

ScooPy: Enhancing Program Synthesis with Nested Example Specifications

Tomer Katz

Technion

Haifa, Israel

tomerkatz@campus.technion.ac.il

Hila Peleg

Technion

Haifa, Israel

hilap@cs.technion.ac.il

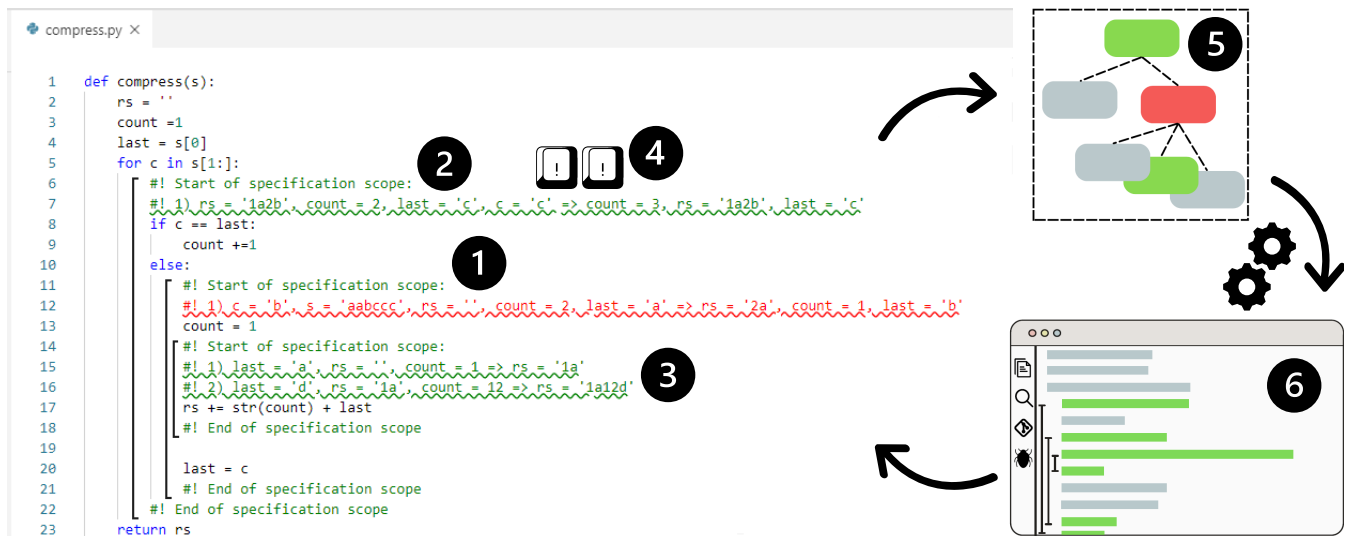


Figure 1. Synthesizing code from nested example specifications using ScooPy: ① an *example scope* with one input-output example. Variable states at input and output are separated by the `=>` arrow. The example is used as a test for the code within the scope, providing live feedback that the code fails for this input-output pair. ②, ③ An outer and an inner scope with code that is correct for their examples. ④ When the programmer moves their cursor to a scope and types `!!`, ⑤ that scope and all the scopes nested within it are sent to the synthesizer. ⑥ A synthesis result is returned to the file. Examples and their nesting are preserved and the code is updated to satisfy the examples.

Abstract

Current IDE-integrated program synthesis leaves no indication of what code was auto-generated, let alone an explanation of why. This makes both identifying and understanding machine-generated code hard. We therefore add *example scopes*, comments enclosing synthesized code that document the input-output examples that created it. This also allows programmers to manually edit examples and re-launch the synthesizer without tediously re-entering the examples. Scopes are simply text, and so can be created anywhere, including inside other scopes. However, synthesizers

can only reason about one flat example set. To address this, we introduce ScooPy, IDE-integrated program synthesis for nested example specifications. ScooPy lets programmers edit example scopes, see live information based on the examples, and call the synthesizer on nested scopes. In two user studies with 6 and 16 participants we see that example scopes increase users' engagement with the code and that ScooPy improves users' ability to synthesize for some types of tasks.

CCS Concepts: • Software and its engineering → Integrated and visual development environments.

Keywords: program synthesis, interaction model

ACM Reference Format:

Tomer Katz and Hila Peleg. 2025. ScooPy: Enhancing Program Synthesis with Nested Example Specifications. In *Proceedings of the 2025 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! '25)*, October 12–18, 2025, Singapore, Singapore. ACM, New York, NY, USA, 20 pages. <https://doi.org/10.1145/3759429.3762619>



This work is licensed under a Creative Commons Attribution 4.0 International License.

Onward! '25, Singapore, Singapore

© 2025 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-2151-9/25/10

<https://doi.org/10.1145/3759429.3762619>

1 Introduction

Program synthesis generates code in response to a user-provided specification. In recent years, program synthesis has been incorporated into development environments where it can accept user specifications and insert the resulting program directly into the user’s code file [1, 13, 14, 16]. Once added to the file, synthesized code is no different than any human-written code. Once time passes, and the code is added to the project’s source control, it cannot be traced back to the synthesizer, only to the user who ran it. In such a case, a programmer—the same one who synthesized the code or a different one—may return to a piece of code in order to debug, modify, or extend it and wonder: did I write this or did the synthesizer? And *why* did it end up like this?

Moreover, it matters not only *that* the code was synthesized but what it was synthesized *from*. If the code was generated as prefix completion from a generative model (e.g., [1]), there is no simple way to explain where it came from. But if the synthesizer is specified using Programming-by-Example (PBE) [5, 11, 16, 18], a modality where the generated program is guaranteed to satisfy a provided set of examples, seeing the examples provided to the synthesizer is helpful for anyone returning to the code. For instance, the user may remember that the code

```
1 t = str(len(s)) + s[0]
```

was generated by a synthesizer, but not why. A programmer using *small-step PBE* [14], i.e., advancing through the problem in small, relatively linear chunks, and generating code from examples for each such chunk, will have many such lines strewn throughout their code.

An immediate solution to this is to document synthesized code as such via code comments. Code generation frameworks [8, 17] use this for the separate, generated code files, as do integrated code generators [33]. For inline synthesized code, the comment needs to serve two needs: to show what input was provided to the synthesizer, and to delineate exactly what code was specified, as this may be several lines. The first need is served by including the programmer-provided specifications as a documentation of the synthesizer’s input in the format *input* => *output*, and the second by the synthesizer outlining the *scope* of the synthesized code. Together, they form an *example scope*:

```
1 #! Start of specification scope:
2 #! 1) s = 'ddd' => t = '3d'
3 #! 2) s = 'cc' => t = '2c'
4 t = str(len(s)) + s[0]
5 #! End of specification scope
```

The information that example scopes contain can be leveraged in a myriad of ways to interact with and comprehend the synthesized code. i) The textual representation of the examples can be edited by the programmer and re-sent to the synthesizer. This means, e.g., correcting a slight error in the

entered specifications does not require the programmer to re-enter all specifications (something users of previous systems complained about [13]). ii) This also creates a keyboard-only mechanism for interacting with the synthesizer, one that does not change the focus to any other windows as previous synthesizers do. iii) Moreover, if the user does not like the result, they can incrementally refine the specification by adding more examples. The persistence of previous examples makes searching for a differentiating example (a task previously shown to be hard [35]) easier for the programmer. iv) Finally, the enclosed code can be executed on its examples, providing *live feedback* [42] that changes as the code or the examples change. This helps programmers better understand the code and raise their confidence in its correctness.

In an unconstrained tool, the programmer can edit any of the text—code or comments—in any way, and call the synthesizer anywhere, and even cut and paste synthesized code (behavior observed by Ferdowsifard et al. [14]). These can lead to example scopes that are nested within each other, as in Figure 1. Both bottom-up and top-down approaches to code construction [9] can cause a new scope to be added around or inside an existing scope, and the result is the same: specifications that cannot be sent to current PBE synthesizers, which can only handle “flat” example sets. To allow unconstrained editing and synthesis, our system must respect and accept nested scopes in both interaction and synthesis.

To support this workflow, we designed ScooPy,¹ a development environment and synthesizer that support hierarchies of example scopes. ScooPy extends the LooPy synthesizer and development environment [13], which neither documents synthesis results with an example scope nor support any nesting of example specifications. ScooPy’s development environment offers the programmer easy ways to create and edit example scopes, and provides live feedback on the code inside the scopes based on those examples. This includes treating examples as tests and highlighting passing and failing tests in the editor. Additionally, since nesting makes it harder to reason about the full specification, ScooPy helps the programmer identify some *contradictions* in their specifications, stopping them from launching the synthesizer for a task that will certainly fail. Finally, ScooPy lets the programmer launch the synthesizer from any example scope to synthesize a result for it and any scopes nested inside of it.

To synthesize with nesting, the ScooPy synthesis algorithm extends LooPy’s, using a syntax-guided approach to ensure all nested scopes of examples are considered by the synthesizer. This breaks the hierarchical specification into *sub-goals*, addressed by several synthesis calls. Each scope is then required to use code that satisfies any nested scopes. This choice satisfies our design goals, that no internal scope goes unaddressed by the synthesizer, and that the result be

¹<https://github.com/tomerkatz2001/ScooPy.git>, VM with ScooPy pre-installed is available at <https://doi.org/10.5281/zenodo.16933581>.

sound: if a program is found, then it is correct, i.e., satisfies all provided examples including all nested examples.

To evaluate ScooPy we conducted two user studies: one, a within-subjects study on 6 participants, evaluating only the liveness and editing of example scopes, and the second, a between-subjects study on 16 participants, evaluating synthesis with ScooPy. Our results show that example scopes increase users' engagement with the code, and users react favorably to live feedback based on them. Moreover, ScooPy's ability to handle nested scopes helped participants better achieve synthesis objectives in some types of tasks.

We also conducted an empirical evaluation of ScooPy's back-end. Using 50 benchmarks, we measured the synthesizer's runtime and the number of nested examples required to solve programming tasks, and compared the results with those of LooPy. Additionally, we evaluated the accuracy and runtime of large language models (LLMs) on the same synthesis tasks and compared their performance with our synthesis algorithm. Our results show that ScooPy solved most of the benchmarks in less time and with fewer examples.

This paper makes the following contributions:

- ▶ Automatic documentation of synthesized code with an *example scope* that also records with what examples the code was synthesized.
- ▶ An interaction model that helps programmers edit example scopes and leverages them to provide live feedback and warnings for synthesis calls that will fail.
- ▶ A new syntax-guided synthesis algorithm that uses the structure of the scopes to ensure nested examples are not discarded and a sound result is returned.
- ▶ Two randomized controlled studies of ScooPy exploring the impact of its interaction and synthesis capabilities.
- ▶ An empirical evaluation showing ScooPy's synthesis algorithm's advantage in speed and number of examples needed.

2 ScooPy by Example

Eunice, a programmer, is asked to write a run-length encoding compression function in Python, the kernel step of which is shown in Section 1. Since she uses program synthesis in her day-to-day work, she employs it in this task. She has a high-level idea of the solution: iterating over the input with three variables: last character seen, count of the number of times it was seen, and an accumulator *rs* to save the result. She first writes the `for` loop and the skeleton for the conditional for checking if the current character is new.

The case where `c == last` (*then* branch) is simple enough, and Eunice writes it herself. The *else* branch is more complicated, and Eunice synthesizes it using LooPy's `??` shortcut:

```
compress.py
1 def compress(s):
2     rs = ''
3     count = 1
4     last = s[0]
5     for c in s[1:]:
6         if c == last:
7             count+=1
8         else:
9             rs, count, last = ??
10    return rs
```

Eunice uses LooPy's live values to provide one example for the full *else* block. The synthesizer inserts the result into the code, documenting it with an *example scope*:

```
if c == last:
    count+=1
else:
    ##! Start of specification scope:
    ##! (1) rs='',count=2,last='a',c='a'=>rs='2a',count=1,last='b'
    count = 1
    rs = "2" + last
    last = c
    ##! End of specification scope
```

The example in the scope is colored green, indicating that when the code inside the scope is run on the example's input, variable values after it match the example's output.


