggered initiation with user-triggered result communication.

Placing the burden of deciding when a specification is sufficient into the hands of novice programmers produced significant learnability obstacles for the synthesizers in this category.

On a basic level, simply remembering how to trigger synthesis was a learning obstacle for our participants. Four of the six participants who used SnipPy struggled significantly to remember how to trigger synthesis (which is done by introducing a hole, ??, in the program text). P3 even asked the same question twice after they saw the ?? work as a trigger for the synthesis pop up: ***"How do I open the box to put in examples again?"*** and ***"Sorry, I forgot how to get the box again, can you remind me?"*** Working with Regae, P1 kept adding input-output examples because they did not know how to trigger synthesis; only after adding twelve input-output examples did they ask,***"Is there a maximum number of examples I can add? What do I do next?"***

On a deeper level, participants often built substantially larger specifications than the synthesizer required—a mode of failure that should never occur in triggerless synthesis initiation. For example, four of the six Regae participants spent significantly more time than necessary adding exhaustive input-output examples in the Regae interface and only clicked the synthesize button when we prompted them to do so; until that time, the synthesizer did not run. After 3 minutes, we nudged P6 to stop adding examples, to which they remarked:

> Oh that makes sense. I wasn't sure when to stop adding examples because I thought I had to add one for every possible input.

P12 had a similar experience, adding ten annotated examples during Regae-Plus-1 before asking, ***"Is this too many inputs? I don't know how many more to add."***

Implication. Synthesizers with user-triggered initiation may need to provide guidance or feedback to users on *how many* examples are likely to be necessary for a given task.

Even more fundamentally, user-triggered synthesis initiation creates a setting that is ripe for user misconceptions about specifications to manifest. For example, the participant behavior we described in the previous paragraph indicates that participants often seemed to hold the belief that the synthesizers they worked with would always perform better if the synthesizers were given more information. We call this belief about synthesizer specifications the *monotonicity belief.*

In fact, the performance of synthesis algorithms may degrade as the sizes of specifications grow large, even when parts of the larger specification are redundant and the specifications are satisfied by the same program [84, 22].

This monotonicity belief led to counterproductive behaviors when working with, for example, Regae. As a synthesizer with user-triggered initiation, Regae required participants to decide—without guidance—how many examples to provide before triggering synthesis. P2 initially entered a set of five redundant examples that caused the synthesizer to return an incorrect result. Instead of modifying these examples to be more representative of the problem domain, P2 immediately jumped to adding ten *more* examples, but the total number of examples P2 entered simply caused Regae to time out.

Implication. User-triggered synthesis lets novice programmers construct incorrect theories about when to trigger synthesis. Designers of such synthesizers may want to consider identifying misconceptions about their tool (such as the belief that larger specifications will improve synthesis performance) via user studies, then refine their tool to proactively combat these misconceptions.

2.6.3 Small Specification Size

In general, participants more easily learned to use—and were delighted by—synthesizers that required relatively small specifications, such as Copilot, Blue-Pencil, and Flash Fill. For example, once Copilot suggested synthesized code for Copilot-Abbreviate-1 after P19 typed only a function declaration (`def abbrev(name):`), P19 exclaimed: ***"Oh wow! Can I just press tab and accept the suggestion?"*** Similarly, after P14 made just one program edit for BluePencil-Rename-2, Blue-Pencil synthesized the remaining edits needed to complete the task, to which P14 responded: ***"I mean, I think it did all of it for me."*** Lastly, after completing FlashFill-Characters-3 by running Flash Fill on just one example, P6 remarked, ***"Oh, cool! That was easy. I didn't know for sure if it could figure out the pattern from [one example]."***

While participants did seem to appreciate a smaller specification size, it is worth noting that we observed that other factors—such as voluntary versus incidental specifications (Section 2.6.1) and triggerless versus user-triggered interactions (Section 2.6.2)—seemed to have a more substantial impact on the learnability of the program synthesizers. Recommended practices for synthesis systems [47] and many existing program synthesizers make small specification size an explicit goal [53, 30, 10, 66], so this finding may suggest a change in emphasis for future algorithmic work.

Implication. Synthesis designers may want to explore factors other than size to reduce the burden of specification, such as collecting incidental specifications, which could elicit large specifications from novice programmers with relative ease.

2.6.4 Interpreting Synthesis Outputs

Participants struggled to use synthesis outputs to figure out if the synthesizer had made useful progress, in particular (i) thinking the synthesizer made progress on a task when it had not and (ii) not realizing that the synthesizer made progress on a task.

This struggle with assessing progress may reflect difficulties decomposing a task into smaller subproblems, understanding when synthesis outputs are solving those subproblems, or some combination of both. If decomposition is at fault, this echoes the difficulties novice programmers face with *plan composition* (putting together fragments of programming knowledge that accomplish particular kinds of tasks [77]) and *task decomposition* in programming more broadly [78, 24, 8]. However, they may also reflect a lack of necessary guidance from the synthesis tool.

Incorrectly Believing the Synthesizer Made Progress

In one case, P14 made an incorrect edit in BluePencil-Point-1 (changing a return statement to return an entire point instead of an x-coordinate). Blue-Pencil then generalized this edit, and P14 accepted Blue-Pencil's suggestion to carry the edit through the rest of the program. However, these edits took P14 farther from the task's solution and ultimately P14 had to make the necessary edits manually, without Blue-Pencil's help.

In another case, P3 overgeneralized their experience working on the first SnipPy task (SnipPy-Abbreviate-1) to the second SnipPy task (SnipPy-Reverse-2) by reusing the step in SnipPy-Abbreviate-1 of splitting a space-separated string into a list of words. As in SnipPy-Abbreviate-1, synthesis succeeded for this query and produced code to separate the string properly. P3 appeared to interpret this result as an encouraging sign and continued by providing input-output examples to reverse the string, not recognizing that the first "successful" result was not a meaningful step towards solving SnipPy-Reverse-2. Due to not having enough of the task broken down properly, SnipPy was unable to synthesize code to satisfy the input-output examples that P3 specified, which led P3 to scrutinize their *second* query to the synthesizer and completely ignore the code generated by their first query, which was actually at fault.

Implication. Because novice programmers may be confused about the difference between synthesis succeeding on a query and making meaningful progress on a task, synthesis designers may need interventions like (i) documentation that stresses the distinction or (ii) scaffolding for problem decomposition.

Incorrectly Believing the Synthesizer Did Not Make Progress

When working on Regae-Plus-1, P17 observed some results that were almost (but not exactly) correct, noting ***"This one is kind of correct because [the examples] can't contain ++, but it still accepts letters."*** However, they did not understand that they could mark these partial solutions to be included in a full solution in Regae; instead, they completely threw away the synthesis results and went back to add and augment examples in their original specification to try to get a more correct synthesis result.

In general, users often missed when the synthesizer had made useful suggestions or even full solutions, even when the synthesis outputs were visible. For instance, P19 noticed that the autocomplete suggestion for Copilot-Occurrences-2 was incorrect, so they started to rename a function in an effort to trigger Copilot to synthesize something different. Although the Copilot pane which shows additional suggestions (beyond the autocomplete one) displayed two correct synthesis outputs, the participant did not realize and continued with their strategy of renaming the function.

2.6.5 What Kind of Specification Does My Synthesizer Want?

Participants did not always know how best to specify their intent to the synthesizers they used.

In particular, participants did not know what *kinds* of specification were best for a given synthesizer. For example, when attempting Copilot-Occurrences-2, P20 was not satisfied with the first synthesis suggestion, but seemed unsure how much of an impact changing their function declaration would have: ***"I feel like this is wrong, and I am a bit confused. If I change the name of the function, would this change the results that Copilot would output?"*** P19 faced a similar situation, receiving an incorrect Copilot autocomplete suggestion, then struggling to figure out how to communicate to Copilot what they wanted to change. They came up with the same strategy of changing a function name, tweaking a portion of their program text to read `def listToDictionary(list)` instead of `def count(list)`.

For triggerless tools, sometimes users could not even figure out how to change their tool use behaviors in order to *trigger synthesis*. For example, P9 could not figure out what to include in their specification to trigger the contextual light bulb in Blue-Pencil, remarking ***"It feels like the light bulb pops up randomly."***

The specification issue was most prominent with participants using example-based synthesizers. As we discussed in Section 2.6.2, participants using user-triggered example-based synthesizers often had the *quantitative* misconception that more examples would always result in a better outcome (the monotonicity belief); however, participants also had *qualitative* misconceptions about what *kinds* of examples to provide as well. For example, P17 provided an extensive set of correctly-annotated input-output examples to Regae, ran the synthesizer, and received only an indication of synthesis failure after a few minutes of waiting. Despite their large synthesis specification, P17 had missed some edge cases while providing an overabundance of redundant examples. P17's response to this situation was to wonder aloud:

> Am I missing any cases? I feel like I covered all of the edge cases. Do I need to add a different example for every letter?

Similarly, P15 initially entered small, underspecified set of input-output examples into Regae. The synthesizer returned an incorrect regex, and P15's reaction was to add significantly more positive and negative input-output examples (approximately 15), but they did not provide the additional, optional annotations on the input-output examples that the synthesizer needed to complete the task. Their endeavor ultimately resulted in another synthesis failure.

Lack of feedback upon synthesis failure. Participants' lack of knowledge about what constitutes a good specification was exacerbated by the lack of informative feedback upon synthesis failure, which echoes findings in the explainability of recommendation systems [82], the learnability of AI systems more broadly (Section 2.3.4), and the notion of the *user-synthesizer gap* [21] that previous usability studies of synthesizers have explored [21, 20].

