Interpretable Program Synthesis

Tianyi Zhang
Harvard University
Cambridge, MA, USA
tianyi@harvard.seas.edu

Zhiyang Chen
University of Michigan
Ann Arbor, MI, USA
zhiychen@umich.edu

Yuanli Zhu
University of Michigan
Ann Arbor, MI, USA
leozhu@umich.edu

Priyan Vaithilingam
Harvard University
Cambridge, MA, USA
pvaithilingam@g.harvard.edu

Xinyu Wang
University of Michigan
Ann Arbor, MI, USA
xwangsd@umich.edu

Elena L. Glassman
Harvard University
Cambridge, MA, USA
glassman@seas.harvard.edu

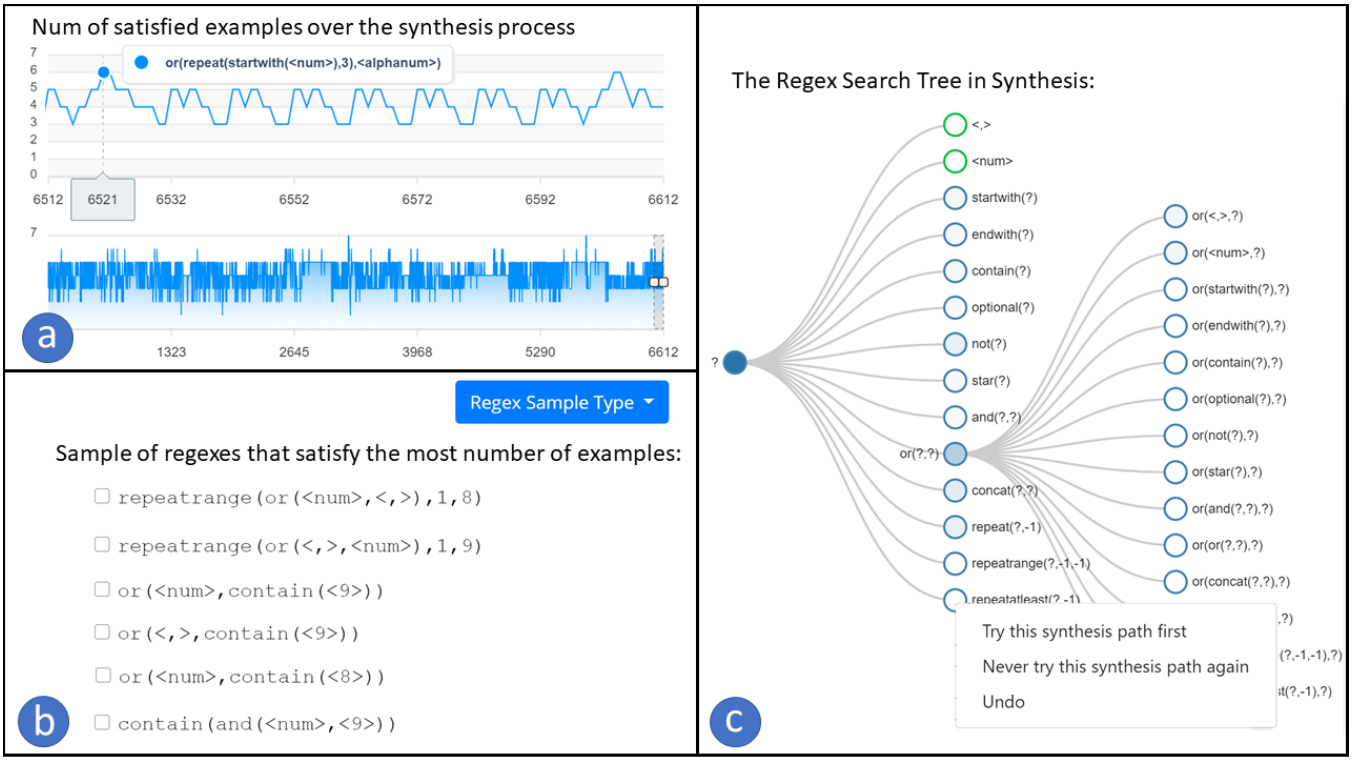


Figure 1: Three representations of the explored program space during a synthesis iteration with different levels of fidelity

ABSTRACT

Program synthesis, which generates programs based on user-provided specifications, can be obscure and brittle: users have few ways to understand and recover from synthesis failures. We propose interpretable program synthesis, a novel approach that unveils the synthesis process and enables users to monitor and guide a synthesizer. We designed three representations that explain the underlying synthesis process with different levels of fidelity. We implemented an interpretable synthesizer for regular expressions and conducted a within-subjects study with eighteen participants on three challenging regex tasks. With interpretable synthesis, participants were able to reason about synthesis failures and provide strategic feedback, achieving a significantly higher success rate compared with a state-of-the-art synthesizer. In particular, participants with a high engagement tendency (as measured by NCS-6) preferred a deductive representation that shows the synthesis process in a search tree, while participants with a relatively low engagement tendency preferred an inductive representation that renders representative samples of programs enumerated during synthesis.

CHI '21, May 8–13, 2021, Yokohama, Japan

ACM ISBN 978-1-4503-8096-6/21/05…$15.00
https://doi.org/10.1145/3411764.3445646

CCS CONCEPTS

• **Human-centered computing → Human computer interaction (HCI)**; **Interactive systems and tools**.

KEYWORDS

Program synthesis; Programming by example; Interpretability

ACM Reference Format:
Tianyi Zhang, Zhiyang Chen, Yuanli Zhu, Priyan Vaithilingam, Xinyu Wang, and Elena L. Glassman. 2021. Interpretable Program Synthesis. In *CHI Conference on Human Factors in Computing Systems (CHI '21), May 8–13, 2021, Yokohama, Japan.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3411764.3445646

1 INTRODUCTION

Program synthesis aims to automatically generate programs that match the user's intent as expressed in high-level specifications, e.g., input-output examples, demonstrations. It has the promise of significantly reducing programming effort and dismantling programming barriers for novices and computer end-users. So far, program synthesis has been investigated in many domains, including data extraction and filtering [38, 72], visualization [70], string transformations [24], and databases [19]. Despite the great progress in recent years such as FlashFill [24], a lot of work still needs to be done to adopt program synthesis in real-world applications.

A major barrier to the adoption of program synthesis is the brittleness and opaqueness of the synthesis process [27, 36, 39, 73]. It is well known that program synthesis may generate plausible programs that are completely wrong due to ambiguity in user specifications. More severely, program synthesis may even get stuck and fail to generate anything if a user provides a specification with conflicts or a specification that is too complex. Unfortunately, most synthesizers follow a *black-box* design—they provide no means for users to reason about synthesis failures, leaving users to guess about productive ways to change the specification they provided or just start over. Though this *black-box* design avoids overloading users with the underlying synthesis details, in practice, it can cause unfavorable outcomes: users' patience and faith in program synthesis are quickly exhausted by recurring synthesis failures and a lack of means to reason and recover from them.

In contrast to traditional *black-box* program synthesis, we propose *interpretable* program synthesis. We hypothesize that, by unveiling the synthesis process, users can build more accurate mental models about how and why a synthesizer fails and thus provide strategic feedback to help it overcome synthesis failures. A key challenge of interpretable program synthesis is that existing synthesis systems are complex. These systems are often equipped with sophisticated search methods and inductive biases in various forms, e.g., hand-crafted heuristics [24, 25], distance-based objective functions [16], and prior distributions [17, 60]. Figuring out how exactly these synthesizers solve a task may even perplex the synthesis designers themselves. However, program synthesis is essentially a search process over a hypothetical program space defined by a domain-specific language (DSL). Thus, instead of explaining to users how exactly a synthesizer works, i.e., its search algorithm and inductive biases, we choose to explain what programs have been tried during synthesis, i.e., the explored program space.

The program space defined by a DSL is enormous; even a small region of it may contain thousands of programs. For instance, our regular expression synthesizer can explore more than 20K programs within 20 seconds. To visualize such a large program space, we designed three representations with different levels of fidelity and potential cognitive demand on users. First, *a live-updated line chart* renders how many programs have been tried by the synthesizer over time and how many user-provided examples each of them satisfies. Since the programs are represented as points in a line chart, this representation requires low intellectual engagement. It communicates the number of explored programs, the speed at which new programs are being tried, and any trends in how many of the specification components, e.g., user-provided input-output examples, are being satisfied. Second, *program samples* that vary both syntactically and semantically are drawn from the explored program space and shown to users. Compared with the line chart, program samples are a more concrete representation of the explored program space, which is sound, but not complete [35]. Since it requires users to read programs, it is more cognitively demanding to engage with. Third, *a search tree* organizes and visualizes all explored programs based on how they are derived from the DSL grammar, i.e., their derivation paths. It provides a holistic view of the synthesis process while still showing concrete programs and partial programs in the tree. It also requires high intellectual engagement as users need to navigate the tree structure and reason about programs in various states of completeness.

Compared with black-box synthesis, exposing the inner workings of a synthesizer also enables users to provide richer feedback to guide the synthesis. In case of synthesis failures, users can inspect partially correct programs explored during synthesis and indicate which regex operator should never be tried or which subexpression looks promising. Furthermore, in the search tree, users now can directly prune the search space by prioritizing or eliminating a search path, so the synthesizer will not waste time on unproductive search paths in the next iteration. We implemented these three representations and adapted an existing interactive synthesizer to support interpretable synthesis [73].

Information comes at the price of attention. Though unveiling the synthesis process would help users gain more insights, the abundance of information presented in interpretable synthesis also requires more attention to process. This may in turn disincentivize users from using interpretable synthesis and affect their performance in programming tasks. We conducted a within-subjects study with eighteen students to investigate how users may respond to interpretable synthesis. When solving challenging programming tasks, participants using interpretable synthesis achieved a statistically significant higher success rate compared with using traditional synthesis. When using interpretable synthesis, participants with either higher engagement tendency (as measured by NCS-6 [44]) or less expertise expressed a stronger preference towards the highest-fidelity representation, i.e, the search tree. Yet participants' engagement tendency had little impact on the task completion rate.