Since only the *else* block is specified, Eunice wraps the whole loop body with another example scope: she selects the entire conditional and presses `Ctrl` + `3`. This creates the scope's header and footer around the block and lets her enter examples. Since Eunice is an experienced synthesis user, she always specifies conditionals with at least two examples, one for each branch. However, Eunice makes a typo when entering the output for the example specifying the *else* branch, entering the string `'2s'` instead of `'2a'`. This causes a *conflict*: there are now two examples with the same input specifying two different outputs. Simply put, there is no longer a program that can satisfy all examples. Despite the fact the two examples are not in the same example scope, ScooPy can help!

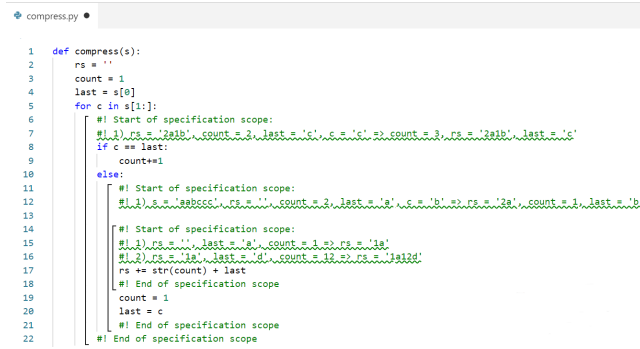
```
for c in s[1:]:
    ##! Start of specification scope:
    ##! (1) rs='',count=2,last='a',c='a'=>rs='2a',count=1,last='b'
    ##! (2) rs='',count=2,last='a',c='a'=>rs='2s',count=1,last='b'
    if c == last:
        count+=1
    else:
        ##! Start of specification scope:
        ##! (1) rs='',count=2,last='a',c='b'=>rs='2a',count=1,last='b'
        count = 1
        rs = "2" + last
        last = c
        ##! End of specification scope
    ##! End of specification scope
```

ScooPy recognizes the conflict and immediately indicates to Eunice which examples are the problem. Moreover, if Eunice tried to call ScooPy's synthesizer now, she would get a warning. Eunice inspects the conflict and decides that the *else* block is sufficiently covered by the inner example, so she deletes the second example of the outer scope, which resolves the conflict.

Eunice now notices a bug in the *else* branch: the synthesized assignment `rs = '2' + last` is overfitted. Eunice deletes

the line and calls the synthesizer again only for this assignment, providing two examples. This synthesizes the correct assignment and brings Eunice to the state of the code shown in Figure 1. However, once this assignment is replaced, the middle block (1) no longer satisfies its example. While Eunice may not immediately see why (the order of the assignments is wrong), she can easily see that the example does not hold: ScooPy provides this information as live feedback by coloring the example in red.

While Eunice could continue debugging the code manually with the help of the live feedback from the example scopes, she instead invokes the synthesizer: placing her cursor at the outer-most scope, she presses , which sends the example scope at the cursor and all its nested scopes to the ScooPy synthesizer (5). ScooPy synthesizes a new solution that satisfies all examples while maintaining their hierarchical nature and replaces it in the editor:



The live feedback for all example scopes is now green, since all code enclosed by a scope satisfies the examples. Since this code is synthesized, this is by construction.

A demo video of this example in ScooPy can be found in the supplementary material.

3 Background and Related Work

In this section, we review the current state of tool-embedded program synthesis, and provide background on the techniques ScooPy extends and builds on.

3.1 Synthesis from Examples

Programming-by-example (PBE) [25] is a program synthesis paradigm where calls to the synthesizer are specified with input variable values and their respective outputs, effects, or variable values after the run, and is handled by a variety of specialty algorithms [2, 4, 18, 43] many of which are domain-specific [6, 44, 47, 52].

PBE tools that output programs into a more general development environment [5, 11, 16, 18, 53] often lose connection between the original specification and the synthesized code [11, 13, 14, 16]. New versions of FlashFill [18] within Microsoft Excel keep cells filled in by synthesis separate from user-provided cells until their content is accepted by

the user, but after that the distinction is lost. Santolucito et al. [37] relies on a specification file kept alongside the code file to provide examples at the function level. Section 4.1 will describe how ScooPy maintains the connection between synthesized code and specification, which enables editing and liveness capabilities in the IDE.

Current PBE synthesizers only handle a “flat” specification: one or more examples all specifying the same code at the same level. *Sub-goals* in PBE remain an open problem, sometimes relying on user-provided task decompositions [14, 21, 49], and otherwise explored for techniques to synthesize recursive programs [2, 32, 34, 50]. *Extending* inner specifications to an outer scope can work forward via execution and even backwards over some language constructs [29], but it requires assuming that the code within the scope is correct. If the programmer is calling the synthesizer to fix a scope where the examples do not hold, this cannot be assumed. Section 4.2 will describe the new ScooPy synthesis algorithm designed to respect example specifications nested inside each other, without any additional assumptions.

3.2 Supporting Users of PBE

Jayagopal et al. [19] showed that programmers favor synthesis tools that allow them to mix editing and synthesis in the same interface. ScooPy incorporates synthesis within the editor, allowing the programmer to mix synthesis with editing of both code and specifications. Specifically, editing the specifications to fix mistakes or refine them has been a user pain point before in PBE [13, 35], especially since PBE synthesizers tackle the inherent underspecification of examples by asking users to provide more examples to clarify their intentions. Some synthesizers help users disambiguate their intent by providing them with new inputs to specify the outputs for [20, 30, 46, 52], but this may not be sufficient to find a *differentiating example* that both exemplifies intended behavior and rules out a current bad program, a problem shown by Peleg et al. [36] to be time-consuming for users. Some tools allow disambiguation via other means than additional examples [35, 36, 51, 52]. However, these disambiguation interactions take place in specialized interfaces.

ScooPy also provides programmers with live feedback by treating example scopes as localized tests. Testing within a unit is not a new concept. It was used to locally test whole functions [12], provide inputs for individual code blocks [23], and even single statements [27]. Liu et al. [26] also support extracting such local tests from separate unit tests. All of these were shown to be helpful in a variety of programming tasks like debugging and code comprehension.

Jayagopal et al. also noted the importance of feedback from the synthesizer in the case of failure to find a program. In PBE one cause of failure is specifying an outright contradiction: the same input mapping to two outputs is a specification with no solution. While some synthesizers explicitly explain why such contradictions hard to test (e.g., Osera and Zdanczewicz

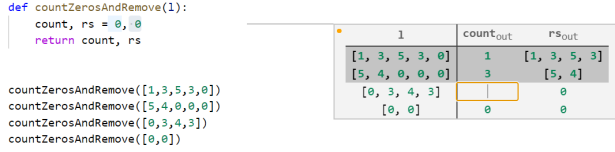


Figure 2. Synthesizing using live values in LooPy: the user assigns output values to variables `count` and `rs` to pair with a live input state from the Projection Box.

[34]) the standard behavior of synthesizers is to assume this is the user’s responsibility, and the search will time out if there is a contradiction. Because contradictions in a nested specification can be harder to identify, ScooPy also provides as much assistance as possible in conflict identification.

3.3 The LooPy Synthesizer

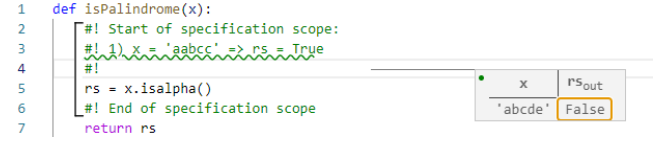
ScooPy extends LooPy [13], a PBE synthesizer embedded in the Live Programming environment Projection Boxes [24] inside VS Code [31]. LooPy leverages the available live values provided by Projection Boxes as inputs, only requiring the programmer to provide outputs. Its user interface for entering examples is seen in Figure 2.

LooPy’s synthesis algorithm can synthesize assignment statements, sequences of assignments, and conditional statements containing assignment sequences. However, to do so, it requires programmers to provide *block-level specifications*: to specify full blocks of code with end-to-end examples that contain output values for all variables in the block. This is in contrast with *small-step PBE* [14] where the programmer advances through the program one statement at a time, thinking through the problem in small chunks.

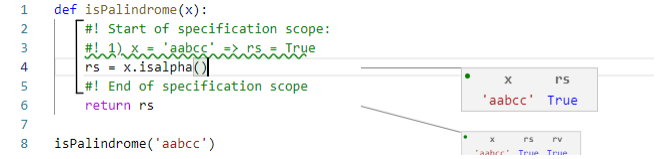
The `compress` function shown in Section 2 was also the motivating example for LooPy. There, the programmer not only had to provide at least five examples in order to solve it, but every example had to describe the behavior of the entire loop iteration, for all variables. ScooPy’s example scopes, however, let the programmer think about smaller pieces of the program, and to exclude variables that do not matter like `s` in all scopes or `c` in the inner-most scope. Not only are these examples easier to provide, the programmer also needs to provide fewer examples overall.

4 The ScooPy System

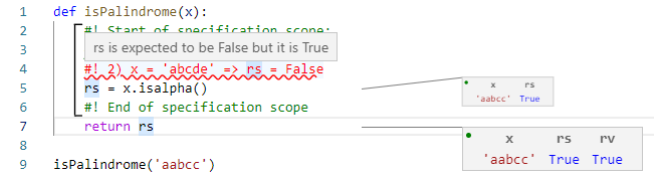
ScooPy is built on top of LooPy’s live PBE interaction model, extending it with four main concepts: i) *persisting* the examples used to call synthesis, ii) *editing* example scopes, iii) *live feedback* for the scopes, and iv) *synthesizing* nested example scopes. Section 4.1 describes the editing and liveness extensions in greater detail, and Section 4.2 describes the changes to the synthesis algorithm.



(a) ScooPy helps the programmer add a new example in the correct format by providing a table to fill.



(b) Line 3 is green to denote that line 4 satisfies the example. The Projection Box for line 4 shows values of live variables.



(c) Line 4 is red to indicate line 5 does not satisfy the example. Hovering over line 4 shows what output variables are incorrect.

Figure 3. Example scopes in VS Code with ScooPy.

4.1 Editing and Liveness of Example Scopes

4.1.1 Persisted Example Specifications. Two use-cases motivate the persisting of synthesis specifications, each contributing goals for our design. First is the need to identify synthesized code and what specification caused it to be generated. This indication should follow the code into version control, and remain even if the programmer edits some of the code. We notice this has no bulletproof solution: programmers who edit synthesized code may wind up entirely replacing the code within an indication of synthesis; likewise, they can manually remove any indication left by the tool that the code was originally synthesized.

Second, as noted by users in the studies for synthesis tools LooPy and SnipPy, once the specification was sent to the synthesizer it disappeared, so if the programmer needed it again to a) reflect, b) fix a typo, c) refine it (i.e., add examples), or d) make a behavioral change, they had to re-enter all examples from scratch. The inability to fix a mistake, in particular, was a great pain point for LooPy’s users [13, Section 8]. The examples themselves, then, should persist alongside the code.

Persisting the examples as code comments is inspired by previous work providing localized execution inputs or tests [12, 23, 26, 27]. These share the approach of storing values as comments in the code file, with IDE support for displaying the values, evaluating, and visualizing the outcome. We choose the format of *input => output*, to be read as “if the input values are so, then the output values will be so”,

where *input* and *output* indicate variable values immediately before and immediately after the code in the scope.

Because snippets returned from the synthesizer can be part of a larger block of code, we cannot follow the lead of previous work [12, 23] and use the end of the function or Python scope to terminate an example scope; instead ScooPy’s example scopes need to open and explicitly close. In the IDE, the start and end of a scope are connected by an indent guide for ease of reading.

4.1.2 Editing and Synthesizing Example Scopes. Example scopes are, at their heart, still example specifications. This means they can be passed back to the synthesizer to *re-synthesize*. When a programmer calls the synthesizer with their cursor at an example scope, that scope and any scopes nested within it will be sent to the synthesizer. Any other code, including other enclosing scopes will not be affected. The resulting code will replace the entire scope, and will still persist the examples, including in nested parts of the result, as shown in Section 2.

Because example scopes are a structured format of Python comments, the programmer can easily edit them. This makes re-running the synthesizer with corrected or extended specifications simple.