For example, after Regae timed out on a synthesis request from P2, P2 asked, ***"Is there a way to check why it failed?"*** Similarly, when attempting SnipPy-Reverse-2, P2 entered "Augusta Ada King" and the same string backward as an input-output example which (for a reason we do not know) caused the synthesizer to fail. When the participant later changed the example to "John Doe" and kept all other aspects of the specification the same, the synthesis query succeeded, and P2 asked ***"Oh that's weird. Do you know why it didn't work before?"***

This lack of feedback on specifications upon failure was particularly problematic in synthesizers with complex, multi-part specifications. For example, the Regae interface requires the user to take several distinct steps: (i) enter input-output examples, (ii) annotate input-output examples as either "general" or "literal," and (iii) decide when there are enough examples to manually trigger the synthesizer. Aside from the fact that longer sequences of actions mean that users will need to wait longer to view the results of synthesis, we observed that tools that require more independent specification components make it harder to debug synthesis failures. For example, a user may wonder: Are the input-output examples correct? Are they correctly annotated? Are there enough examples? Too many? P15 exemplified this confusion: Having completed the steps above for Regae, P15 received an incorrect result from the synthesis engine. Upon receiving the incorrect result, P15 attempted to use the "Mark as Literal" and "Mark as General" buttons more extensively in order to improve the synthesized results. They also tried to add several more input-output examples that were similar to the ones they had already written. Neither of these approaches helped the synthesizer succeed; the specification bug was actually that the examples did not cover all aspects of the requirements.

Implication. Users may benefit from (i) feedback about why a given synthesis run has failed and (ii) interactions that help them understand an incorrect synthesis output and why it was produced.

In summary, participants often did not know what constitutes a good specification, so they invented theories, some of which were incorrect. Without feedback about how to improve their specification when synthesis failed, these wrong theories persisted.

2.6.6 Borrowing Existing Behaviors in Familiar Environments

Three of the five synthesizers in our study were instantiated in existing, mainstream environments: Flash Fill in Excel, and Blue-Pencil and Copilot in Visual Studio Code. When participants used synthesizers in these previously familiar environments, they may have borrowed behaviors and intuitions from their previous exposures to these tools.

For example, participants using Flash Fill were inclined to enter multiple examples and drag the cell or simply manually type in the formula themselves, rather than provide just one example and use the Flash Fill button. P6 remarked, ***"I thought I had to enter in a lot of examples and then drag down on the cell, but this is way easier."*** Similarly, P5 explained,

> I would have probably right clicked on the cell or done a Google search to figure out how to fill in the spreadsheet if you hadn't showed me the Flash Fill button. I don't use Excel too much but that approach usually works for other [programming environments].

When completing BluePencil-Rename-2, P9 knew they could change all variable occurrences to a new name in Visual Studio Code by right clicking on a variable and selecting the "Change All Occurrences" option. They chose this approach that they already knew how to do over using the contextual Blue-Pencil light bulb, which was already present beside the line of code in question.

Finally, P19 did not open the Copilot tab in Visual Studio Code at all prior to the last task. They explained:

> Usually when I use the helper tools in VSCode they don't show anything useful, so I don't use any of the pop ups or features anymore.

Implication. Because novice programmers may apply preexisting strategies from a particular environment to synthesis-augmented variants of the environment, designers may benefit from being aware of common preexisting user behaviors.

2.6.7 More Effort + Less Trust = More Engagement

Synthesis tools provide novice programmers different ways to engage with coding and debugging relative to a traditional educational setting. In particular, participants in our study quickly saw code they had not written themselves, worked to understand and debug code that is partially or entirely incorrect, and used the output of the synthesizer as a tool to refine their understanding of the problem domain. In some ways, synthesizer output is comparable to an instructor-provided exercise solution in a traditional computer science course—both may show the student an example of a working program. But we see a few key differences:

- **More effort.** Program synthesizers require the user to come up with some form of specification.
- **Less trust.** Program synthesizers may return code that satisfies the given specification but does not do what the user intends.

While these points could be viewed as negative attributes of program synthesis, we observed some positive impacts of these differences with our participants. Namely, they may result in more engagement than we expect from students looking at, for example, teacher-written exercise solutions.

Code Engagement

In nearly all cases that the synthesizers returned code, participants investigated the code to some degree to ensure its correctness rather than immediately accepting the code and moving on. For example, P0 completed SnipPy-Abbreviate-1 and manually traced an example input out loud before coming to the conclusion that the synthesized code was incorrect. Similarly, P1 attempted to identify any counterexamples that would imply that the synthesized code was incorrect, ultimately concluding that the code was in fact the correct answer to Regae-Plus-1. On various occasions, P18, P19, and P20 all took at least 60 seconds walking through synthesized code in Copilot before accepting a suggestion. After doing so, both P18 and P20 accepted code they knew did not fully meet the specification for the task at hand and subsequently spent time debugging and tweaking the code to match their needs. P17 describes how this process of engagement unfolded for them in a similar situation:

> This semi walks you through the process of how to get there, and, like, makes you think about what are the edge cases—what do I think about in order to get to that end goal.

While we cannot say for sure that these participants would not have done the same for code instructor-provided solution code, these examples suggest that because synthesizers engender less trust, they increase the engagement with output code.

More generally, many participants expressed signs of engagement such as surprise or delight at the code the synthesizer returned, sometimes learning new ways to tackle the problem. For example, after P3 completed SnipPy-Abbreviate-1, they expressed their understanding of the synthesized code, which included a list comprehension. P3 remarked that they had never seen a list comprehension before, but their stated understanding of the code was correct: ***"I have never seen it done like this before. Does this just mean that it automatically loops over every element without a* `for` *loop?"*** Similarly, P4 remarked of SnipPy,

> After the synthesizer delivered the code for me, I kinda went over it and was like "Ahh! That's a really smart way to do it"—I would have thought of a much longer way to do it.

Implication. Synthesizers require specification effort before providing output programs, and output programs may be incorrect. These factors, or some other factors, may dispose novice programmers to engage more actively with synthesis outputs than with traditional exercise solutions.

Specification Engagement

Participants spent substantial time and energy on refining specifications, shaping them to eventually match their true intent. After engaging with a synthesis output that satisfied their input specification but did not reach their goal, participants had to reexamine their input specification. For example, although the participants described in Section 2.6.5 did not have a correct mental model of what *kinds* of examples would be useful for the synthesizer at hand, they did spend a significant amount of time working and reworking their input specifications.

We observed that Regae participants P2 and P7 found that Regae responded to their synthesis queries by producing results that allowed expressions with an alphabetical letter even though they added a negative example as an input. It was only after they saw this synthesis error that they modified their specification to contain much richer information by annotating their examples as general or literal.

Implication. Because synthesis tools can interactively generate code that satisfies a given specification, they may help novice programmers learn how to write complete and correct specifications. Asking students to ensure that a synthesizer arrives at the correct solution when given their specification may be a reasonable educational intervention for teaching students how to reason about correctness of programs in general, and important correctness concepts such as covering corner cases.

2.7 Limitations

Disentangling effects. As a consequence of using real tools that were not designed to vary one design dimension at a time, we cannot definitively attribute every pattern to a particular design decision. A randomized controlled experiment would be required to achieve definitive conclusions about varying degrees of learnability along the axes that we identified. However, more than attributing success to a particular dimension versus another, this study was aimed at *identifying* and *exploring* dimensions of interest in the first place.

We hope future research on synthesizer learnability will contribute experiments that vary a given synthesizer across the dimensions we identified. We expect this style of research will eventually help the field develop guidelines for how we should design synthesis tools, rather than only evaluating a given tool in isolation or against manual coding.

Effects of think-aloud and investigator presence. Participants completed tasks and thought aloud simultaneously, which prior work [16] indicates could increase cognitive load relative to if the participants had experienced the synthesis tool outside of a lab setting. Further, the first author was present throughout the video meeting and recorded the session for future analysis. If participants felt pressured, their interactions with the synthesizers may have been harder. Conversely, the presence of the first author may have had the opposite effect if it assisted their learning in any way.

Non-evaluative. Some of the tools in our study were not created with the intention of being learnable for novice programmers. We selected the tools in our study with the goal of uncovering design decisions that do and do not work for novices. However, the fact that serving novice programmers is not an explicit goal for some of them is one of several reasons that this study should *not* be read as an evaluation of either the synthesizers we used or whether existing synthesis tools for novices are serving that audience.

External validity. Although the tools used in the study are real and publicly available, the research team selected the tasks that participants completed in sessions. The learnability of synthesis tools for these tasks may or may not resemble the learnability for users' real tasks. Moreover, our participants were drawn from a pool of undergraduates attending an R1 university. While the behavior of this population has generated a useful starting point for understanding the learnability of program synthesizers by novice programmers, it almost almost certainly does not reflect the behavior of *all* novice programmers. We hope future work in this area explores the behaviors of additional groups of novice programmers.

Incompleteness. We expect the insights from this study will be useful for the designers of future program synthesis tools. However, we are certain our study is not the final word on the learnability of program synthesis tools for novice programmers. This study drew on observation of 22 sessions with novice programmers, all of which (i) followed a single, fixed structure and (ii) we analyzed via qualitative analysis alone. Regarding point (i), we are confident that there is much more to learn in this space from additional qualitative and need-finding work; regarding point (ii), our observations offer plausible hypotheses for future, quantitative follow-on work.

2.8 Conclusion

Program synthesis tools have the potential to make programming tasks easier for a variety of audiences. This work explores key barriers that stand between novice programmers and this vision of learnable, helpful synthesizers. We hope reasoning explicitly about design dimensions like voluntary versus incidental specifications, triggerless versus user-triggered synthesis, and triggerless versus user-triggered output will eventually support our community in making synthesis tools more learnable. The common misconceptions we observed may also play an important role in helping future synthesizer authors design specifications that work for real users. Although this work is only a first step toward understanding how to make program synthesizers more learnable by novices, we believe insights from this work can ease the design process for designers who aim to make synthesis tools for use in educational settings and novice-targeted development tools.

Chapter 3

Is There Evidence That Program Synthesizers Improve Learning Outcomes?

This chapter describes a controlled study focused on the impact of synthesis tools on novice programmer learning outcomes.

Description of contributions. Dhanya Jayagopal designed and carried out the experiment. Justin Lubin and Dhanya Jayagopal both worked on the statistical analysis. All authors discussed results and contributed to writing.

