Abstract

Misconceptions about core linguistic concepts like mutable variables, mutable compound data, and their interaction with scope and higher-order functions seem to be widespread. But how do we detect them, given that experts have blind spots and may not realize the myriad ways in which students can misunderstand programs? Furthermore, once identified, what can we do to correct them?

In this paper, we present a curated list of misconceptions, and an instrument to detect them. These are distilled from student work over several years and match and extend prior research. We also present an automated, self-guided tutoring system. The tutor builds on strategies in the education literature and is explicitly designed around identifying and correcting misconceptions.

We have tested the tutor in multiple settings. Our data consistently show that (a) the misconceptions we tackle are widespread, and (b) the tutor appears to improve understanding.

¡ccs2012¿ ¡concept¿ ¡concept$_id >$ 10011007.10011006.10011008 $< /concept_id >< concept_desc >$ $Softwareanditle$

1 Introduction

A large number of widely used modern programming languages share a common semantic basis:

- lexical scope
- nested scope
- eager evaluation
- sequential evaluation (per “thread”)
- mutable first-*order* variables
- mutable first-*class* structures (objects, vectors, etc.)
- higher-order functions that close over bindings
- automated memory management (e.g., garbage collection)

This semantic core can be seen in languages from “object-oriented” languages like C# and Java, to “scripting” languages like JavaScript, Python, and Ruby, to “functional” languages like the ML and Lisp families. Of course, there are sometimes restrictions (e.g., Java has restrictions on closures) and extensions (such as the documented semantic oddities of JavaScript and Python [**bernhardtWat2012**, **guhaEssenceJavascript2010**, **politzPythonFullMonty2013**, **politzTestedSemanticsGett**
Still, this semantic core bridges many syntaxes, and understanding it helps when transferring knowledge from old languages to new ones. In recognition of this deep commonality, in this paper we choose to call this the *Standard Model of Languages* (SMoL).

Unfortunately, this combination of features appears to also be non-trivial for programmers to understand. Consider the following scenario: to create a calculator, we have to construct a callback that can be attached to each button. The following Python program seems to achieve the construction of these callbacks and the act of pushing the buttons in order:

```
button_list = []

for i in range(10):
    button = lambda: print(i)
    button_list.append(button)

for button in button_list:
    button()
```

That is, a user would expect to see `0` through `9`. In fact, however, it prints `9` ten times.

As the related work section (**??**) describes, multiple researchers, in different countries and different kinds of post-secondary educational contexts, have studied how students fare with scope and state. They consistently find that even advanced students have difficulty with such programs and even programs simpler than this.

In fact, these problems are not at all limited to students. This specific looping problem even trips up industrial programmers. The C# language *changed* to produce `0` through `9` [**lippertClosingLoopVariable2009**]. It is now also the focus of a language design change in Go [**chaseFixingLoopsGo2023**] for a new kind of looping construct [**coxSpecLessErrorprone2023**]. It continues to trip up programmers (e.g., [**sharveyCreatingFunctionsLambdas2022**]) despite being documented as a "gotcha" in Python [**reitzHitchhikerGuidePython2016**].

Most of the time, however, these misunderstandings do not lead to language design changes. Nor are changes necessarily desirable: the SMoL feature set has presumably evolved because it is convenient for writing non-trivial programs (e.g., mutable structures) without being too unwieldy (e.g., no dynamic scope). Furthermore, having a good mental model of these features is essential to understand ownership [**clarkeOwnershipTypesFlexible1998**], manage parallelism, and more. Thus, we still need to train students and other programmers on the semantic features *and their interactions* in SMoL.

In response, we have (a) run a multi-year, multi-phase study to identify these misunderstandings while trying to work around the blind spots of experts (and also drawing on prior work), and (b) constructed the SMoL Tutor, an interactive tutor to address these misunderstandings. The SMoL Tutor is not a *programming* tutor; rather, it assumes the user has some basic facility with programming (i.e., that they are already familiar with concepts like vectors, mutation, and higher-order functions). It is built around SMoL's features and their interactions, and draws on cognitive and educational psychology concepts to expressly identify and correct misconceptions.

Concretely, we make the following contributions:

1. We provide a collection of brief programs for detecting misconceptions about the features of SMoL and their interaction. These programs are short and most are readily translatable to a wide variety of programming languages that share the SMoL characteristics.

2. We show that multiple populations have problems with these programs.

3. We present a list of curated misconceptions generated after multiple rounds of analysis.

4. We implement these misconceptions as interpreters, which find weaknesses in our manual analysis and hold potential for generative use in the future.

5. We present a tutoring system that measurably corrects these misconceptions.

A Note on Terminology We use the term “behavior” to refer to the meaning of programs in terms of the answers they produce. A more standard term for this would, of course, be “semantics”. However, the term “semantics tutor” might mislead some readers into thinking it teaches people to read or write a formal semantics, e.g., an introduction to “Greek” notation. Because that is not the kind of tutor we are describing, to avoid confusion, we use the term “behavior” instead.

2 Background: Tutoring Systems and Pedagogic Techniques

There is an extensive body of literature on tutoring systems ([**vanlehnBehaviorTutoringSystems2006a**] is a quality survey), and indeed whole conferences are dedicated to them. We draw on this literature. In particular, it is common in the literature to talk about a “two-loop” architecture [**vanlehnBehaviorTutoringSystems2006a**] where the outer loop iterates through “tasks” (i.e., educational activities) and the inner loop iterates through UI events within a task. We follow the same structure in our tutor (**??**).

Many tutoring systems focus on teaching programming (such as the well-known and heavily studied LISP Tutor [**andersonLISPTutor1985**]), and in the process undoubtedly address some program behavior misconceptions. Our SMoL Tutor differs in a notable way: it does not try to teach programming per se. Instead, it assumes a basic programming background and focuses entirely on program behavior misconceptions and correcting them. We are not aware of a tutoring system (in computer science) that has this specific design.

The SMoL Tutor is firmly grounded in one technique from cognitive and educational psychology. The fundamental problem is: how do you fix a misconception? One approach is to only “present the right answer”, for fear that discussing the wrong conception might actually reinforce it. Instead, there is a body of literature starting from [**posnerTheoryConceptualChange1982**] that presents a theory of conceptual change, at whose heart is the *refutation text*. A refutation text tackles the misconception directly, discussing and providing a refutation for the incorrect idea. Several studies [**schroederRefutationTextFacilitates2022**] have shown their effectiveness in multiple other domains.

The SMoL Tutor’s content structure is also influenced by work on *case comparisons* (which draws analogies between examples). [**alfieriLearningCaseComparisons2013a**] suggests that asking (rather than not asking) students to find similarities between cases, and providing principles *after* the comparisons (rather than before or not at all), are associated with better learning outcomes.

3 The SMoL Language

SMoL is designed to capture common features of many modern languages. The syntax of SMoL is presented in **??**, where `t` stands for terms, `d` stands for definitions, `e` stands for expressions, `c` stands for constants (i.e., number, boolean, and string), and `x` and `f` are identifiers (variables). The last kind of expression (i.e., `(e e ...)`) is function application.

SMoL’s Lispy syntax is an artifact of the course (**??**) using Racket [**friedmanEssentialsProgrammingLanguages2**
krishnamurthiProgrammingLanguagesApplication2007]; however, it also proved to be pedagogically valuable. We have found the parentheses useful when discussing scope in conjunction with local-binding features like `let`. This avoids the various confusing “variable lifting” semantics found in languages like Python ([**politzPythonFullMonty2013**]; see also posts like

```
t ::= d
    | e
d ::= (defvar x e)
    | (deffun (f x ...) body)
e ::= c
    | x
    | (lambda (x ...) body)
    | (let ([x e] ...) body)
    | (begin e ... e)
    | (set! x e)
    | (if e e e)
    | (cond [e body] ... [else body])
    | (cond [e body] ...)
    | (e e ...)
body    ::= t ... e
program ::= t ...
```

Figure 1: The syntax of SMoL.