And, of course, to the synthesizer there is no difference between an example scope created by the synthesizer and one in the same format entered by the user. This creates a keyboard-only interface with the synthesizer, where the programmer manually adds an example scope and sends it to the synthesizer.

For users who do not want to work only in text, ScooPy also supports adding more examples via a table to fill (Figure 3a) similar to the box for collecting examples for a new synthesis task (Figure 2).

4.1.3 Live Feedback from Example Scopes. ScooPy is built on top of LooPy, itself incorporated in the live programming environment Projection Boxes [24]. This means LooPy continually displays runtime values as a means to providing feedback about the program as it is being written. This is seen side by side with ScooPy in Figure 3c. The examples for synthesis can originate from the Projection Boxes runtime values but can also be manually entered via an example scope. Incorporating them into the Projection Boxes can cause confusion.

The live feedback ScooPy offers, then, is centered around the textual comments of the example scope. It constitutes an additional *layer* of liveness for the code. The advantage of this is that this feedback is still available when Projection Boxes are not, e.g., if the current function is not executed.

Examples as tests. In the spirit of Du et al. [12], Lerner [23], and Liu et al. [27], ScooPy considers each example in an example scope as a test for the code inside the scope, including the code inside any nested scopes. The code is

```
1 def f(x):
2     This example is in conflict with the example on line: 4
3     x = 4, y = 8
4     x = 4, y = 10
5     y = x + x + 2
6     #! End of specification scope
7     return y
```

(a) Additional information about a conflict.

```
1 def f(x):
2     #! Start of specification scope:!!
3     x = 4, y = 8
4     x = 4, y = 10
5     y = x + x + 2
6     #! End of specification scope
7     return y
```

Warning! Can't synthesize this scope: this scope or some of its nested scopes contain conflicting examples. Resolve them and try again.

(b) Warning when the user tries launching synthesis with a conflict.

Figure 4. ScooPy highlights the conflict between two examples. Each example in the conflict is highlighted in orange.

continually evaluated on the input values in the examples and compared to the output values. In Figure 3b the example is colored green since the code inside the scope satisfies it, and in Figure 3c the example that does not hold is red. In scoped code inserted by the synthesizer, all examples are initially colored green, as synthesized code is *correct by construction* with respect to the examples provided for synthesis. However, users can edit both the examples and the code inside the scope, including adding more examples that did not participate in synthesis. Any edit can cause a “failing test” that is immediately indicated to the programmer.

Conflict Identification. When a user provides examples, they can wind up specifying an empty set of programs. In PBE, there are many ways to specify something outside the synthesizer’s space, but the only real contradiction is two examples with the same input and different outputs. For the most part, synthesizers assume *consistency* of the provided examples, which is the user’s responsibility. However, nested example scopes are more complex, and the programmer can more easily miss a contradiction if the contradicting examples are not within the same scope, as shown in Section 2.

If the programmer creates a scope with two examples, both with the same input state but with different output states, this is easily identifiable as a contradiction, and ScooPy will highlight the two contradicting examples in orange, as seen in Figure 4. However, the freedom that ScooPy affords users allow for two complications: (1) conflicting examples in different scopes and (2) examples that specify different sets of variables.

If the two examples are in different scopes, the same input with two outputs is no longer necessarily a contradiction. This means ScooPy needs to filter out cases that *may* have a solution. If the two scopes follow each other, for example, they will generate two consecutive assignments and there is no conflict. Likewise, if they enclose each other but the inner scope is in sequence with additional statements: if the solution performs assignments before the inner scope, this disconnects the identical inputs.

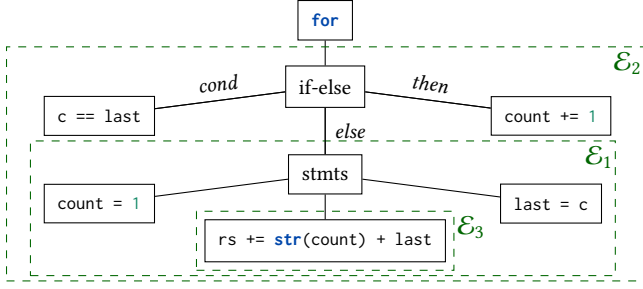


Figure 5. AST of the loop body in Figure 1. Green boxes mark the scope specified by the example scopes.

If the examples with identical inputs specify two separate branches of a conditional, then they specify two unrelated runs, and are not a problem. However, as will be discussed in Section 4.2, while this is not technically a conflict, it will fail when attempting to synthesize the code. As such, ScooPy will still warn about this case.

Scopes can also specify different variables, both in the input and in the output. For example, scope ③ in Figure 1 specifies only `rs` in its output, whereas ① and ② specify `rs`, `count`, and `last`. Likewise, the inputs in ③ do not specify `c`, but ① and ② do. If a variable is unconstrained in both the input and output of an example, the example now describes more than one concrete run of the code. A more constrained example can conflict with a less constrained one if the same run occurs in both; ScooPy therefore tests not only identical inputs, but also *subsuming* inputs, i.e., if all the specified variables in one are specified identically in the other. Likewise for outputs ScooPy tests whether a variable specified in both outputs is specified differently.

Once a conflict has been identified, it will not only be indicated in the editor: ScooPy will warn the programmer if they try running synthesis on the scope or any scope containing it as shown in Figure 4b.

4.2 The ScooPy Synthesizer

To support the potential nesting of specifications, the synthesizer behind our interaction needs to ensure all examples—or synthesized code representing them—reach the outer-most scope where the programmer called synthesis. This ensures the soundness of the solution: a resulting program will satisfy all of the examples contained in the input, as well as their structure. In this section, we describe how the synthesizer *scoops* the internal example scopes outward, creating multiple synthesis tasks that maintain the division into sub-goals. This ensures the nested examples are not lost, and has the potential to make synthesis faster and need fewer examples.

In Section 3.1 we discussed the two problems with extending an example outward to an enclosing scope: i) the need to evaluate code backwards, and in particular to find unknown values before assignments, and ii) the inability to

assume the code is correct. This means we have to look at each example scope where it is. We want the synthesizer to use scopes *locally*, but to also preserve the additional information encapsulated in the nesting. To this end we employ a syntax-guided approach.

The tree structure of specifications. Because specifications are an enclosing structure over parts of the code, they can be parsed along with the language. Figure 5 shows the resulting AST for the specified part of Figure 1, including the example scopes enclosing each sub-tree, denoted \mathcal{E}_1 , \mathcal{E}_2 , and \mathcal{E}_3 matching ①, ②, and ③ in Figure 1, respectively. This can now be incorporated into the tree construction algorithm of the synthesizer, which, like the original LooPy synthesizer, forms program trees bottom-up.

Scooping internal examples. ScooPy’s synthesizer scoops examples by following the AST bottom-up. Each example scope reached is either reserved as examples or synthesized. When synthesized, the way its result needs to be reserved within any enclosing scope is determined by its parent node. A full version of the algorithm appears in Section A. Based on LooPy’s original target grammar, our implementation is concerned with two composition productions: a sequence of statements and a conditional statement. The third production in the target language, a single assignment, is always a leaf in the tree, and so is not involved in scooping.

Scooping conditionals. First, we consider a conditional: an example scope can specify the entirety of a branch in a conditional statement, like \mathcal{E}_1 in Figure 5, or the full conditional, like \mathcal{E}_2 . In total, there may be three example scopes immediately above and immediately below a conditional. Considering them at the same level means they can help the user find a different, better condition for their `if` statement, or help them refactor the code into straight-line code that no longer needs the conditional. Lubin and Chasins [28] observed that programmers often first write code with unnecessary cases, which is then refactored into a single case; aiding in this refactoring is itself a use-case for synthesis.

Example scopes that specify the branches explicitly contain additional information: this is a user-provided hint that if the synthesized solution contains a conditional, these examples should take the same path in the code’s execution. (This is trivially true if the solution does not branch.) In other words, the synthesizer should rule out any boolean condition that separates them. The ScooPy synthesizer, then, can rule out the invalid branchings, proceeding to synthesize as usual afterward.

Scooping sequences. The second production that the synthesizer needs to consider is sequencing: an example scope can be in a sequence with other statements, which themselves may or may not be specified. For example, \mathcal{E}_3 is in a sequence with the statements before and after it inside the `else` block. Assignments before it and possibly-incorrect code after it

mean we cannot “stretch” the examples to the scope of the enclosing block.

Instead, the synthesizer will synthesize any example scopes in the sequence, and pass the results up the tree. Any enclosing scope will then synthesize not only using its own examples but will also be *required* to use the synthesis results from inner synthesis results, implemented using a *retain* predicate [35] in its synthesis specification. In Figure 5, the synthesizer finds `rs += str(count) + last` as a solution to \mathcal{E}_3 , which the next synthesis task, that for the conditional, is then required to use.

More synthesis tasks, fewer examples. One sub-tree with nested specifications can end up triggering several dependent synthesis tasks. However, these tasks fare better than a single large task: both the division into sub-goals and the reduction in the number of examples each individual task handles speed up synthesis. Moreover, because nested specifications are more expressive than top-level examples, the user will need to provide fewer examples across all scopes than they would when specifying just at the top level. These are shown in our additional non-user evaluation in Section 7.

Specifying with different variables. In Section 4.1.3, we described cases where two examples—that can even be in the same scope—have different input variables and different output variables. This is an unusual situation for a PBE synthesizer. To synthesize with such a set of examples, ScooPy’s synthesizer needs to define the meaning of a missing variable v in an example, which is different in an input or an output. In inputs, ScooPy considers this variable to have the value \perp , i.e., be unavailable. Because ScooPy can synthesize conditional statements, the solution can have one branch that handles only examples where v has a value, and so uses v , and another branch that does not use v at all. If v is missing from the output of an example, however, we consider it to be *unconstrained* (with value \top) in that example, i.e., a solution can assign any value to it in a run on the example’s input.

5 User Study 1: Example Scopes

Our first study is a small-scale, online, within-subjects study of ScooPy that isolates example scopes and their liveness and editing features, without the added complexity of synthesis. This gave us an initial assessment of the value of persisted examples, as their existence is the assumption the remainder of the ScooPy system hinges on.² The study aims to explore our preliminary research question:

RQ1 Does persisting specifications in example scopes help programmers better understand and manipulate synthesized code?

5.1 Method

Participants. We recruited a convenience sample of six participants (6M), all students at Technion. Two of the participants were graduate students and four were undergraduate with at least two years of industry experience. All had at least seven years of programming experience, and three years of Python experience. Participants were compensated \$13/hr for their time.

Tasks. We used the four tasks previously used in the user study for SnipPy [14], the predecessor to LooPy, where participants had originally described issues with the ephemeral nature of specifications. In our version of the tasks, code files contained a buggy solution synthesized using small-step specifications. Each task had two versions, synthesized with the same sequence of *small-step PBE* [14] steps: one constructed from LooPy’s results (i.e., no documentation of synthesized snippets) and one constructed by ScooPy, where each small step had its own example scope. Nested example scopes were not synthesized, so the code in the synthesis results was identical between settings. A top-level comment provided several inputs to the function and their expected outputs, one of which is not satisfied by the program.

Task A. Abbreviate: given a name string, return its initials, separated by a ‘.’ and in lower-case. This task contained a *typo in the specifications*, rather than code overfitted to the examples.

Task B. Duplicates: given an input string, return how many characters appear in the input more than once.

Task C. Max and Min: given a string of numbers, return a string containing the largest and smallest.

Task D. Palindrome: return `True` if the input string is a palindrome when rotated zero or more characters to the left.

The initial code that was given to participants for each task can be found in the supplementary material.

Methodology. The study took place over a Zoom call, with full control of the researcher’s computer transferred to participant. To answer RQ1, we asked participants to solve two tasks in LooPy’s editor and two tasks in ScooPy’s. The order of the tools and tasks was counterbalanced using a Latin square.

Participants were first shown an introductory video about program synthesis, where the problem of overfitted synthesis results was explicitly explained to them, and another video about using Projection Boxes and ScooPy’s specifications. For each task, participants were then asked to identify the cause of the failure and *manually* fix the code. They were also asked to inform the researcher when they thought the code works correctly. Participants who stopped at an incorrect solution were given another failing function-level test; tasks had a 25 minute timeout. Participants could not call the synthesizer.

