Identifying and Correcting Programming Language Behavior Misconceptions

Kuang-Chen Lu

November 14, 2023

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?

I propose a plan for finding misconceptions possibly not anticipated by experts, and the design of an automated, self-guided tutoring system. The Tutor builds on strategies in the cognitive and educational literature and is explicitly designed around identifying and correcting misconceptions. Preliminary results suggest (a) the misconceptions I found are widespread, and (b) the Tutor appears to improve understanding.

Contents

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 [Bernhardt, 2012; Guha et al., 2010; Politz et al., 2013, 2012]). 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 proposal I 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 a background section (Section 3) 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` [Lippert, 2009]. It is now also the focus of a language design change in Go [Chase and Cox, 2023] for a new kind

```
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.

of looping construct [Cox, 2023]. It continues to trip up programmers (e.g., sharvey [2022]) despite being documented as a "gotcha" in Python [Reitz and Schlusser, 2016].

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 [Clarke et al., 1998], manage parallelism, and more. Thus, we still need to train students and other programmers on the semantic features *and their interactions* in SMoL.

This led to my **research questions**:

RQ1 What program-behavior misconceptions apply to SMoL and present even in students with prior programming backgrounds?

RQ2 How effective is a tutoring system driven by cognitive and educational principles at correcting these misconceptions?

In the rest of this document, I first define SMoL (Section 2), then present a literature review of misconceptions (Section 3) and tutoring systems (Section 4), and then outline my research plans for the RQs (Sections 5 and 6). After that, I describe the preliminary results (Section 7 and propose my next steps (Section 8).

2 The SMoL Language

SMoL is designed to capture common features of many modern languages. The syntax of SMoL is presented in Figure 1, 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.

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.

SMoL defines a semantic but not the syntax. However, to represent SMoL to students, I have to choose a syntax. I chose a Lispy syntax, which is an artifact of the major population where I conducted studies: a course (Section 5.1) using Racket [Friedman et al., 2001; Krishnamurthi, 2007]. However, the Lispy syntax also proved to be pedagogically valuable. I 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 (Politz et al. [2013]; see also posts like froadie [2022]), 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. Furthermore, I intend to make a multi-lingual Tutor in the future (Section 8).

The semantics of SMoL is as described in Section 1. SMoL includes limited primitive operators (Table 1) 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 [Felleisen et al., 2018]. 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:

https://github.com/shriram/smol

3 Background: 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). Table 2 lists works that seem most relevant to us. Appendix A of Sorva [2012] provides an extensive survey of misconceptions reported in research up to 2012.

Known misconceptions are often detected with a kind of instruments called *concept inventories* [Hestenes et al., 1992; Taylor et al., 2014]. In terms of mechanics, a concept inventory is just an instrument consisting of multiple-choice questions (MCQs), where each question has one correct

Publication	Population	Languages	Topics
Fleury [1991]	Likely CS2 students	Pascal	Scope
Goldman et al. [2008, 2010]	CS1 students	Java, Python, Scheme	Scope and memory model (See their Figures 4 and 5)
Fisler et al. [2017]	Third- and fourth-year undergrads	Java and Scheme	Scope, variable aliasing, and structure aliasing.
Saarinen et al. [2019]	CS2 students	Java	Variable aliasing and structure aliasing.
Strömbäck et al. [2023]	CS masters	Python	Scope, variable aliasing, and structure aliasing.
Strömbäck et al. [2023]	CS undergrads	C++	Scope, variable aliasing, and structure aliasing.

Table 2: Topics of misconceptions found in prior research.

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]

4 Background: Tutoring Systems

There is an extensive body of literature on tutoring systems (VanLehn [2006] is a quality survey), and indeed whole conferences are dedicated to them. I draw on this literature. In particular, it is common in the literature to talk about a "two-loop" architecture [VanLehn, 2006] where the outer loop iterates through "tasks" (i.e., educational activities) and the inner loop iterates through UI events within a task. I follow the same structure in my Tutor (Section 7.5).

Many tutoring systems focus on teaching programming (such as the well-known and heavily studied LISP Tutor [Anderson and Reiser, 1985]), and in the process undoubtedly address some program behavior misconceptions. My 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. I am not aware of a tutoring system (in computer science) that has this specific design.

5 RQ1: Find Misconceptions

RQ1 What program-behavior misconceptions apply to SMoL and present even in students with prior programming backgrounds?

[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.

5.1 In What Populations to Find?

To find misconceptions that "present (even) in students with prior programming backgrounds", I have been collecting data from multiple populations that very likely have prior experience (Section 7.7). I have the following data sources so far:

University 1 includes students in a "Principles of Programming Languages" class at Brown 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.

University 2 includes students at 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).

Textbook includes visitors to the website of a programming languages textbook. Over the course of 8 months, several hundreds of people submitted to the instrument. To protect privacy, I intentionally do not record demographic information, but I conjecture that the population is largely self-learners (who are known to use the accompanying book), including some professional programmers.

I am contacting professors from various other universities. Several of them have expressed interest in using my instruments. So, the final results will likely include more populations.

5.2 Find Misconceptions by Analyzing Student-written Programs