Operators	Meaning
`+ - * /`	Arithmetic Operators
`< > <= >=`	Number comparison
`mvec`	Create a (mutable) vector (a.k.a. array)
`vec-ref`	Look up a vector element
`vec-set!`	Replace a vector element
`vec-len`	Get the length of a vector
`mpair`	Create a 2-element (mutable) vector
`left right`	Look up the first/second element of a 2-element vector
`set-left! set-right!`	Replace the first/second element of a 2-element vector
`eq?`	Equality

Table 1: Primitive operators of SMoL.

[**froadieWhatScopeVariable2022**]), where the actual defined range of a variable is not apparent from the source code.

Nevertheless, most SMoL programs are easy to translate to other languages. We present machine-translated Python and JavaScript versions of our programs in `Translated_Programs.html` in the supplementary materials. Furthermore, we intend to make a multi-lingual tutor in the future (**??**).

The semantics of SMoL is as described in **??**. SMoL includes limited primitive operators (**??**) to work with numbers, strings, and vectors. It provides only one equality operator, which tests for exact equality between atomic values and for pointer equality between other values.

Students are given a working implementation of SMoL, built as a `#lang` language inside Racket [**felleisenProgrammableProgrammingLanguage2018**]. This language provides only the defined syntax and semantics of SMoL, with no (other) Racket features present. (As in Racket, arguments evaluate left-to-right, to give stateful programs an unambiguous semantics.) The implementation is available online [URL anonymized].

4 Paper Roadmap

This paper describes a four-year effort to obtain a quality instrument to measure misconceptions and to build a tutor to address them.

The first step was a two-year process to generate (a) programs that tend to trip up students and (b) incorrect responses to those programs (**??**), using a tool for the purpose (**??**). These were then curated (**??**) to produce an instrument that was tested for one more year (**??**). This underwent one more round of curation to produce the final instrument (**??**).

We then present the tutor (**??**). In particular, we present the list of misconceptions—grounded in student data—that are the focus of this project (**??**). We also evaluate the tutor's effectiveness at correcting them (**??**).

All this work is done with students in a "Principles of Programming Languages" class at a selective, private US university. The class has about 70–75 students per year. It is not required, so students take it by choice. Virtually all are computer science majors. Most are in their third or fourth year of tertiary education; about 10% are graduate students. All have had at least one semester of imperative programming, and most have significantly more experience with it. Most have had close to a semester of functional programming. The student work described here was required, but students were graded on effort, not correctness.

Naturally, we should wonder to what extent the demographic affects the results: we may simply be studying weaknesses at this institution! We therefore work with two other populations (**??**). In both, we find similar results.

5 Generating and Collating Problems

In **??**, we discuss several papers that have provided reports of student misconceptions with different fragments of SMoL. However, it is difficult to know how comprehensive these are. While some are unclear on the origin of their programs, they generally seem to be expert-generated.

The problem with expert-generated lists is that they can be quite incomplete. Education researchers have documented the phenomenon of the *expert blind spot* [**nathanExpertBlindSpot2001**]: experts simply do not conceive of many learner difficulties. Thus, we need methods to identify problems beyond what experts conceive.

Additionally, in this paper, we intentionally use the word mis*conceptions* rather than mis*take*. A mistake can happen for any reason (e.g., selecting the wrong answer from a menu). A misconception, however, implies a conceptual problem: the student has formed an incorrect concept in their head. For instance, they may think that mutable structures are copied on function calls, or that scope is resolved dynamically. This requires probing what they are thinking.

Finally, we are inspired by the significant body of education research on *concept inventories* [**hestenesForceConceptInventory1992**] (with a growing number for computer science, as a survey lists [**taylorComputerScienceConcept2014**]). In terms of mechanics, a concept inventory is just an instrument consisting of multiple-choice questions (MCQs), where each question has one correct answer and several wrong ones. However, the wrong ones are chosen with great care. Each one has been validated so that if a student picks it, we can quite unambiguously determine *what misconception the student has.* For instance, if the question is "What is `sqrt(4)`?", then `37` is probably an uninteresting wrong answer, but if people appear to confuse square-roots with squares, then `16` would be present as an answer.[1]

All these, however, add up to a somewhat challenging demand. We want to produce a list of questions (each one an MCQ) such that

1. we can get past the expert blind spot,
2. we have a sense of what misconceptions students have, and
3. we can generally associate wrong answers with specific misconceptions, approaching a concept inventory.

5.1 Generating Problems Using Quizius

Our main solution to the expert blind spot is to use the Quizius system [**saarinenHarnessingWisdomClasses2019**]. In contrast to the very heavyweight process (involving a lot of expert time) that is generally used to create a concept inventory, Quizius uses a lightweight, interactive approach to obtain fairly comparable data, which an expert can then shape into a quality instrument.

In Quizius, experts create a prompt; in our case, we asked students to create small but "interesting" programs using the SMoL language. Quizius shows this prompt to students and gathers their answers. Each student is then shown a set of programs created by other students and asked to predict (without running it) the value produced by the program.[2] Students are also asked to provide a rationale for why they think it will produce that output.

Quizius runs interactively during an assignment period. At each point, it needs to determine which previously authored program to show a student. It can either "exploit" a given program that already has responses or "explore" a new one. Quizius thus treats this as a multi-armed bandit problem [**katehakisMultiArmedBanditProblem1987**] and uses that to choose a program.

The output from Quizius is (a) a collection of programs; (b) for each program, a collection of predicted answers; and (c) for each answer, a rationale. Clustering the answers is easy (after ignoring some small syntactic differences). Thus, for each cluster, we obtain a set of rationales.

After running Quizius in the course (**??**), we took over as experts. Determining which is the right answer is easy. Where expert knowledge is useful is in *clustering the rationales.* If all the rationales for a wrong answer are fairly similar, this is strong evidence that there is a common misconception that generates it. If, however, there are multiple rationale clusters, that means the program is not discriminative enough to distinguish the misconceptions, and it needs to be further refined to tell them apart. Interestingly, even the correct answer needs to be analyzed, because sometimes correct answers do have incorrect rationales (again, suggesting the program needs refinement to discriminate correct conceptions from misconceptions).

Prior work using Quizius [**saarinenHarnessingWisdomClasses2019**] finds that students do author programs that the experts did not imagine. In our case, we seeded Quizius with programs from prior papers (**??**), which gives the first few students programs to respond to. However, we

[1]Concept inventories are thus useful in many settings. For instance, an educator can use them with clickers to get quick feedback from a class. If several students pick a specific wrong answer, the educator not only knows they are wrong, but also has a strong inkling of *precisely what* misconception that group has and can address it directly. We expect our instruments to be useful in the same way.

[2]In the course (**??**), students were given credit for using Quizius but not penalized for wrong answers, reducing their incentive to "cheat" by running programs. They were also told that doing so would diminish the value of their answers. Some students seemed to do so anyway, but most honored the directive.

found that Quizius significantly expanded the scope of our problems and misconceptions. In our final instrument, most programs were directly or indirectly inspired by the output of Quizius.

5.2 Collating Problems

While Quizius is very useful in principle, it also produced data that needed significant curation for the following reasons:

- A problem may have produced diverse outputs simply because it was written in a very confusing way. Such programs do not reveal any useful *behavior* misconceptions, and must therefore be filtered out. For instance:

  ```
  (defvar x 1)
  (defvar y 2)
  (defvar z 3)
  (deffun (sum a ...) (+ a ...))
  (sum x y z)
  ```

  A reader might think that `sum` takes variable arguments (so the program produces `6`), but in fact `...` is a single variable, so this produces an arity error.

- Some programs relied on (or stumbled upon) underspecified aspects of unimportant (in particular, non-standard) parts of SMoL, such as floating-point versus rational arithmetic.

- A problem may have produced diverse outputs simply because it is hard to parse or to (mentally) trace its execution. One example was a 17-line program with 6 similar-looking and -named functions. As another example:

  ```
  (defvar a (or (/ 1 (- 0.25 0.25)) (/ 1 0.0)))
  (defvar b (and (/ 1 (- 0.25 0.25)) (/ 1 0.0)))
  (defvar c (and (/ 1 0.0) (/ 1 (- 0.25 -0.25))))
  (defvar d (or (/ 1 0) (/ 1 (- 0.25 -0.25))))
  (and (or a c) (or b d))
  ```

  This program is not only confusing, it *also* tests interpretations of (a) exact versus inexact numbers and (b) truthy/falsiness, leading to significant (but not very useful) answer diversity.