²Replication package for both studies is available at <https://doi.org/10.5281/zenodo.16937699>.

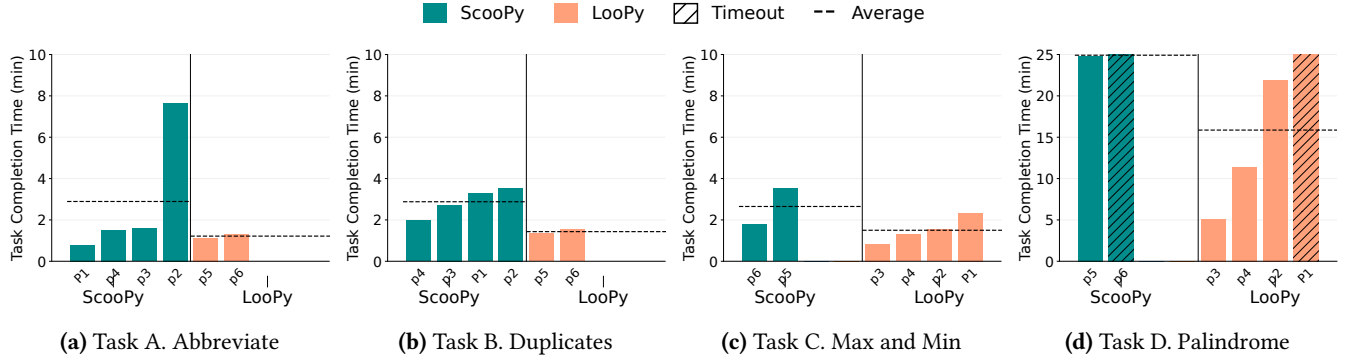


Figure 6. Time to complete each task in Study 1 for each participant. Averages include timeouts.

5.2 Results

While the LooPy baseline allowed users to add more executions of the code that would generate more live values in the Projection Boxes, none did. When using ScooPy, users demonstrated via both actions and verbal comments more engagement with the code and the tool. Some users, even though they verbally expressed a correct understanding of a specific line of code, still added more examples for additional clarity. P5 said, “I don’t think there’s a problem here, but I’ll just add another example”. This heightened engagement with examples also led users to consider new function-level executions and add them to the ones provided with the task.

Even after the program was fixed, participants using ScooPy continued adding both high-level and line-specific specifications. Users preferred adding examples to existing scopes over adding new execution values to view in the Projection Boxes. Two out of the four users who used ScooPy first wanted to add new example scopes. For example, P4 said, “I don’t remember if `sorted` returns a value... how can I add the comments above the line by myself?” All users who used ScooPy before LooPy asked for example scopes back when solving tasks with LooPy (P1, P4, P6).

As an interesting anecdote, the Palindrome task was given to four users in LooPy, and of those, two (P2, P3) chose to delete and re-implement the entire function instead of investing the necessary effort to understand the code. Neither of the two users solving this task with ScooPy did this.

The times shown in Figure 6 reflect when users indicated they were ready to walk away from the task. Almost universally, this time was longer with ScooPy. However, this went hand-in-hand with more apparent willingness to understand the *existing code* rather than just the *task*. Notably, studies such as Gajos and Mamykina [15], Vaithilingam et al. [45] illustrate that lack of engagement with generated code often results in incorrect code and minimal learning.

6 User Study 2: Synthesizing

Our second study is a between-subjects study to examine how users handled the added expressivity of synthesizing with hierarchical specifications. We aim to further support **RQ1**, now considering example scopes and their attached editing and liveness features while synthesizing and not just reading synthesized code. Additionally, it aims to answer our remaining research questions:

RQ2 Do programmers understand the meaning of nested scopes as a synthesis specification?

RQ3 Does synthesis of nested specifications improve programmers’ success with their synthesis objectives?

6.1 Method

Participants. We recruited 16 participants (9F, 7M), all students at Technion, by hanging flyers in communal study areas in the CS department. Of these, one was a master’s student and the rest were undergraduate students. None of the participants had any prior practical experience with program synthesis. For those who responded to the flyer, we performed an initial pretest to guarantee participants had an adequate programming level in Python: participants were asked to find the first index in a string where a second string appears by iterating on the string with a loop. None of the respondents failed to perform this initial task. We denote the participants P7 to P22. Participants were compensated \$13/hr for their time.

Settings. Our experiment required a baseline setting against which to compare ScooPy. We could compare it with the existing LooPy IDE, as in Section 5, i.e., completely without example scopes. However, since our first study showed the documentation and liveness inherent in the scopes to be valuable to users, simply adding synthesis (with different algorithms) to both settings would have confounded the results.

We therefore created an intermediate setting, which we name ScooPy_{flat}: users had full access to the ScooPy interaction, including a synthesizer that generates example scopes,

but with a synthesizer that can only solve “flat” specifications, representing the status-quo in symbolic synthesis. This synthesizer can solve all tasks with sufficient top-level examples, but when invoking `Scoopyflat` on nested example scopes, the synthesizer only respects the top-level examples. This creates a setting where users *can* create nested scopes, and isolates the change in the synthesizer to respect nesting in the study.

Participants in the `Scoopyflat` setting were explicitly shown the behavior of `Scoopyflat` on a nested specification as part of the study’s guided introductory task.

Tasks. Our tasks set out to explore the two scenarios where we hypothesized that nested specifications would be more helpful than “flat” specifications: fixing unfamiliar code and extending existing code. To this end, we have two main study tasks and one additional scaffolding task.

Task A. Compress: Starting with an initial state similar to the incorrect state in Figure 1, participants were asked to fix the synthesized code, which already includes example scopes.

Task B. Brackets (scaffolding): A scaffolding task in which participants create the code they then extend in Task C. This neutralizes the unfamiliarity of the code which is often a confounder in modification tasks. In the initial task we asked users to format a string and surround it with brackets. The task had no initial code, only a top-level example. No nesting was needed to solve this task, only a sequence of assignments, so we expected to see no difference between `ScooPy` and `Scoopyflat`.

Task C. Brackets (test): Starting from the result of task B, participants were asked to extend their code to add a new case where the string uses another formatting.

The initial sketch for the first task and the docstring for the second task can be found in the supplementary material.

Methodology. To answer RQ2–3, we randomly assigned participants to one of the two settings: `ScooPy`, or `Scoopyflat`.

Participants of both groups first watched an introductory video about program synthesis, Projection Boxes, and example scopes. Next, they performed a guided introductory task, where the researcher walked them through using the editing, liveness features and calling the synthesizer.

During this walk-through, participants were instructed to create a two example scopes nested one within the other by wrap an existing example scope with a new scope that specify a missing case. Participants were then asked to predict what would happen if the synthesizer was re-launched on the external scope in order to fix the internal code (in practice: `Scoopyflat` would ignore the internal scopes, while `ScooPy` would not). After they answered, they were instructed to call the synthesizer, and then to explain what actually happened and how it varies from their prediction.

Participants solved all three tasks in order using only editing of examples, adding new example scopes, and synthesis.

Task A had a timeout of 25 minutes, and tasks B and C had timeouts of 15 minutes each.

Finally, participants were asked to fill out a survey reflecting on their experience. Questions alternated positive and negative phrasing. Participants filled out the survey in a form that randomized question order and submitted it without the researcher’s supervision. These are intended to mitigate known biases that occur in participants’ responses.

6.2 Results

Understanding nested specifications. When asked during the guided task what would happen when re-synthesizing a nested specification, participants’ predictions were mixed. Six of them thought the synthesizer would disregard the inner scope, or were unsure but thought it would not take it into account (P21). Even though they thought so, P14 indicated they found this to be undesirable: “I would have wanted it to do something with all the examples”.

The other ten participants thought the synthesizer would use inner examples, saying, e.g., “it will use all the examples” (P19). P19 noted that this is not only their prediction but the *correct* behavior: “it should keep all the examples you provided to it (not just use them)”. Two participants believed that the synthesizer could, or at least should, track the history of synthesis calls and use the data from previous synthesis.

Five participants out of the sixteen predicted the inner example scope would be left completely intact, and the synthesizer “shouldn’t touch the inner one” (P9). This was unrelated to whether they thought the synthesizer would take the inner examples into account or not.

After re-synthesizing the external scope, five of the `ScooPy` participants offered an explanation for what happened during re-synthesis, and all five were correct. The other three offered no explanation, only declarative descriptions of the result. No participant provided an incorrect description.

`Scoopyflat` users described what happened as “deleting” (P14, P17, P22), or “overriding” (P10, P13). Some reacted with confusion (P18, P21), and reconciled it by saying the inner scope did not count as examples (P21). P10, who had initially predicted the inner scope would be used by the synthesizer categorized the result as synthesis having failed.

Using nested specifications. During the Compress task, three participants from each group wrapped code that includes another example scope with a new example scope. However, not all of them did this to re-synthesize: some only did it as a means to test their code. Many participants tried to wrap the top-level of the function in an example scope, but were instructed by the researchers not to do so because the code includes a loop, which is outside of `Scoopyflat` and `ScooPy`’s target language. In the post-study survey, both `Scoopyflat` and `ScooPy` users rated wrapping examples as fairly easy (averages 3.9 and 3.4, where 1 is “very difficult”) and most found it not frustrating (averages both 2, where

1 is “not frustrating at all”). Users also found it simple and valuable to distinguish between scopes of examples, but not as simple and valuable as they found example scopes themselves.

During the two parts of the Brackets task, nine participants from both groups (P9, P12–19) created nested scopes. They synthesized the next assignment inside the scope of the previous one. Since they synthesized an assignment to a new variable, this did not “break the tests”: the variable specified by the outer example scope did not change. And this created an unbalanced tree of example scopes with a long chain to the right. Scoopy_{flat}’s users did this on average 0.25 times per task and ScooPy’s users 0.5 times per task.

At some point during Task A all participants re-synthesized a scope that includes a nested scope. However, ScooPy’s participants did so more often: 2.5 times on average compare to 1.62 times for Scoopy_{flat}’s participants (medians 1.5 and 1, resp.).

In the Brackets (test) task, while many participants created nesting of scopes, participants only synthesized stand-alone or inner-most scopes. This, again, neutralized the difference between the tools.

Synthesizing locally. None of the ScooPy participants added unnecessary input variables to the given inner scope in the first task. Two of them (P13, P14) also added another nested scope and provided only the local and needed variables to specify their intent. On the other hand, almost all Scoopy_{flat} participants at some point re-synthesized the outer scope, removing the inner scope. This led to a loss of locality, as they had to work with examples that required specifying more input variables, rather than focusing on the relevant localized context.

Efficiency and expressiveness of ScooPy. While solving the Compress task, Scoopy_{flat} participants modified their examples and re-launched the synthesizer nearly twice as often as ScooPy participants: 8.25 re-synthesis calls on average and a median of 8.5, compared to 4.37 times on average and a median of 4.5.

Adding and modifying examples. All participants modified existing example scopes, most often as a means to fix buggy examples and re-launch the synthesizer. This is the behavior studied artificially in Task A of our first study (Section 5) demonstrated on the participants’ own synthesis calls. Several (P9, P14, P17) explicitly pointed out that inspecting the example scopes helped them figure this out.

