Abstract of Identifying and Correcting Programming Language Behavior Misconceptions, by Kuang-Chen Lu, Ph.D., Brown University, May 2025.

Modern programming languages share a core set of linguistic concepts, including mutable variables, mutable data structures, and their interactions with scope and higher-order functions. Despite their ubiquity, students often struggle with these topics. How can we identify and effectively correct misconceptions about these cross-language concepts?

This dissertation presents systems designed to identify and correct such misconceptions in a multilingual setting, along with a formal classification of misconceptions distilled from studies with these systems. At the core of this work are: (1) the idea that formally defining misconceptions allows for more effective identification and correction, and (2) a self-guided tutoring system that diagnoses and addresses misconceptions. Grounded in established educational strategies, this tutor has been tested in multiple settings. My data show that (a) the misconceptions addressed are widespread and (b) the tutor improves student understanding on some topics and does not hinder learning on others.

Additionally, I introduce a program-tracing tool that explains programming language behavior with the rigor of formal semantics while addressing usability concerns. This tool supports the tutoring system in correcting misconceptions and appears to be valuable in its right.

Identifying and Correcting Programming Language Behavior Misconceptions

by

Kuang-Chen Lu
B.S., Shanghai Jiao Tong University, 2018
M.S., Indiana University, 2020

A dissertation submitted in partial fulfillment of the requirements
for the Degree of Doctor of Philosophy
in the Department of Computer Science at Brown University

PROVIDENCE, RHODE ISLAND
May 2025

This dissertation by Kuang-Chen Lu is accepted in its present form
by the Department of Computer Science as satisfying the
dissertation requirement for the degree of Doctor of Philosophy.

Date _____ _____
Shriram Krishnamurthi, Advisor

Recommended to the Graduate Council

Date _____ _____
Kathi Fisler, Reader

Date _____ _____
Juha Sorva (Aalto University), Reader

Date _____ _____
R. Benjamin Shapiro (University of Washington), Reader

Approved by the Graduate Council

Date _____ _____
Thomas A. Lewis, Dean of the Graduate School

Kuang-Chen Lu

lukuangchen.github.io

Education

- **Ph.D.**, Brown University, RI, USA 2020–2025
- **M.S.**, Indiana University, IN, USA 2018–2020
- **B.S.**, Shanghai Jiao Tong University, Shanghai, China 2014–2018

Publications

- Lu, Kuang-Chen, and Shriram Krishnamurthi. *Identifying and Correcting Programming Language Behavior Misconceptions*. OOPSLA, 2024.
- Lu, Kuang-Chen, Shriram Krishnamurthi, Kathi Fisler, and Ethel Tshukudu. *What Happens When Students Switch (Functional) Languages (Experience Report)*. Proceedings of the ACM on Programming Languages 7, no. ICFP (2023): 796-812.
- Lu, Kuang-Chen, Ben Greenman, Carl Meyer, Dino Viehland, Aniket Panse, and Shriram Krishnamurthi. *Gradual Soundness: Lessons from Static Python*. The Art, Science, and Engineering of Programming 7, no. 1 (2022).
- Lu, Kuang-Chen, Ben Greenman, and Shriram Krishnamurthi. *Types for Tables: A Language Design Benchmark*. The Art, Science, and Engineering of Programming 6, no. 2 (2022).
- Lu, Kuang-Chen. *Equivalence of Cast Representations in Gradual Typing*. Master's thesis, Indiana University, 2020.

- Lu, Kuang-Chen, Jeremy G. Siek, and Andre Kuhlenschmidt. *Hypercoercions and a framework for equivalence of cast calculi.* In Workshop on Gradual Typing. 2020.
- Ma, Weixi, Lu, Kuang-Chen, and Daniel P. Friedman. *Higher-order Logic Programming with λKanren.* In Proceedings of the 2020 miniKanren and Relational Programming Workshop.
- Lu, Kuang-Chen, Weixi Ma, and Daniel P. Friedman. *Towards a miniKanren with fair search strategies.* In Proceedings of the 2019 miniKanren and Relational Programming Workshop, pp. 1-15. 2019.
- Hu, Zhiqiang, Chen Sun, Kuang-chen Lu, Xixia Chu, Yue Zhao, Jinyuan Lu, Jianxin Shi, and Chaochun Wei. *EUPAN enables pan-genome studies of a large number of eukaryotic genomes.* Bioinformatics 33, no. 15 (2017): 2408-2409.
- Sun, Chen, Zhiqiang Hu, Tianqing Zheng, Kuangchen Lu, Yue Zhao, Wensheng Wang, Jianxin Shi, Chunchao Wang, Jinyuan Lu, Dabing Zhang, Zhikang Li, Chaochun Wei. *RPAN: rice pan-genome browser for ~3000 rice genomes.* Nucleic acids research 45, no. 2 (2017): 597-605.