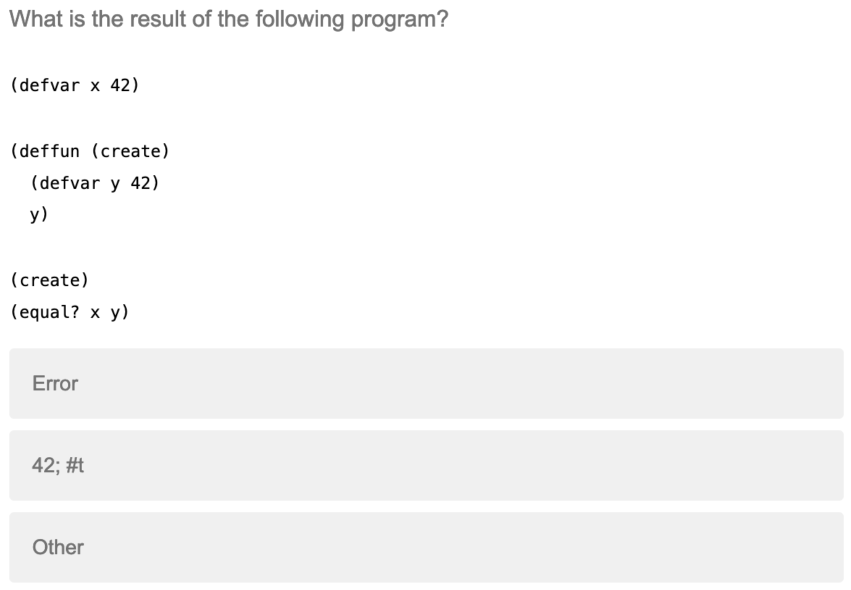
- As noted above, a program's wrong (or even correct) answers may correspond to multiple (mis)conceptions. In these cases, the program must be refined to be more discriminative.

and so on. We therefore manually curated the Quizius output to address these issues.

5.3 The SMoL Quizzes

Having curated the output, we had to confirm that these programs were still effective! That is, they needed to actually find student errors.

We therefore turned the programs into a set of quizzes (in the US sense: namely, a brief test of knowledge) that we call the SMoL Quizzes. There were three quizzes, ordered by linguistic complexity. The first consisted of only basic operators and first-order functions. The second added variable and structure mutation. The third added `lambda` and higher-order functions.



What is the result of the following program?

```
(defvar x 42)

(deffun (create)
  (defvar y 42)
  y)

(create)
(equal? x y)
```

Error

42; #t

Other

Figure 2: Screenshot of a SMoL Quiz question.

The goal of the SMoL Quizzes was to confirm that the aforementioned processes of cleansing and enriching the problems was successful. The quizzes were therefore administered in the third year of this project in the same course. **??** shows a sample question. Every question got an "Other" option. If chosen, the quiz gave the user a text box with the caption "Please specify". The goal here was to record any other answers, which in turn might lead to fresh misconceptions.

Question orders were partially randomized. We wanted students to get some easy, warm-up questions initially, so those were kept at the beginning. Similarly, we wanted programs that are syntactically similar to stay close to each other in the quiz. This is so that, when students got a second such program, they would not have to look far to find the first one and confirm that they are indeed (slightly) different, rather than wonder if they were seeing the same program again.

Students only received feedback after having completed a whole quiz. At the end of each quiz, they received both summative feedback *and* a refutation text that explained every program that appeared in the quiz. Students were also encouraged to run the programs, but we have little reason to believe that they did (and certainly they asked few questions on the class forum about them).

Due to space limitations, we present the entire instrument in `SMoL Quizzes/instrument` in the supplementary materials. Here we focus on a few programs where student choices correspond to misconceptions identified earlier, thereby also showing that the curated programs are still effective. Syntactically, `#t` and `#f` are true and false, while `#(...)` is a vector. We use a * to indicate the correct answer (which also matches the implementation's output).

Questions	**Frequency Table of Answers**	
`(defvar x 0)`	†59%	`2 0`
`(defvar y x)`	*32%	`Error`
`(defvar x 2)`	4%	`Other: 2 2`
`x`	3%	"Nothing is printed"
`y`	2%	`0 0`
`(defvar x 42)`		
`(deffun (create)`	*65%	`Error`
`  (defvar y 42)`	†29%	`42 #t`
`  y)`	6%	Other
`(create)`		
`(eq? x y)`		
`(defvar x 5)`	*53%	`#(6 5) 5`
`(deffun (reassign var_name new_val)`	*20%	`Error`
`  (defvar var_name new_val)`	†11%	`#(6 6) 5`
`  (mpair var_name x))`	†11%	`#(6 6) 6`
`(reassign x 6)`	5%	"Nothing is printed" or Other
`x`		

Table 2: Some questions and student answers from Quiz 1. Each table row is a program and the (relative) frequencies of student answers. Answers that can be considered correct are marked with $*$. Wrong answers that we discuss are marked with †.

5.3.1 Quiz 1: Basic Operators and First-Order Functions

?? lists programs in Quiz 1 that we consider the most interesting. These data confirm the presence of scope-related misconceptions:

1. At least 59% of students incorrectly believe that a variable can be defined twice in one block.
2. 29% of students believe that a variable defined in a function will be available in the top-level (or global) environment after the function is called. That is, these students may have a dynamic scope misconception.
3. 22% (11% + 11%) of students believe variables themselves can be passed as arguments and redefined inside the function. They disagree on whether the redefinition persists after the function call.

5.3.2 Quiz 2: Adding Variable and Structure Mutation

?? lists interesting programs after the addition of state. These data suggest the students have aliasing-related misconceptions:

- Up to 50% of students think vectors are copied rather than aliased.
- 16% of students think trying to construct and print a self-referring vector would error. (This program is ambiguous: *constructing* works in most SMoL languages, but *printing* may well cause a problem. These identified ambiguities are addressed in **??**.)

Question	Frequency Table of Answers	
`(defvar x (mvec 2 3))`	†50%	`#(#(2 #(2 3)) #(2 3))`
`(set-right! x x)`	*29%	`x=#(x x)` or something similar. Both `(left x)` and `(right x)` are `x` itself.
`(set-left! x x)`	†16%	`Error`
`x`	5%	Other
`(defvar v (mvec 1 2 3 4))`	*62%	`#(#(100 2 3 4) #(100 2 3 4))`
`(defvar vv (mvec v v))`	†26%	`#(#(1 2 3 4) #(100 2 3 4))`
`(vec-set! (vec-ref vv 1) 0 100)`	6%	`Error`
`vv`	4%	`#(#(1 2 3 4) #(1 2 3 4))`
	2%	Other
`(defvar x (mvec 123))`	*59%	`#(10)`
`(let ([y x])`	†36%	`#(123)`
`  (vec-set! y 0 10))`	5%	Other
`x`		
`(defvar x 12)`		
`(deffun (f x)`	*65%	`12`
`  (set! x 0))`	†31%	`0`
`(f x)`	4%	Other
`x`		
`(defvar x 5)`		
`(deffun (set1 x y)`		
`  (set! x y))`	*59%	Other: `5 7`
`(deffun (set2 a y)`	†27%	`6 7`
`  (set! x y))`	11%	`5 5`
`(set1 x 6)`	2%	Other: `Error`
`x`	1%	Other: `5 6`
`(set2 x 7)`		
`x`		

Table 3: Some questions and student answers from Quiz 2. Each table row is a program and the (relative) frequencies of student answers. Answers that can be considered correct are marked with *. Wrong answers that we discuss are marked with †.

- 31% of students think a variable is aliased by a parameter if the two variables have the same name. Perhaps interestingly, fewer students (27%) think the variable aliasing would happen if the two variables had different names.