In Section 3, I 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* [Nathan et al., 2001]: experts simply do not conceive of many learner difficulties.

To overcome such bias, I analyze a University 1 dataset produced by a two-year process to find (a) student-written programs that tend to trip up students and (b) incorrect responses to those programs using a tool for the purpose (Section 7.1). I have been cleaning up those programs (Section 7.2) and confirming that these programs are still tricky in the same way. Section 7.3 presents the final list of misconceptions found with this process.

5.3 Find Misconceptions by Testing Students with Expert-written Programs

I have another source of misconceptions. I have been creating a Tutor to answer my RQ2, which I explain in detail in Section 6. The Tutor includes many tasks that ask students to predict the result(s) of running a program. Although these programs originate from student-written programs (Section 5.2), they have been modified to fit tutoring purposes. Perhaps because of the modification or because of the population variance, I have found several unanticipated patterns of errors (Section 7.3), which suggest unknown misconceptions.

I plan to change the Tutor so that it collects data to uncover these unknown misconceptions and confirm known ones.

5.4 Expected Results

Eventually, I expect to build a table of misconceptions. Each misconception comes with multiple programs. For each of these programs, the wrong result that corresponds to the misconception should be commonly predicted by students from various populations. Furthermore, these students should give similar explanations for all the wrong answers. For example, I might find a misconception called **DefByRef** (See Tables 4 to 6 in Section 7.3 for a list of misconceptions that I have found): when students are asked to predict the result of

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

X% of them responded `12 0`; when asked to predict

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

Y% of them responded `0 12`; and these students all explained something like "`y` is `x`".

To be reasonably specific about misconceptions, I have created a definitional interpreter for each misconception that I have found. I plan to do the same things for any new misconceptions. I call these interpreters *misinterpreters*. Having definitional interpreters makes it convenient (and less error-prone) to associate wrong answers with misconceptions (Section 7.3).

6 RQ2: Correct Misconceptions

RQ2 How effective is a tutoring system driven by cognitive and educational principles at correcting these misconceptions?

To answer this RQ, I plan to create a Tutor (Section 6.1) and argue that the Tutor is effective (Section 6.2).

6.1 Design a Tutor for Correcting Misconceptions

To answer this RQ, I have done some literature review on what pedagogic techniques are helpful (Section 7.4). My current takeaway is that refutation text, case comparison, notional machines, and

Tutorial	Language Constructs
scope: Lexical scope	Primitives, variables, `defvar`, and `deffun`.
mut-vars: Mutable variables	Adding `set!`.
vectors: Vectors and vector updates	Adding vector operators.
lambda: Lambda expressions	Adding `lambda`.
local: Local binding forms	Adding `let`.

Table 3: Tutorials and Their Language Constructs

language levels are helpful. I have created a Tutor that incorporates all these ideas (Section 7.5) and have collected data from all populations.

I would like to first explain key aspects of the Tutor (Sections 6.1.1 and 6.1.2) and then my plan to incorporate ideas of refutation text (Section 6.1.3), case comparison (Section 6.1.3), and notional machines (Section 6.1.3). Finally, I present my plan to evaluate the effectiveness of the Tutor and how evaluation impacts the Tutor design (Section 6.2).

6.1.1 Tutorials and Language Levels

There are many misconceptions to correct. To make the tutoring process manageable, I have designed the Tutor as a sequence of tutorials. Inspired by prior research on language levels (Section 7.4), my sequence of tutorials is ordered, and each tutorial covers one or more new language constructs (Table 3). Each tutorial might be delivered as multiple modules to avoid forcing students to continuously work for more than 30 minutes.

6.1.2 Interpreting Tasks

The Tutor mostly consists of *interpreting tasks*: multiple-choice questions (MCQs) that ask students to interpret program. The options include (among other options that I will explain shortly) the correct answer and wrong answers that correspond to known misconceptions. Because students might conceive wrong answers that I do not anticipate, the Tutor has been providing an "Other" option that, once chosen, allows students to enter arbitrary answers.

Let's reconsider this program from Section 5.4

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

In this case, the options should include the correct answer `12 0` and a wrong answer `0 0`, which corresponds to the **DefByRef** misconception. If an MCQ only has few options like this example, students have a great chance (in this case, 50%) of just guessing the correct answer or successfully using a process of elimination. So I have been adding additional options that are not obviously wrong. The Tutor is adding many options, but with an ad hoc rule. I plan to revise the rules of adding extra options.

Penalty to Wrong Choices To discourage students from making a random choice, the Tutor has been giving penalties. Designing the strategy of assigning penalties is non-trivial:

- A common strategy is to grade students by their correctness. However, grading by correctness is not suitable for a tutoring system because I do not mind students doing badly on earlier tasks as long as they eventually master the concepts. Actually, if students generally succeed in all tasks, the tutoring system is failing – it does not teach anything to the students.
- An alternative common strategy is to grade by completion. However, grading by completion alone encourages students to rush through the tutorials and, hence, might encourage students to make random choices.

In short, I want to make students feel as follows:

- They won't lose grades as long as they finish the tutorials.
- They should avoid giving wrong answers.

To encourage this feeling, I have been doing the following: when I deployed the Tutor in my institution, University 1, I graded by completion[2]; when a student fails a task, the Tutor gives an *extra* task:

1. The program 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, I want to force reflection on the explanation just given, whereas with an MCQ, students could just guess. Second, I feel that students would find typing more onerous than clicking. In case students had just guessed on a question, my hope is that the penalty of having to type means, on *subsequent* tasks, they would be more likely to pause and reflect before choosing.

6.1.3 Design with Pedagogic Techniques Known to Work

Refutation Text and Other Explanations The Tutor has been presenting a refutation text when a student chooses a wrong answer that corresponds to known misconception(s). Let's reconsider the program in Section 6.1.2. If a student answers `0 0`, which suggests they hold the **DefByRef** misconception, the Tutor responds

> `y` was bound to `12` (i.e., the value of `x`) rather than to `x`. So changing the value of `y` does not change the value of `x`.

If a student gives a wrong answer that does not correspond to a misconception (either by choosing one of the generated options or by choosing the "Other" option), the Tutor presents a generic explanation. For example, if a student answers `12 12`, the Tutor responds

> 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 `y`, so `y` is now bound to `0`.

If a student gives the correct answer, the Tutor also presents the generic explanation.

[2] I encourage collaborators to grade by completeness, but I have no real control over how they actually use the Tutor.

Case Comparison The Tutor provides opportunities for making case comparisons: each interpreting task presents a SMoL program and the program's result. These program-result pairs are concrete cases of the semantics of SMoL. To incorporate the idea of case comparison, the Tutor currently prompts students to compare these cases and then to summarize the rules of program behavior at the end of *some* tutorials.

Notional Machine Prior research argues (Section 7.4)that illustrating how programs run with notional machines (NMs) is beneficial for students. I have been giving students opportunities and sometimes explicitly requiring them to run programs in the NM visualization tool.

6.2 Evaluate the Tutor

To fully answer my research questions, I should also argue that the Tutor is effective at correcting misconceptions. In this section, I describe how I want to refine the design of the Tutor to help me make the argument, what data I want to collect, and what (statistical) analysis I want to perform on the data.

I can infer whether students hold a misconception by analyzing their responses to interpreting tasks. There are three possible responses to an interpreting task:

1. They give the correct answer.
2. They give a wrong answer that corresponds to some misconceptions.
3. They give a wrong answer that does not correspond to any (known) misconceptions.

If a student gives a wrong answer that corresponds to exactly one misconception *and* an explanation that matches the misconception, I deduce that the student holds the misconception when doing the task.

Given this observation, I plan to set up the Tutor such that

- Every wrong answer corresponds to up to one misconception. In this case, I say the wrong answer *represents* the misconception.
- Every misconception is represented by some questions.
- In each tutorial, a misconception is either not represented by any interpreting tasks or represented by at least two tasks.
- Within each tutorial, the order of interpreting tasks is randomized.

For each tutorial and each misconception, I plan to do a model comparison (with the standard AIC and BIC criteria) on the following models/theories:

TaskSpecific Some students are more likely to choose the representing wrong answers in certain tasks.

PositionSpecific Students are more likely to choose the representing wrong answer in task that appears earlier (or later) in the tutorial.

If the **TaskSpecific** model best explains the data, I deduce that some programs are more confusing and that I need further investigation into the relationship between the tasks and misconceptions. If the **PositionSpecific** model best explains *and* students perform better in later tasks, I deduce that the Tutor seems to be correcting misconceptions. If students perform worse in later tasks, the Tutor is potentially doing harm.

To know students well, I would like to have as many tasks per misconception as possible. But in practice, the number of tasks is subject to the following limitations:

1. I don't want the length of any tutorial section to exceed 30 minutes.

2. Programs shouldn't be too long. Some misconceptions are really similar and, as a consequence, have few short programs to tell them apart.

The current Tutor is not prepared for this analysis yet. I describe how I plan to change the Tutor in Section 8.

7 Preliminary Results

7.1 Generating Problems Using Quizius

As said in Section 5, my main solution to the expert blind spot is to use the Quizius system [Saarinen et al., 2019]. 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 my case, students were 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.[3] 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 [Katehakis and Veinott, 1987] 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, I obtained a set of rationales.

After running Quizius in the course (Section 5.1), I took over as an expert. Determining which is the right answer is easy, whereas expert knowledge is useful 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).

[3] In the course (Section 5.1), 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.

Prior work using Quizius [Saarinen et al., 2019] finds that students do author programs that the experts did not imagine. In my case, I seeded Quizius with programs from prior papers (Section 3), which gives the first few students programs to respond to. However, I found that Quizius significantly expanded the scope of my problems and misconceptions. In my final instrument, most programs were directly or indirectly inspired by the output of Quizius.

The Quizius instrument was set up by other people. I analyzed the collected data.

7.2 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.

- I felt the existing programs were insufficient to cover the ideas I wanted students to work through. For instance, I found no program that aliases vectors using `defvar`, so we added this program:

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

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

- I deferred the local binding forms to the end, since I felt they were less important. I therefore rewrote programs that use it to not depend on it.

- I 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)
  ```