3.1 Introduction

With the rise of the applicability of program synthesizers, it can be tempting to believe that program synthesis is a magic bullet for a variety of goals. In this short chapter, we describe (i) the state of the art in educational program synthesis and (ii) that no research has offered evidence that educational synthesis tools improve student learning. There is little research that attempts to evaluate the impact of program synthesizers on learning outcomes, and none provides evidence of improved learning outcomes. Thus this chapter centers on exploring the question: Is there evidence that program synthesizers can improve computer science learning outcomes for novice programmers?

To answer our research question, we performed a literature review of papers at the intersection of program synthesis and education. Additionally, we designed a program synthesizer explicitly for integrating into a lesson plan for novice programmers; we then performed a between-subjects randomized controlled trial assessing whether using this synthesizer influenced participant learning over the course of a lesson.

Through our literature review and our own study, we could not find any evidence that program synthesizers have improved learning outcomes for novice programmers thus far. We stress that this should not be taken as evidence that synthesis tools *cannot* help novice programmers learn, but rather that (1) no existing tool has provided evidence that it does so and (2) simply adding program synthesis into an educational interaction does not automatically boost learning outcomes. Instead, these results highlight that although synthesizers may help computer science students (and programmers more broadly) reach functioning programs more efficiently, *this outcome is not the same as improving learning outcomes.*

In this paper, we are focused on the impact of synthesis-aided tools on novice programmers' learning outcomes. However, we acknowledge that this is only one possible role that synthesizers can play in programming education. Other roles include but are not limited to: (i) generating automatic feedback on a programming task, even if the feedback does not affect student learning; (ii) making teaching staff more efficient at grading; (iii) making teaching staff more efficient at annotating student submissions with feedback.

Contributions. This chapter presents the following contributions:

- A **literature review of research at the intersection of program synthesis and computer science education**, focused in particular on identifying evidence of effects of program synthesizers on learning outcomes for novice programmers.

- A **randomized controlled trial of novice programmers**, which assesses learning outcomes with and without access to a program synthesizer.

3.2 Background

Concept Inventories Concept Inventories (CI) are test-based multiple choice assessments of a set of concepts that measure student learning [81]. The incorrect answer choices presented to the student are generally based around common misconceptions around the concepts being tested. A *validated* concept inventory is one that has been (1) reviewed by numerous researchers and (2) accepted as a state-of-the-art assessment of the concept in question. There are only a few validated concept inventories within the field of computer science [29]. We rely on one of the existing validated concept inventories to assess learning outcomes.

Intelligent Tutoring Systems This paper focuses on the role that synthesizers could play in affecting student learning outcomes. Even though synthesizers specifically have not been studied extensively within this context, there is concrete evidence to suggest that other tools and technologies do in fact improve student learning outcomes. For example, there is a large body of work on how Intelligent Tutoring Systems (ITS) affect student learning. ITS systems generally (1) guide students through a personalized curriculum by suggesting problems based on certain criteria, and then (2) provide feedback on submissions [14]. These systems are integrated into a large number of courses, so a great deal of data on how they influence student grades and performance on assessments exists. Several studies show that Intelligent Tutoring Systems do increase grades and scores on average [14, 13].

3.3 Literature Review: Synthesis-Backed Tools in Education

There has been an increasing amount of research surrounding the use of program synthesizers in education. However, none of the literature evaluates their impact on learning outcomes. Rather, evaluations focus on other metrics such as efficiency or accuracy. To date, *we have found no research providing evidence of improved learning outcomes.* Our literature review includes summaries of the papers in this space.

In the paper, **"Automated feedback generation for introductory programming assignments"** [75], the authors present an ‘automated technique to provide feedback for introductory programming assignments." The motivation behind this tool is to make programming education more accessible worldwide. The research evaluated the tool by looking at the percentage of incorrect solutions for which it could provide a synthesized correction: 64%. The work did not evaluate the impact that this sort of automated feedback has on student learning.

Error Correction Synthesis Coverage

In the paper **"What Would Other Programmers Do? Suggesting Solutions to Error Messages"** [34], the authors present an error-correction synthesis tool called HelpMeOut which is a "new strategy of collecting and presenting crowdsourced suggestions for programming errors inside an IDE." The authors completed a user study in which participants were amateur programmers. They measured (1) the number of times HelpMeOut provided a suggestion when queried and (2) how useful the suggestions were, as assessed by the the authors of the tool. HelpMeOut was found to provide *a* suggestion 87% of the time. Approximately 50% of those suggestions were deemed 'useful' by the authors of HelpMeOut. While this provides an evaluation of the synthesizer's coverage, it does not provide evidence that such synthesizers can or do improve learning outcomes. The paper proposes that "future evaluation should clarify the impact that explanations have on transfer performance. Literature on analogical problem solving suggests that programmers seeking to transfer a fix suggestion to their code will be aided by an explanation that states the principle that the fix demonstrates, but we have yet to measure this effect."
Error Correction Synthesis Coverage Researcher-Rated Usefulness

In the paper, **"Learning Syntactic Program Transformations from Examples"** [71], the authors present Refazer: "a technique for synthesizing syntactic transformations from examples." The authors state that the intended use case for Refazer is to fix student bugs automatically, without the manual intervention of an instructor. The results of their user study report that Refazer automatically fixed 87% of student program submissions for one task and learned the necessary code transformations for 84% of another task. While Refazer can edit many programs automatically, the authors observe that the fixes may not be stylistically desirable. The paper does not attempt to assess how the replacement of an instructor with Refazer would affect student learning outcomes.
Error Correction Synthesis Coverage Correctness

In the paper **"Qlose: Program Repair with Quantitative Objectives"** [15], the authors present Qlose, a synthesizer built on top of Sketch [76]. The motivation for this work is built around the observation (see above) that many synthesis tools do not provide "desirable" repairs—in particular, repairs that make only small changes to the original program structure. They are interested specifically in handling programs from educational platforms such as CodeHunt and edX. The authors performed an evaluation of Qlose on 11 buggy student program submissions sourced from introductory programming courses. They conclude that Qlose's approach to synthesizing programs using on a distance metric based on syntactic *and* semantic distance is "practically feasible for small student solutions and leads to more desirable repairs" relative to the semantic and syntactic baselines they introduced. The authors manually inspected synthesis outputs to mark them as desirable or undesirable. This study does not provide evidence on the impact of Qlose on student learning outcomes.
Error Correction Synthesis Researcher-Rated Usefulness

In the paper **"Writing Reusable Code Feedback at Scale with Mixed-Initiative Program Synthesis"** [35], the authors evaluate systems that "use program synthesis to learn bug-fixing code transformations and then cluster incorrect submissions by the transformations that correct them," including a system backed by Refazer, described above. The user studies measured participants' assessment of fix "value" and reusability of the feedback. Participants were teachers. In the first study, teachers assessed fixes in isolation—i.e., without sending them on to students. Teachers rated 19% of Refazer's solutions poor, called some fixes "dangerous," and identified fixes that would cause student solutions to pass tests but still include logical failures. Teachers were also asked to rate how many solutions

in a tool-provided cluster shared the same misconception. Answers ranged from 50–100%. A second study asked teachers to use another tool for propagating fixes to student solutions. In this study, teachers decided whether or not to use particular fixes as feedback for students, and teachers reported on the acceptability of fixes. No element of either study assessed effects on student learning.
Error Correction Synthesis Synthesizer Execution Time User-Rated Usefulness

In the paper **"Automated Clustering and Program Repair for Introductory Programming Assignments"** [32], the authors present a "fully automated program repair algorithm for introductory programming assignments" called Clara. The authors then conducted two evaluations: (1) an evaluation on MOOC data and (2) an evaluation of the usefulness of synthesized corrections. Clara automatically generated repairs for nearly 98% of the submissions scraped from the MOOC courses. In the user study, participants were asked to solve a set of programming problems. During this process, they might see repairs from Clara. *When Clara successfully produced a repair*, participants were asked: "How useful is the generated repair-based feedback?" The response options were: "1 ("Not useful at all") to 5 ("Very useful")" The average grade that Clara received when it had produced a repair was a 3.5. Unlike the works above, this work collected an assessment of usefulness *from programmers seeing the feedback*. However, it did not attempt to measure impact on learning outcomes.
Error Correction Synthesis User-Rated Usefulness

In the paper **"Neuro-Symbolic Program Corrector for Introductory Programming Assignments"** [6], the authors present "a technique to combine RNNs with constraint-based synthesis to repair programs with syntax errors … to predict token sequences for finding syntax repairs for student submissions." The authors ran the tool on nearly 15,000 edX submissions of programming assignments and found that their "method is able to repair syntax errors in 60% (8689) of submissions, and finds functionally correct repairs for 23.8% (3455) submissions." This work did not assess the impact that the tool would have on student learning outcomes.
Error Correction Synthesis Coverage Correctness

In the paper **"CPSGrader: Synthesizing Temporal Logic Testers for Auto-Grading an Embedded Systems Laboratory"** [44], the authors "formalized the auto-grading problem for laboratory assignments in cyber-physical systems, and presented a formal, algorithmic approach to solve it based on parameter synthesis." In particular, the autograder synthesizes test cases for a particular class of problems. The autograder, CPSGrader, was evaluated on a dataset of programs submitted at various stages of program development by 50 student groups during a lab session in an introductory Embedded Systems course. The evaluation assesses whether the synthesized test benches correctly label student submissions as faulty or correct. The work does not assess any impacts on learning outcomes.
Test Suite Synthesis Correctness

The paper **"Exploring the Design Space of Automatically Synthesized Hints for Introductory Programming Assignments"** [79] aims to pave"the way for future deployments of automatic, pedagogically-useful programming hints driven by program synthesis." The work presents "(1) a characterization of five types of hints that can be generated by state-of-the-art synthesis techniques," informed by an analysis of 132 online Q&A posts and an interview of a teacher; and "(2) the implementation of transformation, location, and data hints in an interactive debugging interface." The paper states "we have not evaluated how pedagogically useful these hints are." The tool aims to generate automatic feedback that mimics manual feedback as closely as possible—i.e., generates similar learning outcomes—but the work has not been evaluated.