5.3.3 Quiz 3: Adding Closures and Higher-Order Functions

?? lists interesting programs after the addition of closures and higher-order functions. It extends what we saw with loops in **??**: that students have misconceptions about their interaction with mutable variables:

- 17% of students think `lambda` functions can't refer to free variables.
- 21% of the students think mutating a variable defined outside a lambda can't possibly change

Question	Frequency Table of Answers	
`(deffun (f x)`	*82%	3
`  (lambda (y) (+ x y)))`	†17%	`Error`
`((f 2) 1)`	1%	Other
`(defvar x 1)`		
`(defvar f`		
`  (lambda (y)`	*74%	4
`    (+ x y)))`	†21%	3
`(set! x 2)`	6%	Other: `Error`
`(f x)`		
`(defvar x 1)`		
`(deffun (f y)`	*88%	4
`  (+ x y))`	†10%	3
`(set! x 2)`	3%	Other: `Error`
`(f x)`		
`(deffun (make-counter)`		
`  (let ([count 0])`		
`    (lambda ()`		
`      (begin`		
`        (set! count (+ count 1))`		
`        count))))`	*62%	`1 1 2 3 2`
`(defvar f (make-counter))`	†34%	`1 1 1 1 1`
`(defvar g (make-counter))`	4%	Other
	1%	`1 1 2 3 4`
`(f)`		
`(g)`		
`(f)`		
`(f)`		
`(g)`		

Table 4: Some questions and student answers from Quiz 3. Each table row is a program and the (relative) frequencies of student answers. Answers that can be considered correct are marked with `*`. Wrong answers that we discuss are marked with †.

the behavior of the lambda. Perhaps interestingly, this misconception seems to depend on how the lambda is constructed (compare the middle two programs).

- Student understanding of the `let`-over-`lambda` pattern is weak.

6 The SMoL Tutor

So far, we have focused on identifying problems. As noted earlier (**??**), we provided students with refutation texts after the quizzes, but it is unclear to what extent students read, understood, or internalized these. We also wanted to determine whether other populations run into these issues, and it was unclear whether we would be able to administer passive quizzes to them.

Furthermore, while the quizzes may be a useful diagnostic, our goal is not only to find faults but to improve the understanding of basic programming language behavior. We believe (and presumably so does anyone else who writes a formal semantics!) that an understanding of these basic program behaviors is important, and even more so when it comes to understanding concurrency, ownership, and other advanced features. Even more simply, misunderstanding these clearly impacts development and debugging time.

In response, we decided to create a *tutor*: the SMoL Tutor. It is built around our quiz instruments and the detected misconceptions. We describe the Tutor from a user's perspective in **??**, and explain how it was populated from the SMoL Quizzes in **??**. We then discuss what we learned from its data (**??**), and finally evaluate its effectiveness as a *tutor* (**??**).

6.1 The User Experience

The Tutor covers five major topics, shown on the left in **??**. The larger topics are further broken down into 2–3 modules. The goal was for students to spend at most 20–30 minutes per module. Our data show that in practice, students spent about 9.8 (median) minutes.

Each tutorial consists of a sequence of questions. Most questions in the Tutor are **interpreting questions**.[3] These questions (illustrated in **??**) are versions of the SMoL Quizzes (obtained by the process described in **??**). In each of these questions, students are shown a program and asked to predict the program's running result(s) by answering an MCQ (with an "Other" option). After making a choice, students receive feedback. If a student chooses incorrectly, they are (a) given an explanation that is *based on the misconception associated with that wrong answer* (or a generic one, if there is not a specific misconception), and then (b) asked to answer a second question:

1. The second question is semantically the same as the first, but with superficial changes (e.g., variable names, constants, and operators are changed) so that the student cannot immediately guess the answer.

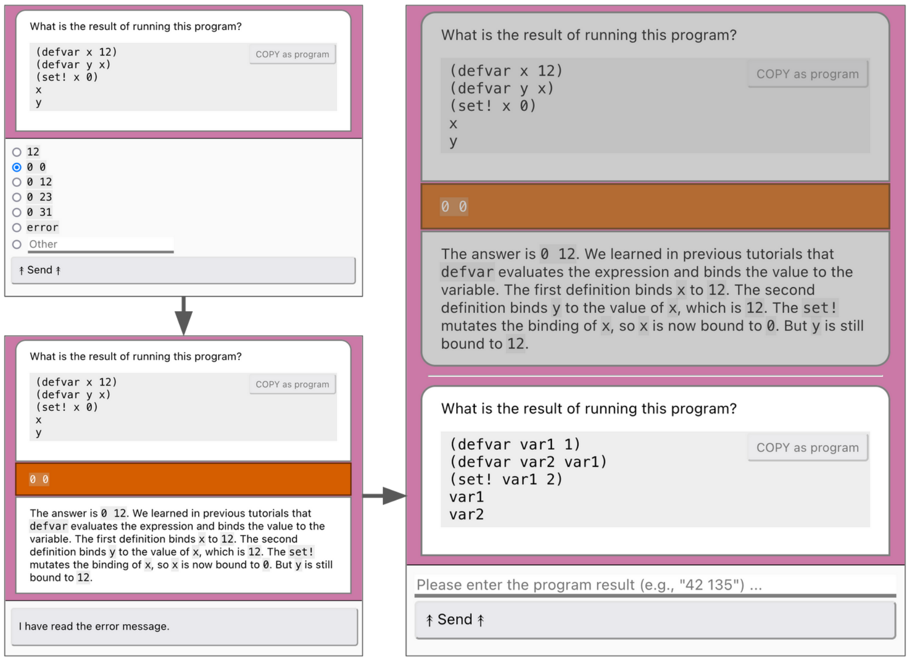
2. Instead of multiple-choice, students must *type* the answer into a text box. (The Tutor normalizes text to accommodate variations.) This is intentional. First, we wanted to force reflection on the explanation just given, whereas with an MCQ, students could just guess. Second, we felt that students would find typing more onerous than clicking. In case students had just guessed on a question, our hope is that the penalty of having to type means, on *subsequent* tasks, they would be more likely to pause and reflect before choosing.

In addition to asking students questions, the Tutor along the way introduces terminology and states the true conceptions. These are the teaching goals of the Tutor. We therefore refer to these as **goal sentences**. **??** lists the (abbreviated) goal sentences for each tutorial. Readers can find the full Tutor in `SMoL Tutor/instrument` in the supplementary materials.

Some later tutorials include questions about earlier tutorials so that we can check whether students remember the concepts across modules. In particular, the **mut-vars** tutorial starts with questions about **scope**, and **lambda** starts with questions about **mut-vars** and **vectors**.

Of the modules, **local** is the least portable across languages. While local binding is of course present in other languages, this is mostly covered using nested `defvar`s in earlier modules, starting with **scope**. The distinction central to this module (between three local-binding constructs with slightly different scopes) is primarily a focus of Lispy languages. Therefore, we exclude this module from our analysis and instruments, and indeed plan to deprecate the module in future versions.

[3]The Tutor also asked students to perform some other activities, like showing programs and asking for their heap content, which we do not cover in this paper.


What is the result of running this program?

```
(defvar x 12)
(defvar y x)
(set! x 0)
x
y
```

- 12
- [✓ 0 0]
- 0 12
- 0 23
- 0 31
- error
- Other

What is the result of running this program?

```
(defvar x 12)
(defvar y x)
(set! x 0)
x
y
```

```
0 0
```

The answer is `0 12`. We learned in previous tutorials that `defvar` evaluates the expression and binds the value to the variable. The first definition binds `x` to `12`. The second definition binds `y` to the value of `x`, which is `12`. The `set!` mutates the binding of `x`, so `x` is now bound to `0`. But `y` is still bound to `12`.

What is the result of running this program?

```
(defvar x 12)
(defvar y x)
(set! x 0)
x
y
```

```
0 0
```

The answer is `0 12`. We learned in previous tutorials that `defvar` evaluates the expression and binds the value to the variable. The first definition binds `x` to `12`. The second definition binds `y` to the value of `x`, which is `12`. The `set!` mutates the binding of `x`, so `x` is now bound to `0`. But `y` is still bound to `12`.

What is the result of running this program?

```
(defvar var1 1)
(defvar var2 var1)
(set! var1 2)
var1
var2
```

Please enter the program result (e.g., "42 135") ...