- I resolved ambiguities in some programs by adding answer choices or even other questions to tease out different interpretations.

- To reduce the number of concepts, I removed programs that relied upon *im*mutable vectors and lists, because they did not seem to create problems. (For brevity, I leave these out of the presentation of SMoL in Section 2, though they are in the implementation.)

- I removed questions related to function equality, which was not a focus of the Tutor.

- I removed programs of a "Lispy" nature.

7.3 Preliminary List of Misconceptions

I now examine what I learned from the Tutor's interpreting tasks in terms of program behavior misconceptions.

I iteratively created our final catalog of misconceptions (from the perspective of the data in this document). I started with misinterpreters representing the misconceptions for which we have reasonable validation (due to the prose in Quizius), so I call them *grounded* misconceptions. I then looked at wrong answers not covered by these but chosen by students, and did my best to distill these into misconceptions. These are *surmised* misconceptions (which we identify with a ‡), which need to be validated in the future. I then re-ran the misinterpreters against the chosen answers. I 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 me to be "Lispy" and hence not of broad interest.

Tables 4 to 6 list the final catalog. For each, I 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.

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

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, I surmise that students might have another misconception, which I define as **Lazy**‡. (Recall that SMoL is eager, but even in many

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%
		**4%
			`error`
			`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%
		**11%
		4%
		1%
			`4`
			`error`
			`2`
			`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%
		**23%
			`3`
			`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%
		**14%
		9%
		1%
			`3`
			`error`
			Run out of memory or time.
			`+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%
		**24%
		9%
			`#(1 2 3)`
			`#(1 2 4)`
			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%
		24%
		**6%
		6%
			`#0=#(#0# #0#)` (Racket circular object notation)
			`#(#(2 3)#(2 3))`
			`#(#(2 #(2 3))#(2 3))`
			error

Table 5: Ground misconceptions identified by the SMoL Tutor. Answers marked with "**" represent the misconception. (Part II)

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)
```

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

```
  (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 my existing misconceptions. Based on this, I surmise a new misconception, **NestedDef**‡. (Though its frequency falls below our threshold, my reading of answers suggests this may be more widespread, and I feel it needs to be investigated more.)

Once I turned this into a misinterpreter, I found that it unexpectedly captured the program-answer pair for the following program — 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, I 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. I 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 I uncovered the new (surmised) misconception.

7.4 What Pedagogic Techniques are Effective?

The fundamental problem about correcting misconceptions 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 Posner et al. [1982] 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 [Schroeder and Kucera, 2022] have shown their effectiveness in multiple other domains.

Case comparisons draw analogies between examples. Alfieri et al. [2013] suggest 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.

Several groups of authors have argued for the benefits of visualizing program executions [Dickson et al., 2022; Guo, 2013; Karnalim and Ayub, 2017; Naps et al., 2002]. Visualization seems to increase engagement and help students understand how programs work. These works often mention *notional machines* [Du Boulay et al., 1981], which refers to the underlying mental models rather than the visual representations.

Prior research argues for the benefits of *language levels* [Findler et al., 2002; Crestani and Sperber, 2010]: a language has better be taught as a sequence of languages, where each later language includes more language constructs than the previous one.

7.5 The SMoL Tutor

This section describes the current status of the Tutor.

The Tutor includes five tutorials, shown on the left in Table 7. 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. My data show that in practice, students spent about 9.8 minutes (median).

Figure 2 illustrates an interpreting task as described in Section 6.1.2. The Tutor also asked students to perform some other activities, like showing programs and asking for their heap content, which I do not cover in this document.

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. I, therefore, refer to these as **goal sentences**. Table 7 lists the (abbreviated) goal sentences for each tutorial. I plan to keep goal sentences in the Tutor.

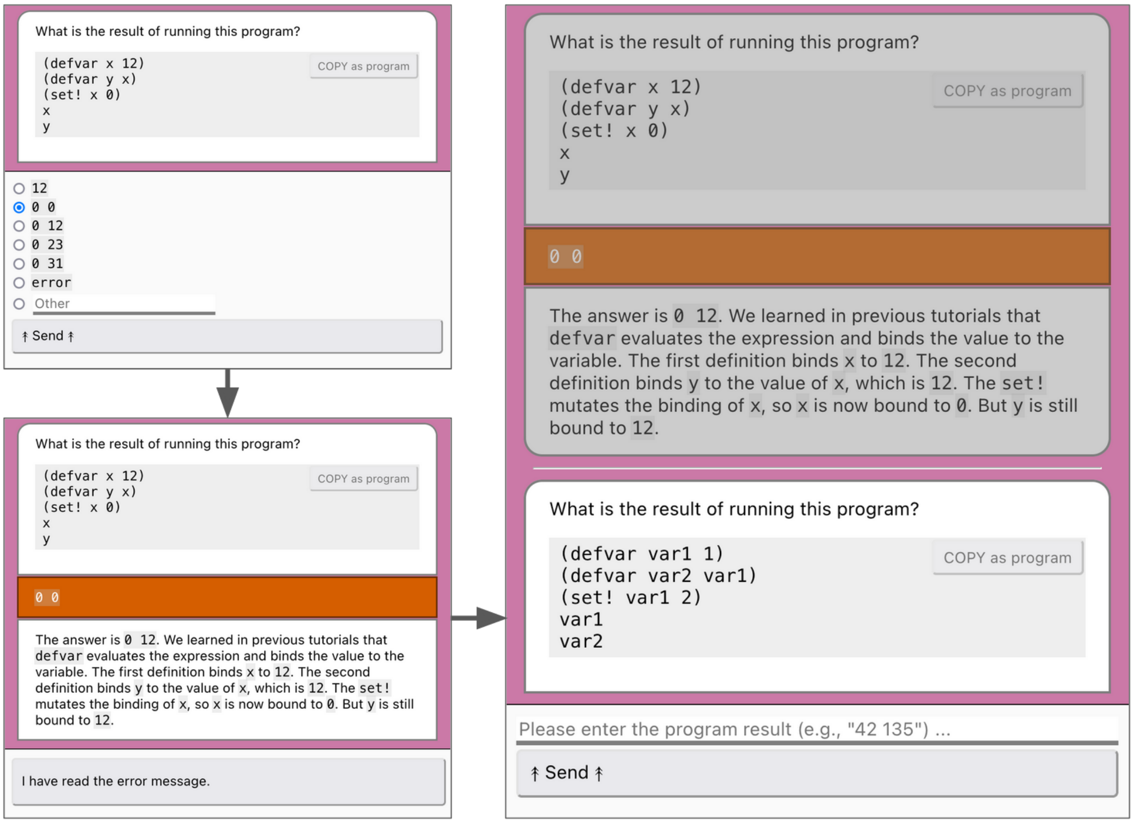
Some later tutorials include questions about earlier tutorials so that I can check whether students remember the concepts across modules. In particular, the **mut-vars** tutorial starts with questions about **scope**, and **lambda** starts with questions about **mut-vars** and **vectors**. I plan to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

7.6 How Effective Is the Tutor?

This section presents part of my preliminary answer to **RQ2** (Section 6): is the Tutor effective?