In the paper **“Towards Answering ‘Am I on the Right Track?’ Automatically using Program Synthesis”** [19], the authors present a new tool that provides “feedback automatically for programming assignments using program synthesis,” based on a study of TA responses to student questions during office hours. The evaluation of the tool addresses three questions: (1) is the synthesizer able to provide full or partial feedback, (2) do students find the feedback helpful, and (3) how do students interact with the synthesizer? While these questions cover some elements of user experience, the evaluation does not assess the impact of the tool on student learning outcomes.
Feedback Synthesis Coverage User-Rated Usefulness

In the paper **“Human-Centric Program Synthesis”** [12], the author “[sketches] a human-centric vision for program synthesis where programmers explore and learn languages and APIs aided by a synthesis tool.” The author writes, “By explaining its code-generating decisions, a synthesis tool can move beyond code that just works, to code that teaches how it works.” The paper does not introduce a tool or include evidence that program synthesizers teach languages or APIs.

A central theme of many of these papers is *minimizing the burden on computer science instructors without limiting the accessibility of computer science education.* This suggests two distinct goals: (1) creating tools that automate instructor behavior and (2) evaluating the impact of such tools on the student learning outcomes. So far, we have seen a few evaluations tackling the first goal; however, we have not seen any studies that provide evidence that synthesis tools have a positive (or even non-negative) impact on learning outcomes.

3.4 Randomized Controlled Trial of a Lesson With and Without a Synthesizer

Motivated by our findings from the literature review, we ran a randomized controlled trial to evaluate the impact on student learning of introducing a program synthesis tool into a particular computer science lesson.

3.4.1 Participants and Recruitment

We conducted virtual study sessions over Zoom with 29 participants, all of whom were in their first year of studying computer science.[1] We screened participants (recruited from Piazza, Facebook groups, and Slack workspaces) via a survey for little to no external programming experience beyond their limited coursework in order to limit our participant pool to novice programmers.

3.4.2 Study Protocol

We randomly assigned each participant to go through exactly one of three conditions (each of which we describe in the next section, Section 3.4.3), to conduct a between-subjects randomized controlled trial. During each study session, we had participants complete the following three steps:

1. Complete a concept inventory pre-test.

[1] We ran the study with 30 participants, but our software crashed for one of the participants, so we excluded the participant’s data from all analyses.

2. Go through the version of our binary search tree lesson plan that corresponds to their randomly assigned condition.

3. Complete a concept inventory post-test.

For our concept inventory, we selected three questions related to binary search trees from an existing validated concept inventory for basic data structures [67]. The pre-test and post-test had the same questions, but with both the questions and answer choices reordered. We administered the concept inventory virtually and proctored each participant over Zoom. We provide the questions we used as well as the scoring mechanisms in Tables 3.4, 3.4, and 3.6, located at the end of this chapter. All questions were scored on a scale from 0–1.

3.4.3 Experimental Conditions: Lesson Plan and Synthesizer

We designed and developed an interactive lesson plan for binary search tree ourselves based on introductory course material from R1 universities throughout the United States. In addition to the written materials, we made a programming-by-demonstration synthesizer for the two functionalities tested in our concept inventory questions: binary search tree (BST) *search* and BST *insertion*. Participants were not able to progress beyond each task unless the interface accepted their code as correct via a hidden test suite. Both the written materials and the custom synthesizer were presented within a single webpage.

Our study included three conditions. The interface for each condition is (partly) displayed in Figure 3.3 at the end of the chapter.

1. Control: Participants did not have access to a synthesizer and had to write code for BST search and BST insertion manually. *(9 participants total.)*

2. Full: Participants were shown an interactive binary search tree visualization. Their interactions with the visualization were logged by the tool and used as the specification for the custom synthesizer. Once participants felt they had done enough demonstrations of how the function should behave on different inputs, they could click a button to trigger synthesis. *(10 participants total.)*

3. Half: Participants were shown the interactive binary search tree visualization, and they could complete all of the same interactions with it that participants in the Full condition could complete. However, the tool did not synthesize any code based on these interactions. *(10 participants total.)*

All versions of the webpage had a text window for participants to write and edit code, as well as an interface for writing test cases. We did not show the labels Control, Half, or Full to the participants.

We note that the synthesizer behind the Full condition is not a standalone tool; it is a single-purpose synthesizer that supports only the two tasks in our target tutorial, BST search and BST insertion. The synthesizer is available with the rest of the interface implementation and lesson plan webpage at the following link:

`https://github.com/dhanyajayagopal/synthesis_in_cs_education`

3.4.4 Research Questions

We operationalize the amount that a participant learned in regard to a given question as the difference between their score on that question on the post-test and their score on that question on the pre-test. We call this quantity the *learning outcome* for a given question. Our research question is thus:

Table 3.1: *One-way analysis of variation (ANOVA) in learning outcome among the three conditions for each question in our quizzes.*

Question	Mean (Control)	Mean (Half)	Mean (Full)	F-value	p-value
1	0.0444	0.100	0.220	0.800	0.460
2	0.111	0.000	0.000	0.204	0.817
3	−0.222	0.100	0.000	1.488	0.244

Table 3.2: *One-way analysis of variation (ANOVA) in time spent (in minutes) among the three conditions for each portion of our learning module.*

Portion	Mean (Control)	Mean (Half)	Mean (Full)	F-value	p-value	
BST Search (Portion 1)	9.861	10.275	10.317	0.013	0.987	
BST Insertion (Portion 2)	16.825	11.404	2.641	6.555	0.007	*

RQ1. Is there significant variation in learning outcome for any of the three questions across the three conditions of our study?

As a secondary research question, we were interested in whether learning was accelerated by access to a synthesizer. Even if learning outcomes are the same across conditions, a change in speed could indicate that synthesis access affects learning. Thus we also asked:

RQ2. Is there significant variation in time spent on the learning module (either the search or insertion portion) among any of the three conditions of our study?

3.4.5 Results

Throughout this section, we take our confidence level $\alpha = 0.95$.

For **RQ1** results, see the bar chart of the average learning outcomes for each question (Figure 3.1). We performed a one-way analysis of variations (ANOVA) test among the three experimental conditions (Table 3.1). We discovered that there was no statistically significant variation among any of the conditions. *Our results do not support the hypothesis that there is a significant difference in learning outcomes from using a synthesis tool compared to programming without synthesis support.*

For **RQ2** results, see the box plot of the time taken for each portion of the learning module (Figure 3.2). We performed a one-way ANOVA among the three experimental conditions (Figure 3.2). For the first portion (search), there was no statistically significant variation among the conditions. However, *for the second portion (insertion), we*

Table 3.3: *Post-hoc Tukey's honest significant difference (HSD) test for time taken on the second portion of the learning module (BST insertion).*

Condition Comparison	Statistic	p-value	
Control–Half	5.421	0.467	
Control–Full	14.185	0.010	*
Half–Full	8.763	0.059	

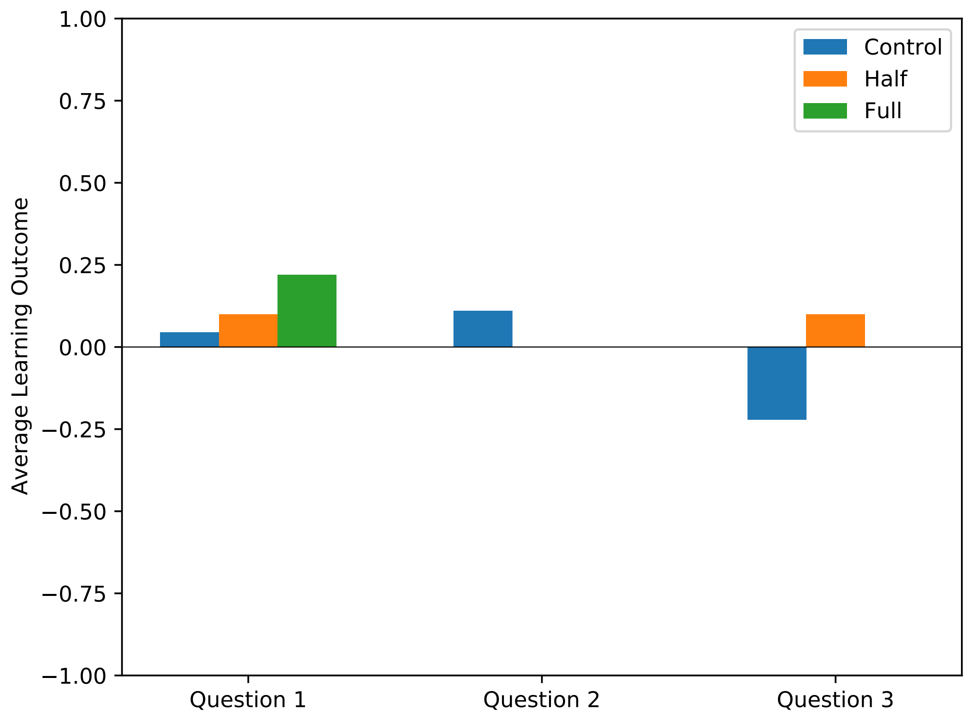


Figure 3.1: *Learning outcomes by question and condition. Question 1 relates to insertion, Question 2 to search, and Question 3 to both insertion and search.*

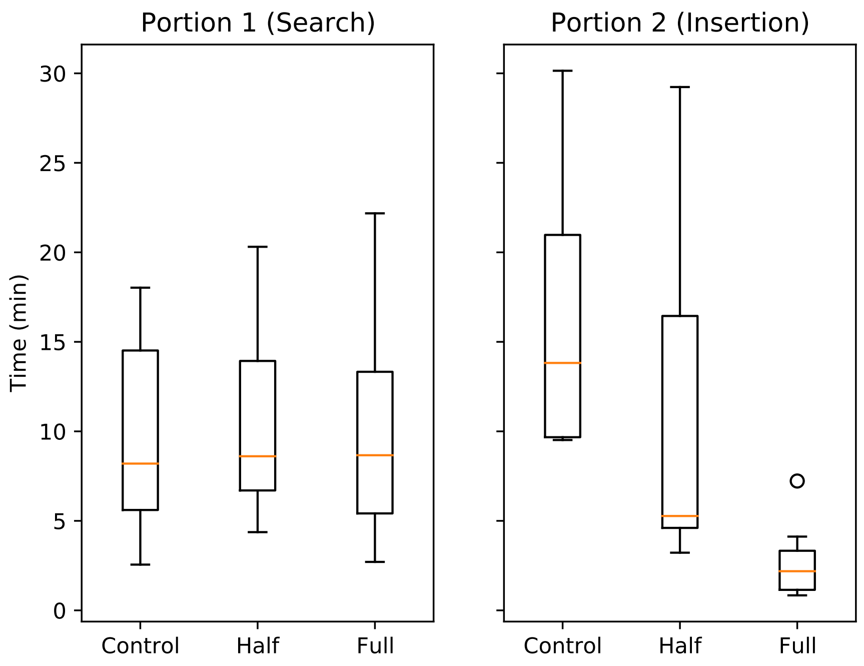


Figure 3.2: *Time (in minutes) spent on each portion of the lesson plan.*

discovered that there was statistically significant variation.