Participants of both groups responded favorably in the post-study survey to adding and modifying examples. In Table 1 all but one of the participants marking adding and modifying examples as non-frustrating or neutral activities (all averages under 2, where 1 is “not frustrating at all”), and all but two participants ranked the ease of adding and modifying examples as easy or neutral (all averages over 3, where 1 is “very difficult”). Moreover, participants found





Table 1. Participant responses to the post-study survey about features available in Scoopy_{flat} (F) and ScooPy (S).

	avg		distribution	
	F	S	F	S
Rate the ease of the following action in the editor 1=very difficult, 5=very easy				
Adding examples	3.8	4.1		
Modifying examples	4.5	3.3		
Wrapping code with examples	3.9	3.4		
How frustrating was performing the following action in the editor? 1=not frustrating at all, 5=very frustrating				
Adding examples	1.6	1.5		
Modifying examples	1.9	1.6		
Wrapping code with examples	2	2		
How valuable did you find the following feature? 1=not valuable at all, 5=very valuable				
Coloring examples according to their evaluation result	4.5	4.9		
The tool’s ability to visually distinguish between different scopes of examples	3.6	3.5		
Keeping past synthesis examples in the editor for future synthesis	4.6	4.5		
How complicated was it to understand the following feature? 1=not complicated at all, 5=very complicated				
Coloring examples according to their evaluation result	1.6	1.5		
The tool’s ability to visually distinguish between different scopes of examples	2.4	2.1		
Keeping past synthesis examples in the editor for future synthesis	1.6	2		
Did you trust synthesized code? 1=completely distrusted, 5=fully trusted				
When synthesizing with ?? (from scratch)	2.3	3		
When re-synthesizing a single scope of examples	2.8	3.6		
When re-synthesizing multiple nested example scopes	3	2.4		

keeping the examples in the editor for future synthesis to be valuable (averages 4.5 and 4.6, where 1 is “not valuable at all”) and simple to understand (averages 1.6 and 2, where 1 is “not complicated at all”).

Using live feedback. Participants from both groups considered the live feedback for example scopes valuable (averages 4.5 and 4.9, where 1 is “not valuable at all”), and simple to understand (averages 1.6 and 1.5, where 1 is “not complicated at all”). Only one participant marked the coloring of tests as complicated to understand, but did not mark them as not valuable.

Table 2. Participant responses to the post-study survey summarizing their experience with Scoopy_{flat} (F) and Scoopy (S).

	avg		distribution	
	F	S	F	S
1=ineffective, 2=slightly effective, 5=very effective				
How effective were the examples in helping you understand code you did not write?	3.3	2.8		
1=very dissatisfied, 5=very satisfied				
How satisfied are you with the overall user experience of the tool?	3.3	3.6		

As in our first study, some users (P10, P19) added more examples to their code after they reached a solution in order to check these examples would hold as well. This is a behavior specifically enabled by the live feedback. P11, who used ScooPy, also pointed to one of the input-output examples provided in a comment for the task description and said, “I wish it would appear up here with color”.

Specifically, conflict identification was more important than we anticipated: participants created in 50% of ScooPy sessions and in 38% of ScooPy sessions. The majority of these conflicts were conflicts between examples in the same scope. Only a handful were between examples of nesting scopes. Participants who got a conflict warning quickly recovered and modified or deleted one of the conflicting examples.

Times and success rates. Figure 7 shows the times until participants completed each of the tasks. One ScooPy participant, P15, timed out performing the scaffolding Brackets task and could not continue on to Brackets (test), indicated with a -. One Scoopy_{flat} participant, P7, left the experiment after the first task, and is indicated as- for the remaining tasks. Figure 7 also shows the average time to task completion, including participants who timed out as the task timeout, 15 or 25 minutes.

Excluding timeouts, 5 of 8 ScooPy participants finished the Compress task in an average of 19.5 minutes, compared to one successful Scoopy_{flat} participant who finished the task in 25 minutes, seconds before the timeout. Figure 7 shows averages counting timeouts as 25 minutes, the average completion time with ScooPy was 21.5min and for Scoopy_{flat} the average was 25min. The difference between the two groups is statistically significant (unpaired t-test, $p = 0.0326$).

In the scaffolding Brackets task, the two settings are near equivalent—synthesis is always of the next step and no nested scopes need to be synthesized to complete it. It is expected, then, that participants in both settings performed similarly; the difference between the groups is not statistically significant ($p = 0.7603$).

In the Brackets (test) task, Scoopy_{flat} participants finished in average of 2.3min. With a timeout counted as 15min,

ScooPy participants finished in an average of 6min. However, this difference is not statistically significant ($p = 0.1091$).

User feedback. After seeing its effects in the guided task, Scoopy_{flat} participants were wary of any nesting they encountered during the study itself. P10 said, “if I touch the outer scope it will disrupt all of the inner scope”. P14 was also cautious, “if I [synthesize the outer scope] it would delete everything”.

In the post-study survey, ScooPy participants expressed a slight distrust of code synthesized from a nested specification compared to Scoopy_{flat} participants, as shown in Table 1. We interpret the difference as Scoopy_{flat} participants using this feature less, because they understood it would delete inner examples along with the code. In contrast, ScooPy users knew that internal scopes would be considered but had not yet developed enough intuition to be as confident about how.

Overall, ScooPy users were more satisfied with their experience (Table 2), on which the large gap in success rates of Task A likely had some impact.

7 Non-User Evaluation

In this section, we show our non-user evaluations of ScooPy’s synthesizer. We aim to answer the following research questions:

- RQ4** Do nested specifications require fewer top-level examples and fewer examples in general to synthesize correct programs?
- RQ5** Does the addition of nested specifications and the division into sub-tasks allow the synthesizer to search its space faster than synthesizing a non-hierarchical specification?
- RQ6** Can an LLM serve as the synthesizer for nested example specifications instead of ScooPy’s synthesis algorithm?

Benchmarks. For the experiments in this section, we constructed two benchmark sets:

- *main*: a set of 50 programming tasks inspired by tasks in the SyGuS competition [3] and from other PBE synthesizers [13, 22]. We turned each benchmark into a hierarchical ScooPy task by synthesizing small parts of each task’s solution and manually writing other parts, in a workflow similar to the motivating example in Section 2. We then added a top-level example scope with a few examples from the original benchmark.
- *flat*: to compare ScooPy to a synthesizer without hierarchical capabilities, we converted every benchmark in the *main* set to a flat benchmark: given a benchmark with k total examples in all nesting levels, we retained only the examples in the outer-most scope (examples from the original task from the literature) then ran the LooPy (flat) synthesizer on these examples. If the synthesizer returned an overfitted program another example from the original

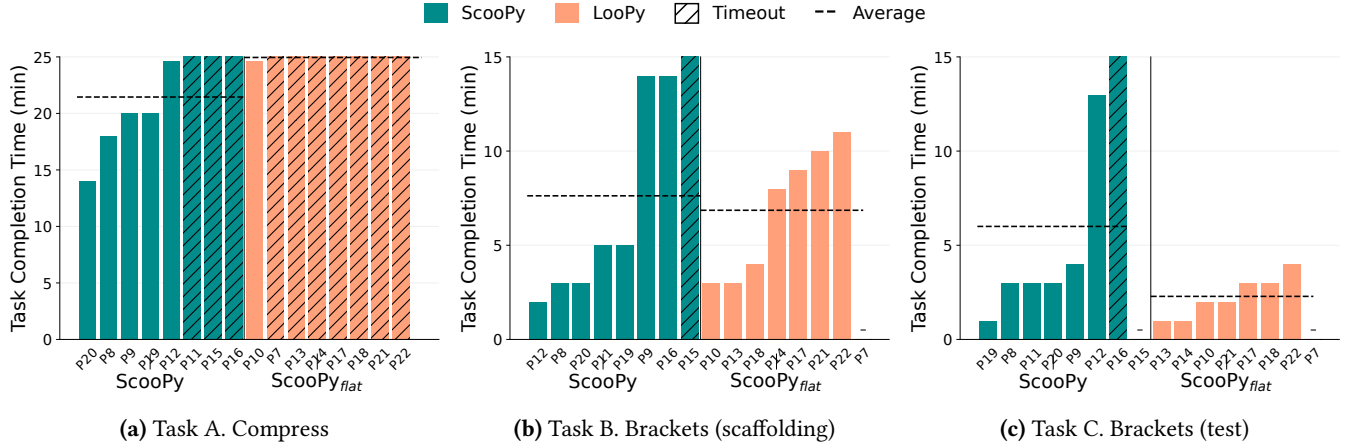


Figure 7. Time to complete each task in Study 2 for each participant. A participant who did not participate in the task, e.g., because of timing out in the scaffolding task, is denoted with a “-”. Averages include timeouts.

SyGuS task was added to the set. We continued adding examples until a the synthesizer found the target program, up to a limit of $k + 5$ total examples, i.e., giving the flat benchmark up to five *additional* examples over the total number of nested examples.

Experimental setup. Experiments that refer to a “flat” synthesizer use the LooPy synthesizer. In a flat setting (i.e., no nesting), LooPy and ScooPy synthesizers are equivalent except for ScooPy’s support of the *retain* predicate, which does not feature in runs without nesting.

Times were recorded as an average of the synthesizer run-time over five runs. We define a benchmark as solved if it was solved in over half (i.e., three or more) of the runs, but in practice no benchmark was split between solved and timeout results.

All benchmarks were run on a laptop computer with 12th Gen Intel(R) i7-1260P 2.10 GHz processor and 32 GB of RAM. We ran the benchmarks on the with a timeout of 10 seconds.

7.1 RQ4: Difference in expressiveness

7.1.1 Method. To test RQ4 we constructed the *flat* benchmark for each benchmark in the *main* set, and attempted to synthesize the same code with as few top-level examples as possible and no nested examples. To compare it to ScooPy in terms of expressiveness we ran the *flat* version of each benchmark on the flat synthesizer, and examined the *delta*: the difference in the number of examples required for each benchmark to be solved. For ScooPy, we count examples as both the outer-most and any nested examples in inner scopes, i.e., anything a user would have specified. We ran each benchmark on both tools with a timeout of 10 seconds.

7.1.2 Results. The number of benchmarks solved at each delta is shown in Figure 8.

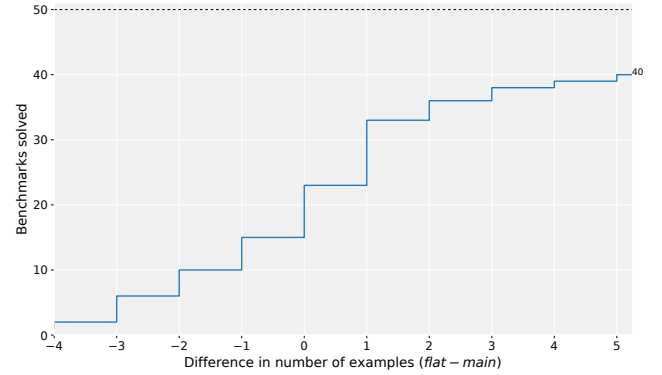


Figure 8. Number of benchmarks solved at each size of delta between *flat* and *main* (cumulative). Benchmarks solved at a positive difference (i.e., need more examples for *flat*) or remain unsolved show ScooPy’s additional expressiveness.

In 15 of the 50 benchmarks, the flat synthesizer needed a smaller total number of examples than ScooPy’s synthesizer. This is because the *main* set was constructed to mimic human action, rather than to minimize the number of examples used, whereas the *flat* set was methodically constructed from it.

In eight additional benchmarks the number was equal, and in 17 benchmarks, the flat synthesizer needed more examples at the top-level than ScooPy needed at the top level and in all nested scopes. Finally, in the 10 remaining benchmarks, either 5 additional examples were not sufficient or a timeout was reached, meaning the flat synthesizer could not find a solution.

Even when looking only at the number of examples, we see that nested specifications’ ability to specify locally allows many tasks to be solved with fewer examples. This was originally the philosophy of Small-Step PBE [14]—which is

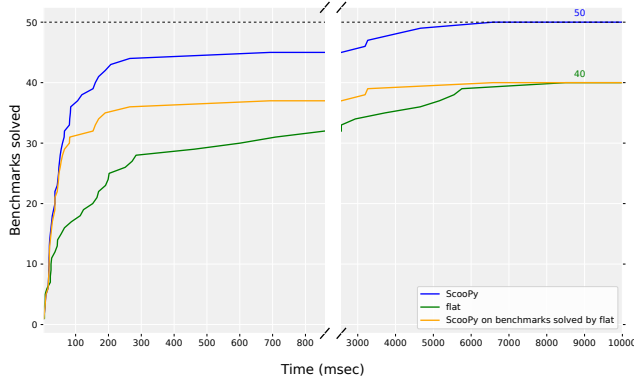


Figure 9. Number of benchmarks solved, cumulative over time (higher is better). ScooPy runs on benchmarks from the *main* set while the flat synthesizer runs on their *flat* counterparts. The orange line shows ScooPy only on benchmarks that the flat synthesizer can solve, and is contained in the full ScooPy (blue) run.

here extended to nesting. We find that often hierarchical specifications *can* be more expressive.