Teaching Experience

- **CSCI 1730 Programming Languages**, Brown University: Fall 2021, Fall 2022, Fall 2023, Fall 2024
- **CSCI 1260 Compilers**, Brown University: Spring 2021
- **C311/B521 Programming Languages**, Indiana University: Spring 2019, Fall 2019

Professional Experience

- **R&D Intern**, RelationalAI, Remote May 2023 – Aug. 2023

PREFACE

Shriram observed that modern programming languages share a common semantic core, which he called SMoL—the Standard Model of Languages. I resonate with this idea—learning a new language often feels like "just learning the new syntax". Many programmers likely share this intuition, as evidenced by the abundance of resources that teach one language by comparing its syntax to another.

When I joined Shriram's group in 2020, I became increasingly interested in grounding my PL research more firmly in human concerns. Programming languages are mathematical artifacts, created with arbitrary design choices. How, then, do we determine which PL research directions are valuable? Since this question first crossed my mind, my answer has always been: humans. This led me to focus on teaching the semantic core, which eventually became the central theme of my PhD.

This dissertation represents the culmination of my efforts in this direction: a structured study of programming misconceptions, a tracing tool designed to make language behavior more transparent, and an automated tutor aimed at diagnosing and correcting misunderstandings.

All work in this dissertation was done jointly with my advisor. Throughout this dissertation, 'we' and 'us' include the reader to foster a shared perspective, while 'I' and 'me' refer to both my advisor and me.

ACKNOWLEDGMENTS

Thanks to Shriram for his guidance and his support in both research and life.

Thanks to Kathi Fisler and Ben Greenman for close collaboration on many projects.

Thanks to Will Crichton for inspiring conversations, including those that helped develop the idea of misinterpreters.

Thanks to my PhD peers—Elijah, Gavin, Siddhartha, Skyler, and Yanyan—for their suggestions and support.

Thanks to Sorawee Porncharoenwase and Mark Guzdial for many thoughtful exchanges.

Thanks to Tom Van Cutsem, Denis Carnier, Filip Strömbäck, Pontus Haglund, Edwin Flórez-Gómez, David Bremner, John Clements, Srikumar Subramanian, Paulo Carvalho, Suzanne Rivoire, Kimball Germane, Lukas Ahrenberg, Otto Seppälä, Juha Sorva, and Kathi Fisler for using or attempting to use my systems, and for sharing datasets and feedback.

Thanks to the students and friends who used my systems and provided encouraging, insightful feedback.

Thanks to Sam Saarinen for creating Quizius.

Thanks to the NSF for partially funding this work.

Thanks to Dan Friedman for introducing me to the world of Programming Languages.

Thanks to Lumeng for companionship, joy, and love, and to my parents for their unconditional support throughout my life.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF ILLUSTRATIONS

CHAPTER 1

INTRODUCTION

1.1 The Standard Model of Languages (SMoL)

A large number of widely used modern programming languages behave in a common way:

- Variables are lexically scoped.
- Expressions are evaluated eagerly and sequentially (per thread).
- Mutable data structures (e.g., arrays) are aliased (at least by default).
- Mutable variables are *not* aliased (at least by default).
- Some higher-order values (e.g., functions and objects) are first-class.
- First-class higher-order values can close over (variable) bindings.

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 [42]) and deviation (such as the documented semantic oddities of JavaScript and Python; see Sections 8.2.1 and 10.8 for further discussion). 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 work I choose to call this the **Standard Model of Languages (SMoL)**.

1.2 Why Should You Care About Teaching SMoL?

Unfortunately, this combination of features appears to be non-trivial to understand. CS-major students and even professional programmers do not understand these behaviors well. In Section 5.2, I discuss how multiple researchers, across different countries and various post-secondary educational contexts, have studied students' difficulties with scope and state. They consistently find that even advanced students struggle with brief programs involving SMoL topics. Similar confusions are also common on platforms like StackOverflow [28, 84, 86].

Not understanding SMoL can lead to costly consequences: Related bugs can easily take hours to fix; Students not understanding SMoL cannot possibly understand advanced topics (e.g., asynchronous functions, generators, threads, and ownership), which often rely on combination of these behaviors.