Figure 3: Screenshots of an interpreting question in SMoL Tutor. The top-left shows the initial state, where the question is presented as an MCQ. If a student chooses a wrong answer, they will receive feedback (bottom-left) and will be asked a similar question (right). The similar question must be answered by typing.

?? discusses the partial randomization employed by the SMoL Quizzes. In contrast, the SMoL Tutor does *not* randomize question order. This is because the Tutor consists of more than just questions: it also has explanatory text. This text is based on the preceding problems. Authoring it is therefore somewhat like writing a textbook, with complex dependencies that can easily be broken.

6.2 Collating Problems for the Tutor

The SMoL Tutor's interpreting questions are the final instrument of this paper. We bootstrap it from the SMoL Quizzes, but made the following alterations:

1. In the process of developing the goal sentences, we felt the existing programs were insufficient to cover the ideas we wanted students to work through. For instance, SMoL Quizzes did not alias vectors using `defvar`, so we added this program:

```
(defvar x (mvec 100))
(defvar y x)
(vec-set! x 0 200)
y
```

Tutorial	Goal Sentences
scope: Variable definitions and function definitions	1. Referring to an unbound variable leads to an error. 2. Define "blocks". 3. SMoL is lexically scoped rather than dynamically scoped. 4. SMoL disallows defining a variable twice in one block. 5. SMoL is eager (and evaluates from left to right) rather than lazy, reactive, or relational.
mut-vars: Variable updates	1. Variable assignment respects the hierarchical structure of blocks. 2. Variables do not alias.
begin: Sequencing expressions	A sequencing expression evaluates its sub-expressions from left to right and returns the value of the last sub-expression.
vectors: Vectors and vector updates	1. Define heap and memory addresses. 2. Vectors can be referred to by vectors, including themselves. 3. Vectors are not copied but aliased by bindings. 4. Constructing vectors doesn't alter environments. 5. Introducing bindings doesn't alter the (value part of) heap.
lambda: Lambda expressions	1. Functions are (first-class) values. 2. Functions remember the environment where they were created. 3. Functions can be created with lambda expressions. 4. A function definition can be viewed as a variable definition plus a lambda expression.
local: Local binding forms	Introduce `let`, `letrec`, and `let*`.

Table 5: SMoL Tutor tutorials and their goal sentences.

2. We renamed vector operators to avoid confusion between `vset!`, which replaces a vector element, and `set!`, which mutates variables. We rename all vector operators from, for example, `vset!` to `vec-set!`.

3. We deferred the local binding forms to the end, since we felt they were less essential. We therefore rewrote programs that use it to not depend on it.

4. We removed some programs that relied on underspecified aspects or unimportant aspects of SMoL. For example, some depended on whether operations on pairs could be applied to arbitrary vectors:

```
(pair? (mpair 1 2))
(pair? (mvec 1 2))
(pair? '#(1 2))
(pair? '(1 2))
```

As another example, one hinged on whether or not a function's formal parameter could have the same name as the function itself:

```
(deffun (f f) f)
(f 5)
```

5. We resolved ambiguities in some programs, either adding answer choices or even adding other questions to tease out different interpretations.

6. To reduce the number of concepts, we removed programs that relied upon *im*mutable vectors and lists, because they did not seem to create problems. (For brevity, we leave these out of the presentation of SMoL in **??**, though they are in the implementation.)

7. We removed questions related to function equality, which was not a focus of the Tutor.[4]

8. We removed programs of a "Lispy" nature, such as one where the answer depended on whether the reader correctly understood this inequality:

```
    (filter (lambda (n) (> 3 n)) '(1 2 3 4 5))
```

   This checks whether `n` $<$ `3`, but some presumably vocalized it as "greater than `3`, `n`?".

Most of these steps either modify or elide programs. Two cases, filling gaps in light of the goal sentences and resolving ambiguities, introduce programs. Starting with the 37 programs in the SMoL Quizzes, we added 52 more programs to arrive at a total of 89.

In addition to generating this set of programs, we also modified the answer options. In addition to retaining the correct and incorrect answers from the SMoL Quizzes (and constructing our best guess of analogous incorrect answers for the new problems), we added more wrong answers.

The reason is as follows. The SMoL Quizzes often have very few wrong choices: of the 37 tasks, 26 have only three choices (including the correct answer and the "Other" option[5]), seven have 4, and only four have 5 or 6. Thus, in most cases, students have a 25% or even 33% chance of just guessing the right answer or successfully using a process of elimination. By increasing the number of options, we hoped to greatly reduce the odds of getting the right answer by chance or by elimination.

It was important to add wrong answers that are not utterly implausible, because those would become easy to eliminate. Therefore, we added numeric constants mentioned in the problem, permuted some of the values in case of multiple outputs, and so on. In general, we ended up increasing the number of choices substantially: only 8 out of the 89 questions have three or four choices; 70 questions have 5–8 choices; and 11 questions have 9–14 choices. We hope this reduced both guessing and elimination, and forced students to actually think through the program. Of course, these new answers do not have a clear associated misconception, so their mistakes are given the generic explanation.

[4]However, we believe there is a good deal of confusion about notions of equality. We intend to add that as a major tutorial component in future versions.

[5]In a few cases, we removed the *correct* answer from the choices. A student would therefore have to click on "Other" and type it in. The idea was to make the "Other" option more salient and plausible.

7 Misconceptions Detected by the SMoL Tutor

We now examine what we learned from the interpreting questions in terms of program behavior misconceptions. But first, we explain how we associate incorrect answers with misconceptions.

We do not present the results of **begin** and **local** (**??**), because they focus on constructs not usually found in non-Lispy languages. They were included in the Tutor to help students write programs for the course, but are not interesting from a SMoL perspective.

7.1 Misinterpreters

In principle, an expert can identify what misconceptions a particular incorrect answer might correspond to. In practice, we found this rather difficult for three reasons. First, with 89 programs, each with several answers, it's easy to make mistakes. Second, our own expert blind spot may prevent us from seeing an interpretation that would lead to an additional association. Finally, and specific to our case, since we had either curated or written all the programs and answers, we were likely to miss some associations we had not intended.

To address this problem, we formalized misconceptions as interpreters. That is, for each misconception we created a corresponding *misinterpreter*. This is an interpreter for the SMoL syntax that intentionally has a semantic error corresponding to that misconception. Put differently, a misinterpreter is a like a "definitional interpreter for a misconception".

By running programs through the misinterpreters, we can more uniformly and rigorously identify *all* the misconceptions associated with a wrong answer. Furthermore, if we identify a new misconception (or alter a misinterpreter), it is easy to automatically re-classify all the answers. By using misinterpreters, we indeed found new interpretations for existing program-answer pairs. We provide all the misinterpreters in the artifact.

7.2 A Catalog of Misconceptions

We iteratively created our final catalog of misconceptions (from the perspective of the data in this paper). We started with misinterpreters representing the misconceptions described in **??**. These are misconceptions for which we have reasonable validation (due to the prose in Quizius), so we call them *grounded* misconceptions. We then looked at wrong answers not covered by these but chosen by students, and did our best to distill these into misconceptions. These are *surmised* misconceptions (which we identify with a ‡), which need to be validated in the future.[6] We then re-ran the misinterpreters against the chosen answers. We terminated when all the remaining wrong answers were either (a) found in very few students (we found a gap between 23% and 13%, and hence took 14% as the threshold), (b) difficult for us to attribute to a misconception, or (c) appeared to us to be "Lispy" and hence not of broad interest.

?????? list the final catalog. For each, we also present a Tutor question for which the marked wrong answer can be explained by *only* the named misconception. That is, that program-answer pair is a representative example of that misconception.[7]

[6] To keep the time spent in each module reasonable, the SMoL Tutor does not ask students for rationales for their programs. However, we could easily modify it to do so a small number of times per user, to make progress towards this need.

[7] An astute reader may well imagine another misinterpretation; but we presumably did not find evidence of it in our data. We do welcome other misinterpretations, for which we can add more misinterpreters!

Confirmed Misconceptions The Tutor confirmed all the misconceptions from Quizius via the SMoL Quizzes.