7.2 RQ5: Accelerating the search

7.2.1 Method. To test whether nesting can accelerate the search of the synthesizer’s space, we compared ScooPy and the flat synthesizer: ScooPy on the *main* version of each benchmark and the flat synthesizer on the corresponding *flat* version. We ran each benchmark on both tools with a timeout of 10 seconds. For each benchmark we measure the synthesizer’s run time until solving.

7.2.2 Results. The number of benchmarks solved by each synthesizer over time is shown in Figure 9. The graph shows three lines: benchmarks solved by ScooPy, benchmarks solved by the flat synthesizer, and an additional line showing the progress of ScooPy on the 40 benchmarks that both synthesizers can solve.

The flat synthesizer was able to solve 40 out of the 50 benchmarks. ScooPy solved all of these faster than the flat synthesizer, as well as the remaining 10 benchmarks in the set that time out in the flat synthesizer.

This shows the power of nesting to accelerate the search by providing synthesis with simpler sub-goals and hints, and allows us to answer RQ5 in the affirmative.

7.3 RQ6: ScooPy vs. LLMs

7.3.1 Method. To test RQ6 we compared ScooPy’s synthesizer to a LLM with a PBE prompting strategy on a set of benchmarks with hierarchical specifications.

Experimental setup. We used the OpenAI API for Python for this experiment. We tried several models and chose gpt4-o, as it performed the best of the available models. Our

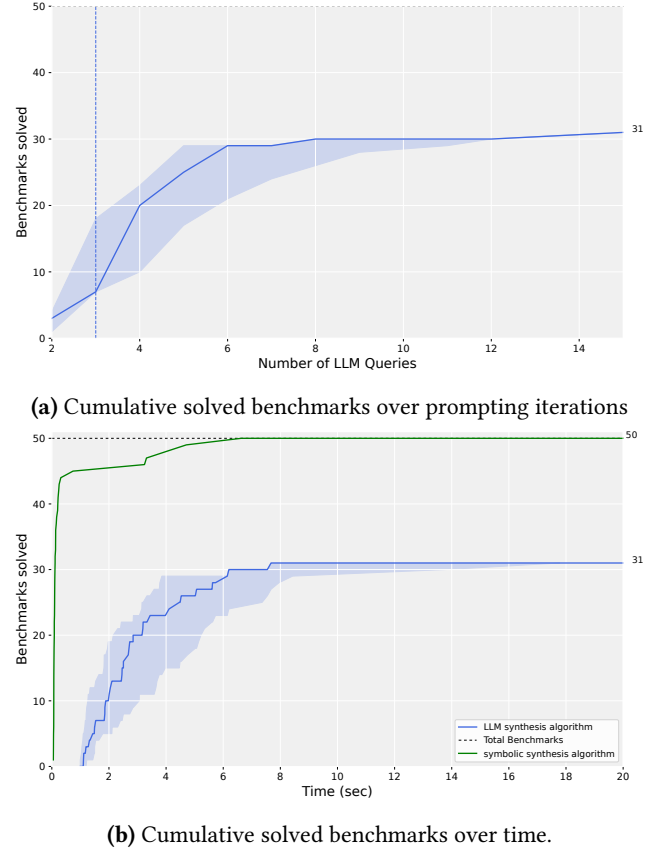


Figure 10. Comparing ScooPy to gpt4-o (higher is better). For gpt4-o line shows the median run out for each benchmark, with the surrounding range delineating best and worst runs.

prompt used the same role and description of the task in all benchmarks. Our test program also validated the response by counting the number of returned examples, verifying the format of the response and evaluating the examples on the code itself.

Prompting strategy. To prompt the model, we used a Chain of Thought and Refinement style of prompting [41, 48]. Each prompt contained the delineating comment for one example scope, and nesting was constructed bottom-up: successful results from nesting level $n-1$ were provided with the prompt to nesting level n , asking the model to use the previous level’s result in the current level. The prompt asked the model for code and to return the example scope provided surrounding the code, to mimic ScooPy’s creation of example scopes. Using input-output examples as part of the prompt was preformed in multiple fields such as agents refinement [41] and programs repairs [40].

If code could be extracted from the response, it was verified by evaluation on the examples in the specification, and returned to the model if the code did not run or the examples

did not hold. Additionally, preservation of examples in the example scope was tested. Every time the model failed, we provided it with the resulting error—failing to satisfy the examples or a missing example scopes—as feedback. The LLM then used this information in the next call to attempt to fix the previous program. This iterative feedback loop is widely used and was shown improve the accuracy of LLM-generated code [7, 10, 38]. Benchmarks had a timeout of 20 seconds.

Data collected. We ran the above setup on all benchmarks in the *main* set and collected the number of iterations needed, the type of error for error iterations, and the time. Since prompting an LLM is not deterministic, we ran each benchmark 5 times. We also compare these to the runtimes of ScooPy on the *main* set collected in Section 7.2.

7.3.2 Results. The results are shown in Figure 10. In all runs, the LLM was able to solve correctly 31 out of the 50 benchmarks. While the maximum number of scopes in the benchmark set (i.e., minimum number of queries at which all benchmarks could be solved in the experimental setup) was 5, the LLM required up to 15 queries to solve the tasks it did solve. Likewise, the LLM took substantially longer to solve the benchmarks—unsurprising given the number of iterations with the LLM—only five tasks were solved in under a second, whereas ScooPy solves 45 tasks in under a second.

93% of queries to the LLM returned some sort of erroneous response. We analyzed the errors and found a variety of reasons. In 74% the model did not return examples along with the code, or changed the specification examples in the example scope, in 13% the response did not satisfy the examples provided, 11% of the responses contained no code at all, and 2% had other errors.

8 Discussion and Future Work

Usefulness of example scopes (RQ1). Users in both studies were unequivocally pleased with example scopes as a tool for testing and comprehension, both with and without access to synthesis. The live feedback for example scopes also encouraged users to add more examples for new potentially-edge cases they considered, finishing the task only after several of those were green; this behavior was observed in both of the studies. This meant task times were a little longer when the method of solution was equivalent (i.e., the first study and the Brackets task of the second study), but participants were more certain of their solution.

Moreover, the presence of the examples surrounding a synthesis result allowed them to reflect on a synthesis query they made, and make fast judgements about whether the code is overfitted. Example scopes were frequently used to re-synthesize, and as a result, users trusted re-synthesized code more than initially synthesized code.

We therefore answer RQ1 in the affirmative: ScooPy’s introduction of example scopes helps users understand and interact with synthesized code more effectively.

Understanding nested specifications (RQ2). Not all ScooPy users correctly predicted what would happen the first time they encountered synthesis with nested specifications in the guided task of the second study. However, all correctly understood and explained back what had happened once it was demonstrated to them. This base level understanding was likely not enough to garner trust: ScooPy users did not fully trust code synthesized with a nested specification, whereas ScooPy_{flat} users who understood exactly how the synthesizer would treat nesting (i.e., ignore it) trusted the results more. We believe this difference (average of 2.4 compared to 3 for ScooPy_{flat}, Table 1) would improve over continued use.

ScooPy participants rated the general principle as not complicated to understand (Table 1), and were still more pleased with their experience with synthesis. While we cannot answer RQ2 with an unequivocal affirmative, we see the results as promising, and believe that with time synthesis with nested examples will feel like a less opaque action, even though it is not a completely simple one.

Usefulness of synthesizing nested examples (RQ3). Our second study found ScooPy to be very useful in a debugging task, but not improving time or success in a code modification task. We do not know whether this is due to the nature of the task or some characteristic of the task itself. However, this does show that there *are* tasks where ScooPy makes a big difference, and in others it is equivalent to ScooPy_{flat}.

The overall satisfaction with ScooPy was higher. Users reported less frustration and higher levels of understating of the interaction. On average participants who used ScooPy also added fewer examples and edited previous ones until synthesis returned a desirable result: the additional expressiveness of ScooPy and its ability to provide specifications locally lowers the burden of providing examples. This accentuates the fact that though nested specifications are not the best tool for every type of task, they do provide programmers with the ability to synthesize and express specifications locally, itself a powerful tool.

To answer RQ3, then, synthesis with nested examples *sometimes* improves on synthesis with flat examples.

Other ways to form nesting. Our initial hypothesis was that the two main cases where nesting would form given example scopes were using top-down and bottom-up construction of the code. However, during our second study, 9 of the 16 participants in both groups managed to create nested examples even though they essentially solved the task as *small-step PBE* [14], synthesizing a sequence of assignments one at a time, by synthesizing a new assignment *at the end* of an enclosing scope. While unexpected as a behavior, this only shows how easily example scopes saved in the code file can wind up nested inside each other by unintentional

programmer actions. This further motivates the need for a tool that accepts nested examples when the programmer later tries to interact with them.

A separate layer of liveness. Separating ScooPy’s live feedback into a separate layer of liveness separate from Projection Boxes was confusing to some users (P13, P15, P19). Moreover, of those some noted that the values in the Projection Boxes were simply not useful to them (P13, P19). While the values in the Projection Boxes were used by many participants to synthesize single scopes, this is not a necessity: the choice to separate ScooPy into its own liveness layer in this way means that example scopes and synthesis with them can be used on their own, or in synthesis tools other than LooPy.

Preserving or prioritizing existing code. There was a specific tension in how users viewed synthesis with example scopes: on the one hand, they expected incorrect code to be fixed by the synthesizer. On the other hand, they wanted the synthesizer to take into account code that exists inside the scope, e.g., not replace an existing condition if it can be used in the solution, and generally consider user-provided code. While previous work on this in the context of synthesis-driven program repair [35] was only moderately successful, and mainly led to weird results, it is worth exploring given new results in probabilistic grammars and the different domain.

Provenance and nesting of LLM-generated code. Though large language models drive many available coding assistants, we chose not to compare ScooPy against these assistants directly. The initial problem that ScooPy tackles—the provenance of code—is non-trivial with an LLM to an extent that it constitutes a disparate and interesting research direction: how can provenance be stored when it should contain a huge prompt? And how should the non-determinism of LLM results factor into such documentation? On the other hand, simply comparing to developing with, e.g., Copilot, would not tell us much about the value of solutions to either problem.

Moreover, our results in Section 7.3 show once nesting has formed, LLMs may not be the right tools to tackle it: querying a model took longer, returned worse results for PBE tasks, and required many iterations and corrections to both generate correct code and label it for the programmer.

Limitations. The external validity of our studies is limited by our choice of participants. While Tahaei and Vaniea [39] found CS students to be a cost-effective population for programming studies, we tried to further mitigate the threat by recruiting participants with Python experience and, in our second study, adding a programming pretest. We also acknowledge the results of the second study are limited to tasks that are entirely within the scope of the tested synthesizers. A more open-ended study where the tool only supports some of users’ calls to the synthesizer would have been more ecologically valid, but would have generated less

data about ScooPy’s novel features over the course of a single session.

9 Conclusion

We presented ScooPy, a synthesis tool that records its specifications in *example scopes* and leverages these scopes to give programmers live feedback. Because nesting of example scopes can easily occur, ScooPy also supports synthesizing with nested example scopes without losing the information in the inner scopes. Our user studies demonstrate that example scopes increase users’ engagement with the code, that ScooPy is effective in improving success with synthesis in some tasks, and that example scopes and live feedback from them are helpful and favored by programmers.

Acknowledgments

We would like to thank Elena L. Glassman, who was instrumental in every step of this project, and the participants of our user studies. This work is funded by the European Union (EXPLOSYN, 101117232). Views and opinions expressed are however those of the authors only and do not necessarily reflect those of the European Union or the European Research Council Executive Agency. Neither the European Union nor the granting authority can be held responsible for them.