This paper makes the following contributions:

- A novel interpretable program synthesis approach that unveils the underlying synthesis process and visualizes the explored program space with different levels of fidelity

- An implementation of an interpretable synthesizer for regular expressions
- A within-subjects study that shows the usefulness and usability of interpretable synthesis
- An in-depth analysis of how users with different levels of expertise and engagement tendency, as measured by NCS-6 [44], responded to interpretable synthesis

2 RELATED WORK

Program synthesis is not a new idea. Back in the 1960s, Waldinger and Lee presented a program synthesizer called PROW that automatically generated LISP programs based on user-provided specifications in the form of a predicate calculus [68]. PROW, and many synthesis techniques that followed [23, 45], require users to provide a complete, formal specification, which is shown to be as complicated as writing the program itself [27]. Later, programming-by-example (PBE) or programming-by-demonstration (PBD) systems were introduced to allow users to demonstrate desired program behavior in the form of input-output examples or a sequence of actions [13, 49, 50, 53, 54], as summarized in [14] and [43]. Compared with formal specifications, examples or demonstrations are much easier for users to provide, significantly reducing the barrier of making using of synthesis in practice. Since then, progress has been made to push program synthesis closer to real-world applications [5, 10, 21, 24, 32, 37, 41, 42, 58, 71]. For example, FlashFill [24], a PBE system for string transformation, has been seamlessly integrated into Microsoft Excel and used by millions of Excel users. From a few examples given by a user, FlashFill can learn a string transformation procedure such as separating first and last names from a single column, and then process the rest of the data in a spreadsheet with the learned transformation procedure.

Despite great progress in recent years, user studies on program synthesis systems have revealed several major usability issues and challenges [26, 36, 39, 52, 73]. Myers and McDaniel point out that one major obstacle of using PBE/PBD systems was *the lack of confidence and trust in synthesized programs*, since users were not able to see or understand the synthesized programs [52]. In a large-scale online study, Lee et al. observed two common usability issues of a PBE system—*ambiguous user examples* and *synthesis failures* [39]. In the former case, participants often failed to provide sufficient examples that precisely described how a desired program should behave in different scenarios, leading the synthesizer to generate plausible programs that only satisfied the given examples but did not generalize to unseen data as the user intended. In the latter case, the PBE system failed to generate any programs when participants did not provide crucial values in their examples or when they did not properly decompose a complex task to simpler sub-tasks. Zhang et al. made a similar observation—"*even without any user mistakes, the synthesizer may still fail to return anything to the user, if user examples are too complicated to generalize or if a task is too hard to solve within a given time budget*" [73].

Many interactive approaches have been proposed to address *lack of confidence on synthesized programs* and *user intent ambiguity*. However, all of them make one fundamental assumption—the synthesizer must return at least one program that satisfies all user-provided examples (*satisfiable programs* henceforth), so users can inspect and provide feedback on it. These approaches do not change the *all-or-nothing* nature of traditional synthesis. When synthesis fails, users have nothing to inspect or provide feedback on. We summarize these approaches below.

Techniques to communicate synthesized programs to users. Two major approaches have been investigated to present programs in a readable format to target users: translation to a more familiar language and rendering programs graphically. The first approach includes SmallStar [28], CoScripter [41], and FlashProg [46], which translate arcane instructions in a synthesized program to English-like text descriptions to help computer end-users understand the program semantics. Wrex [15] translates programs written in a DSL to a programming language, Python, that its target users, data scientists, are more familiar with. The second approach includes Topaz [51], Pursuit [48], and Rousillon [10], which show their synthesized programs in a graphical representation. Specifically, Pursuit shows the before and after states of a synthesized program in a comic-book style, while Rousillon shows synthesized programs in a block-based visual format like Scratch [61].

End-user debugging techniques [3, 33, 47, 62, 67] allow users to interactively understand and reason about the functionality of a synthesized program and discover erroneous behaviors, i.e., bugs. End-user debugging still requires a satisfiable program to be generated first and cannot be used to debug a synthesizer or a synthesis process. Unlike interpretable synthesis, end-user debugging helps users investigate programs created on their behalf, instead of helping users create the program they want.

Techniques to resolve ambiguity in user-provided examples. Most PBE and PBD systems allow users to actively provide additional examples to disambiguate their intent, which is cognitively demanding. Several approaches automate this process by generating input examples that distinguish multiple plausible programs [32, 46, 69]. Peridot [49] used question-and-answer dialogues to confirm the intent of users during inference. Yet this interaction design turned out not to be effective, since users tended to simply answer yes to every question. In a new interaction model by Peleg et al. [58], users can directly specify which parts of a synthesized program must be included or excluded in the next synthesis iteration. Following [58], Zhang et al. proposed an alternative approach that allows users to directly annotate their input examples instead of synthesized programs to disambiguate user intent [73].

Techniques to handle synthesis failures. So far, little work has been done to handle *synthesis failures*. The most related systems to our approach are Chinle [11] and Bester [57]. Chinle [11] is a PBD system that supports partial learning—if a problem is too hard to solve in a single shot, it learns a program with holes in it and present it users. This idea of partial learning is similar to partial programs shown in our search tree. The difference is that in our approach, partial programs are used to organize concrete programs to visualize an explored search space. Bester [57] relaxes the synthesis objective of identifying a program that satisfies all user-provided examples to identify a program that satisfies the most number of examples, called *best-effort* programs. Unlike Bester, we propose a tree representation that visualizes the explored program space more completely and faithfully than just rendering best-effort program samples only.

To our best knowledge, explaining the synthesis process to users has not been investigated in the literature. The key of our approach is to trace and visualize the massive amount of programs enumerated during synthesis, so users can directly see the explored program space and understand what has and has not been tried by the synthesizer. This is related to program execution tracing and program visualization, both of which have been well studied in domains such as time-travel debugging [33, 59], visual debugging [30], and code example visualization [22]. However, none of these techniques have been applied to interpret the synthesis process. A main challenge is that the number of programs enumerated during synthesis, even within a short period of time, can be enormous. For instance, our regex synthesizer can explore more than 20K programs in 20 seconds. FlashProg [46] and Scout [65] have explored how to visualize many alternative programs in a program space, but not at the scale of thousands of programs. Furthermore, both of them still require a synthesizer to generate programs that satisfy a user specification in the first place. Therefore, they are not designed to handle synthesis failures nor trace partially correct programs that have been generated and tested during synthesis.

3 USAGE SCENARIO

This section illustrates interpretable synthesis based on a realistic task from Stack Overflow [1]. Suppose Alex is a data scientist and she wants to extract strings with comma-separated numbers from a large text file. The text data is noisy, with some invalid values such as ",12", "12,", and "ab13,14". She needs to write a regular expression to only match valid comma-separated numbers such as "1,23,5". Alex has only used regexes a couple of times before, so she decides to use a regex synthesizer instead.

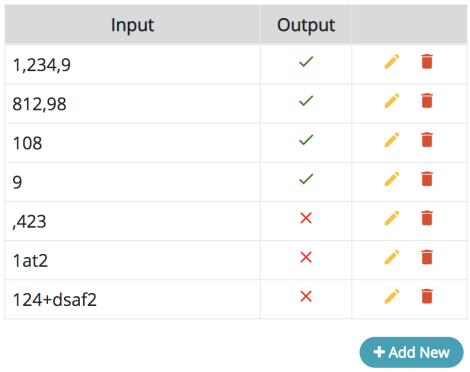


Figure 2: Alex enters some positive and negative examples

Alex enters several input examples that should be accepted or rejected by the desired regex, as shown in Figure 2. Within a blink, the synthesizer finds five regexes (Figure 3a) that satisfy all her examples. However, Alex quickly realizes that none of the synthesized solutions are correct. Since all the positive examples she gave contain the two digits, 8 and 9, the synthesized solutions all try to match one of these two digits, overfitting her examples.

Alex adds several counterexamples, i.e., positive examples that do not contain 8 or 9, and starts synthesis again. This time, the synthesizer seems puzzled by her counterexamples. Alex stares at a live-updated line chart (Figure 1a), which shows the number of regex candidates the synthesizer has tried so far and the number of examples each regex candidate satisfies. The overall trend has been oscillating up and down for a while. After exploring over 6612 regex candidates in 20 seconds, the synthesizer prompts Alex that it has not found any regexes that satisfy all of her examples and asks whether she wants it to continue to synthesize. While being impressed that the synthesizer has tried so many candidates in the back-end, Alex suspects the synthesizer may have wasted a lot of time in some unproductive directions. Therefore, instead of giving more counterexamples, she decides to find out what kinds of programs the synthesizer has tried. Alex clicks on the *sample* view (Figure 1b). The sample view shows some partially correct programs that satisfy the most number of examples in the explored program space. Given that the synthesizer did not return any regexes satisfying all the examples this time, Alex finds it helpful to at least see some concrete regexes that the synthesizer has tried. When hovering her mouse on each regex, her examples are highlighted in red or green to indicate which examples are passed or failed respectively (Figure 5b). Alex notices that some sampled regexes can only match numbers in her examples but not commas, while some can match both numbers and commas but cannot reject some corner cases she included in her examples. Though not ideal, these samples give Alex some hints, e.g., the regex must enforce that commas appear after some numbers but not at the end of the string.

These samples only give Alex a fragmented view of what kinds of programs have been tried by the synthesizer. Though helpful, Alex is eager to know more about the explored program space and figure out why the synthesizer got stuck. So she clicks on the *Search Tree* tab (Figure 1c). This tree view shows all the paths the synthesizer has explored. Each tree node is labeled with a partial regex with some question marks in it. The question mark represents a program hole to be filled in the following search steps. For example, `or(?,?)` represents regexes that starts with the operator `or`. By expanding the node of `or(?,?)`, Alex sees some regexes that are derived from `or(?,?)`, such as `or(<,>,?)`, `or(<num>,?)`, and `or(startwith(?),?)`. In addition, the background color of each tree node indicates how many regex candidates have been tried along each search path. Alex notices the synthesizer has mainly explored in the paths of `or(?,?)` and `not(?)`, which seems irrelevant to her task. To guide the synthesizer towards her intuition, she marks the two search paths, `or(?,?)` and `not(?)`, to be excluded from the next synthesis iteration. She also prioritizes the search path of `concat(?,?)`, since `concat` can be used to enforce the ordering between two sub-strings. In this way, Alex prunes two unproductive search paths in which the synthesizer spent a lot of time in the previous synthesis iteration and also informs the synthesizer to focus on a search path that looks promising to her.

With Alex's guidance, the synthesizer identifies 4 regexes that satisfy all the examples (Figure 3b). She notices that these regexes can match numbers but fail to recognize the second part of the pattern (i.e, a comma followed by numbers). Instead, it checks for certain characters present in a string input. To fix this, Alex uses the search tree to exclude the search path of `concat(?,not(?))`. Alex also prioritizes `concat(repeatatleast(<num>,1),?)`, since the first part of this pattern matches with one or more numbers,

and the second part should be something that repeatedly matches comma followed by numbers.