Added Misconceptions The misinterpreters helped us find misinterpretations that we had overlooked. For instance, consider this program from Quiz 2:

```
(defvar x 12)
(deffun (f x)
  (set! x 0))
(f x)
x
```

We had assumed the wrong answer `0` is caused by **CallByRef**. However, our misinterpreters made us realize it can also be explained by **FlatEnv**. This could also explain why we see a difference in error rate when the formal and actual parameter have the same name (as mentioned in **??**).

A New Potential Misconception For the following program, added in the Tutor:

```
  (defvar y (+ x 2))
  (defvar x 1)
  x
  y
```

56% of students asserted that it produces `1 3`. Based on this, we surmise that students might have another misconception, which we define as **Lazy**‡. (Recall that SMoL is eager, but even in many lazy languages, this would be an error.)

Another Potential Misconception, and Its Effect on Interpreting Descriptions. Consider this program from the SMoL Tutor:

```
(defvar x 1)
(deffun (main)
  (deffun (get-x) x)
  (defvar x 2)
  (get-x))

(main)
```

This program, suitably translated, would produce `2` in a wide variety of languages (Python, JavaScript, Racket, Java, etc.), because `get-x` and the inner `x` are in the same scope block. The answer `1` cannot be explained by any of our existing misconceptions. Based on this, we surmise a new misconception, **NestedDef**‡. (Though its frequency falls below our threshold, our reading of answers suggests this may be more widespread, and we feel it needs to be investigated more.)

Once we turned this into a misinterpreter, we found that it unexpectedly captured the program-answer pair for the following program from Quiz 1—which should be an error, due to the double-binding of `x` in the same scope block—and the answer `2 0`:

```
  (defvar x 0)
  (defvar y x)
  (defvar x 2)
  x
  y
```

Previously, we had interpreted this only as **DefOrSet**, because students had stated that the second `(defvar x ....)` "mutates" or "redefines" `x`. This is reminiscent of the behavior of languages like Python, which use the same syntax both for binding new variables and for mutating existing ones.

The problem here is that the word "redefine" underspecifies how the second definition is interpreted. We had interpreted it as *mutating* the binding established by the first definition, which fits **DefOrSet**. However, another possibility is that it *shadows* the binding (i.e., establishes a new scope block). We did not recognize that the original misconception is underspecified until we uncovered the new (surmised) misconception.

Summary To summarize, we used the SMoL Tutor, with an expanded set of programs, to investigate student misconceptions. We implemented the idea of misinterpreters to help us properly classify student performance. We were able to reconfirm all the previously identified misconceptions, refine some of them, and also identify potential new ones (that need further investigation).

8 Is The Tutor Effective?

Recall that the SMoL Tutor is not only a collection of MCQs: it is also a *tutor*! So far, we have investigated the value of the MCQs. Now we examine its tutoring aspect. Concretely, we ask:

RQ How effective is the Tutor at correcting each misconception?

To study this, we perform the following analysis per misconception. Using misinterpreters, we identify those questions whose wrong answers fit only one misconception (i.e., only one output matches that produced by that misinterpreter, and it matches only that misinterpreter). We then examine student performance over time across those questions. This would let us examine how they do on just that topic, in isolation, over time.

We present the result of this analysis in two forms. Graphically, we show plots in **??**. Each figure shows the percentage of students who chose the answer corresponding to that misconception. Ideally, we would like to see these percentages diminish.

Indeed, that is what we see in most of the graphs. The exceptions are **CallsCopyStructs**, **NestedDef**[‡], and **StructsCopyStructs**, which have only one problem (and hence no trend), and **DefOrSet** and **FunNotVal**, which show an increase. The lack of improvement for **FunNotVal** is unsurprising because the Tutor does not explicitly address this issue, focusing on closures created by `lambda`s rather than named functions. (However, this does not explain the increase!)

We also perform a logistic regression to see whether these improvements are significant (at a $p < 0.05$ threshold); details are in `Paper.html` in the supplementary materials. Of the 11:

- Of the nine seemingly improved (i.e., decreased) misconceptions:
  - Two are *not* significant: **CallByRef** (p-value = .074); **NoCircularity** (p-value = .075).
  - The other seven *are* significant.
- However, the two with an increasing trend (**DefOrSet** and **FunNotVal**) are *also* significant.

These data broadly suggest that the Tutor is a net positive. Ultimately, however, what these data really show is a need for improvement in the Tutor. When designing the Tutor questions, many more were intended to be representative of single misconceptions. However, as noted in **??**, it is easy to be incomplete (or incorrect) in ascribing misconceptions. Furthermore, as their set grows, it is difficult to reassess all the problems manually. Once evaluated using misinterpreters, we found far fewer problems than we would have liked. Concretely, only 40 of the 71 eligible problems (after removing the non-SMoL modules) were useful in the above analysis.

We therefore view the above data as purely formative: they suggest that the Tutor *most likely did not do harm* and perhaps even *may have done some good*. However, it would be improper to read too much into the analysis. It is quite possible that some of the other problems would have found issues. Rather, what we really see is value in the misinterpreter concept. It is not only useful for analysis, it is also valuable for *problem design*: in the next iteration of the Tutor, we will use misinterpreters actively to shape the incorrect answers and, as necessary, update the programs as well. We therefore hope to have much more thorough analyses in future work.

9 Performance on Other Populations

There is, of course, a significant danger that the data above have been overfitted to only one institution (**??**, henceforth University 1), thereby actually reflecting the state of its curriculum rather than some greater truth about program behavior understanding. We already have some reason to believe this is not the case: the related work discussed in **??** is drawn from many institutions in multiple countries with different educational preparations, levels, and demographics. Nevertheless, that gives us only limited information about the specific questions and misconceptions described above.

Fortunately, we were able to deploy the Tutor on two other populations:

- University 2 is a primarily public university in the US. It is one of the largest Hispanic-serving institutions in the country. As such, its demographic is extremely different from those whose data were used above. The Tutor was used in one course in Spring 2023, taken by 12 students. The course is a third-year, programming language course. The students are required to have taken two introductory programming courses (C++ focused).

- A separate instance of the Tutor was published on the website of a programming languages textbook [ANON]. Over the course of 8 months, 597 people started with the first module and 103 users made it to the last one. To protect privacy, we intentionally do not record demographic information, but we conjecture that the population is largely self-learners (who are known to use the accompanying book), including some professional programmers. It is extremely unlikely to be the students from either university, because they would not get credit for their work on the public instance; they needed to use the university-specific instance. Furthermore, since they were not penalized for wrong answers, it would make little sense to do a “test run” on the public instance. Finally, we note that there is no overlap between the dates of submission on the public instance and the semester at University 1.

These two populations are therefore at least somewhat different from the original population, and help us assess whether the problems we identify are merely an artifact of the first institution.

To evaluate, we computed a Spearman’s rank correlation ρ, ranking questions by what percentage of students got the question right. Between the original university and University 2, we obtain a p-value = 2.013e-07. Between the original university and the online population, we obtain a p-value ¡ 2.2e-16. These show that the other two populations performed similarly to the original one. While further validation on other populations remains essential, these suggest that the questions are *finding misconceptions that may be universal.*

10 Related Work

Misconceptions Misconceptions related to scope, mutation, and higher-order functions have been widely identified in varying populations (from CS1 students to graduate students to users of online forums) over many years (since at last 1991) in varying programming languages (from Java to Racket) and in different countries (such as the USA and Sweden). **??** lists works that seem most relevant to us.

There are some small differences. [**fleuryParameterPassingRules1991**] identifies a dynamic scope misconception that is different from **FlatEnv**:

CallerEnv‡ Function values don't remember their environments. When a function is called, the function body is evaluated in an environment that extends from the *caller's* environment.

We don't include **CallerEnv**‡ in our analysis because in our data, all wrong answers that can be explained by **CallerEnv**‡ are also explainable by **FlatEnv**.