I present the result in two forms. Graphically, we show plots in Figure 3. 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 I see in most of the graphs. The exceptions are **CallsCopyStructs**, **NestedDef**‡, and **StructsCopyStructs**, which have only one problem (and hence no trend), and



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 2: 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.

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!)

I also perform a logistic regression to see whether these improvements are significant (at a $p < 0.05$ threshold). 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.

Overall, the data suggest that the Tutor *most likely did not do harm* and perhaps even *may have done some good*. Revisions on the Tutor (Section 8), in light of misinterpreters, is necessarily

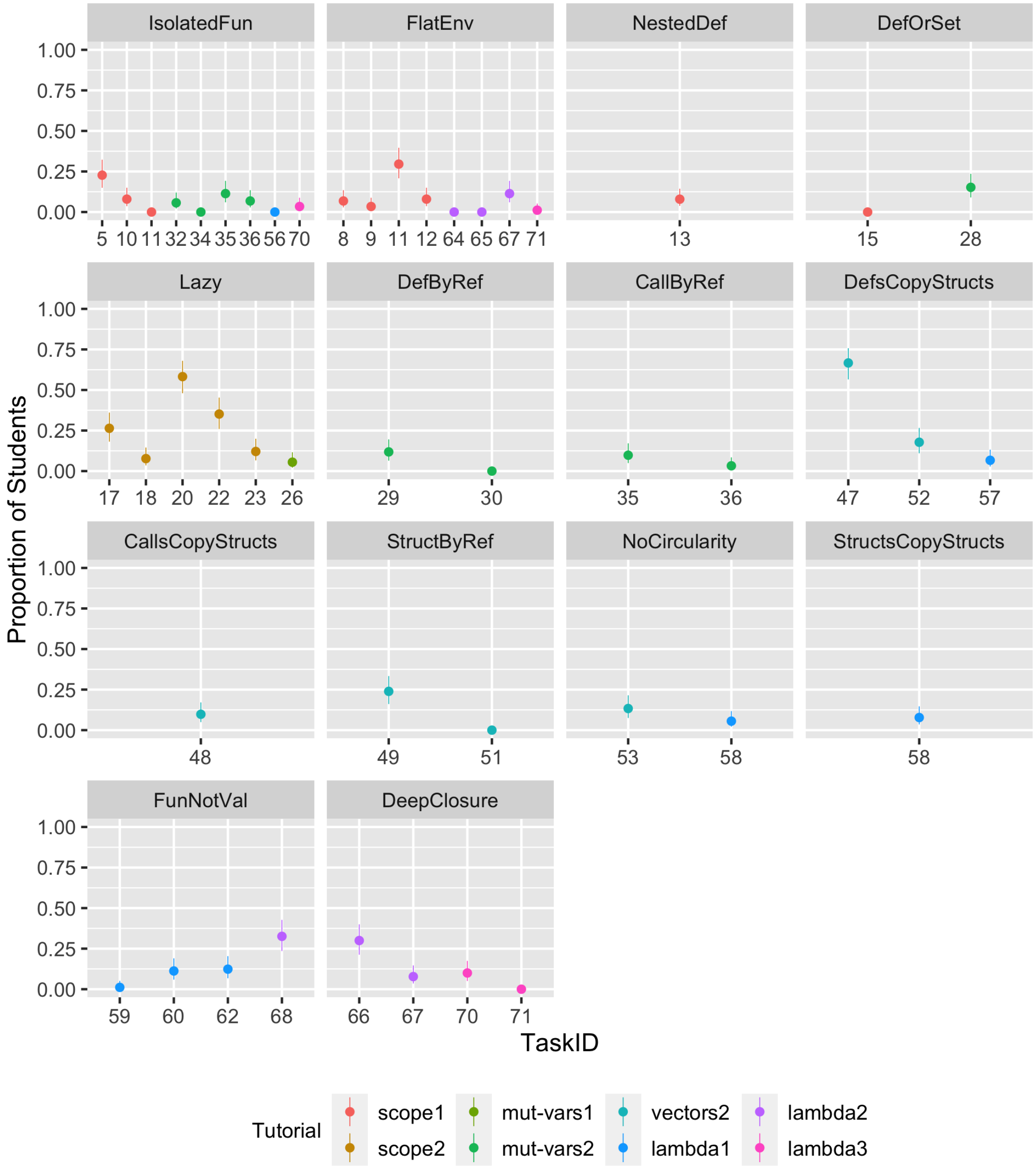


Figure 3: How many students chose a wrong answer that (uniquely) represents a misconception? (Downward tendency suggests improvement over time.)

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 7: SMoL Tutor tutorials and their goal sentences.

to give a more accurate evaluation. For example, I found far fewer problems than I would have liked. Concretely, only 40 of the 71 eligible problems (after removing the non-SMoL modules) were useful in the above analysis.

7.7 How Generalizable are My Results?

I 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, I obtained a p-value = 2.013e-07. Between the original university and the online population, I obtained 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.*

Of course, other threats to generalizability remain: both University 1 and University 2 use the same textbook; SMoL's Lispy syntax might have biases in probing errors. I plan to address these issues in the future (Section 8).

8 Future Work & Timeline

8.1 Keep Exploring More Pedagogic Techniques

I have learned (Section 7.4) that refutation text, case comparison, notional machines, and language levels are helpful. I plan to explore more literature and improve the Tutor if I find new helpful techniques.

8.2 Implemented But Not Evaluated Improvements in the Tutor

Ease Access to the Notional Machine The preliminary Tutor provides a "Copy This Program" button that helps students to run the program in DrRacket, where a SMoL implementation is provided.