Given these additional search path annotations, the synthesizer returns 5 new regexes (Figure 3c). The first 4 regexes check for a `not` condition after the comma, which is incorrect. Alex confirms this hypothesis by clicking the "Show Me Familiar Examples" button and observing how these regexes accept or reject some addition examples, as presented in our previous work [73]. However, the last regex, `concat(repeatatleast(<num>,1),star(<,>,repeatatleast(<num>,1)))` looks promising. Alex clicks the "Show Me Corner Cases" button and verifies this solution on some corner cases. Finally, she decides this is the right solution.

- ☐ `endwith(or(<8>,<9>))`
- ☐ `endwith(or(<9>,<8>))`
- ☐ `contain(or(<0>,<9>))`
- ☐ `contain(or(<8>,<9>))`
- ☐ `contain(or(<9>,<0>))`

(a) Synthesis results in the first iteration

- ☐ `concat(repeatatleast(<num>,1),not(endwith(<2>)))`
- ☐ `concat(repeatatleast(<num>,1),not(contain(<low>)))`
- ☐ `concat(repeatatleast(<num>,1),not(contain(<let>)))`
- ☐ `concat(repeatatleast(<num>,1),not(contain(<a>)))`

(b) Synthesis results after prioritizing the search path of `concat(?,?)` and excluding `or(?,?)` and `not(?)`

- ☐ `concat(repeatatleast(<num>,1),star(concat(<,>,not(<a>))))`
- ☐ `concat(repeatatleast(<num>,1),star(concat(<,>,not(<d>))))`
- ☐ `concat(repeatatleast(<num>,1),star(concat(<,>,not(<f>))))`
- ☐ `concat(repeatatleast(<num>,1),star(concat(<,>,not(<+>))))`
- ☐ `concat(repeatatleast(<num>,1),star(concat(<,>, repeatatleast(<num>, 1))))`

(c) Synthesis results after prioritizing the search path of `concat(repeatatleast(<num>,1),?)` and excluding `concat(?,not(?))`

Figure 3: Synthesis results over iterations

4 APPROACH

This section describes how to adapt a black-box synthesizer to support interpretable synthesis. We use an interactive synthesizer for regular expressions as an example and elaborate on our implementation details. In this work, we choose the domain of regular expressions (regexes) since regexes are widely used and also known to be hard to understand and construct, even for experienced programmers [8, 9, 64]. Furthermore, regexes are known to be challenging to synthesize [56]. Hence, users may run into synthesis failures more frequently compared with other domains, providing a fertile ground for experimenting with interpretable synthesis.

4.1 A Standard Regular Expression Synthesizer

We adopt a standard regex synthesizer from existing work [12, 73]. It generates regular expressions in a predefined domain-specific language (DSL), as shown in Figure 4. This DSL includes *character classes* as basic building blocks. For instance, `<num>` is a character class that matches any digits from `0` to `9`. Similarly, `<let>` is a character class that matches any English letters. We also have `<low>`, `<cap>`, `<any>` that match lower-case letters, upper-case letters, and any characters, respectively. In addition to these general character classes, this DSL also includes specific character classes that match only one character, e.g., `<a>` only matches letter `a`. Using these character classes, the DSL allows us to create more complex regular expressions. For example, `contain(<let>)` recognizes any strings that contain an English letter, and `star(<num>)` matches a sequence of digits of arbitrary length. This DSL provides high-level abstractions that are essentially wrappers of standard regex. This makes DSL programs more amenable to program synthesis as well as readable to users. Note that this DSL has the same expressiveness power compared to standard regular languages.

$$
\begin{aligned}
e :=\ & \texttt{<num>} \mid \texttt{<let>} \mid \texttt{<low>} \mid \texttt{<cap>} \mid \texttt{<any>} \mid \ldots \mid \texttt{<a>} \mid \texttt{<b>} \mid \ldots \\
& \mid \texttt{startwith}(e) \mid \texttt{endwith}(e) \mid \texttt{contain}(e) \mid \texttt{concat}(e_1, e_2) \\
& \mid \texttt{not}(e) \mid \texttt{or}(e_1, e_2) \mid \texttt{and}(e_1, e_2) \\
& \mid \texttt{optional}(e) \mid \texttt{star}(e) \\
& \mid \texttt{repeat}(e, k) \mid \texttt{repeatatleast}(e, k) \mid \texttt{repeatrange}(e, k_1, k_2)
\end{aligned}
$$

Figure 4: The DSL for regular expressions.

Given this DSL, our synthesizer generates regexes in this DSL that match the user-provided examples. The synthesis algorithm is shown in Algorithm 1. At a high-level, it performs enumerative search over the space of regexes defined by the DSL grammar and returns the first regex that matches all examples. The algorithm maintains a worklist of *symbolic regexes*. Here, a symbolic regex is a partial regular expression that has at least one symbol that is not resolved to a DSL operator or a character class. The worklist is initialized with a symbolic regex with only one symbol, denoting an empty program (line 1). During each iteration, the algorithm removes a symbolic regex p from the worklist (line 3). It checks whether p is fully resolved, i.e., containing no more symbols, and whether p is consistent with all examples E (line 4). If so, it considers p as a satisfying program and returns it to the user. Otherwise, the algorithm chooses an unresolved symbol s in p (line 7) and resolves s to either a DSL operator or a character class (line 8). It then adds all resulting symbolic regexes into the worklist (lines 9-12). Note that the algorithm performs pruning at line 10 whenever we determine that a symbolic regex cannot lead to a regex that is consistent with E. We skip the detail of how the `resolve` function at line 8 and the pruning at line 10 work but refer interested readers to [12].

4.2 Interpretable Regular Expression Synthesis

The standard regex synthesizer in Section 4.1 follows a *black-box* design. That is, it does not communicate with users until a satisfying program is found. If no satisfying program is found within a time bound, the synthesizer will inform users that it failed. Since no program is returned, users cannot do much other than allocating

Algorithm 1: Enumeration-based synthesis algorithm

```
Input  : a set of string examples E
Output : a regular expression that matches E
worklist := {e};
while worklist is not empty do
    p := worklist.dequeue();
    if p is fully resolved then
        if p matches E then return p;
    else
        s := chooseSymbol (p);
        worklist' := resolve (p, s);
        for p' in worklist' do
            if p' is infeasible then continue;
            else worklist.add(p');
        end
    end
end
```

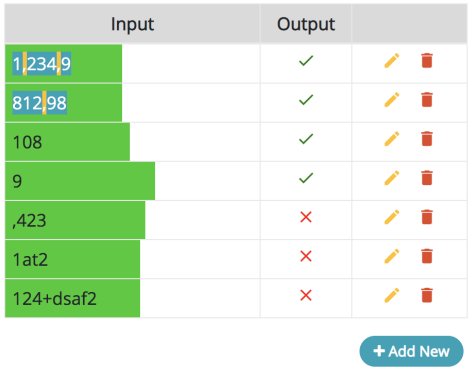
more synthesis time or changing their own examples by guesswork. In interpretable synthesis, we introduce three representations that explain the synthesis process with different levels of fidelity, in order to help users understand the explored search space and diagnose where the synthesizer is stuck.

A Live-Updated Line Chart. During synthesis, the line chart (Figure 1a) gives users a live update about the synthesis progress. The x-axis of the line chart shows the number of program candidates that have been tried by the synthesizer so far. The y-axis shows the number of examples each program satisfies. Users can inspect the concrete program at each point by hovering the mouse over it. This view allows users to monitor the synthesis progress over time and make quick decisions about whether to interrupt the synthesis process. For example, if the line chart suddenly plunges or has been zig-zaging for a while, it may hint to the user that the search could use additional guidance to help it make progress.

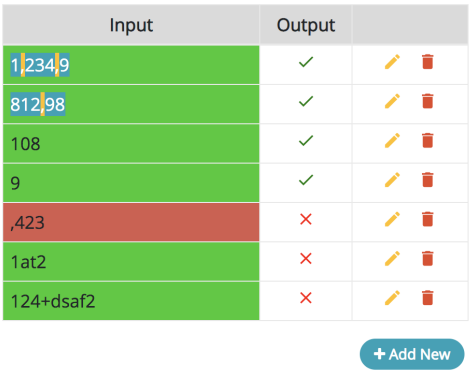
As individual programs are each rendered as a point on the line chart whose composition is only revealed by hovering, the line chart is most suited for showing any overall synthesis trends in satisfying the user's specification, but does not give a holistic view of the space of programs tried so far. It is simple to comprehend, requiring little intellectual engagement.

A Sample of Representative Programs. As it is too overwhelming to unroll all candidate programs during synthesis, the sample view shows a subset of representative ones instead. Our interface draws three kinds of programs from the explored program space—a *syntactically diverse sample*, a *semantically diverse sample*, and a *best-effort sample*. The syntactically diverse sample includes a set of programs that cover all regex operators a synthesizer has tried. The semantically diverse sample includes a set of programs, each of which satisfies a subset of user-provided examples but together satisfy all examples. The best-effort sample includes a set of programs that satisfy the most number of user-provided examples.

Compared with the all-or-nothing design in traditional synthesis, the sample view shows users some partially correct program candidates in case of synthesis failures. To help users better troubleshoot, our interface highlights which input examples a sampled program passes (green color) and fails (red color), as shown in Figure 5b, as



(a) How often each of my examples is satisfied?



(b) Which example does this program candidate pass or fail on?

Figure 5: A user troubleshoots his own examples and some program tried by the synthesizer.

they hover the mouse over it. By default, the interface shows how many program candidates in the explored program space satisfy each of the user-provided examples in a histogram-like view (Figure 5a). By showing the distribution of how frequently an example is satisfied, users can make a better guess about which example may be causing the synthesizer to get stuck and then make more well-informed changes to their examples. For example, if an example is rarely satisfied, it indicates that the example may be too complex to be handled by the synthesizer or have some conflict with other examples. Therefore, the user may want to modify it first. Compared with the line chart, these samples form a more concrete view of the explored program space and thus has a higher fidelity. But it also requires higher intellectual engagement, since users need to inspect individual samples to understand them.