Appendix A of [**sorvaVisualProgramSimulation2012**] provides an extensive survey of misconceptions reported in research up to 2012. There are overlaps between our survey and theirs. For instance, our **CallerEnv**‡ is their No. 47. However, because their descriptions are brief, it is difficult to tell whether a misconception in their survey matches our misconceptions. Because they provide neither misinterpreters nor representative program-output pairs, it is difficult to determine the overlaps precisely (showing the value of providing these two machine-runnable descriptions). At any rate, we certainly find no equivalent of **FlatEnv**, **DeepClosure**, and **DefByRef** in their survey.

Goal Sentences Our idea of "goal sentences" is not substantially different from the *rules of program behavior* from [**duranRulesProgramBehavior2021**]. We only used these sentences (or rules) to guide the design of the Tutor and as text in the Tutor itself. [**duranRulesProgramBehavior2021**] argue for other uses of such sentences.

Misinterpreters Our idea of misinterpreters is related to mystery languages [**diwanPLdetectiveSystemTeaching20**
pombrioTeachingProgrammingLanguages2017]. Both approaches use evaluators that represent alternative semantics to the same syntax. However, the two are complementary. In mystery languages, instructors design the space of semantics with pedagogic intent, and students must create programs to explore that space. Misinterpreters, in contrast, are driven by student input, while the programs are provided by instructors. The two approaches also have different goals: mystery languages focus on encouraging students to experiment with languages; misinterpreters aim at capturing students' misconceptions.

11 Threats to Validity

11.1 Construct Validity

Did we measure the right thing? While our goal is to study student understanding of language behavior, what we actually measure is performance on MCQs on select programs. MCQs as a mechanism introduce various clear biases, though we add the "Other" options to somewhat alleviate them. The small size, syntactic details (such as intentionally meaningless variable names), and other aspects of the programs may also impact our ability to measure understanding. (This could go both ways: students may have no trouble understanding behavior in a small program but may struggle to do so in a larger one.)

A particularly notable threat is the use of Lispy syntax. We chose it for two reasons: both because it helps clarify scope (**??**) and because the rest of the course used Racket. However, students may well perform differently with more familiar syntaxes. It seems unlikely their performance would be *too* different given the many languages that have produced similar misconceptions (**??**). Nevertheless, we intend to use a variety of syntaxes to examine this issue further.

Finally, we curated programs by hand, so they may well reflect our own biases about misconceptions. It may be possible to mitigate this problem by synthesizing MCQ programs using the misinterpreters.

11.2 Internal Validity

Is our reasoning valid? We have applied standard techniques and measurements for evaluating student responses. However, our instruments still lack the validity of a proper concept inventory. Their creation requires heavyweight processes (such as Delphi methods [**goldmanSettingScopeConcept2010**]) that require many hours of expert attention as well as conversations with learners, and can hence be prohibitive in cost. The Quizius method (**??**) was created precisely to be an inexpensive method that provides a good proxy.

Ultimately, our goal is to provide a reasonable instrument for widespread use. While the set of all misconceptions could be unbounded, we believe our tasks, especially as embedded in the Tutor, provide a good starting point for others. In particular, if students select a wrong answer, that is still of *some* use to an educator, even if the precise misconception cannot be pinned down with the highest accuracy. We therefore believe our instruments, and our misinterpreter technique, are of general value.

A failure in our Tutor's logging infrastructure led us to miss some responses. These are unlikely to be task-specific because the Tutor has a generic framework that should perform the same across tasks. Moreover, on average only 0.4% (sd = 0.6%) of values are missing. Therefore, we do not believe this had a noticeable impact. (In addition, every wrong answer is still wrong! We may just not have exactly the right proportions of them.)

11.3 External Validity

How well do our results generalize? Certainly there is reason to question whether our results would apply to other populations. **??** provides preliminary evidence that the results are not specific to one institution, and **??** suggests these issues are widespread. Nevertheless, much more broad testing is needed to confirm our specific instruments.

The other major concern is the tie to Lispy syntax. Learners in other settings may do worse or even better with it. Building a Tutor that supports multiple, and more traditional, syntaxes should help address this issue. We defer this to future work.

12 Discussion

The paper has already identified several areas for future work:

- testing on more varied populations;
- using other syntaxes;
- having more questions that uniquely identify minsconceptions; and,
- enabling textual responses in the Tutor so we can better characterize wrong answers.

These can help get us even closer to a good approximation of a true concept inventory. We would also like the Tutor to make more use of the education theories discussed in **??**.

We focus here on some issues that we think warrant broader discussion.

The State-Aliasing-Function Triangle Our problematic programs hardly include any "advanced" programming features: there are no threads, asynchrony, sophisticated type systems, ownership, inversion of control, etc. Indeed, many of those features typically build on an understanding of these basics (e.g., it is hard to make sense of ownership [**clarkeOwnershipTypesFlexible1998**] without a good understanding of this triangle). But many populations seem to struggle even with this, which may explain why concepts like ownership are considered hard [**crichtonGroundedConceptualModel2023**].

It is worth noting that we find problems even without all three components, as **??** shows! Nevertheless, we think it would be useful for curricula to focus on achieving mastery of this triangle. This may require a deep revision of widely accepted pedagogy. For instance, it is common to explain variables as "boxes". But if taken seriously by a learner, this metaphor may cause more harm than good. As prior research [**putnamSummaryMisconceptionsHigh1986**, **groverMeasuringStudentLearning2017**, **hermansThinkingOutBox2018**] shows, students expect that a variable can then contain more than one value, removing one makes the other accessible, etc. That research has not explored aliasing, but the metaphor may affect that too. A box is a closed object, so a (mutable) value put "into" it clearly can't be modified by another "box" (variable)—i.e., it not only doesn't explain but is antithetical to aliasing.

Terminology It is common to teach programming using the terms "call-by-value" and "call-by-reference". The reader will note that we instead have *three* **ByRef** misconceptions.

The emphasis on "calling" suggests that the semantics is associated *only* with calls. That leaves open what happens when one simply binds a variable. In a reasonable language (and definitionally, in SMoL), the behavior is exactly the same: the formal parameter can be viewed as a binding to the value of the actual. But students often form inconsistent views because the "call" terminology breaks this deep similarity. We therefore recommend that languages in general use the term *bind-by-*, to emphasize the underlying semantic unity of these syntactically different mechanisms.

We feel that further confusion is caused by terms like "pass" and "return". We often vocalize the call `(f x)` as "passing `x` to `f`"; saying "passing *the value of* `x` to `f`" is a mouthful. But is it then surprising that students think `x` is being aliased? Similarly, consider a statement like `return y` (in Python syntax). Of course, semanticists understand that it is the *value* of `y`, not `y` itself, that is being returned. Nevertheless, it is not surprising if this also results in assumptions that `y` is either aliased or that it has escaped from the function (leading to an interpretation of dynamic scope). We believe there is a need for significant research to investigate these kinds of effects, which are similar to but not strictly the same as "vernacular misconceptions" [**nationalresearchcouncilScienceTeachingReconsidered1997**].

Misconception	Question	Table of Answers	
CallByRef Function calls alias variables.	`(defvar x 12)` `(deffun (set-and-return y)` `(set! y 0)` `x)` `(set-and-return x)`	78%	`12`
		11%	`error`
		**10%	`0`
		1%	`23`
CallsCopyStructs Function calls copy data structures.	`(defvar x (mvec 1 0))` `(deffun (f y)` `(vec-set! y 0 173))` `(f x)` `x`	90%	`#(173 0)`
		**10%	`#(1 0)`
DeepClosure Closures copy the *values* of free variables.	`(defvar x 1)` `(defvar f` `(lambda (y)` `(+ x y)))` `(set! x 2)` `(f x)`	86%	`4`
		**10%	`3`
		3%	`error`
		1%	`lambda`
DefByRef Variable definitions alias variables.	`(defvar x 12)` `(defvar y x)` `(set! x 0)` `x` `y`	85%	`0 12`
		**12%	`0 0`
		2%	`error`
		1%	depends on implementation.
DefOrSet Both definitions and variable assignments are interpreted as follows: if a variable is not defined in the current environment, it is defined. Otherwise, it is mutated to the new value.	`(set! foobar 2)` `foobar`	85%	`error`
		**15%	`2`
DefsCopyStructs Variable definitions copy structures recursively.	`(defvar x (mvec 100))` `(defvar y x)` `(vec-set! x 0 200)` `y`	**67%	`#(100)`
		30%	`#(200)`
		1%	`#(300)`
		1%	`error`