Given the prevalence and persistence of SMoL misconceptions—and the steep cost of holding them—it is crucial to better align human understanding with common language behavior. While language design improvements might reduce the likelihood of SMoL misunderstandings, the pervasiveness of these behaviors makes it unlikely that programming languages will undergo substantial changes. Therefore, it is essential to teach SMoL effectively and efficiently.

1.3 Thesis Statement and Contributions

The thesis of my dissertation is:

> Misconceptions about SMoL behavior[1] are widespread, but a tutoring system with carefully designed questions and feedback can effectively correct them.

At the heart of this dissertation is the **SMoL Tutor** (Chapter 4), an interactive, self-paced tutorial designed to correct misunderstandings about SMoL. The Tutor draws on concepts from cognitive and educational psychology to explicitly identify and address misconceptions.

Chapter 5 presents the **misconceptions** targeted by the Tutor and describes how this collection was developed.

The Tutor incorporates **Stacker** (Chapter 6), a tool for tracing the execution of SMoL programs. The Stacker is also useful as a standalone visualization tool.

Since SMoL is a cross-language concept, teaching it effectively requires presenting essentially the same program in multiple languages. If students are exposed to only one, they may fixate on its syntax, mistakenly believe their learning is confined to it, or struggle to transfer concepts across languages. To support multilingual learning, I introduce the **SMoL Language** (Chapter 3), along with the **SMoL Translator** (Chapter 8), a tool that converts SMoL programs into several commonly used programming languages. Both the Tutor and Stacker rely on the Language and Translator to provide multilingual support. To distinguish between SMoL (Section 1.1) and the SMoL Language, I sometimes refer to the former as **SMoL Characteristics**.

Chapter 7 introduces **misinterpreters**, a novel way of representing misconceptions that makes working with them much more scalable.

[1] I use the term "behavior" to refer to the meaning of programs in terms of the answers they produce. A more standard term would be "semantics." However, the term "semantics tutor" might mislead some readers into thinking it teaches people to read or write a formal semantics, such as an introduction to "Greek" notation. Because that is not the kind of tutor I am describing, I use the term "behavior" instead to avoid confusion.

The remaining chapters introduce relevant terminology (Chapter 2), describe the study populations involved in the user studies of the Tutor and the Stacker (Chapter 9), and provide an overall discussion (Chapter 10) and conclusion (Chapter 11).

An artifact containing all the aforementioned systems is available at:

`https://cs.brown.edu/research/pubs/theses/phd/2025/Kuang-Chen_Lu.zip`

CHAPTER 2

BACKGROUND

This chapter provides background information and discusses related work that informs this dissertation. It covers key concepts in programming languages, tracing, rules of program behavior, misconceptions, notional machines, tutoring systems, and pedagogic techniques.

2.1 Misconceptions and Mistakes

A **misconception** is an incorrect mental model, whereas a **mistake** is an incorrect action. Mistakes do not necessarily indicate misconceptions, as they can occur for various incidental reasons (e.g., a typo or a misclick). However, if students consistently make the same kind of mistake or provide reasoning that suggests a flawed understanding, the students likely have a misconception.

This dissertation addresses misconceptions about programming language behavior by identifying, defining, and correcting them.

2.2 Tutoring Systems

There is an extensive body of literature on tutoring systems ([85] 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 [85] where the outer loop iterates through "tasks" (i.e., educational activities) and the inner loop iterates through user interface (UI) events within a task. I follow the same structure in the SMoL Tutor (Chapter 4).

Many tutoring systems focus on teaching programming (such as the well-known and heavily studied LISP Tutor [4]), and in the process undoubtedly address some program behavior misconceptions. The 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. I am not aware of a tutoring system (for programming concepts) that has this specific design.

2.3 Programming Language Concepts

In the Programming Languages community, **semantics** refers to the formal description of meaning.

Some languages have a non-trivial static phase, such as the type systems found in languages like Java, OCaml, and TypeScript. In this dissertation, I focus solely on the **dynamics** of programming languages—that is, how programs are executed [35]—and do not study their statics. Chapter 10 discusses potential future work that could extend my system to also teach statics.

States and Transitions There are multiple ways to define dynamics. A commonly used approach is structural dynamics (also known as structural operational semantics or small-step semantics), which describes execution as a sequence of **states** and **transitions** between

them [35]. Dynamics describe how programs generally execute. Therefore, structural dynamics describe rules of forming states and transitions rather than their concrete instances.