This approach is inconvenient both for the students and for me. For students, they have to click, paste, and run, which is less convenient than a single click.

Given these shortcomings, I have created a web-based notional machine

https://lukuangchen.github.io/stacker-2023

The notional machine can be configured with URL parameters. For example, this link opens the NM with a program embedded.

The latest Tutor provides a button on the top-right of each program. If students click the button, they will open a new browser tab that shows the NM populated with the program. I have deployed the latest Tutor to the latest population at Brown (i.e., University 1), but I have not analyzed the data, so the new Tutor is not presented in the preliminary results.

Reduce Syntactic Bias with Multiple Syntax The processes described in Sections 5.2 and 5.3 have been conducted with SMoL's Lispy syntax (Section 2). This can potentially introduce biases to the found misconceptions. For example, I found that some students think the following program produces `'(4 5)` rather than `'(1 2)`.

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

This checks whether `3` > `n`, but those students presumably vocalized it as "greater than `3`, `n`?". Although I have tried to remove programs like this from the Tutor, there is still a risk of having biases in student errors. This semester (Fall 2023), I am already deploying a newer SMoL Tutor at Brown. This new Tutor is able to display programs in JavaScript and Python, in addition to the Lispy syntax. I plan to compare in Spring 2024 the new Tutor data with the current results, i.e., a comparison between uni-syntax Tutor and multi-syntax Tutor.

8.3 The Rest of Fall 2023: Improve the Tutor

Log Access to the Notional Machine I have not changed the logging systems to log when students click the buttons to access the NM. I plan to make the change before Spring 2024.

Limit Access to the Notional Machine Another issue with the current design is that students might run a program before giving an answer. I don't know how often this happens. But, given that students receive a penalty (Section 6.1.2) for giving wrong answers, they might do so when the program is overly complicated. This problem is more of an issue after the switch to a web-based notional machine because now the "cost" of running a program is cheaper. My plan is to change the Tutor so that it provides a program-specific link *only* after each interpreting task.

Use Case Comparisons More Consistently The Tutor currently prompts students to compare program-output pairs and then to summarize the rules of program behavior at the end of *some* tutorials.

I plan to expand this to *all* tutorials and use a consistent prompt. The prompt should ask about the behavior of new language constructs and their interaction with all previously introduced constructs.

The current scope tutorial uses the following prompt:

> When I see a variable reference, how do I find the corresponding declaration, if any?

It is unclear to adapt it to other tutorials.

So I plan to change the question format into a more consistent style. So I plan to replace the **scope** prompt with

> How do variable referencess (e.g., the `x` in `(+ x 1)`) , `defvar`s, and `deffun`s interact with each other?

I plan to use a similar prompt in later tutorials. For example, in **mut-vars**, I would want the Tutor to ask

> How do `set!`s interact with other language constructs (namely, `defvar`s, `deffun`s, and variable references)?

Use More Sensible Options in Interpreting Tasks The Tutor is adding extra options to interpreting tasks in an ad hoc way. Eventually, I plan to generate options by moving and/or copying numbers in the current options. In the case of the example question in Section 6.1.2, my plan would add `12 12` and `0 12`. In summary, each MCQ has the following options:

- the correct answer,
- wrong answers that correspond to known misconceptions,
- options generated by moving and/or copying numbers in the current options
- the "Other" option (which, once chosen, allows students to enter arbitrary answers)

Ask For Explanation on Choices The Tutor currently does not ask students to explain when they give a wrong answer in an interpreting question. This limits my ability to find new misconceptions and to confirm that the tasks do detect the anticipated misconceptions. I plan to change the Tutor so that it asks for an explanation in each interpreting task.

Clean-Up the Program Set As I said in Section 7.6, for many of the misconceptions, the Tutor has too few questions. On the other hand, there are many questions that do not produce useful data because their wrong answers do not uniquely point to a misconception. I plan to adjust the collection of interpreting questions so that every misconception has at least two questions that aim to correct the misconception.

Randomize the Order of Interpreting Tasks The Tutor is not randomizing the order of questions. This makes it difficult to compare questions for one misconception – earlier questions might be difficult simply because the programs are somehow more difficult. I plan to randomize the order of interpreting tasks within each tutorial module. After randomizing question orders, I would need to check carefully the dependencies between explanations and resolve these dependencies, if any.

Clean-Up Variable Names The Tutor uses many variable names; some are separated by dashes (e.g., `get-x`). Students might misread these variables. For example, some students might misread `(get-x)` (a function named `get-x` is called with no argument) as `(get x)` (a function named `get` is called with one argument `x`). These misreadings add noise to the data. So I plan to use consistent and simple variable names in all programs.

8.4 Spring 2024: Collect More Data

Several faculties from other institutions have expressed interest in using the Tutor in Spring 2024. My goal is to collect data from at least two other institutions.

I am officially taking a personal leave to do military service in my country in Spring 2024, so I do not plan to make more progress other than data collection.

8.5 Summer 2024: Data Analysis and Write a Paper

During this time, I plan to analyze the data I collected in Fall 2023 and Spring 2024 and start working on a paper about the changes in the Tutor.

8.6 Fall 2024: Collect More Data and Write a Paper

During Fall 2024, I plan to collect a new round of data from Brown, analyze data that I collected in Spring 2024, and hopefully finish writing the new paper.

8.7 Spring 2025: Defense

I plan to defend in Spring 2025.

9 Related Work

9.1 Program Behavior

Our idea of "goal sentences" is not substantially different from the *rules of program behavior* from Duran et al. [2021]. I only used these sentences (or rules) to guide the design of the Tutor and as text in the Tutor itself. Duran et al. [2021] argue for other uses of such sentences.