We performed a post-hoc Tukey's honest significant difference (HSD) test (Table 3.3) and discovered that participants in the Full condition completed the second portion of the learning module faster than those in the Control condition with statistical significance. No other comparisons were statistically significant. There are a few possible explanations for this result:

- The synthesizer made students working through the second portion disproportionately more confident that they understood the material quickly.
- The synthesizer made the second portion disproportionately easier.
- Once students became familiar with the synthesizer on the first portion, they could use it more efficiently in the second portion.

Although students did not demonstrate average learning outcomes above even a quarter of a point for our study, this experiment suggests an interesting future direction: can we use program synthesizers to *speed up* student learning, even if the learning outcomes do not differ significantly?

3.4.6 Additional Qualitative Observations

In addition to our quantitative results, we observed a few qualitative observations during the study:

- In the Control version, most participants attempted to come up with an iterative solution to the problems. However, in the Half and Full versions, more participants attempted a recursive solution.
- Participants who were using the Half and Full versions did not know when to stop adding demonstrations and when to start writing or synthesizing code. This resonates with the consequences of user-triggered synthesis we observed in Section 2.6.2.
- Almost every participant referenced the lesson notes before coding and after interacting with the demonstration portion of the module.
- In all versions, most participants struggled to remember the null case. Participants generally remembered and added a null case after the interface marked their submission as incorrect.

3.5 Discussion

The outcome of our study is an important negative result, especially because it sounds a clear note of caution in the synthesis plus education space, which has become popular over the past few years.

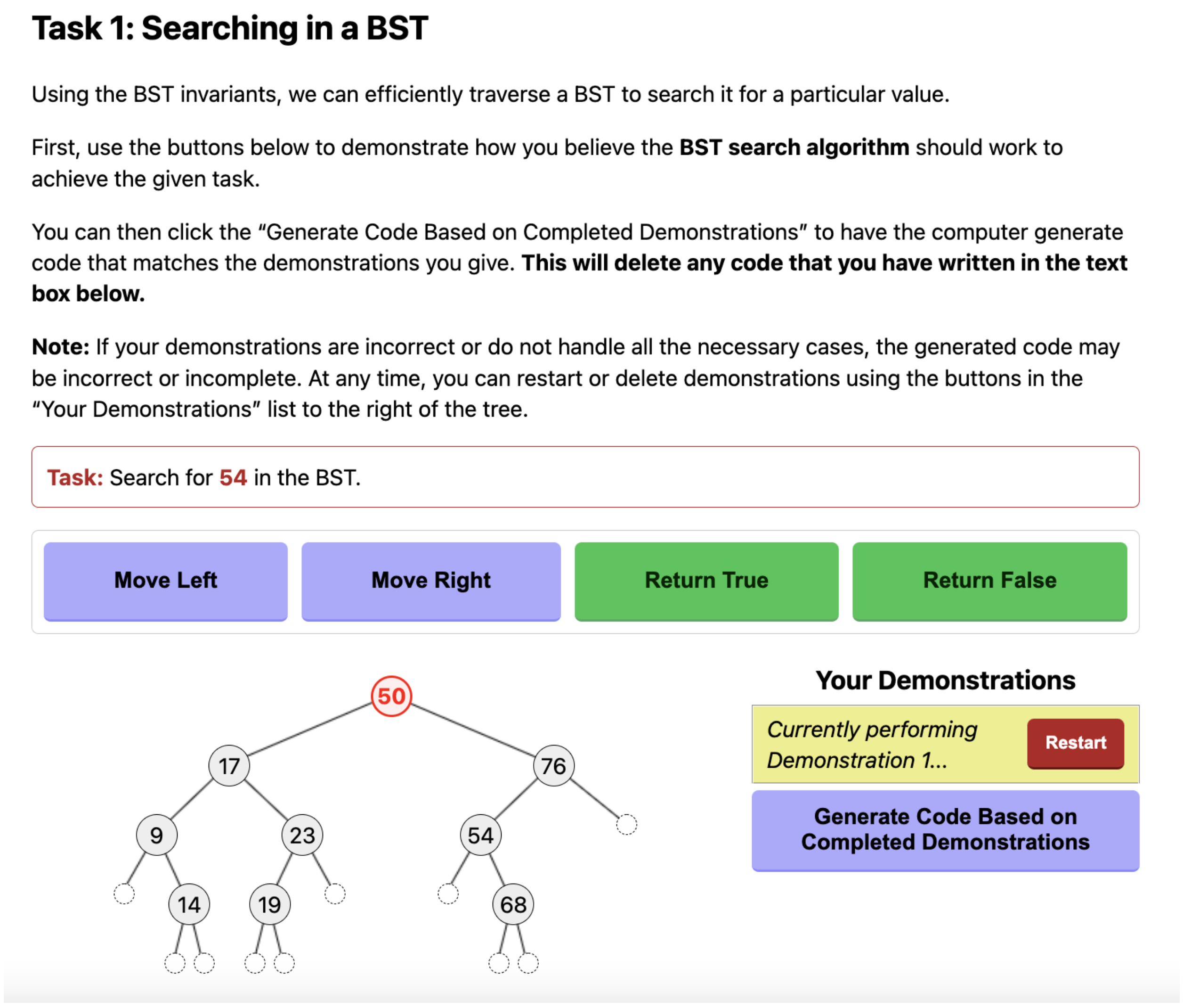
The past few years has shown that program synthesis is an effective way to complete coding tasks [41, 85, 50, 10, 80, 18, 57, 36, 72, 25, 88, 60]. Past work further indicates that synthesis tools are learnable [26, 10, 85, 40, 21] and usable [51, 83, 20] ways to complete tasks. Because these tools are accessible to non-programmers and novice programmers, these findings have been followed by speculation that synthesis tools may help students *learn* programming. Up until the execution of this study, no studies have tested this hypothesis. Nevertheless, there has been a spate of research introducing synthesis tools targeted at CS students [34, 19, 79, 71, 44, 15, 32, 75, 6]. These works have tested important metrics like whether the tools find bugs, but not whether the tools affect student learning outcomes, either positively or negatively.

This study assesses a single synthesis tool and clearly should not be taken as evidence that synthesis tools *cannot* help students learn. However, it offers an important lesson for the future of educational program synthesis tools: simply adding program synthesis into an educational interaction does not automatically boost learning outcomes. Although these tools may help students reach functioning programs, this is not the same as helping students learn.

3.6 Conclusion

Program synthesizers are becoming more and more commonplace, even in the context of programming education. A plethora of synthesizers are being built for this space—nearly all of them focused on synthesizing error corrections for student program submissions. While these tools have been found to work (i.e they successfully synthesize code for a high percentage of tested inputs), there has been no research on their impact on learning outcomes.

In this chapter we presented an overview of the literature that sits at the intersection of program synthesis and computer science education. We then presented a short study to answer our research question of whether or not there exists evidence to support the claim that program synthesizers improve learning outcomes.


Task 1: Searching in a BST

Using the BST invariants, we can efficiently traverse a BST to search it for a particular value.

First, use the buttons below to demonstrate how you believe the **BST search algorithm** should work to achieve the given task.

You can then click the "Generate Code Based on Completed Demonstrations" to have the computer generate code that matches the demonstrations you give. **This will delete any code that you have written in the text box below.**

Note: If your demonstrations are incorrect or do not handle all the necessary cases, the generated code may be incorrect or incomplete. At any time, you can restart or delete demonstrations using the buttons in the "Your Demonstrations" list to the right of the tree.

Task: Search for **54** in the BST.

Move Left
Move Right
Return True
Return False
50
17
76
9
23
54
14
19
68
Your Demonstrations
Currently performing Demonstration 1...
Restart
Generate Code Based on Completed Demonstrations

Figure 3.3: *Part of the FULL version of our learning module.* The HALF version does not have the column labeled "Your Demonstrations" (and thus the user cannot "Generate Code Based on Completed Demonstrations") or any text related to those features. The CONTROL version does not have any of the interactive features shown in this figure.

Table 3.4: *Question 1 of our quiz. - Relating to Insertion.* We did not explain or show the scoring mechanism to the participants.

QUESTION TEXT	Consider a BINARY SEARCH TREE T_0, containing N numerical values whose root is the median value in the tree. (The median value is the value such that half of the values are less and the other half are greater.)
	Suppose that N additional values were added to the tree T_0, using the standard insertion algorithm, to create an updated tree T_1. The newly-added values are all larger than the largest value that T_0 stored, and they are added in an unknown order.
	Note: N refers to both the original number of nodes in T_0 and the number of nodes added to create T_1. Hence, T_1 contains $2N$ nodes. You may assume $N > 0$.
	Which of these statements is/are true about the creation of T_1?
	Select **ALL** that apply:
POSSIBLE ANSWERS	☐ The root of T_1 holds the median value of T_1.
	☐ There are more values in the right subtree of T_1 than in the left subtree.
	☐ Adding the last new value could take time proportional to $\log N$.
	☐ Adding the last new value could take time proportional to N.
	☐ Adding the last new value could take time proportional to $N \log N$.
SCORING MECHANISM	**Scoring was from 0–1.** All participants started with a score of 1 and 0.2 points were deducted for each incorrect categorization (i.e selected an incorrect answer choice or failed to select a correct answer choice).