This dissertation, for the purpose of teaching dynamics, presents a tool (Chapter 6) that explains how program executes by presenting states and transitions.

Interepreters Another approach to define dynamics is **interpreters**, which are a program that executes other programs.

This dissertation presents several interpreters for defining the correct dynamics as well as incorrect dynamics (Chapter 7).

2.4 Tracing and Program Trace Visualization Tools

A **trace** is a representation of a program's execution, showing its **states** and the **transitions** (or **steps**) between them. The process of producing a trace is called **tracing**.

While tracing can be done manually, many programming language communities provide tools that automate the process. These tools typically offer interactive visualizations of program traces and are commonly referred to as **Program Visualization Tools** (e.g., the tool by Cai et al. [7], Online Python Tutor [33], UUhistle [75], VILLE [63], and Jeliot 3 [45]).

Although widely used, this term is not always precise. I refer to these tools—along with the Stacker (introduced in Chapter 6)—as **Program Trace Visualization Tools**. As Rajala et al. [63] noted, program visualization tools do not necessarily depict program traces (italic in original):

> Unlike *dynamic* visualization tools …, static tools don't visualize program execution step by step, but instead focus on visualizing program structure and the relations between program components.

Comparing dynamic tools that visualize execution traces to static visualization tools is not particularly meaningful, as they represent fundamentally different types of information.

2.5 Notional Machines

The term "Notional Machine" was introduced by Du Boulay, O'Shea, and Monk [20], who defined it as "the idealized model of the computer implied by the constructs of the programming language". Later, Du Boulay [19] described it as "the general properties of the [physical] machine that one is learning to control" in the context of students learning programming.

Early definitions suggest that a notional machine focuses on semantics. However, later work blurs the distinction between semantics and other aspects of physical machine behavior. For example, Du Boulay [19] includes confusion about terminal displays as a misunderstanding of the notional machine, which extends beyond semantics:

> …it is often unclear to a new user whether the information on the terminal screen is a record of prior interactions between the user and the computer or a window onto some part of the machine's innards.

More recent works (summarized in [41, 64, 73]) typically restrict the term to semantics. This dissertation focuses specifically on the semantics aspect of notional machines.

Duran, Sorva, and Seppälä [21] introduces a related concept, *Rules of Program Behavior* (RPBs), defined as "written statements about how computer programs of a particular kind behave when executed". RPBs specify learning objectives. The authors note that the term "notional machine" is often used inconsistently, with many papers labeled as notional machine research actually focusing on tracing, RPBs, or misconceptions.

To avoid confusion, this dissertation minimizes the use of the term "notional machine" and instead uses more specific terms (e.g., "program trace visualization tools" and "RPBs") where applicable.

2.6 Pedagogic Techniques

The Tutor (Chapter 4) is firmly grounded in one technique from cognitive and educational psychology. The fundamental problem is: how do you tackle a misconception? One approach is to present "only the right answer", for fear that discussing the wrong conception might actually reinforce it. Instead, there is a body of literature starting from [57] 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 (see [67, 79] for summary) have shown their effectiveness in multiple other domains.

The Tutor's content structure is also influenced by work on **case comparisons** (which draws analogies between examples). [3] 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.

CHAPTER 3

THE SMOL LANGUAGE

This chapter presents the **SMoL Language**, a programming language designed to illustrate SMoL Characteristics. Rather than reusing an *existing* language, I designed a *new* one because, although SMoL is a representative model, no existing language fully aligns with it (Section 1.1).

The syntax of the Language is presented in Figure 3.1, where

- `s` represents statements,
- `d` represents definitions,
- `e` represents expressions,
- `c` represents constants (i.e., numbers and booleans),
- `x` represents identifiers (variables), and
- `o` represents primitive operators.