9.2 Misinterpreters

Our idea of misinterpreters is related to mystery languages [Diwan et al., 2004; Pombrio et al., 2017]. 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.

References

Louis Alfieri, Timothy J. Nokes-Malach, and Christian D. Schunn. Learning Through Case Comparisons: A Meta-Analytic Review. *Educational Psychologist*, 48(2):87–113, April 2013. ISSN 0046-1520, 1532-6985. doi: 10.1080/00461520.2013.775712.

John R. Anderson and Brian J. Reiser. The LISP tutor. *Byte*, 10(4):159–175, 1985. URL `http://act-r.psy.cmu.edu/wordpress/wp-content/uploads/2012/12/113TheLISPTutor.pdf`.

Gary Bernhardt. Wat, 2012. URL `https://www.destroyallsoftware.com/talks/wat`.

David Chase and Russ Cox. Fixing For Loops in Go 1.22 - The Go Programming Language, September 2023. URL `https://go.dev/blog/loopvar-preview`.

David G. Clarke, John M. Potter, and James Noble. Ownership types for flexible alias protection. In *Proceedings of the 13th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '98, pages 48–64, New York, NY, USA, October 1998. Association for Computing Machinery. ISBN 978-1-58113-005-8. doi: 10.1145/286936.286947.

Russ Cox. Spec: Less error-prone loop variable scoping · Issue #60078 · golang/go, May 2023. URL `https://github.com/golang/go/issues/60078`.

Marcus Crestani and Michael Sperber. Experience report: Growing programming languages for beginning students. *ACM SIGPLAN Notices*, 45(9):229–234, September 2010. ISSN 0362-1340, 1558-1160. doi: 10.1145/1932681.1863576.

Paul E. Dickson, Tim Richards, and Brett A. Becker. Experiences Implementing and Utilizing a Notional Machine in the Classroom. In *Proceedings of the 53rd ACM Technical Symposium on Computer Science Education - Volume 1*, volume 1 of *SIGCSE 2022*, pages 850–856, New York, NY, USA, February 2022. Association for Computing Machinery. ISBN 978-1-4503-9070-5. doi: 10.1145/3478431.3499320.

Amer Diwan, William M. Waite, Michele H. Jackson, and Jacob Dickerson. PL-detective: A system for teaching programming language concepts. *Journal on Educational Resources in Computing*, 4(4):1–es, December 2004. ISSN 1531-4278. doi: 10.1145/1086339.1086340.

Benedict Du Boulay, Tim O'Shea, and John Monk. The black box inside the glass box: Presenting computing concepts to novices. *International Journal of man-machine studies*, 14(3):237–249, 1981. URL `https://www.sciencedirect.com/science/article/pii/S0020737381800569`.

Rodrigo Duran, Juha Sorva, and Otto Seppälä. Rules of Program Behavior. *ACM Transactions on Computing Education*, 21(4):33:1–33:37, November 2021. doi: 10.1145/3469128.

Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Shriram Krishnamurthi, Eli Barzilay, Jay McCarthy, and Sam Tobin-Hochstadt. A programmable programming language. *Communications of the ACM*, 61(3):62–71, February 2018. ISSN 0001-0782. doi: 10.1145/3127323.

Robert Bruce Findler, John Clements, Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, Paul Steckler, and Matthias Felleisen. DrScheme: A programming environment for Scheme. *Journal of Functional Programming*, 12(2):159–182, March 2002. ISSN 1469-7653, 0956-7968. doi: 10.1017/S0956796801004208.

Kathi Fisler, Shriram Krishnamurthi, and Preston Tunnell Wilson. Assessing and teaching scope, mutation, and aliasing in upper-level undergraduates. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '17, pages 213–218, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 978-1-4503-4698-6. doi: 10.1145/3017680.3017777.

Ann E. Fleury. Parameter passing: The rules the students construct. *ACM SIGCSE Bulletin*, 23 (1):283–286, 1991. ISSN 0097-8418. doi: 10.1145/107005.107066.

Daniel P. Friedman, Mitchell Wand, and Christopher Thomas Haynes. *Essentials of Programming Languages*. MIT press, 2001. ISBN 978-0-262-56067-2. URL `https://mitpress.mit.edu/9780262560672/essentials-of-programming-languages/`.

froadie. What's the scope of a variable initialized in an if statement?, December 2022. URL `https://stackoverflow.com/q/2829528`.

Ken Goldman, Paul Gross, Cinda Heeren, Geoffrey Herman, Lisa Kaczmarczyk, Michael C. Loui, and Craig Zilles. Identifying important and difficult concepts in introductory computing courses using a delphi process. In *Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '08, pages 256–260, New York, NY, USA, March 2008. Association for Computing Machinery. ISBN 978-1-59593-799-5. doi: 10.1145/1352135.1352226.

Ken Goldman, Paul Gross, Cinda Heeren, Geoffrey L. Herman, Lisa Kaczmarczyk, Michael C. Loui, and Craig Zilles. Setting the Scope of Concept Inventories for Introductory Computing Subjects. *ACM Transactions on Computing Education*, 10(2):5:1–5:29, June 2010. doi: 10.1145/1789934.1789935.

Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. The essence of javascript. In *Proceedings of the 24th European Conference on Object-oriented Programming*, ECOOP'10, pages 126–150, Berlin, Heidelberg, June 2010. Springer-Verlag. ISBN 978-3-642-14106-5.

Philip J. Guo. Online python tutor: Embeddable web-based program visualization for cs education. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education*, SIGCSE '13, pages 579–584, New York, NY, USA, March 2013. Association for Computing Machinery. ISBN 978-1-4503-1868-6. doi: 10.1145/2445196.2445368.