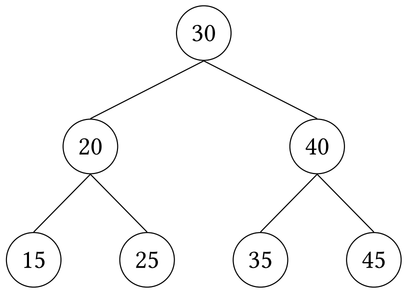
Table 3.5: *Question 2 of our quiz. - Relating to Search.* We did not explain or show the scoring mechanism to the participants.

Question Text

Suppose you just wrote the `search` method for your `BinarySearchTree` implementation. The specification is below:

`search(e)`
Input: An element *e*
Description: Determines if *e* is in the tree.
Return: `True` if *e* is in the tree; `False` if not.

To test your new method, you build the tree below:



You are not sure if your implementation has flaws. Please select the four integer values that you should search for in this tree to gain as much confidence as possible in the correctness of your algorithm.

Select **ONE**:

Possible Answers

- ○ 10, 15, 20, 30
- ○ 15, 25, 35, 45
- ○ 15, 20, 30, 40
- ○ 20, 30, 40, 50
- ○ 25, 30, 40, 50

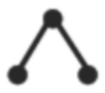
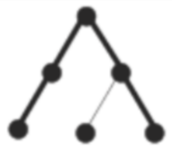
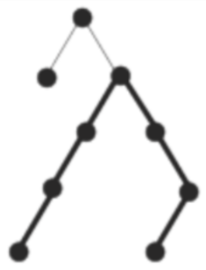
Scoring Mechanism

Scoring was from 0–1. A point was given if and only if the selected answer choice was correct.

Table 3.6: *Question 3 of our quiz. - Relating to Search & Insertion.* We did not explain or show the scoring mechanism to the participants.

Question Text

The *width* of a `BinaryTree` is the maximal number of nodes on a path from one node in the tree to another. (The width of an empty `BinaryTree` is 0.) Here are some examples, with the relevant path indicated in bold. Note in particular that the relevant path does not necessarily include the root of the binary `BinaryTree`.

width	1	3	5	7
tree				

The *height* of a node in a `BinaryTree` is the number of nodes on the longest path from node down to leaf. (**The height of a leaf is 1.**) Assume that each `BinaryTree` node contains its height in an instance variable named `height`. You may also assume you have a function, `max`, which returns the greater of two values.

Assume you are writing a function named `width` that, when given a `node` as an argument, returns the width of the `BinaryTree` rooted at that `node`. In authoring this function, consider a `node` with both left and right children. What is the width of the `BinaryTree` rooted at that `node`? We have started the answer for you.

The `width` of a `node` is the maximum of:

1. `width(node.left)`
2. `width(node.right)`
3. ____________________

Select **ONE**:

Possible Answers

- ○ `node.left.height + node.right.height`
- ○ `width(node.left) + width(node.right)`
- ○ `1 + node.left.height + node.right.height`
- ○ `1 + width(node.left) + width(node.right)`
- ○ `1 + max(width(node.left) + node.right.height, node.left.height + width(node.right))`

Scoring Mechanism

Scoring was from 0–1. A point was given if and only if the selected answer choice was correct.

Chapter 4

Conclusion

This thesis covers two dimensions of research within the realm of program synthesizers in educational contexts: (1) learnability of program synthesizers and (2) how program synthesizers affect student learning. The first dimension goes over different design decisions and their impact on learnability, and the second dimension explores a question that has little research addressing it to date.

The goal of investigating the learnability of existing program synthesizers is to be able to identify the aspects of program synthesizers that contribute to and detract from their learnability by novice programmers. The goal of investigating the impact of program synthesizers on students is to understand how and when these tools can serve novice programmers best. We hope that future work will be able to both (1) identify additional design dimensions that impact learnability and (2) find evidence of program synthesizers improving student learning outcomes.

Bibliography

[1] D. Alan Allport, Elizabeth A. Styles, and Shulan Hsieh. 1994. Shifting Intentional Set: Exploring the Dynamic Control of Tasks. In *Attention and Performance 15: Conscious and Nonconscious Information Processing*. The MIT Press, 421–452.

[2] Saleema Amershi, Dan Weld, Mihaela Vorvoreanu, Adam Fourney, Besmira Nushi, Penny Collisson, Jina Suh, Shamsi Iqbal, Paul N. Bennett, Kori Inkpen, Jaime Teevan, Ruth Kikin-Gil, and Eric Horvitz. 2019. Guidelines for Human-AI Interaction. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems (CHI '19)*. Association for Computing Machinery, 1–13. https://doi.org/10.1145/3290605.3300233

[3] Matej Balog, Alexander Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. 2017. DeepCoder: Learning to Write Programs. In *Proceedings of ICLR'17*.

[4] Osbert Bastani, Xin Zhang, and Armando Solar-Lezama. 2019. Synthesizing Queries via Interactive Sketching. *CoRR* abs/1912.12659 (2019). arXiv:1912.12659 http://arxiv.org/abs/1912.12659

[5] Victoria Bellotti and Keith Edwards. 2001. Intelligibility and Accountability: Human Considerations in Context-Aware Systems. *Human–Computer Interaction* 16, 2-4 (Dec. 2001), 193–212. https://doi.org/10.1207/S15327051HCI16234_05

[6] Sahil Bhatia, Pushmeet Kohli, and Rishabh Singh. 2018. Neuro-Symbolic Program Corrector for Introductory Programming Assignments. In *Proceedings of the 40th International Conference on Software Engineering* (Gothenburg, Sweden) *(ICSE '18)*. Association for Computing Machinery, New York, NY, USA, 60–70. https://doi.org/10.1145/3180155.3180219

[7] Virginia Braun and Victoria Clarke. 2006. Using Thematic Analysis in Psychology. *Qualitative Research in Psychology* 3, 2 (Jan. 2006), 77–101. https://doi.org/10.1191/1478088706qp063oa

[8] Francisco Enrique Vicente Castro and Kathi Fisler. 2016. On the Interplay Between Bottom-Up and Datatype-Driven Program Design. In *SIGCSE Technical Symposium*. https://doi.org/10.1145/2839509.2844574

[9] Francisco Enrique Vicente Castro and Kathi Fisler. 2020. *Qualitative Analyses of Movements Between Task-Level and Code-Level Thinking of Novice Programmers*. Association for Computing Machinery, 487–493. https://doi.org/10.1145/3328778.3366847

[10] Sarah E. Chasins, Maria Mueller, and Rastislav Bodik. 2018. Rousillon: Scraping Distributed Hierarchical Web Data. In *Proceedings of the 31st Annual ACM Symposium on User Interface Software and Technology (UIST '18)*. Association for Computing Machinery, 963–975. https://doi.org/10.1145/3242587.3242661

[11] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code. (2021). arXiv:2107.03374 [cs.LG]

[12] Will Crichton. 2019. Human-Centric Program Synthesis. *CoRR* abs/1909.12281 (2019). arXiv:1909.12281 `http://arxiv.org/abs/1909.12281`

[13] Tyne Crow, Andrew Luxton-Reilly, and Burkhard Wuensche. 2018. Intelligent Tutoring Systems for Programming Education: A Systematic Review. In *Proceedings of the 20th Australasian Computing Education Conference* (Brisbane, Queensland, Australia) *(ACE '18)*. Association for Computing Machinery, New York, NY, USA, 53–62. `https://doi.org/10.1145/3160489.3160492`

[14] Loris D'antoni, Dileep Kini, Rajeev Alur, Sumit Gulwani, Mahesh Viswanathan, and Björn Hartmann. 2015. How Can Automatic Feedback Help Students Construct Automata? *ACM Trans. Comput.-Hum. Interact.* 22, 2, Article 9 (mar 2015), 24 pages. `https://doi.org/10.1145/2723163`

[15] Loris D'Antoni, Roopsha Samanta, and Rishabh Singh. 2016. Qlose: Program Repair with Quantitative Objectives. In *Computer Aided Verification*. Springer International Publishing, 383–401. `https://doi.org/10.1007/978-3-319-41540-6_21`

[16] Simon P. Davies and Adrian M. Castell. 1994. From Individuals to Groups Through Artifacts: The Changing Semantics of Design in Software Development. In *User-Centred Requirements for Software Engineering Environments*, David J. Gilmore, Russel L. Winder, and Françoise Détienne (Eds.). `https://doi.org/10.1007/978-3-662-03035-6_2`

[17] Asaf Degani. 1996. *Modeling Human-Machine Systems: On Modes, Error, and Patterns of Interaction.* Ph.D. Dissertation. Georgia Institute of Technology.

[18] Ian Drosos, Titus Barik, Philip J. Guo, Robert DeLine, and Sumit Gulwani. 2020. *Wrex: A Unified Programming-by-Example Interaction for Synthesizing Readable Code for Data Scientists.* Association for Computing Machinery, 1–12. `https://doi.org/10.1145/3313831.3376442`

[19] Molly Q Feldman, Yiting Wang, William E. Byrd, François Guimbretière, and Erik Andersen. 2019. Towards Answering "Am I on the Right Track?" Automatically Using Program Synthesis. In *Proceedings of the 2019 ACM SIGPLAN Symposium on SPLASH-E* (Athens, Greece) *(SPLASH-E 2019)*. Association for Computing Machinery, New York, NY, USA, 13–24. `https://doi.org/10.1145/3358711.3361626`

[20] Kasra Ferdowsifard, Shraddha Barke, Hila Peleg, Sorin Lerner, and Nadia Polikarpova. 2021. LooPy: Interactive Program Synthesis with Control Structures. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 153 (Oct. 2021), 29 pages. `https://doi.org/10.1145/3485530`

[21] Kasra Ferdowsifard, Allen Ordookhanians, Hila Peleg, Sorin Lerner, and Nadia Polikarpova. 2020. *Small-Step Live Programming by Example.* Association for Computing Machinery, 614–626. https://doi.org/10.1145/3379337.3415869

[22] John K. Feser, Swarat Chaudhuri, and Isil Dillig. 2015. Synthesizing Data Structure Transformations from Input-Output Examples. *ACM SIGPLAN Notices* 50, 6 (June 2015), 229–239. https://doi.org/10.1145/2813885.2737977

[23] Leah Findlater and Joanna McGrenere. 2004. A Comparison of Static, Adaptive, and Adaptable Menus. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '04).* Association for Computing Machinery, 89–96. https://doi.org/10.1145/985692.985704

[24] Kathi Fisler and Francisco Enrique Vicente Castro. 2017. Sometimes, Rainfall Accumulates: Talk-Alouds with Novice Functional Programmers. In *International Conference on Computing Education Research (ICER).* https://doi.org/10.1145/3105726.3106183

[25] Nat Friedman. 2021. Introducing GitHub Copilot: your AI pair programmer. https://github.blog/2021-06-29-introducing-github-copilot-ai-pair-programmer/. Accessed: 2022-04-04.