A Search Tree. The search tree (Figure 1c) provides a faithful representation of the program space that has been explored so far during the synthesis process. The search tree organizes program candidates based on how they are derived from the regex DSL (Figure 4). Internal nodes are labeled with programs that are partially derived, i.e., *partial programs*, while leaf nodes are labeled with programs that are fully derived, i.e., *concrete programs*. Partial programs have one or more question marks (?), which denote placeholders that need to be resolved during synthesis. The root

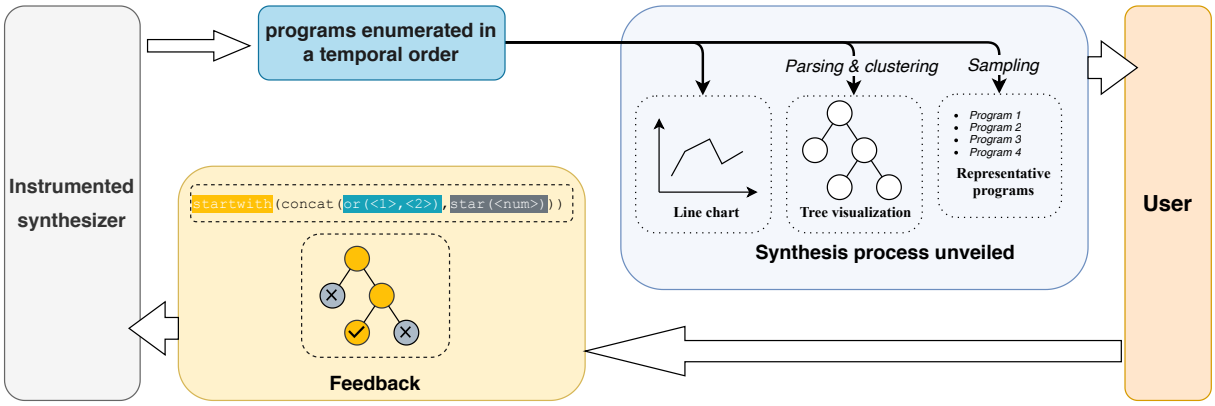


Figure 6: The overview of interpretable synthesis

of the search tree is always set as ?, indicating an empty program. The children of this root node are partial programs derived from ?, e.g., `startwith(?)`, `endwith(?)`, `or(?,?)`, etc. Users can continue to expand an internal node such as `or(?, ?)` to find out what programs are further derived from it, e.g., `or(concat(?,?),?)`.

Essentially, each internal tree node represents a sub-region of the explored program space in a hierarchical way. For example, `or(?,?)` represents all explored programs that starts with `or`. As the number of explored programs increases exponentially over depth, we limit the tree view to render at most 20 nodes in each hierarchical level to avoid overwhelming users. The background opacity of each tree node indicates how many concrete programs have been derived from the corresponding partial program of the node. Hence, users can immediately tell which search path the synthesizer spends most time on. If a user believes that the synthesizer is searching in a wrong direction, she can right click the corresponding tree node and select the "Never try this synthesis path again" option (Figure 1c). On the other hand, if she finds a search path promising, she can right click and select "Try this synthesis path first" to prioritize it for the next synthesis iteration. This allows users to prune the search space and directly pinpoint search directions for the synthesizer based on their own hypotheses. Interacting with this view has the highest cognitive load, since it not only requires users to understand abstract concepts such as a search tree and a program space, but also requires users to navigate the tree and assess partial programs.

4.3 Open the Box: From Black-box Synthesis to Interpretable Synthesis

The previous two sections describe how a standard black-box regex synthesizer and our interpretable synthesizer work respectively. Now, let us describe how to adapt a black-box synthesizer to make it interpretable. Figure 6 shows the pipeline.

Logging Program Candidates Enumerated During Synthesis. To open the box, a key step of our approach is to instrument the black-box synthesizer to log program candidates that have been considered by the synthesizer. In particular, we instrument line 5 of Algorithm 1 to record each concrete, fully resolved regex during each synthesis iteration and count the number of examples that it satisfies. Below we describe how we parse the logged programs to generate the three representations that are described in Section 4.2.

Rendering Logged Programs in a Line Chart. The logged programs are parsed into a sequence of data points (x, y), where x means the x-th program in the log file and y is the number of user-provided examples the x-th program satisfies. Then, the sequence of dots is plotted in the line chart. Since the log file is constantly updated with new programs enumerated by a synthesizer, the line chart is updated accordingly every one second to reflect the synthesis progress.

Drawing Samples from Logged Programs. We have implemented three sampling methods for users to choose from. Though sampling more programs can more completely represent the explored program space, it also induces more cognitive demand. For the syntactically diverse sample, our objective is to sample a minimal set of logged programs that cover all regex operators and constants tried by the synthesizer. Similarly, for the semantically diverse sample, our objective is to sample a minimal set of logged programs that satisfy all user-provided examples together, though each of them may only satisfy a subset of those examples in case of synthesis failures. We formulate the sampling problem into a set cover problem and solve it with an off-the-shelf set-cover solver [2]. This solver employs an iterative heuristic approximation method, combining the greedy and Lagrangian relaxation algorithms. Best-effort programs can be easily identified by counting and comparing the number of satisfied examples of each logged program.

Parsing and Clustering Logged Programs to a Search Tree. Figure 7 illustrates how to generate a search tree from logged programs. We first parse each logged program into a parse tree. Then, we infer the derivation path of each parse tree by performing a pre-order traversal. For instance, consider a regex `concat(<A>,<B>)`. The derivation path for this program is `?` → `concat(?,?)` → `concat(<A>,?)` → `concat(<A>,<B>)`, where `?` is a placeholder for further derivation. Once we obtain the derivation path for each program, we cluster all of them into a single search tree by merging identical nodes. For example, `concat(<A>,<C>)` and `concat(<A>,<B>)` have three identical nodes, `?`, `concat(?,?)`, `concat(<A>,?)`), in their derivation paths, which will be merged into single ones in the search tree.

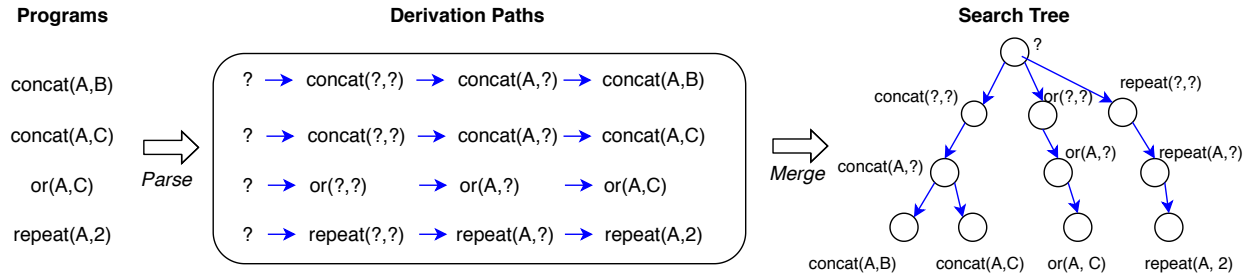


Figure 7: The pipeline of parsing explored program candidates and registering them into the search tree

4.4 Supporting New User Feedback in Interpretable Synthesis

Since our interpretable synthesizer is adapted from an existing interactive synthesizer [73], the user feedback elicitation features in the original synthesizer—*semantic augmentation*, *data augmentation*, and *syntactic annotation* (originally proposed by [58])—have also been inherited by our interpretable synthesizer. Here we briefly introduce these interactive features. Please refer to [73] and [58] for technical details. *Semantic augmentation* enables users to annotate how input examples should or should not be generalized to other similar contexts, which is designed to resolve intent ambiguity in user-provided examples. *Data augmentation* automatically generates additional input examples and corner cases for a selected program. It is designed to help computer end-users and novice programmers quickly understand and validate synthesized programs without the need of inspecting and fully understanding their syntax. *Syntactic annotation* enables users to mark parts of a synthesized program to be included or excluded in the next synthesis iteration.

Rendering the explored search space to users also brings in opportunities to solicit new kinds of user feedback to guide the synthesizer. Given an explored program space in the form of a search tree, users can analyze the search paths taken by the synthesizer and mark some search paths, i.e., partial programs represented by tree nodes, as prioritized or prohibited. If one or more search paths are prioritized, their corresponding partial programs will be used to initialize the worklist (line 1 in Algorithm 1). For each search path, the synthesizer will search a constant depth (3 by default). If no satisfiable results are found within the given depth, the synthesizer will fall back and explore other unprohibited paths. This fall-back mechanism is designed for cases where users develop wrong hypotheses and guide the synthesizer towards a wrong direction, which is very common in challenging programming tasks. On the other hand, if a search path is prohibited, the synthesizer will add it into an exclude set. We add a filter at line 10 in Algorithm 1 to compare symbolic regexes in the worklist with those in the exclude set. The synthesizer will stop exploring a search path if any symbolic regex in the worklist is matched with or derived from a partial program in the exclude set. In this way, the synthesizer ensures any search paths marked as prohibited will be pruned from the search space.

5 USER STUDY

We conducted a within-subjects study with eighteen participants to evaluate the usefulness and usability of interpretable synthesis. We selected a state-of-the-art interactive synthesizer for regular expressions [73] as the comparison baseline. Like traditional synthesizers, this baseline synthesizer does not expose any internals of the synthesis process, other than showing the elapsed time in a progress bar. We investigated the following research questions:

- RQ1. Compared with traditional synthesis, to what extent does interpretable synthesis augment users' problem solving capability?
- RQ2. How do users with various expertise and levels of engagement tendency respond to interpretable synthesis?
- RQ3. How are representations with different fidelity of the synthesis process perceived by users?
- RQ4. What obstacles have users encountered when using interpretable synthesis?

5.1 Participants

We recruited eighteen students (eight female and ten male) through the mailing lists of the School of Engineering and Applied Sciences at Harvard University. Sixteen participants were graduate students and two were undergraduate students. Regarding their familiarity with regular expressions, eleven participants said they knew regex basics but only used it several times, while seven said they were familiar with regexes and had used it many times. As part of the pre-study survey, we measured participants' engagement tendency using the Need for Cognition Scale-6 (NCS-6) questionnaire [44]. The questionnaire contained six statements that describe a person's Need for Cognition characteristics in different situations, such as "*I would prefer complex to simple problems.*" Appendix A lists all six statements. For each statement, participants were asked to rate to what extent the statement is characteristic of them on a 7-point scale from "not at all like me (1)" to "very much like me (7)." The final score for a participant was the average of the participant's ratings across all six statements. The higher the Need for Cognition score, the more a participant tends to enjoy activities that involve intellectual engagement and thinking. Among eighteen participants, six participants have a medium level of engagement tendency (3 to 5 points), while twelve participants have a high engagement tendency (above 5 points). Participants received a $25 Amazon gift card as compensation for their time.

5.2 Tasks

We selected three tasks from the regular expression programming benchmark in [73]. This benchmark was derived from realistic

regex questions asked in Stack Overflow. The descriptions of these three tasks and their solutions in both our DSL and standard regex grammar are listed below. Note that these tasks may have multiple correct solutions. During the study, if participants reached any correct solution to a task, we considered them as having completed the task successfully.