References

- [1] [n.d.]. GitHub Copilot, your AI pair programmer. <https://copilot.github.com/>.
- [2] Aws Albarghouthi, Sumit Gulwani, and Zachary Kincaid. 2013. Recursive Program Synthesis. In *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 8044)*, Natasha Sharygina and Helmut Veith (Eds.). Springer, 934–950. doi:10.1007/978-3-642-39799-8_67
- [3] Rajeev Alur, Dana Fisman, Rishabh Singh, and Armando Solar-Lezama. 2016. SyGuS-Comp 2016: Results and Analysis. 229 (2016), 178–202. doi:10.4204/EPTCS.229.13
- [4] Shraddha Barke, Hila Peleg, and Nadia Polikarpova. 2020. Just-in-time learning for bottom-up enumerative synthesis. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 227:1–227:29. doi:10.1145/3428295
- [5] José Cambronero, Sumit Gulwani, Vu Le, Daniel Perelman, Arjun Radhakrishna, Clint Simon, and Ashish Tiwari. 2023. FlashFill++: Scaling Programming by Example by Cutting to the Chase. *Proc. ACM Program. Lang.* 7, POPL (2023), 952–981. doi:10.1145/3571226
- [6] 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 (Berlin, Germany) (UIST '18)*. Association for Computing Machinery, New York, NY, USA, 963–975. doi:10.1145/3242587.3242661
- [7] Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. 2024. Teaching Large Language Models to Self-Debug. (2024). <https://openreview.net/forum?id=KuPixlqPiq>
- [8] Robert Corbett. 2022. *GNU Bison (software)*. <https://git.savannah.gnu.org/cgit/bison.git>
- [9] Françoise Détienné. 2001. *Software design cognitive aspects*. Springer. <http://www.springer.com/computer/swe/book/978-1-85233-253-2>
- [10] Yangruibo Ding, Marcus J. Min, Gail E. Kaiser, and Baishakhi Ray. 2024. CYCLE: Learning to Self-Refine the Code Generation. *Proc. ACM Program. Lang.* 8, OOPSLA1 (2024), 392–418. doi:10.1145/3649825

- [11] 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* (Honolulu, HI, USA) (CHI '20). Association for Computing Machinery, New York, NY, USA, 1–12. doi:10.1145/3313831.3376442
- [12] Justin Du, Mandeep Syal, and Thanh-Nha Tran. 2022. PBUUnit: A Live Programming Environment for Unit Testing. In *Programming Language Design and Implementation (PLDI) Student Research Competition*. Poster.
- [13] Kasra Ferdowsifard, Shraddha Barke, Hila Peleg, Sorin Lerner, and Nadia Polikarpova. 2021. LooPy: Interactive Program Synthesis with Control Structures. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 153 (Oct. 2021), 29 pages. doi:10.1145/3485530
- [14] Kasra Ferdowsifard, Allen Ordoookhanians, Hila Peleg, Sorin Lerner, and Nadia Polikarpova. 2020. Small-Step Live Programming by Example. In *UIST '20: The 33rd Annual ACM Symposium on User Interface Software and Technology, Virtual Event, USA, October 20-23, 2020*, Shamsi T. Iqbal, Karon E. MacLean, Fanny Chevalier, and Stefanie Mueller (Eds.). ACM, 614–626. doi:10.1145/3379337.3415869
- [15] Krzysztof Z. Gajos and Lena Mamykina. 2022. Do People Engage Cognitively with AI? Impact of AI Assistance on Incidental Learning. In *IUI 2022: 27th International Conference on Intelligent User Interfaces, Helsinki, Finland, March 22 - 25, 2022*, Giulio Jacucci, Samuel Kaski, Cristina Conati, Simone Stumpf, Tuukka Ruotsalo, and Krzysztof Gajos (Eds.). ACM, 794–806. doi:10.1145/3490099.3511138
- [16] Joel Galenson, Philip Reames, Rastislav Bodik, Björn Hartmann, and Koushik Sen. 2014. CodeHint: dynamic and interactive synthesis of code snippets. In *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*, Pankaj Jalote, Lionel C. Briand, and André van der Hoek (Eds.). ACM, 653–663. doi:10.1145/2568225.2568250
- [17] Google. 2023. *Protocol Buffers (Protobuf)*. <https://github.com/protocolbuffers/protobuf>
- [18] Sumit Gulwani. 2011. Automating String Processing in Spreadsheets Using Input-output Examples. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Austin, Texas, USA) (POPL '11). ACM, New York, NY, USA, 317–330. doi:10.1145/1926385.1926423
- [19] Dhanya Jayagopal, Justin Lubin, and Sarah E. Chasins. 2022. Exploring the Learnability of Program Synthesizers by Novice Programmers. In *Proceedings of the 35th Annual ACM Symposium on User Interface Software and Technology* (Bend, OR, USA) (UIST '22). Association for Computing Machinery, New York, NY, USA, Article 64, 15 pages. doi:10.1145/3526113.3545659
- [20] Susmit Jha, Sumit Gulwani, Sanjit A. Seshia, and Ashish Tiwari. 2010. Oracle-guided component-based program synthesis. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE 2010, Cape Town, South Africa, 1-8 May 2010*, Jeff Kramer, Judith Bishop, Premkumar T. Devanbu, and Sebastián Uchitel (Eds.). ACM, 215–224. doi:10.1145/1806799.1806833
- [21] Sean Kandel, Andreas Paepcke, Joseph Hellerstein, and Jeffrey Heer. 2011. Wrangler: interactive visual specification of data transformation scripts. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Vancouver, BC, Canada) (CHI '11). Association for Computing Machinery, New York, NY, USA, 3363–3372. doi:10.1145/1978942.1979444
- [22] Woosuk Lee. 2021. Combining the Top-down Propagation and Bottom-up Enumeration for Inductive Program Synthesis. *Proc. ACM Program. Lang.* 5, POPL, Article 54 (jan 2021), 28 pages. doi:10.1145/3434335
- [23] Sorin Lerner. 2020. Focused Live Programming with Loop Seeds. In *UIST '20: The 33rd Annual ACM Symposium on User Interface Software and Technology, Virtual Event, USA, October 20-23, 2020*, Shamsi T. Iqbal, Karon E. MacLean, Fanny Chevalier, and Stefanie Mueller (Eds.). ACM, 607–613. doi:10.1145/3379337.3415834
- [24] Sorin Lerner. 2020. Projection Boxes: On-the-fly Reconfigurable Visualization for Live Programming. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems* (Honolulu, HI, USA) (CHI '20). Association for Computing Machinery, New York, NY, USA, 1–7. doi:10.1145/3313831.3376494
- [25] Henry Lieberman. 2001. *Your wish is my command: Programming by example*. Morgan Kaufmann.
- [26] Yu Liu, Pengyu Nie, Anna Guo, Milos Gligoric, and Owolabi Legunsen. 2023. Extracting Inline Tests from Unit Tests. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSSTA 2023, Seattle, WA, USA, July 17-21, 2023*, René Just and Gordon Fraser (Eds.). ACM, 1458–1470. doi:10.1145/3597926.3598149
- [27] Yu Liu, Pengyu Nie, Owolabi Legunsen, and Milos Gligoric. 2022. Inline Tests. In *37th IEEE/ACM International Conference on Automated Software Engineering, ASE 2022, Rochester, MI, USA, October 10-14, 2022*, ACM, 57:1–57:13. doi:10.1145/3551349.3556952
- [28] Justin Lubin and Sarah E. Chasins. 2021. How statically-typed functional programmers write code. *Proc. ACM Program. Lang.* 5, OOPSLA (2021), 1–30. doi:10.1145/3485532
- [29] Justin Lubin, Nick Collins, Cyrus Omar, and Ravi Chugh. 2020. Program sketching with live bidirectional evaluation. *Proc. ACM Program. Lang.* 4, ICFP (2020), 109:1–109:29. doi:10.1145/3408991
- [30] Mikael 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* (Charlotte, NC, USA) (UIST '15). Association for Computing Machinery, New York, NY, USA, 291–301. doi:10.1145/2807442.2807459
- [31] Microsoft. 2024. *Visual Studio Code*. <https://code.visualstudio.com/>
- [32] Anders Miltner, Adrian Trejo Nuñez, Ana Brendel, Swarat Chaudhuri, and Isil Dillig. 2022. Bottom-up Synthesis of Recursive Functional Programs Using Angelic Execution. *Proc. ACM Program. Lang.* 6, POPL, Article 21 (jan 2022), 29 pages. <https://doi.org/10.1145/3498682>
- [33] Cyrus Omar, YoungSeok Yoon, Thomas D. LaToza, and Brad A. Myers. 2012. Active code completion. In *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*, Martin Glinz, Gail C. Murphy, and Mauro Pezzè (Eds.). IEEE Computer Society, 859–869. doi:10.1109/ICSE.2012.6227133
- [34] Peter-Michael Osera and Steve Zdancewicz. 2015. Type-and-example-directed program synthesis. (2015), 619–630. doi:10.1145/2737924.2738007
- [35] Hila Peleg, Roi Gabai, Shachar Itzhaky, and Eran Yahav. 2020. Programming with a Read-Eval-Synth Loop. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (11 2020). doi:10.1145/3428227
- [36] Hila Peleg, Sharon Shoham, and Eran Yahav. 2018. Programming not only by example. In *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, Michel Chaudron, Ivica Crnkovic, Marsha Chechik, and Mark Harman (Eds.). ACM, 1114–1124. doi:10.1145/3180155.3180189
- [37] Mark Santolucito, William T. Hallahan, and Ruzica Piskac. 2019. Live Programming By Example. In *Extended Abstracts of the 2019 CHI Conference on Human Factors in Computing Systems, CHI 2019, Glasgow, Scotland, UK, May 04-09, 2019*, Regan L. Mandryk, Stephen A. Brewster, Mark Hancock, Geraldine Fitzpatrick, Anna L. Cox, Vassilis Kostakos, and Mark Perry (Eds.). ACM. doi:10.1145/3290607.3313266
- [38] Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems*, A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine (Eds.), Vol. 36. Curran Associates, Inc., 8634–8652. https://proceedings.neurips.cc/paper_files/paper/2023/file/1b44b878bb782e6954cd888628510e90-Paper-Conference.pdf