Table 6: Ground misconceptions identified by the SMoL Tutor. Answers marked with "**" represent the misconception. (Part I)

Misconception	Question	Table of Answers

FlatEnv There is only one environment, the global environment. (This misconception is a kind of dynamic scope.)	`(deffun (addy x)`
	`  (defvar y 200)`
	`  (+ x y))`
	`(+ (addy 2) y)`
		96% `error`
		**4% `402`

FunNotVal Functions are *not* considered first-class values. They can't be bound to other variables, passed as arguments, or referred to by data structures.	`(deffun (twice f x)`
	`  (f (f x)))`
	`(deffun (double x)`
	`  (+ x x))`
	`(twice double 1)`
		83% `4`
		**11% `error`
		4% `2`
		1% `8`

IsolatedFun Functions can't refer to free variables except for the built-in ones.	`(defvar y 1)`
	`(deffun (addy x)`
	`  (+ x y))`
	`(addy 2)`
		77% `3`
		**23% `error`

NoCircularity Data structures can't (possibly indirectly) refer to themselves.	`(defvar x (mvec 1 0 2))`
	`(vec-set! x 1 x)`
	`(vec-len x)`
		76% `3`
		**14% `error`
		9% Run out of memory or time.
		1% `+inf`

StructByRef Data structures might refer to variables by their references.	`(defvar x 3)`
	`(defvar v (mvec 1 2 x))`
	`(set! x 4)`
	`v`
		67% `#(1 2 3)`
		**24% `#(1 2 4)`
		9% error

StructsCopyStructs Storing data structures into data structures makes copies.	`(defvar x (mvec 2 3))`
	`(set-right! x x)`
	`(set-left! x x)`
	`x`
		65% `#0=#(#0# #0#)` (Racket circular object notation)
		24% `#(#(2 3) #(2 3))`
		**6% `#(#(2 #(2 3)) #(2 3))`
		6% error

Table 7: Ground misconceptions identified by the SMoL Tutor. Answers marked with "**" represent the misconception. (Part II)

Misconception	**Question**	**Table of Answers**

NestedDef‡ Sequences of definitions are interpreted as if they are written in nested blocks. A definition is not in the scope of later definitions.	`(defvar x 1)`
	`(deffun (main)`
	`  (deffun (get-x) x)`
	`  (defvar x 2)`
	`  (get-x))`
	`(main)`
		92% `2`
		**8% `1`

Lazy‡ Expressions are only evaluated when their values are needed.	`(defvar y (+ x 2))`
	`(defvar x 1)`
	`x`
	`y`
		**57% `1 3`
		43% `error`

Table 8: Surmised misconceptions identified by the SMoL Tutor. Answers marked with "**" represent the misconception.

Publication	**Population**	**Languages**	**Misconceptions**
[**fleuryParameterPassingRules1991**]	Unclear, likely CS2 students	Pascal	**CallerEnv**‡; **IsolatedFun**; **DefOrSet** (See their RULEs 2, 3a, and 3b in TABLE 2; RULE 1 doesn't apply to us.)
[**goldmanIdentifyingImportantDifficult2008**, **goldmanSettingScopeConcept2010**]	CS1 students	Java, Python, Scheme	Scope and memory model (See their Figures 4 and 5)
[**fislerAssessingTeachingScope2017**]	Third- and fourth-year undergrads	Java and Scheme	**FlatEnv**; **CallByRef**; **CallsCopyStruct**; **DefByRef** (See their Section 4)
[**saarinenHarnessingWisdomClasses2019**]	CS2 students	Java	**StructByRef** (Their G2); **DefByRef** or **DefCopyStructs** (Their G3); **CallsCopyStruct** (Their G4).
[**strombackProgressionStudentsAbility2023**]	CS masters	Python	**FlatEnv**; **CallByRef**; **DefsCopyStruct** (See their section 4.2)
[**strombackProgressionStudentsAbility2023**]	CS undergrads	C++	**FlatEnv**; **CallByRef**; **DefsCopyStruct** (See their section 4.2)

Table 9: Similar misconceptions found in prior research.

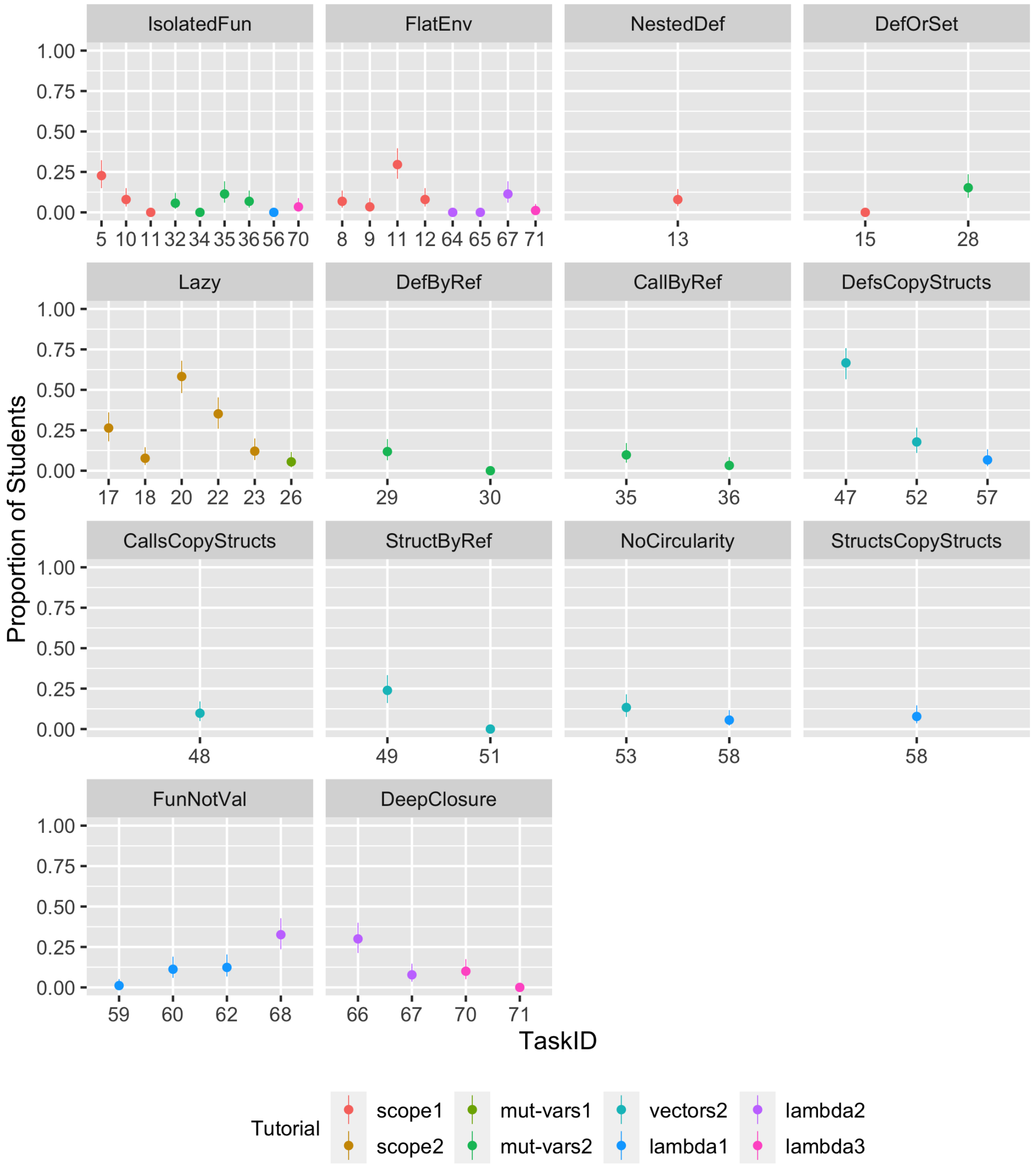


Figure 4: How many students chose a wrong answer that (uniquely) represents a misconception? (Downward tendency suggests improvement over time.)