Task 1. Write a regular expression that validates a month with a leading zero, e.g., 01, 02, 10, 12, but not 00, 1, 13. [Post 2878010]

```
or(concat(<0>,<num1-9>),concat(<1>,or(<2>,or(<1>,<0>))))
```

```
^(0[1-9]|1[0-2]$
```

Task 2. Write a regular expression that accepts numeric values that are seven or ten digits. [Post 2908527]

```
or(repeat(<num>,7),repeat(<num>,10))
```

```
^\d{7}|\d{10}$
```

Task 3. Write a regular expression that validates a string of comma-separated numbers, e.g., "1,2,3", "12", but not "1,", ",1", or "12,,3".[Post 5262196]

```
concat(repeatatleast(<num>,1),star(concat(<,>,repeatatleast(
<num>,1))))
```

```
^\d+(,\d+)*$
```

Compared with Task 2, Task 1 and Task 3 are much more challenging. Their solutions involve more regex operators and constants, and also require more computation power to synthesize. Without human guidance, the baseline synthesizer can solve Task 2 in 5 minutes. However, Task 1 and Task 3 cannot be solved even after 100 minutes. According to the supplementary material of [73], Task 1 and Task 3 require a divide-and-conquer strategy to be solved.

5.3 Protocol

We recorded each user study session with the permission of participants. A study session took 85 minutes on average. In each session, a participant completed one of the three tasks using interpretable synthesis (i.e., the experiment condition) and another task using the traditional synthesizer [73] (i.e., the control condition). To mitigate the learning effect, both the order of task assignment and the order of synthesizer assignment were counterbalanced across participants through random assignment. In total, six participants tried each of the three tasks in each condition. Before each task, participants first watched a pre-recorded tutorial video of the synthesizer they would use. Then they were given 20 minutes to finish the assigned task. The task was considered failed if participants did not guide the synthesizer to find a correct regex within 20 minutes.

After each task, participants answered a survey to reflect on the usability of the assigned synthesizer. As part of the survey, they were asked to answer five NASA Task Load Index questions [29] to rate the cognitive load of using the assigned synthesizer to complete the task. After finishing both tasks, participants answered a final survey to directly compare the two synthesizers. The first author performed open-coding on participants' responses to identify themes and then discussed with co-authors to refine the themes in multiple sessions. These themes were then used to explain the quantitative results such as why participants performed better with interpretable synthesis in the following section.

6 USER STUDY RESULTS

6.1 User Performance Improvement

When using interpretable synthesis, 11 of 18 participants successfully guided the synthesizer to a correct regex solution, while only 5 of 18 participants finished the task when using traditional synthesis. Fisher's exact test shows that this performance difference is not statistically significant (p=0.09222). Further investigation shows that it is because in both conditions, participants did very well in the easy task (Task 2)—all six participants who were assigned Task 2 using interpretable synthesis finished it, and five of six participants assigned Task 2 using traditional synthesis finished it. This is not surprising because the baseline synthesizer has already been proven quite efficient to solve average programming tasks [73].

For the two challenging tasks (Task 1 and Task 3), none of the participants finished the two most challenging tasks using traditional synthesis. By contrast, using interpretable synthesis, two out of six participants finished Task 1 (16:42 min on average) and three of six participants finished Task 3 (17:25 min on average). Fisher's exact test shows that the performance difference in these two challenging tasks is statistically significant (p=0.03727).

There are four main reasons that participants performed much better with interpretable synthesis in challenging programming tasks. First, 10 participants self-reported that they gained more insights of the synthesis process, indicating that they built a more detailed mental model. P3 wrote in the post survey, "*it [the interpretable synthesizer] was more helpful because it was more obvious to directly see which search paths the algorithm went down, and to then edit search paths the algorithm should not have gone down.*" Second, the interface affords participants using their more accurate mental model to strategically add more examples or prune unproductive search paths during synthesis. P5 wrote, "*the sample view and tree view can give me useful information, such as the search order and explored regex or partial regex. This information can easily help me refine my inputs (e.g. examples, annotations) to guide the synthesizer and/or help me find the correct solution by myself.*" Third, in challenging tasks, the synthesizer often failed to find any satisfying programs within a given time budget. In such cases, interpretable synthesis provided more means of guiding the synthesizer, e.g., annotating representative regexes that are partially correct, prioritizing or prohibiting some search paths, etc. By contrast, participants with traditional synthesis had very few options other than (1) increasing the time budget and (2) adding, deleting, or modifying input-output examples by guesswork. P7 complained, "*the lack of information about why a search was failing was frustrating. It felt like I had to* go back to go forward *by eliminating examples to recover regex candidates.*" Fourth, in the post-study survey, three participants reported they got inspiration from some partially correct regexes and intermediate results in the search tree. When facing a new language, i.e., the regex DSL in our tool, seeing the programs tried by the synthesizer helped them learn the language and gave them hints about possible correct solutions. P3 said, "*when I look at the search tree, it can give me hints of what sub-expressions should be included in the final regex and may even let me know the correct regex before the synthesizer actually finds the solution.*"

Despite the advantages brought by interpretable synthesis, seven participants still failed to complete assigned tasks within 20 minutes.

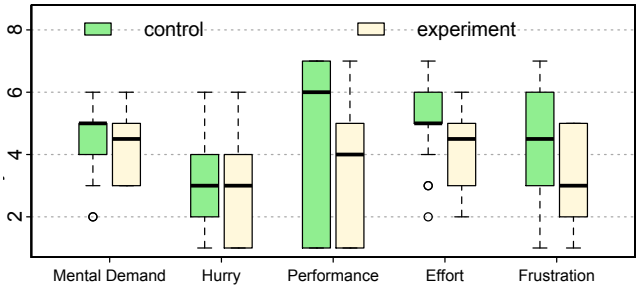


Figure 8: Cognitive load measured by NASA TLX

We manually analyzed the screen recordings and identified several reasons. First, some participants misunderstood the DSL syntax and the semantics of some regex operators such as `startwith` and `repeat`. As a result, they wasted a lot of time to recover from their own misunderstandings and did not have enough time to finish the task. For example, two participants thought DSL regexes should be constructed from left to right as in the standard regex grammar. It took them a while to realize they could use operators such as `or` and `concat` to connect sub-expressions in a top-down manner. Second, some participants engaged in a wrong direction for too long before realizing it was a dead end. In fact, 8 of 18 participants started with wrong hypotheses in their mind. Some of them quickly realized they were wrong by observing how the synthesized regexes following those hypotheses failed on their own examples. Yet others just took much longer time to realize it. Third, participants felt overloaded with too many options to guide the synthesizer, not knowing which one option would be the best to communicate their intent to the synthesizer. They ended up spending a lot of time inspecting information in different views and switching back and forth.

Figure 8 shows participants' responses to the five NASA TLX questions [29]. When using interpretable synthesis, participants felt less frustrated, spent less effort, and gave themselves much better performance ratings. Welch's t-test on the five comparisons in Figure 8 shows that the mean differences of performance, effort, and frustration are statistically significant (p=0.04885, p=0.03737, and p=0.02105). Yet the mean differences of mental demand and hurry are not (p=0.6717 and p=0.5495). This indicates that the perception of better performance, less effort and less frustration is more consistent across participants, while there is more diversity in participants' perception of mental demand and hurry.

Figure 9 shows the overall preference and usefulness ratings given by our participants. Among 18 participants, 14 of them preferred or strongly preferred interpretable synthesis, and all but one participants find interpretable synthesis more useful than traditional synthesis. In the post-study survey, 10 participants explained that they were able to get more insights into the synthesis process and observe how their feedback impacted the search strategy, which helped them better figure out what to do next. P8 said, "*it provided a sense of what the synthesizer was doing when it failed instead of just a blank wall and an error message. Towards the end of the first [traditional synthesis] session, I felt almost helpless.*" 11 participants appreciated that they had more agency to control how and where to search in interpretable synthesis, especially when the synthesizer seemed off-track. P1 wrote, "*[interpretable synthesis] allows me to partially guide the synthesizer if I know the next few steps—I don't have to know the entire solution, but* ***I know how to start and I can let the synthesizer fill in the holes****.*"

6.2 Performance and Responses of Different Kinds of Users

We characterize different kinds of users with two factors—regex expertise and engagement tendency, and investigate both their objective performances and subjective preferences in the user study.

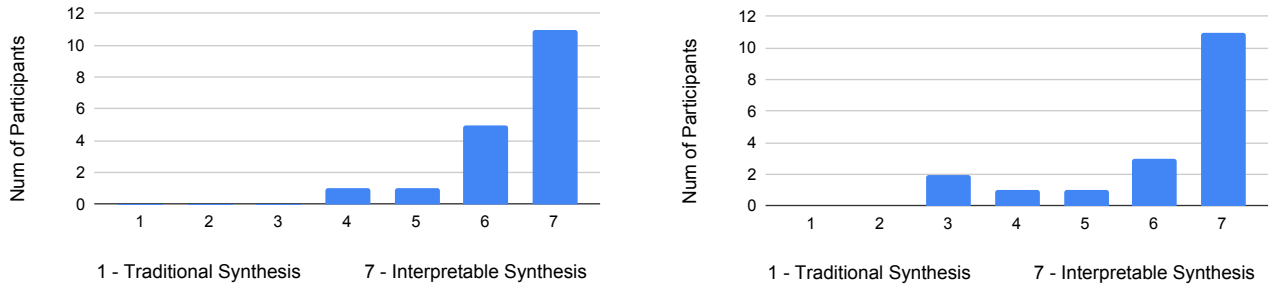
Table 1: The performance of different kinds of participants when using interpretable synthesis.

	Expertise		Engagement Tendency	
	Novice	Expert	Medium	High
Task 1	1/5	1/1	2/3	0/3
Task 2	4/4	2/2	3/3	3/3
Task 3	1/2	2/4	0/2	3/4
Total	6/11	5/7	5/8	6/10

Performance Difference. Table 1 shows the performance, i.e., task success rate, of different kinds of participants using interpretable synthesis. Using interpretable synthesis, regex experts were only marginally better than regex novices: 71% of regex experts (5/7) completed the task assigned to them using interpretable synthesis, while 55% of novices (6/11) completed their tasks using interpretable synthesis. This result is surprising to us. We initially hypothesized that experts would do much better than novices, since experts have more domain knowledge and interpretable synthesis provides a way to incorporate their domain knowledge into the synthesis process. Section 7 offers a discussion on this.

As for engagement tendency, there was little performance difference among participants, regardless of their estimated place on that spectrum: 63% of participants (5/8) with high engagement tendency completed their tasks using interpretable synthesis, while a comparable 60% of participants (6/10) with medium-level engagement tendency completed their tasks using interpretable synthesis. This suggests that engagement tendency has little impact on user *performance*. This is in contrast to our initial hypothesis that users with high engagement tendency would perform much better as the abundance of information presented in interpretable synthesis requires intense intellectual engagement. We discussed one possible explanation in Section 7.