[26] Joel Galenson, Philip Reames, Rastislav Bodik, Björn Hartmann, and Koushik Sen. 2014. CodeHint: Dynamic and Interactive Synthesis of Code Snippets. In *Proceedings of the 36th International Conference on Software Engineering (ICSE 2014).* Association for Computing Machinery, 653–663. https://doi.org/10.1145/2568225.2568250

[27] Ivan Gavran, Eva Darulova, and Rupak Majumdar. 2020. Interactive Synthesis of Temporal Specifications from Examples and Natural Language. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 201 (Nov. 2020), 26 pages. https://doi.org/10.1145/3428269

[28] GitHub Inc. 2021. GitHub Copilot: Your AI pair programmer. https://copilot.github.com/. Accessed: 2022-03-31.

[29] Ken Goldman, Paul Gross, Cinda Heeren, Geoffrey L. Herman, Lisa Kaczmarczyk, Michael C. Loui, and Craig Zilles. 2010. Setting the Scope of Concept Inventories for Introductory Computing Subjects. *ACM Trans. Comput. Educ.* 10, 2, Article 5 (jun 2010), 29 pages. https://doi.org/10.1145/1789934.1789935

[30] Sumit Gulwani. 2011. Automating String Processing in Spreadsheets Using Input-Output Examples. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '11).* Association for Computing Machinery, 317–330. https://doi.org/10.1145/1926385.1926423

[31] Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. 2011. Synthesis of Loop-Free Programs. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '11).* Association for Computing Machinery, 62–73. https://doi.org/10.1145/1993498.1993506

[32] Sumit Gulwani, Ivan Radiček, and Florian Zuleger. 2018. Automated Clustering and Program Repair for Introductory Programming Assignments. *SIGPLAN Not.* 53, 4 (jun 2018), 465–480. https://doi.org/10.1145/3296979.3192387

[33] Sumit Gulwani and Ramarathnam Venkatesan. 2009. *Component Based Synthesis Applied to Bitvector Circuits.* Technical Report MSR-TR-2010-12. Microsoft Research.

[34] Björn Hartmann, Daniel MacDougall, Joel Brandt, and Scott R. Klemmer. 2010. What Would Other Programmers Do: Suggesting Solutions to Error Messages. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Atlanta, Georgia, USA) *(CHI '10)*. Association for Computing Machinery, New York, NY, USA, 1019–1028. https://doi.org/10.1145/1753326.1753478

[35] Andrew Head, Elena Glassman, Gustavo Soares, Ryo Suzuki, Lucas Figueredo, Loris D'Antoni, and Björn Hartmann. 2017. Writing Reusable Code Feedback at Scale with Mixed-Initiative Program Synthesis. In *Proceedings of the Fourth (2017) ACM Conference on Learning @ Scale* (Cambridge, Massachusetts, USA) *(L@S '17)*. Association for Computing Machinery, New York, NY, USA, 89–98. https://doi.org/10.1145/3051457.3051467

[36] Brian Hempel and Ravi Chugh. 2016. Semi-Automated SVG Programming via Direct Manipulation. In *Proceedings of the 29th Annual Symposium on User Interface Software and Technology* (Tokyo, Japan) *(UIST '16)*. Association for Computing Machinery, New York, NY, USA, 379–390. https://doi.org/10.1145/2984511.2984575

[37] Austin Z. Henley, Julian Ball, Benjamin Klein, Aiden Rutter, and Dylan Lee. 2021. *An Inquisitive Code Editor for Addressing Novice Programmers' Misconceptions of Program Behavior.* IEEE Press, 165–170. https://doi.org/10.1109/ICSE-SEET52601.2021.00026

[38] K. Höök. 2000. Steps to Take before Intelligent User Interfaces Become Real. *Interacting with Computers* 12, 4 (2000), 409–426. https://doi.org/10.1016/S0953-5438(99)00006-5

[39] Eric Horvitz. 1999. Principles of Mixed-Initiative User Interfaces. In *Proceedings of CHI '99, ACM SIGCHI Conference on Human Factors in Computing Systems, Pittsburgh, PA, ACM Press.* 159–166.

[40] Thibaud Hottelier, Ras Bodik, and Kimiko Ryokai. 2014. Programming by Manipulation for Layout. In *Proceedings of the 27th Annual ACM Symposium on User Interface Software and Technology (UIST '14)*. Association for Computing Machinery, 231–241. https://doi.org/10.1145/2642918.2647378

[41] Jingmei Hu, Priyan Vaithilingam, Stephen Chong, Margo Seltzer, and Elena L. Glassman. 2021. Assuage: Assembly Synthesis Using A Guided Exploration. In *The 34th Annual ACM Symposium on User Interface Software and Technology (UIST '21)*. Association for Computing Machinery, 134–148. https://doi.org/10.1145/3472749.3474740

[42] A. T. Jersild. 1927. Mental Set and Shift. *Archives of Psychology* 14, 89 (1927).

[43] Susmit Jha, Sumit Gulwani, Sanjit A. Seshia, and Ashish Tiwari. 2010. Oracle-Guided Component-Based Program Synthesis. In *2010 ACM/IEEE 32nd International Conference on Software Engineering*, Vol. 1. 215–224. https://doi.org/10.1145/1806799.1806833

[44] Garvit Juniwal, Alexandre Donzé, Jeff C. Jensen, and Sanjit A. Seshia. 2014. CPSGrader: Synthesizing Temporal Logic Testers for Auto-Grading an Embedded Systems Laboratory. In *Proceedings of the 14th International Conference on Embedded Software* (New Delhi, India) *(EMSOFT '14)*. Association for Computing Machinery, New York, NY, USA, Article 24, 10 pages. https://doi.org/10.1145/2656045.2656053

[45] Etienne Kneuss, Ivan Kuraj, Viktor Kuncak, and Philippe Suter. 2013. Synthesis modulo Recursive Functions. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA '13)*. Association for Computing Machinery, 407–426. https://doi.org/10.1145/2509136.2509555

[46] Todd Kulesza, Margaret Burnett, Weng-Keen Wong, and Simone Stumpf. 2015. Principles of Explanatory Debugging to Personalize Interactive Machine Learning. In *Proceedings of the 20th International Conference on Intelligent User Interfaces (IUI '15)*. Association for Computing Machinery, 126–137. https://doi.org/10.1145/2678025.2701399

[47] Tessa Lau. 2009. Why Programming-By-Demonstration Systems Fail: Lessons Learned for Usable AI. *AI Magazine* 30, 4 (Oct. 2009), 65–65. https://doi.org/10.1609/aimag.v30i4.2262

[48] Sorin Lerner. 2020. *Projection Boxes: On-the-Fly Reconfigurable Visualization for Live Programming*. Association for Computing Machinery, 1–7. https://doi.org/10.1145/3313831.3376494

[49] Colleen M. Lewis. 2012. The Importance of Students' Attention to Program State: A Case Study of Debugging Behavior. In *Proceedings of the Ninth Annual International Conference on International Computing Education Research (ICER '12)*. Association for Computing Machinery, 127–134. https://doi.org/10.1145/2361276.2361301

[50] Toby Jia-Jun Li, Amos Azaria, and Brad A. Myers. 2017. SUGILITE: Creating Multimodal Smartphone Automation by Demonstration. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems (CHI '17)*. Association for Computing Machinery, 6038–6049. https://doi.org/10.1145/3025453.3025483

[51] Toby Jia-Jun Li, Marissa Radensky, Justin Jia, Kirielle Singarajah, Tom M. Mitchell, and Brad A. Myers. 2019. PUMICE: A Multi-Modal Agent That Learns Concepts and Conditionals from Natural Language and Demonstrations. In *Proceedings of the 32nd Annual ACM Symposium on User Interface Software and Technology (UIST '19)*. Association for Computing Machinery, 577–589. https://doi.org/10.1145/3332165.3347899

[52] Brian Y. Lim and Anind K. Dey. 2009. Assessing Demand for Intelligibility in Context-Aware Applications. In *Proceedings of the 11th International Conference on Ubiquitous Computing (UbiComp '09)*. Association for Computing Machinery, 195–204. https://doi.org/10.1145/1620545.1620576

[53] Justin Lubin, Nick Collins, Cyrus Omar, and Ravi Chugh. 2020. Program Sketching with Live Bidirectional Evaluation. *Proceedings of the ACM on Programming Languages* 4, ICFP (Aug. 2020), 109:1–109:29. https://doi.org/10.1145/3408991

[54] Ewa Luger and Abigail Sellen. 2016. "Like Having a Really Bad PA": The Gulf between User Expectation and Experience of Conversational Agents. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems (CHI '16)*. Association for Computing Machinery, 5286–5297. https://doi.org/10.1145/2858036.2858288

[55] David Mandelin, Lin Xu, Rastislav Bodík, and Doug Kimelman. 2005. Jungloid Mining: Helping to Navigate the API Jungle. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '05)*. Association for Computing Machinery, 48–61. https://doi.org/10.1145/1065010.1065018

[56] Guillaume Marceau, Kathi Fisler, and Shriram Krishnamurthi. 2011. 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* (Portland, Oregon, USA) *(Onward! 2011)*. Association for Computing Machinery, New York, NY, USA, 3–18. https://doi.org/10.1145/2048237.2048241

[57] Mikaël Mayer, Gustavo Soares, Maxim Grechkin, Vu Le, Mark Marron, Oleksandr Polozov, Rishabh Singh, Benjamin Zorn, and Sumit Gulwani. 2015. User Interaction Models for Disambiguation in Programming by Example. In *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology (UIST '15)*. Association for Computing Machinery, 291–301. https://doi.org/10.1145/2807442.2807459

[58] Anders Miltner, Sumit Gulwani, Vu Le, Alan Leung, Arjun Radhakrishna, Gustavo Soares, Ashish Tiwari, and Abhishek Udupa. 2019. On the Fly Synthesis of Edit Suggestions. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 143 (Oct. 2019), 29 pages. https://doi.org/10.1145/3360569

[59] Chelsea Myers, Anushay Furqan, Jessica Nebolsky, Karina Caro, and Jichen Zhu. 2018. Patterns for How Users Overcome Obstacles in Voice User Interfaces. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems (CHI '18)*. Association for Computing Machinery, 1–7. https://doi.org/10.1145/3173574.3173580

[60] Sujata Narayana. 2020. Power BI Desktop August 2020 Feature Summary: Text/CSV By Example (preview). https://powerbi.microsoft.com/en-us/blog/power-bi-desktop-august-2020-feature-summary/#_text_csv. Accessed: 2022-04-05.