David Hestenes, Malcolm Wells, and Gregg Swackhamer. Force concept inventory. *The Physics Teacher*, 30(3):141–158, March 1992. ISSN 0031-921X, 1943-4928. doi: 10.1119/1.2343497.

Oscar Karnalim and Mewati Ayub. The Use of Python Tutor on Programming Laboratory Session: Student Perspectives. *Kinetik: Game Technology, Information System, Computer Network, Computing, Electronics, and Control*, pages 327–336, October 2017. ISSN 2503-2267, 2503-2259. doi: 10.22219/kinetik.v2i4.442.

Michael N. Katehakis and Arthur F. Veinott. The Multi-Armed Bandit Problem: Decomposition and Computation. *Mathematics of Operations Research*, 12(2):262–268, May 1987. ISSN 0364-765X, 1526-5471. doi: 10.1287/moor.12.2.262.

Shriram Krishnamurthi. *Programming Languages: Application and Interpretation*. Shriram Krishnamurthi, 2007. URL `https://cs.brown.edu/~sk/Publications/Books/ProgLangs/`.

Eric Lippert. Closing over the loop variable considered harmful, part one, November 2009. URL `https://ericlippert.com/2009/11/12/closing-over-the-loop-variable-considered-harmful-part-one/`.

Thomas L. Naps, Guido Rößling, Vicki Almstrum, Wanda Dann, Rudolf Fleischer, Chris Hundhausen, Ari Korhonen, Lauri Malmi, Myles McNally, Susan Rodger, and J. Ángel Velázquez-Iturbide. Exploring the role of visualization and engagement in computer science education. *ACM SIGCSE Bulletin*, 35(2):131–152, June 2002. ISSN 0097-8418. doi: 10.1145/782941.782998.

Mitchell J. Nathan, Kenneth R. Koedinger, and Martha W. Alibali. Expert Blind Spot : When Content Knowledge Eclipses Pedagogical Content Knowledge. In *Proceedings of the Third International Conference on Cognitive Science*, volume 644648, pages 644–648, 2001.

Joe Gibbs Politz, Matthew J. Carroll, Benjamin S. Lerner, Justin Pombrio, and Shriram Krishnamurthi. A tested semantics for getters, setters, and eval in JavaScript. In *Proceedings of the 8th Symposium on Dynamic Languages*, DLS '12, pages 1–16, New York, NY, USA, October 2012. Association for Computing Machinery. ISBN 978-1-4503-1564-7. doi: 10.1145/2384577.2384579.

Joe Gibbs Politz, Alejandro Martinez, Mae Milano, Sumner Warren, Daniel Patterson, Junsong Li, Anand Chitipothu, and Shriram Krishnamurthi. Python: The full monty. *ACM SIGPLAN Notices*, 48(10):217–232, October 2013. ISSN 0362-1340. doi: 10.1145/2544173.2509536.

Justin Pombrio, Shriram Krishnamurthi, and Kathi Fisler. Teaching programming languages by experimental and adversarial thinking. In *2nd Summit on Advances in Programming Languages (SNAPL 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017. URL `https://drops.dagstuhl.de/opus/volltexte/2017/7117/`.

George J. Posner, Kenneth A. Strike, Peter W. Hewson, and William A. Gertzog. Toward a theory of conceptual change. *Science education*, 66(2):211–227, 1982.

Kenneth Reitz and Tanya Schlusser. *The Hitchhiker's Guide to Python: Best Practices for Development*. O'Reilly Media, Inc., 2016. ISBN 978-1-4919-3322-0. URL `https://docs.python-guide.org/`.

Sam Saarinen, Shriram Krishnamurthi, Kathi Fisler, and Preston Tunnell Wilson. Harnessing the wisdom of the classes: Classsourcing and machine learning for assessment instrument generation. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*, pages 606–612, Minneapolis MN USA, 2019. ACM. ISBN 978-1-4503-5890-3. doi: 10.1145/3287324.3287504.

Noah L. Schroeder and Aurelia C. Kucera. Refutation Text Facilitates Learning: A Meta-Analysis of Between-Subjects Experiments. *Educational Psychology Review*, 34(2):957–987, June 2022. ISSN 1040-726X, 1573-336X. doi: 10.1007/s10648-021-09656-z.

sharvey. Creating functions (or lambdas) in a loop (or comprehension), September 2022. URL `https://stackoverflow.com/q/3431676`.

Juha Sorva. *Visual Program Simulation in Introductory Programming Education*. Aalto University, 2012. ISBN 978-952-60-4626-6. URL `https://aaltodoc.aalto.fi/handle/123456789/3534`.

Filip Strömbäck, Pontus Haglund, Aseel Berglund, and Erik Berglund. The Progression of Students' Ability to Work With Scope, Parameter Passing and Aliasing. In *Proceedings of the 25th Australasian Computing Education Conference*, ACE '23, pages 39–48, New York, NY, USA, January 2023. Association for Computing Machinery. ISBN 978-1-4503-9941-8. doi: 10.1145/3576123.3576128.

C. Taylor, D. Zingaro, L. Porter, K.C. Webb, C.B. Lee, and M. Clancy. Computer science concept inventories: Past and future. *Computer Science Education*, 24(4):253–276, October 2014. ISSN 0899-3408. doi: 10.1080/08993408.2014.970779.

Kurt VanLehn. The Behavior of Tutoring Systems. *International Journal of Artificial Intelligence in Education*, 16(3):227–265, January 2006. ISSN 1560-4292. URL `https://content.iospress.com/articles/international-journal-of-artificial-intelligence-in-education/jai16-3-02`.