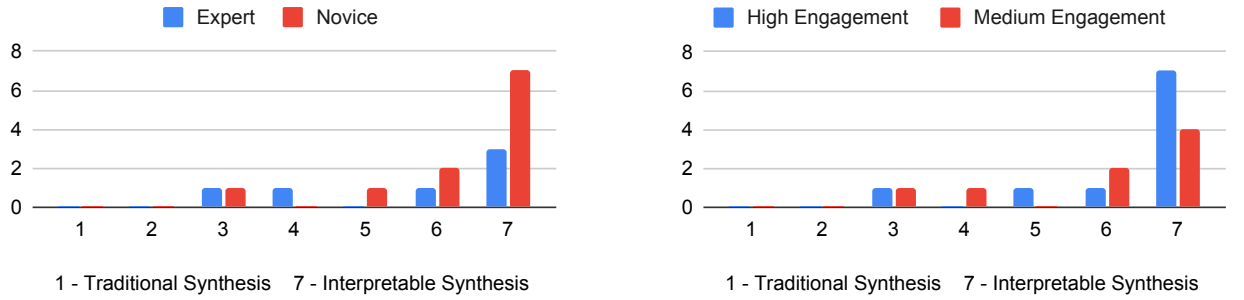
Preference Difference. While interpretable synthesis enabled participants at various levels of expertise and engagement tendency to perform at similar levels, the participants' subjective experience, as measured by their stated preferences, was less uniform. The participants that expressed the strongest preference toward interpretable synthesis were those with either less expertise or higher engagement tendency (see Figure 10). The preference difference as a function of engagement tendency was consistent with our initial hypothesis, since the abundance of information in interpretable synthesis indeed required more intellectual engagement



(a) Which interface did you prefer to use?

(b) Which interface was more useful?

Figure 9: Most participants found interpretable synthesis more useful and preferred to use it over traditional synthesis.



(a) The preference of experts vs. novices

(b) The preference of high vs. medium engagement tendency

Figure 10: Participants with less expertise or higher engagement tendency preferred interpretable synthesis more.

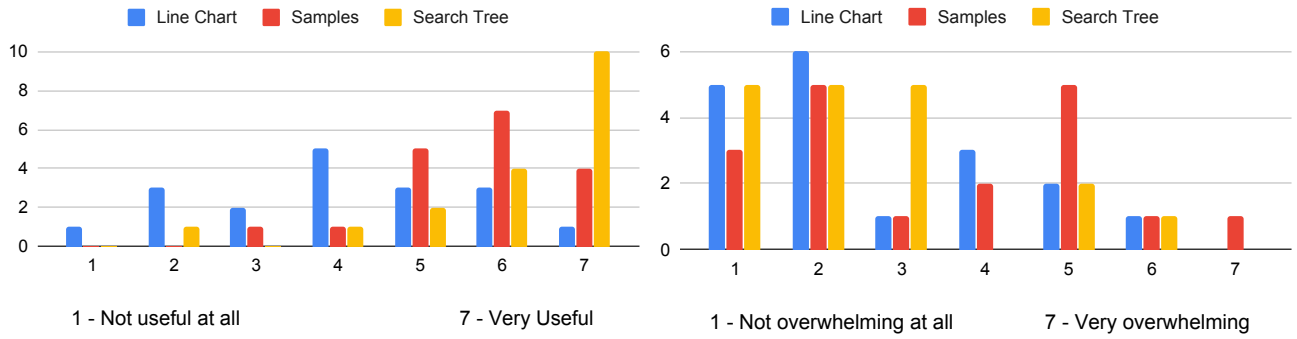
than traditional synthesis. However, regarding expertise, we initially hypothesized experts would prefer interpretable synthesis more than novices, since experts had enough knowledge and expertise to understand the information in search trees and samples. By contrast, novices seemed to appreciate it more. After analyzing the post-study surveys, we realized that experts often had already developed some hypotheses (e.g., partial programs) in their mind after reading a task description. As a result, they wanted to directly give the synthesizer some starting points to work on and rapidly update their own hypotheses based on the synthesis result, rather than waiting for the synthesizer to initiate the interaction and navigating sampled programs or search trees. Unlike experts, novices did not have strong instincts or expertise to rely on and thus showed more appreciation to the synthesis details such as partially correct programs and search trees that helped them initialize and refine their own hypotheses about the correct solution.

6.3 Responses to Different Representations of Explored Program Space

Our interface includes three representations that explain the synthesis process with different levels of fidelity: the live-updated line chart, the various samples of programs tried so far, and the search tree. In the post-study survey, we asked participants to report their preference and rate the usefulness and cognitive overload of each representation. Among 18 participants, 13 of them preferred to navigate and annotate the search tree, 5 of them preferred to inspect and annotate representative programs (i.e., samples), while none of them preferred the live-updated line chart.

We coded participants' explanations to understand why each representation was preferred. The search tree was most preferred (13/18) for two main reasons: it gave them fine-grained control and a holistic view. Six participants mentioned that the search tree provided fine-grained control of the synthesis process. P6 explained, "*the user can easily control what patterns should be considered first. It's much easier than guessing more input examples or including/excluding regex operators.*" While interacting with the synthesizer and observing how different kinds of regexes pass or fail their own examples of desired behavior, participants often developed their own hypotheses about what the final regex might be. Compared with annotating individual regex operators, they found it more convenient to annotate the search tree to give the synthesizer their preferred starting point. By observing the synthesis result from the starting point they gave, they could quickly validate and refine their hypotheses. The search tree was also specifically called out by four participants as giving them a holistic view of the synthesis process. P4 wrote, "*the tree gives me a whole view of available expressions and helps me construct the right expression and guide the synthesizer better. With the sample view, I am restricted to just thinking about the functions that the synthesizer has [already] tried.*"

Still, five participants preferred samples over the search tree. Two participants said the sample view makes it very tangible to figure out how their own examples relate to various possible programs. One participant found the different kinds of programs in the



(a) How useful was each representation?

(b) How overwhelming was each representation?

Figure 11: The search tree was considered the most useful, while the samples were considered the most overwhelming.

sample view inspiring. One participant found it intuitive to annotate sampled programs to narrow down the search space, compared with navigating the search tree.

The line chart was the least preferred because participants felt the line chart only told them the synthesizer was making some progress but did not provide actionable information compared with the sample view and the search tree.

We hypothesized that the search tree would be the most overwhelming representation, since it required navigation over a tree hierarchy. The concept of partial regular expressions might not be easy to reason about, especially for participants with the least expertise. Surprisingly, the majority of participants (15/18) did not find the search tree overwhelming. In Figure 11b, participants' sense of overwhelm as a function of different representations indicates that the search tree was considered no more overwhelming than the line chart, while the sample representation was considered the most overwhelming.

We did observe a relationship between users' engagement tendency and their preferences for different presentations. As shown in Column Engagement Tendency in Table 2, only half of the participants with lower engagement tendencies preferred the search tree, while nearly all the participants with high engagement tendency preferred it.

When considering the expertise of participants, we expected the regex experts preferred the search tree more than novices, since the experts might be more comfortable looking at *partial* regular expressions. Contrary to our expectations, there was not much difference between experts' and novices' preferences (Column Expertise in Table 2). In fact, two out of the five experts in the study liked samples more. Since they were familiar with regular expressions, these two participants found it more intuitive to simply read the synthesized regexes and then check which examples these regexes did and did not satisfy, compared with navigating through the search tree.

6.4 RQ4. Obstacles and Lessons

The user study also revealed several obstacles of using interpretable synthesis:

The last mile problem. Among the seven participants who did not complete the task using interpretable synthesis, three of them

Table 2: The preference of individual representations from different kinds of users

	Expertise		Engagement Tendency	
	Novice	Expert	Medium	High
Line Chart	0	0	0	0
Sample	3	2	4	1
Search Tree	8	5	4	9

guided the synthesizer to generate a regex close to the correct solution before running out of time. For example, in Task 1, P5 reached to a regex, `or(concat(<0>,<num1-9>),concat(<1>,or(<0>,<1>)))`. He only needed to substitute the last subexpression `or(<0>,<1>)` with `or(<0>,or(<1>,<2>))` to get the correct answer. Though this substitution looks trivial to a human, it is much harder for a synthesizer without human guidance, because, at this point, the synthesizer has searched deeply in the program space. Every step forward means exponentially more possible programs to explore, which would take significant time for the synthesizer. We made similar observations in other study sessions—participants made quick progress with the synthesizer in the first 5 to 10 minutes, then they felt the synthesizer got slower and slower, and in turn they needed to provide more and more guidance to reach the final complex regex. This phenomenon is quite similar to *the last mile problem* in telecommunication, supply chain management, and public transportation.

In the example above, prioritizing a search path such as `or(concat(<0>,<num1-9>),concat(<1>,?))` would significantly reduce the program space to explore in the next iteration. On the other hand, if a user could precisely locate such a complex search path, the user is likely to have already known the final regex. In fact, three participants complained that the synthesizer needed too much guidance in the last few steps, which seemed unnecessary as they believed they already knew the answer. In the end, they felt they spent most of time helping the synthesizer rather than being helped.

This poses a fundamental question to synthesis designers—if a user has mentally developed a correct program while interacting with the synthesizer, is it necessary for the user to continue to use the synthesizer? In a real-world setting, the answer is likely no. If a

user has mentally reached a correct program before the synthesizer, we argue that it is not necessary to stick with the synthesizer and force it to solve the last mile problem. As the synthesis process is opened up in interpretable synthesis, users are now able to inspect partially correct programs or observe how a synthesizer succeeds or fails in different search paths. By pinpointing a search path, they could even ask the synthesizer to try out their own hypotheses and rapidly update their own mental models of the space of correct programs. Therefore, we recommend synthesis designers and users to treat a synthesizer as an *artificial pair programmer* that generates, validates, and refines various hypotheses together with a user symbiotically, rather than an *all-or-nothing* tool.

Lack of flexibility to directly suggest a partial program. In the post-study survey, seven participants wished they could enter their own hypotheses (i.e., partial programs) and ask the synthesizer to start its search from there. P4 wrote, "*while the search tree is immensely helpful in choosing the right direction to search from, I would still like to give initial seed functions that it should start from. This would potentially narrow down search space from the beginning*." In the current interface, they have to wait for relevant components appearing in sampled programs or the search tree before they can annotate them. This is much slower compared with directly communicating their own hypotheses to the synthesizer.

More support for navigating samples and search trees. In the post-study survey, four participants mentioned the difficulty of navigating sampled programs and search trees. P6 wrote, "*it is hard to find the right search path with the [interpretable] synthesizer—especially, it's difficult to get the right pattern from the samples.*" Indeed, both the sample view and the search tree view present programs naively and provide little support for rapid navigation or comparing and contrasting. More research is needed in presenting many programs in a more visual and easy-to-interpret manner, such as using code alignment and color highlighting [22].