[61] Greg L. Nelson, Benjamin Xie, and Amy J. Ko. 2017. Comprehension First: Evaluating a Novel Pedagogy and Tutoring System for Program Tracing in CS1. In *Proceedings of the 2017 ACM Conference on International Computing Education Research (ICER '17)*. Association for Computing Machinery, 2–11. https://doi.org/10.1145/3105726.3106178

[62] Donald A. Norman. 1994. How Might People Interact with Agents. *Commun. ACM* 37, 7 (July 1994), 68–71. https://doi.org/10.1145/176789.176796

[63] Peter-Michael Osera and Steve Zdancewic. 2015. Type-and-Example-Directed Program Synthesis. *SIGPLAN Not.* 50, 6 (June 2015), 619–630. https://doi.org/10.1145/2813885.2738007

[64] Hila Peleg, Shachar Itzhaky, and Sharon Shoham. 2018. Abstraction-Based Interaction Model for Synthesis. In *Verification, Model Checking, and Abstract Interpretation*, Isil Dillig and Jens Palsberg (Eds.). Springer International Publishing, 382–405. https://doi.org/10.1007/978-3-319-73721-8_18

[65] Hila Peleg, Sharon Shoham, and Eran Yahav. 2018. Programming Not Only by Example. In *Proceedings of the 40th International Conference on Software Engineering (ICSE '18)*. Association for Computing Machinery, 1114–1124. https://doi.org/10.1145/3180155.3180189

[66] Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. 2016. Program Synthesis from Polymorphic Refinement Types. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '16)*. Association for Computing Machinery, 522–538. https://doi.org/10.1145/2908080.2908093

[67] Leo Porter, Daniel Zingaro, Soohyun Nam Liao, Cynthia Taylor, Kevin C. Webb, Cynthia Lee, and Michael Clancy. 2019. BDSI: A Validated Concept Inventory for Basic Data Structures. In *Proceedings of the 2019 ACM Conference on International Computing Education Research* (Toronto ON, Canada) *(ICER '19)*. Association for Computing Machinery, New York, NY, USA, 111–119. https://doi.org/10.1145/3291279.3339404

[68] Jef Raskin. 2000. *The Humane Interface: New Directions for Designing Interactive Systems.* Addison-Wesley Professional.

[69] Ruth Ravichandran, Sang-Wha Sien, Shwetak N. Patel, Julie A. Kientz, and Laura R. Pina. 2017. Making Sense of Sleep Sensors: How Sleep Sensing Technologies Support and Undermine Sleep Health. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems (CHI '17)*. Association for Computing Machinery, 6864–6875. https://doi.org/10.1145/3025453.3025557

[70] Robert D. Rogers and Stephen Monsell. 1995. Costs of a Predictible Switch between Simple Cognitive Tasks. *Journal of Experimental Psychology: General* 124, 2 (1995), 207–231. https://doi.org/10.1037/0096-3445.124.2.207

[71] Reudismam Rolim, Gustavo Soares, Loris D'Antoni, Oleksandr Polozov, Sumit Gulwani, Rohit Gheyi, Ryo Suzuki, and Björn Hartmann. 2017. Learning Syntactic Program Transformations from Examples. In *Proceedings of the 39th International Conference on Software Engineering* (Buenos Aires, Argentina) *(ICSE '17)*. IEEE Press, 404–415. https://doi.org/10.1109/ICSE.2017.44

[72] Chad Rothschiller. 2012. Flash Fill. https://www.microsoft.com/en-us/microsoft-365/blog/2012/08/09/flash-fill/. Accessed: 2022-04-04.

[73] Eric Schkufza, Rahul Sharma, and Alex Aiken. 2013. Stochastic Superoptimization. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '13)*. Association for Computing Machinery, 305–316. https://doi.org/10.1145/2451116.2451150

[74] Steven C. Shaffer. 2005. Ludwig: An Online Programming Tutoring and Assessment System. *SIGCSE Bull.* 37, 2 (June 2005), 56–60. https://doi.org/10.1145/1083431.1083464

[75] Rishabh Singh, Sumit Gulwani, and Armando Solar-Lezama. 2013. Automated Feedback Generation for Introductory Programming Assignments. *SIGPLAN Not.* 48, 6 (jun 2013), 15–26. https://doi.org/10.1145/2499370.2462195

[76] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. 2006. Combinatorial Sketching for Finite Programs. *SIGARCH Comput. Archit. News* 34, 5 (Oct. 2006), 404–415. https://doi.org/10.1145/1168919.1168907

[77] Elliot Soloway and Kate Ehrlich. 1986. Empirical Studies of Programming Knowledge. In *Readings in Artificial Intelligence and Software Engineering*, Charles Rich and Richard C. Waters (Eds.). https://doi.org/10.1016/B978-0-934613-12-5.50042-2

[78] James C. Spohrer and Elliot Soloway. 1986. Novice Mistakes: Are the Folk Wisdoms Correct? *Communications of the ACM* (July 1986). https://doi.org/10.1145/6138.6145

[79] Ryo Suzuki, Gustavo Soares, Elena Glassman, Andrew Head, Loris D'Antoni, and Björn Hartmann. 2017. Exploring the Design Space of Automatically Synthesized Hints for Introductory Programming Assignments. In *Proceedings of the 2017 CHI Conference Extended Abstracts on Human Factors in Computing Systems* (Denver, Colorado, USA) *(CHI EA '17)*. Association for Computing Machinery, New York, NY, USA, 2951–2958. https://doi.org/10.1145/3027063.3053187

[80] Amanda Swearngin, Chenglong Wang, Alannah Oleson, James Fogarty, and Amy J. Ko. 2020. *Scout: Rapid Exploration of Interface Layout Alternatives through High-Level Design Constraints.* Association for Computing Machinery, 1–13. https://doi.org/10.1145/3313831.3376593

[81] Cynthia Taylor, Michael Clancy, Kevin C. Webb, Daniel Zingaro, Cynthia Lee, and Leo Porter. 2020. *The Practical Details of Building a CS Concept Inventory.* Association for Computing Machinery, New York, NY, USA, 372–378. https://doi.org/10.1145/3328778.3366903

[82] Nava Tintarev and Judith Masthoff. 2015. Explaining Recommendations: Design and Evaluation. In *Recommender Systems Handbook*, Francesco Ricci, Lior Rokach, and Bracha Shapira (Eds.). Springer US, 353–382. https://doi.org/10.1007/978-1-4899-7637-6_10

[83] Priyan Vaithilingam and Philip J. Guo. 2019. Bespoke: Interactively Synthesizing Custom GUIs from Command-Line Applications By Demonstration. In *Proceedings of the 32nd Annual ACM Symposium on User Interface Software and Technology (UIST '19).* Association for Computing Machinery, 563–576. https://doi.org/10.1145/3332165.3347944

[84] Chenglong Wang, Yu Feng, Rastislav Bodik, Alvin Cheung, and Isil Dillig. 2019. Visualization by Example. *Proceedings of the ACM on Programming Languages* 4, POPL (Dec. 2019), 49:1–49:28. https://doi.org/10.1145/3371117

[85] Chenglong Wang, Yu Feng, Rastislav Bodik, Isil Dillig, Alvin Cheung, and Amy J Ko. 2021. Falx: Synthesis-Powered Visualization Authoring. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems (CHI '21).* Association for Computing Machinery, Article 106, 15 pages. https://doi.org/10.1145/3411764.3445249

[86] Jacqueline Whalley and Nadia Kasto. 2014. A Qualitative Think-Aloud Study of Novice Programmers' Code Writing Strategies. In *Proceedings of the 2014 Conference on Innovation & Technology in Computer Science Education (ITiCSE '14).* Association for Computing Machinery, 279–284. https://doi.org/10.1145/2591708.2591762

[87] Brian Whitworth. 2005. Polite Computing. *Behaviour & Information Technology* 24, 5 (Sept. 2005), 353–363. https://doi.org/10.1080/01449290512331333700

[88] Mark Wilson-Thomas. 2019. Refactoring made easy with IntelliCode! https://devblogs.microsoft.com/visualstudio/refactoring-made-easy-with-intellicode/. Accessed: 2022-04-04.

[89] Rayoung Yang, Eunice Shin, Mark W. Newman, and Mark S. Ackerman. 2015. When Fitness Trackers Don't 'Fit': End-User Difficulties in the Assessment of Personal Tracking Device Accuracy. In *Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing (UbiComp '15).* Association for Computing Machinery, 623–634. https://doi.org/10.1145/2750858.2804269

[90] Tianyi Zhang, London Lowmanstone, Xinyu Wang, and Elena L. Glassman. 2020. *Interactive Program Synthesis by Augmented Examples.* Association for Computing Machinery, 627–648. https://doi.org/10.1145/3379337.3415900