- [39] Mohammad Tahaei and Kami Vaniea. 2022. Recruiting Participants With Programming Skills: A Comparison of Four Crowdsourcing Platforms and a CS Student Mailing List. In *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems* (New Orleans, LA, USA) (CHI '22). Association for Computing Machinery, New York, NY, USA, Article 590, 15 pages. doi:10.1145/3491102.3501957
- [40] Hao Tang, Keya Hu, Jin Zhou, Sicheng Zhong, Wei-Long Zheng, Xujie Si, and Kevin Ellis. 2024. Code Repair with LLMs gives an Exploration-Exploitation Tradeoff. (2024). http://papers.nips.cc/paper_files/paper/2024/hash/d5c56ec4f69c9a473089b16000d3f8cd-Abstract-Conference.html
- [41] Hao Tang, Darren Key, and Kevin Ellis. 2024. World-Coder, a Model-Based LLM Agent: Building World Models by Writing Code and Interacting with the Environment. (2024). http://papers.nips.cc/paper_files/paper/2024/hash/820c61a0cd419163ccb2c33b268816e-Abstract-Conference.html
- [42] Steven L. Tanimoto. 2013. A perspective on the evolution of live programming. In *Proceedings of the 1st International Workshop on Live Programming, LIVE 2013, San Francisco, California, USA, May 19, 2013*, Brian Burg, Adrian Kuhn, and Chris Parnin (Eds.). IEEE Computer Society, 31–34. doi:10.1109/LIVE.2013.6617346
- [43] Abhishek Udupa, Arun Raghavan, Jyotirmoy V. Deshmukh, Sela Mador-Haim, Milo M. K. Martin, and Rajeev Alur. 2013. TRAnsIT: specifying protocols with concolic snippets. (2013), 287–296. doi:10.1145/2491956.2462174
- [44] Priyan Vaithilingam and Philip J. Guo. 2019. Bespoke: Interactively Synthesizing Custom GUIs from Command-Line Applications By Demonstration. In *Proceedings of the 32nd Annual ACM Symposium on User Interface Software and Technology* (New Orleans, LA, USA) (UIST '19). Association for Computing Machinery, New York, NY, USA, 563–576. doi:10.1145/3332165.3347944
- [45] Priyan Vaithilingam, Tianyi Zhang, and Elena L. Glassman. 2022. Expectation vs. Experience: Evaluating the Usability of Code Generation Tools Powered by Large Language Models. In *Extended Abstracts of the 2022 CHI Conference on Human Factors in Computing Systems* (New Orleans, LA, USA) (CHI EA '22). Association for Computing Machinery, New York, NY, USA, Article 332, 7 pages. doi:10.1145/3491101.3519665
- [46] 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* (Chicago, Illinois, USA) (SIGMOD '17). Association for Computing Machinery, New York, NY, USA, 1631–1634. doi:10.1145/3035918.3058738
- [47] Chenglong Wang, Yu Feng, Rastislav Bodik, Isil Dillig, Alvin Cheung, and Amy J Ko. 2021. Falx: Synthesis-Powered Visualization Authoring. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems* (Yokohama, Japan) (CHI '21). Association for Computing Machinery, New York, NY, USA, Article 106, 15 pages. doi:10.1145/3411764.3445249
- [48] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. 2022. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. In *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*, Sanmi Koyejo, S. Mohamed, A. Agarwal, Danielle Belgrave, K. Cho, and A. Oh (Eds.). http://papers.nips.cc/paper_files/paper/2022/hash/9d5609613524ecf4f15af0f7b31abca4-Abstract-Conference.html
- [49] Kuat Yessenov, Shubham Tulsiani, Aditya Menon, Robert C. Miller, Sumit Gulwani, Butler Lampson, and Adam Kalai. 2013. A colorful approach to text processing by example. In *Proceedings of the 26th Annual ACM Symposium on User Interface Software and Technology* (St. Andrews, Scotland, United Kingdom) (UIST '13). Association for Computing Machinery, New York, NY, USA, 495–504. doi:10.1145/2501988.2502040
- [50] Yongwei Yuan, Arjun Radhakrishna, and Roopsha Samanta. 2023. Trace-Guided Inductive Synthesis of Recursive Functional Programs. *Proc. ACM Program. Lang.* 7, PLDI, Article 141 (jun 2023), 24 pages. doi:10.1145/3591255
- [51] Tianyi Zhang, Zhiyang Chen, Yuanli Zhu, Priyan Vaithilingam, Xinyu Wang, and Elena L. Glassman. 2021. Interpretable Program Synthesis. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems* (Yokohama, Japan) (CHI '21). Association for Computing Machinery, New York, NY, USA, Article 105, 16 pages. doi:10.1145/3411764.3445646
- [52] Tianyi Zhang, London Lowmanstone, Xinyu Wang, and Elena L. Glassman. 2020. Interactive Program Synthesis by Augmented Examples. In *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology* (Virtual Event, USA) (UIST '20). Association for Computing Machinery, New York, NY, USA, 627–648. doi:10.1145/3379337.3415900
- [53] Xiangyu Zhou, Rastislav Bodik, Alvin Cheung, and Chenglong Wang. 2022. Synthesizing analytical SQL queries from computation demonstration. In *PLDI '22: 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, San Diego, CA, USA, June 13 - 17, 2022*, Ranjit Jhala and Isil Dillig (Eds.). ACM, 168–182. doi:10.1145/3519939.3523712

A The ScooPy Synthesis Algorithm

While Section 4.2 gives a high-level description of the ScooPy synthesizer's algorithm, in this section we describe it more formally. We provide two equivalent descriptions of the algorithm, one using inference rules and one in pseudocode.

ScooPy in inference rules. ScooPy is *syntax-guided*. Its grammar, defined in Figure 11, expands Python's syntax to include example scopes that can wrap constructs in the target language of the synthesizer. The *scopeable* nonterminal also describes the target language of the ScooPy synthesizer, as every synthesis result will be a *scope*.

ScooPy's algorithm, then applies a set of inference rules bottom-up until the top-level scope where the programmer called the synthesizer. Figure 12 shows the inference rules, whose operations are described in Section 4.2: each rule is applied to a production with child nodes that have already computed a triple of information $\langle \hat{\mathcal{E}}, T, \mathcal{R} \rangle$, where $\hat{\mathcal{E}}$ are examples that remain to be synthesized, T are example pairs synthesis must consider together when branching, and \mathcal{R} is code to retain, and computes such a triple for the current node. The first four rules, SCOPE, ASSIGN, COND, and SEQ handle tree nodes with *scopeable* children, whereas the last two, TO-SCOPEABLE and TO-SCOPEABLE2 turn an example scope into a scopeable to allow it to compose with statements.

The SEQ and TO-SCOPEABLE rules employ *synthesize*, a call to the base flat synthesizer to generate one snippet satisfying a triple $\langle \hat{\mathcal{E}}, T, \mathcal{R} \rangle$.

Creating scopeables. There is a choice between two ways to turn a scope into a scopeable: if the scope is directly under a conditional, TO-SCOPEABLE2, which is essentially a *nop*, is used so that the COND rule can make use of the scope's examples. Otherwise, TO-SCOPEABLE is applied, calling the flat synthesizer.

The top-most example scope will be reached which would call the SCOPE rule, yielding a triple $\langle \hat{\mathcal{E}}, T, \mathcal{R} \rangle$ representing the final step to be taken. *synthesize*($\langle \hat{\mathcal{E}}, T, \mathcal{R} \rangle$) is then called a final time, and the result is returned.

ScooPy in pseudocode. An equivalent description of ScooPy's algorithm is shown in pseudocode in Algorithm 1: a recursive traversal of the AST that uses the same triple of information from child nodes—examples, pairs of examples to consider together, and code to retain—until the top-level is reached. The algorithmic formulation shows more clearly that a sequence of statements that interleaves example scopes is the only non-top-level case where the flat synthesizer is called.

```

scopeable :=  $\mathcal{E}(\text{scopeable})$ 
scopeable := assign #ASSIGN rule
           | if cond: scopeable else: scopeable #COND rule
           | scopeable; ...; scopeable #SEQ rule
           | scope #TO-SCOPEABLE rule
stmt := scope
      | ... #All Python statements

```

Figure 11. The Python language grammar extended with example scopes. $\mathcal{E}(\dots)$ describes code wrapped in an example scope with example set \mathcal{E} . The three statements in the *scopeable* nonterminal are the target language of the original LooPy synthesizer.

$$\begin{array}{c}
\frac{s : \langle \hat{\mathcal{E}}, T, \mathcal{R} \rangle \quad \mathcal{E} = \{i \rightarrow o\} \quad \hat{\mathcal{E}}' = \mathcal{E} \cup \hat{\mathcal{E}}}{\mathcal{E}(s) : \langle \hat{\mathcal{E}}', T, \mathcal{R} \rangle} \text{SCOPE} \\
\\
\frac{}{x = e : \langle \emptyset, \emptyset, \emptyset \rangle} \text{ASSIGN} \\
\\
\frac{s_1 : \langle \hat{\mathcal{E}}_1, T_1, \mathcal{R}_1 \rangle \quad s_2 : \langle \hat{\mathcal{E}}_2, T_2, \mathcal{R}_2 \rangle \quad \hat{\mathcal{E}} = \hat{\mathcal{E}}_1 \cup \hat{\mathcal{E}}_2 \quad \mathcal{R} = \mathcal{R}_1 \cup \mathcal{R}_2 \quad T = \hat{\mathcal{E}}_1 \times \hat{\mathcal{E}}_1 \cup \hat{\mathcal{E}}_2 \times \hat{\mathcal{E}}_2}{\text{if cond: } s_1 \text{ else: } s_2 : \langle \hat{\mathcal{E}}, T, \mathcal{R} \rangle} \text{COND} \\
\\
\frac{s_1 : \langle \hat{\mathcal{E}}_1, T_1, \mathcal{R}_1 \rangle \quad \dots \quad s_n : \langle \hat{\mathcal{E}}_n, T_n, \mathcal{R}_n \rangle \quad \mathcal{R} = \bigcup \{ \text{synthesize}(\langle \hat{\mathcal{E}}_i, T_i, \mathcal{R}_i \rangle) \mid \mathcal{E}_i \neq \emptyset \}_i}{s_1; \dots; s_n : \langle \emptyset, \emptyset, \mathcal{R} \rangle} \text{SEQ} \\
\\
\frac{\mathcal{E}(s) : \langle \hat{\mathcal{E}}, T, \mathcal{R} \rangle \quad \mathcal{R}' = \text{synthesize}(\langle \hat{\mathcal{E}}, T, \mathcal{R} \rangle)}{\mathcal{E}(s) : \langle \emptyset, \emptyset, \mathcal{R}' \rangle} \text{TO-SCOPEABLE} \\
\\
\frac{\mathcal{E}(s) : \langle \hat{\mathcal{E}}, T, \mathcal{R} \rangle}{\mathcal{E}(s) : \langle \hat{\mathcal{E}}, T, \mathcal{R} \rangle} \text{TO-SCOPEABLE2}
\end{array}$$

Figure 12. Inference rules employed in ScooPy's syntax-guided approach. The result of processing each AST node will be a triple $\langle \hat{\mathcal{E}}, T, \mathcal{R} \rangle$ where \mathcal{E} are examples being scooped up, T are pairs of examples that need to be considered together when branching, and \mathcal{R} is code from previous synthesis calls to be retained.

Algorithm 1: The syntax-guided synthesis algorithm for ScooPy, using a flat synthesizer `synthesize`.

```

function Scoop(node: ASTNode): // Syntax-guided extraction of specifications
  node match :
    case  $\mathcal{E}$ (scopeable): // Scope node
       $\langle \hat{\mathcal{E}}, T, \mathcal{R} \rangle \leftarrow \text{Scoop}(\text{scopeable})$ 
      return  $\langle \hat{\mathcal{E}} \cup \mathcal{E}, T, \mathcal{R} \rangle$  // add example scope's examples to any specification inside
    end
    case assign: // Assign node
      return  $\langle \emptyset, \emptyset, \emptyset \rangle$ 
    end
    case if cond: scopeable1 else: scopeable2: // Cond node
       $\langle \hat{\mathcal{E}}_1, T_1, \mathcal{R}_1 \rangle \leftarrow \text{Scoop}(\text{scopeable}_1)$  // Scoop then specification
       $\langle \hat{\mathcal{E}}_2, T_2, \mathcal{R}_2 \rangle \leftarrow \text{Scoop}(\text{scopeable}_2)$  // Scoop else specification
      // Gather examples and code to retain from both sides
       $\hat{\mathcal{E}} \leftarrow \hat{\mathcal{E}}_1 \cup \hat{\mathcal{E}}_2$ 
       $\mathcal{R} \leftarrow \mathcal{R}_1 \cup \mathcal{R}_2$ 
      // Require examples from each branch be together in the result
       $T \leftarrow \hat{\mathcal{E}}_1 \times \hat{\mathcal{E}}_1 \cup \hat{\mathcal{E}}_2 \times \hat{\mathcal{E}}_2$ 
      return  $\langle \emptyset, \emptyset, \emptyset \rangle$ 
    end
    case scopeable1 ; ... ; scopeablen: // Sequence node
       $\mathcal{R} \leftarrow \emptyset$ 
      foreach scopeablei :
         $\langle \hat{\mathcal{E}}_i, T_i, \mathcal{R}_i \rangle \leftarrow \text{Scoop}(\text{scopeable}_i)$ 
        if  $\hat{\mathcal{E}}_i \neq \emptyset$  :
           $\text{res} \leftarrow \text{synthesize}(\langle \hat{\mathcal{E}}_i, T_i, \mathcal{R}_i \rangle)$  // Call flat synthesizer on current child
           $\mathcal{R} \leftarrow \mathcal{R} \cup \text{res}$  // retain resulting code
        end
      end
      return  $\langle \emptyset, \emptyset, \mathcal{R} \rangle$  // All examples were synthesized for children, only retained code is passed up
    end
  end
end

function synthesizeTopLevel(topLevel: Example scope): // Synthesize selected example scope
   $\langle \hat{\mathcal{E}}, T, \mathcal{R} \rangle \leftarrow \text{Scoop}(\text{topLevel})$  // Scoop handles everything but the top level scope
  return  $\text{synthesize}(\langle \hat{\mathcal{E}}, T, \mathcal{R} \rangle)$ 
end

```