7 DISCUSSION AND FUTURE WORK

Though synthesis failures are often attributed to user mistakes or the complexity of a problem, Padhi et al. proved that the chance of running into synthesis failures was also significantly impacted by a fundamental design choice—*the expressiveness of a synthesizer* [55]. Increasing grammar expressiveness allows a synthesizer to solve more problems, but it also makes the underlying program space exponentially bigger. This inevitably increases its chance of failing on some previously solvable problems given the same amount of time. Over the years, the PL and ML communities have devoted significant effort to optimizing synthesis algorithms. There is no doubt that synthesis algorithms will continuously be improved. Yet, as pointed out by Gulwani et al. [26], ultimately, there will be limits to complexity that no algorithm improvements can address. Since the problem space in the real world is infinite, there will always be problems that are too complex for a synthesizer to solve, especially within a short time that a user is willing to wait. In such a case, interpretable synthesis offers an alternative way to address the complexity of real-world problems. Compared with the black-box design, interpretable synthesis helps users gain more insights about when and where a synthesizer is stuck, so they can provide more strategic feedback based on their hypotheses and domain knowledge.

Though it is possible that a user may reach a correct solution before a synthesizer, it is still beneficial to have an interpretable synthesizer as a *mental debugger* to help users develop, validate, and refine their hypotheses of the correct solution. When solving challenging tasks, it is common that programmers may start with a wrong mental state or hypothesis. In our user study, eight participants started in a wrong direction when solving a task. For example, in Task 1, P8 initially thought the correct solution should contain the `repeatatleast` operator. After prioritizing the search path of `repeatatleast(?,-1)`, he quickly realized that it was a dead end since the synthesizer enumerated thousands of regexes along that search path but none of them satisfied his examples. By observing how various possible regexes were enumerated and then rejected by the synthesizer, participants said they gained a better understanding of a task and also some hints for the final solution.

We are well aware that, compared with traditional synthesis, the interpretable synthesis interface renders much more code-rich information and synthesis details, imposing more cognitive load on our users. Surprisingly, participants' responses suggest the opposite—when using interpretable synthesis, participants actually felt slightly less mental demand (Figure 8). One explanation is that although interpretable synthesis showed more information and had a more complex UI, participants gained more insights about the synthesis process, which in turn made it easier for them to provide effective feedback to the synthesizer to make progress. By contrast, when using traditional synthesis, participants had to think much harder to make good guesses. Due to a lack of insights about the synthesis process, in case of synthesis failures, participants had to guide the synthesizer in a trial-and-error manner, guessing which examples might puzzle the synthesizer and then changing or deleting them. If it did not work, they had to try modifying another example until the synthesizer finally returned something. This requires a lot of guesses and observations.

We are also surprised by the marginal performance difference between novices and experts and between participants with a high engagement tendency and participants with a relatively low engagement tendency (Section 6.2). One plausible reason is that the abundance of information in interpretable synthesis was still consumable for participants with less expertise or lower engagement tendency, leading to effective actions in the tasks. This implies that, when synthesis details are presented in an understandable way, the benefits indeed outweigh the cost, bridging the gap between users with different expertise and engagement tendencies.

Many users of program synthesis techniques are still end users, who may not be comfortable looking at programs in any programming language. It is an interesting yet challenging task to extend the design of interpretable synthesis to communicate the synthesis process to them. As discussed in Section 2, prior work has explored several ways to communicate the end result, i.e., synthesized programs, to end-users, but not in an interpretable synthesis process. For example, FlashProg [46] translates synthesized string transformations to natural language descriptions to explain their functionality to end-users. In future work, we could adapt this technique to, e.g., communicate the search tree representation of a synthesis

process by translating each search path to natural language descriptions, such as "the synthesizer first tries the `repeatatleast` operator." Communication support, such as pop-up tool tips, linked help pages, and video demonstration snippets, can also be considered to help end-users understand how to respond to code-rich information displays.

During the user study, some participants broke down a task into smaller pieces to solve in our interface. For example, in Task 2, some participants first guided the synthesizer to generate a regex to match 7 digits, marked it to be included in the next synthesis iteration, and then focused on synthesizing another regex that matches 10 digits. In future work, we plan to investigate tool support for task decomposition, such as allowing users to enter partial examples and synthesize regex operators one by one.

The focus of this paper is to understand the value of interpretable program synthesis. Prior work has shown that participants achieved a significantly higher success rate with the assistance of a synthesizer on regex tasks (73% vs 28%) [12]. Given this prior evidence, we chose a comparison baseline in which participants finished tasks with a traditional synthesizer with no interpretability support, rather than finishing tasks manually.

Our interpretable synthesis approach requires instrumenting a synthesizer to log candidate programs that have been generated and tested during synthesis. It is applicable to program synthesis tools whose backbone algorithm involves iterative search, such as enumeration-based search [4, 21, 66], stochastic search [34, 63], or an explicit search process guided by a SMT solver [31] or a probabilistic model [6, 7, 18, 20, 40]. However, some synthesizers completely reduce this search problem to other problems such as satisfiability modulo theories and statistical machine translation. These synthesizers cannot be easily instrumented to log enumerated programs as described in Section 4.3, since they encode user specifications and programs to logical formulae or vectors, and then delegate the search process to a SMT solver or a sequence-to-sequence model. As a result, we would need to further instrument the underlying SMT solver and machine learning model, which remains as an interesting topic to investigate in the future.

8 CONCLUSION

This paper presents a new synthesis design called interpretable synthesis. Unlike traditional black-box synthesis, interpretable synthesis communicates the program space explored by a synthesizer to users, so users can gain more insights of the synthesis process and provide more well-informed guidance to the synthesizer based on their own hypotheses, domain knowledge, and mental model of the synthesizer. Interpretable synthesis is especially useful when a synthesizer fails to generate a program that is consistent with user-provided specifications in an acceptable time. It provides a way of inspecting the explored program space and identifying unproductive search directions that the synthesizer is wasting time on. In a lab study with eighteen participants, we evaluated the usefulness of interpretable synthesis and measured the cost of rendering the abundance of synthesis details to users. The results are promising: the availability of information about—and the ability to act on—the internals of the synthesis process enabled more users to complete challenging programming tasks with the synthesizer.

ACKNOWLEDGMENTS

We would like to thank anonymous participants for the user study and anonymous reviewers for their valuable feedback.

REFERENCES

[1] [n.d.]. Regular expression to check comma separted number values in Flex. https://stackoverflow.com/questions/5262196/regular-expression-to-check-comma-separted-number-values-in-flex. Accessed: 2020-07-21.
[2] [n.d.]. SetCoverPy: A heuristic solver for the set cover problem. https://github.com/guangtunbenzhu/SetCoverPy. Accessed: 2020-06-04.
[3] Robin Abraham and Martin Erwig. 2005. Goal-directed debugging of spreadsheets. In *2005 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'05)*. IEEE, 37–44.
[4] Aws Albarghouthi, Sumit Gulwani, and Zachary Kincaid. 2013. Recursive program synthesis. In *International conference on computer aided verification*. Springer, 934–950.
[5] Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo MK Martin, Mukund Raghothaman, Sanjit A Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. 2013. *Syntax-guided synthesis*. IEEE.
[6] Rajeev Alur, Arjun Radhakrishna, and Abhishek Udupa. 2017. Scaling enumerative program synthesis via divide and conquer. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 319–336.
[7] Matej Balog, Alexander L Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. 2016. Deepcoder: Learning to write programs. *arXiv preprint arXiv:1611.01989* (2016).
[8] Carl Chapman and Kathryn T Stolee. 2016. Exploring regular expression usage and context in Python. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*. 282–293.
[9] Carl Chapman, Peipei Wang, and Kathryn T Stolee. 2017. Exploring regular expression comprehension. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 405–416.
[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*. 963–975.
[11] Jiun-Hung Chen and Daniel S Weld. 2008. Recovering from errors during programming by demonstration. In *Proceedings of the 13th international conference on Intelligent user interfaces*. 159–168.
[12] Qiaochu Chen, Xinyu Wang, Xi Ye, Greg Durrett, and Isil Dillig. 2020. Multi-modal Synthesis of Regular Expressions. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.
[13] Allen Cypher. 1995. Eager: Programming repetitive tasks by example. In *Readings in human–computer interaction*. Elsevier, 804–810.
[14] Allen Cypher and Daniel Conrad Halbert. 1993. *Watch what I do: programming by demonstration*. MIT press.
[15] 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. In *Proceedings of the 2020 CHI conference on human factors in computing systems*. 6038–6049.
[16] Loris D'Antoni, Roopsha Samanta, and Rishabh Singh. 2016. Qlose: Program repair with quantitative objectives. In *International Conference on Computer Aided Verification*. Springer, 383–401.
[17] Kevin Ellis, Armando Solar-Lezama, and Josh Tenenbaum. 2016. Sampling for bayesian program learning. In *Advances in Neural Information Processing Systems*. 1297–1305.
[18] Yu Feng, Ruben Martins, Osbert Bastani, and Isil Dillig. 2018. Program synthesis using conflict-driven learning. *ACM SIGPLAN Notices* 53, 4 (2018), 420–435.
[19] Yu Feng, Ruben Martins, Jacob Van Geffen, Isil Dillig, and Swarat Chaudhuri. 2017. Component-based synthesis of table consolidation and transformation tasks from examples. *ACM SIGPLAN Notices* 52, 6 (2017), 422–436.
[20] Yu Feng, Ruben Martins, Jacob Van Geffen, Isil Dillig, and Swarat Chaudhuri. 2017. Component-based synthesis of table consolidation and transformation tasks from examples. *ACM SIGPLAN Notices* 52, 6 (2017), 422–436.
[21] John K Feser, Swarat Chaudhuri, and Isil Dillig. 2015. Synthesizing data structure transformations from input-output examples. *ACM SIGPLAN Notices* 50, 6 (2015), 229–239.
[22] Elena L Glassman, Tianyi Zhang, Björn Hartmann, and Miryung Kim. 2018. Visualizing API usage examples at scale. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*. 1–12.
[23] Cordell Green. 1981. Application of theorem proving to problem solving. In *Readings in Artificial Intelligence*. Elsevier, 202–222.
[24] Sumit Gulwani. 2011. Automating string processing in spreadsheets using input-output examples. *ACM Sigplan Notices* 46, 1 (2011), 317–330.
[25] Sumit Gulwani, William R Harris, and Rishabh Singh. 2012. Spreadsheet data manipulation using examples. *Commun. ACM* 55, 8 (2012), 97–105.

[26] Sumit Gulwani, José Hernández-Orallo, Emanuel Kitzelmann, Stephen H Muggleton, Ute Schmid, and Benjamin Zorn. 2015. Inductive programming meets the real world. *Commun. ACM* 58, 11 (2015), 90–99.

[27] Sumit Gulwani, Oleksandr Polozov, Rishabh Singh, et al. 2017. Program synthesis. *Foundations and Trends® in Programming Languages* 4, 1-2 (2017), 1–119.

[28] Daniel C Halbert. 1993. SmallStar: programming by demonstration in the desktop metaphor. In *Watch what I do: Programming by demonstration*. 103–123.

[29] Sandra G Hart and Lowell E Staveland. 1988. Development of NASA-TLX (Task Load Index): Results of empirical and theoretical research. In *Advances in psychology*. Vol. 52. Elsevier, 139–183.

[30] Jane Hoffswell, Arvind Satyanarayan, and Jeffrey Heer. 2016. Visual debugging techniques for reactive data visualization. In *Computer Graphics Forum*, Vol. 35. Wiley Online Library, 271–280.

[31] Jinseong Jeon, Xiaokang Qiu, Armando Solar-Lezama, and Jeffrey S Foster. 2015. Adaptive concretization for parallel program synthesis. In *International Conference on Computer Aided Verification*. Springer, 377–394.

[32] 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. IEEE, 215–224.

[33] Andrew J Ko and Brad A Myers. 2004. Designing the whyline: a debugging interface for asking questions about program behavior. In *Proceedings of the SIGCHI conference on Human factors in computing systems*. 151–158.

[34] John R Koza and John R Koza. 1992. *Genetic programming: on the programming of computers by means of natural selection*. Vol. 1. MIT press.

[35] Todd Kulesza, Simone Stumpf, Margaret Burnett, Sherry Yang, Irwin Kwan, and Weng-Keen Wong. 2013. Too much, too little, or just right? Ways explanations impact end users' mental models. In *2013 IEEE Symposium on Visual Languages and Human Centric Computing*. IEEE, 3–10.

[36] Tessa Lau. 2009. Why programming-by-demonstration systems fail: Lessons learned for usable ai. *AI Magazine* 30, 4 (2009), 65–65.

[37] Tessa Lau, Steven A Wolfman, Pedro Domingos, and Daniel S Weld. 2003. Programming by demonstration using version space algebra. *Machine Learning* 53, 1-2 (2003), 111–156.

[38] Vu Le and Sumit Gulwani. 2014. FlashExtract: a framework for data extraction by examples. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 542–553.

[39] Tak Yeon Lee, Casey Dugan, and Benjamin B Bederson. 2017. Towards understanding human mistakes of programming by example: an online user study. In *Proceedings of the 22Nd International Conference on Intelligent User Interfaces*. 257–261.

[40] Woosuk Lee, Kihong Heo, Rajeev Alur, and Mayur Naik. 2018. Accelerating search-based program synthesis using learned probabilistic models. *ACM SIGPLAN Notices* 53, 4 (2018), 436–449.

[41] Gilly Leshed, Eben M Haber, Tara Matthews, and Tessa Lau. 2008. CoScripter: automating & sharing how-to knowledge in the enterprise. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. 1719–1728.

[42] 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*. 6038–6049.

[43] Henry Lieberman. 2001. *Your wish is my command: Programming by example*. Morgan Kaufmann.

[44] Gabriel Lins de Holanda Coelho, Paul HP Hanel, and Lukas J. Wolf. 2018. The very efficient assessment of need for cognition: Developing a six-item version. *Assessment* (2018), 1073191118793208.

[45] Zohar Manna and Richard J Waldinger. 1971. Toward automatic program synthesis. *Commun. ACM* 14, 3 (1971), 151–165.

[46] 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*. 291–301.

[47] Robert C Miller and Brad A Myers. 2001. Outlier finding: Focusing user attention on possible errors. In *Proceedings of the 14th annual ACM symposium on User interface software and technology*. 81–90.

[48] Francesmary Modugno and Brad A. Myers. 1997. Visual programming in a visual shell—A unified approach. *Journal of Visual Languages & Computing* 8, 5-6 (1997), 491–522.

[49] Brad A Myers. 1990. Creating user interfaces using programming by example, visual programming, and constraints. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 12, 2 (1990), 143–177.

[50] Brad A Myers. 1991. Graphical techniques in a spreadsheet for specifying user interfaces. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. 243–249.

[51] Brad A Myers. 1998. Scripting graphical applications by demonstration. In *Proceedings of the SIGCHI conference on Human factors in computing systems*. 534–541.

[52] Brad A Myers and Richard McDaniel. 2001. Demonstrational interfaces: sometimes you need a little intelligence, sometimes you need a lot. In *Your wish is my command*. Morgan Kaufmann Publishers Inc., 45–60.

[53] Brad A Myers, Richard G McDaniel, and David S Kosbie. 1993. Marquise: Creating complete user interfaces by demonstration. In *Proceedings of the INTERACT'93 and CHI'93 Conference on Human Factors in Computing Systems*. 293–300.

[54] Brad A Myers, Brad Vandcr Zanden, and Roger B Dannenberg. 1989. Creating graphical interactive application objects by demonstration. In *Proceedings of the 2nd annual ACM SIGGRAPH symposium on User interface software and technology*. 95–104.

[55] Saswat Padhi, Todd Millstein, Aditya Nori, and Rahul Sharma. 2019. Overfitting in synthesis: Theory and practice. In *International Conference on Computer Aided Verification*. Springer, 315–334.

[56] Rajesh Parekh and Vasant Honavar. 1996. An incremental interactive algorithm for regular grammar inference. In *International Colloquium on Grammatical Inference*. Springer, 238–249.

[57] Hila Peleg and Nadia Polikarpova. 2020. Perfect is the Enemy of Good: Best-Effort Program Synthesis. (2020), 30 pages.

[58] Hila Peleg, Sharon Shoham, and Eran Yahav. 2018. Programming not only by example. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 1114–1124.

[59] Khoo Yit Phang, Jeffrey S Foster, and Michael Hicks. 2013. Expositor: scriptable time-travel debugging with first-class traces. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 352–361.

[60] Veselin Raychev, Pavol Bielik, Martin Vechev, and Andreas Krause. 2016. Learning Programs from Noisy Data. *SIGPLAN Not.* 51, 1 (Jan. 2016), 761–774. https://doi.org/10.1145/2914770.2837671

[61] Mitchel Resnick, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay Silver, Brian Silverman, et al. 2009. Scratch: programming for all. *Commun. ACM* 52, 11 (2009), 60–67.

[62] Joseph R Ruthruff, Amit Phalgune, Laura Beckwith, Margaret Burnett, and Curtis Cook. 2004. Rewarding" Good" Behavior: End-User Debugging and Rewards. In *2004 IEEE Symposium on Visual Languages-Human Centric Computing*. IEEE, 115–122.

[63] Eric Schkufza, Rahul Sharma, and Alex Aiken. 2013. Stochastic superoptimization. *ACM SIGARCH Computer Architecture News* 41, 1 (2013), 305–316.

[64] Eric Spishak, Werner Dietl, and Michael D Ernst. 2012. A type system for regular expressions. In *Proceedings of the 14th Workshop on Formal Techniques for Java-like Programs*. 20–26.

[65] 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. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*. 1–13.

[66] Abhishek Udupa, Arun Raghavan, Jyotirmoy V Deshmukh, Sela Mador-Haim, Milo MK Martin, and Rajeev Alur. 2013. TRANSIT: specifying protocols with concolic snippets. *ACM SIGPLAN Notices* 48, 6 (2013), 287–296.

[67] Earl J Wagner and Henry Lieberman. 2004. Supporting user hypotheses in problem diagnosis. In *Proceedings of the 9th international conference on Intelligent user interfaces*. 30–37.

[68] Richard J Waldinger and Richard CT Lee. 1969. PROW: A step toward automatic program writing. In *Proceedings of the 1st international joint conference on Artificial intelligence*. 241–252.

[69] Chenglong Wang, Alvin Cheung, and Rastislav Bodik. 2017. Interactive query synthesis from input-output examples. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 1631–1634.

[70] Chenglong Wang, Yu Feng, Rastislav Bodik, Alvin Cheung, and Isil Dillig. 2019. Visualization by Example. *Proc. ACM Program. Lang.* 4, POPL, Article 49 (Dec. 2019), 28 pages. https://doi.org/10.1145/3371117

[71] Chenglong Wang, Yu Feng, Rastislav Bodik, Alvin Cheung, and Isil Dillig. 2019. Visualization by example. *Proceedings of the ACM on Programming Languages* 4, POPL (2019), 1–28.

[72] Xinyu Wang, Sumit Gulwani, and Rishabh Singh. 2016. FIDEX: filtering spreadsheet data using examples. *ACM SIGPLAN Notices* 51, 10 (2016), 195–213.

[73] Tianyi Zhang, London Lowmanstone, Xinyu Wang, and Elena L Glassman. 2020. Interactive Program Synthesis by Augmented Examples. (2020), 15 pages.

A THE NEED FOR COGNITION SCALE-6 (NCS-6) QUESTIONNAIRE

This questionnaire is composed of six statements that describe a person's need-for-cognition characteristics. Responses are given on a 7-point scale from "not at all like me (1)" to "very much like me (7)". Four of the six statements are positively associated with need for cognition, while the other two statements are negatively associated (i.e., reverse scored). Asterisks designate the statements that are reverse scored.

1. I would prefer complex to simple problems.

2. I like to have the responsibility of handling a situation that requires a lot of thinking.

3. Thinking is not my idea of fun.*

4. I would rather do something that requires little thought than something that is sure to challenge my thinking abilities.*

5. I really enjoy a task that involves coming up with new solutions to problems.

6. I would prefer a task that is intellectual, difficult, and important to one that is somewhat important but does not require much thought.

B THE COGNITIVE LOAD QUESTIONNAIRE

This questionnaire includes five of the six questions in the original NASA TLX questionnaire [29]. The question "How physically demanding was the task?" is excluded, since the programming tasks in our user study do not involve much physical effort.

Q1. How mentally demanding was using this tool? (1—Very Low, 7—Very High)

Q2. How hurried or rushed were you during the task? (1—Very Low, 7—Very High)

Q3. How successful would you rate yourself in accomplishing the task? (1—Perfect, 7—Failure)

Q4. How hard did you have to work to achieve your level of performance? (1—Very Low, 7—Very High)

Q5. How insecure, discouraged, irritated, stressed, and annoyed were you? (1—Very Low, 7—Very High)