The `...` symbol indicates that the preceding syntactic unit may be repeated zero or more times. For example, a conditional can have zero or more branches, each consisting of a

Figure 3.1: The syntax of the SMoL Language

```
s ::= d
    | e
d ::= (defvar x e)                          ;; variable definition
    | (deffun (x x ...) body)                ;; function definition
e ::= c
    | x
    | (set! x e)                            ;; variable mutation
    | (and e ...)
    | (or e ...)
    | (if e e e)
    | (cond [e body] ... [else body])        ;; conditional
    | (cond [e body] ...)                    ;; conditional w/o 'else'
    | (begin e ...)                          ;; the comma (,) operator
    | (lambda (x ...) body)                  ;; anonymous function
    | (let ([x e] ...) body)                 ;; let expression
    | (let* ([x e] ...) body)                ;; nested 'let'
    | (letrec ([x e] ...) body)              ;; recursive 'let'
    | (o e ...)                              ;; primitive operation
    | (e e ...)                              ;; function call
body    ::= s ... e
program ::= s ...
```

Table 3.1: Primitive operators in the SMoL Language

Operator	**Inputs**	**Output**	**Meaning**
`+ - * /`	Int	Int	Arithmetic
`< <= > >=`	Int × Int	Bool	Compare numbers
`mvec`	Any ...	Vec	Make a vector
`vec-len`	Vec	Int	Get the length of the vector
`vec-ref`	Vec × Int	Any	Get an element
`vec-set!`	Vec × Int × Any	None	Replace an element
`mpair`	Any × Any	Vec	Make a 2-element vector
`left`	Vec	Any	Get the first element
`right`	Vec	Any	Get the second element
`set-left!`	Vec × Any	None	Replace the first element
`set-right!`	Vec × Any	None	Replace the second element
`=`	Any × Any	Bool	Structural equality on boolean and numbers, and pointer equality on vectors

condition (e) and a body. For convenience, I collectively refer to `programs` and `bodys` as **blocks**.

Most syntax rules in Figure 3.1 are common to modern programming languages. However, the SMoL Language includes three types of `let` expressions, which are more characteristic of Lisp-derived languages. While they are not the primary focus of this dissertation, interested readers can refer to external resources such as Racket's documentation for more details:

docs.racket-lang.org/reference/let.html

The parenthetical syntax in the SMoL Language originates from the course where I conducted my earlier study. Moreover, this syntax offers several benefits:

- Presenting the same program in multiple syntaxes likely aids learning. I suspect that incorporating additional syntaxes—especially those that differ significantly from existing ones—could further enhance this effect.
- Parenthetical syntax is easy for computers to parse, reducing the cost of developing tools (e.g., the translator described in Section 8.1) for processing SMoL programs.
- Parenthetical syntax may be helpful when discussing scope alongside local-binding features like `let`. It avoids the confusing "variable lifting" semantics found in languages like Python [55], where the actual binding range of a variable is not always apparent from the source code (see also posts like [55]).

The SMoL Language includes a limited set of primitive operators (Table 3.1) for working with Booleans, numbers, and vectors. To clarify the table:

- "Any" refers to "any values", which corresponds to the "top" type in some typed language (e.g., Java's `Object` type).
- "Vectors", also known as *arrays*, are fixed-length mutable data structures.[1]

[1] "Arrays" is likely a better name, as it avoids confusion with "vectors" in mathematics and physics.

- "None" is equivalent to *void* or *unit*.
- The "..." symbol denotes "zero or more" arguments. For instance, the operator `mvec` can take an arbitrary number of inputs.

The semantics of the SMoL Language is the SMoL Characteristics (section 1.1) plus the following details:

- Arguments are evaluated from left to right.
- Function parameters are considered declared in the same block as function bodies.
- If a name is declared twice in one block, the program outputs an error.
- Primitive operators are not defined for non-integer numbers. Division is only defined if the first operand is a multiple of the second operand, in which case the quotient is returned, or if the second operand is zero, in which case division raises an error.
- Primitive vector operators expect index arguments to be within-range integers. Using out-of-range integers as indexes triggers errors.

A definitional interpreter for the SMoL Language is provided in Appendix A.

CHAPTER 4

SMOL TUTOR

This chapter introduces the SMoL Tutor, an interactive, self-paced tutoring system designed to address SMoL misconceptions. Section 4.1 provides a guided tour of the system. Section 4.2 summarizes the user studies conducted. Sections 4.3 to 4.5 present the results of these studies from multiple perspectives, demonstrating the Tutor's effectiveness in identifying and correcting misconceptions. Finally, Section 4.6 explores the design space of the system.

4.1 A Guided Tour of SMoL Tutor

4.1.1 Choosing a Syntax

When users open the Tutor, they first encounter a dialog prompting them to select their preferred *primary syntax* (Figure 4.1). The dropdown menu lists the syntaxes supported by the SMoL Translator (Section 8.1). Changing the syntax updates the example program accordingly. The default syntax, and whether the menu appears at all, are configurable (Section 4.1.6).

Figure 4.1: The SMoL Tutor window for selecting the primary syntax

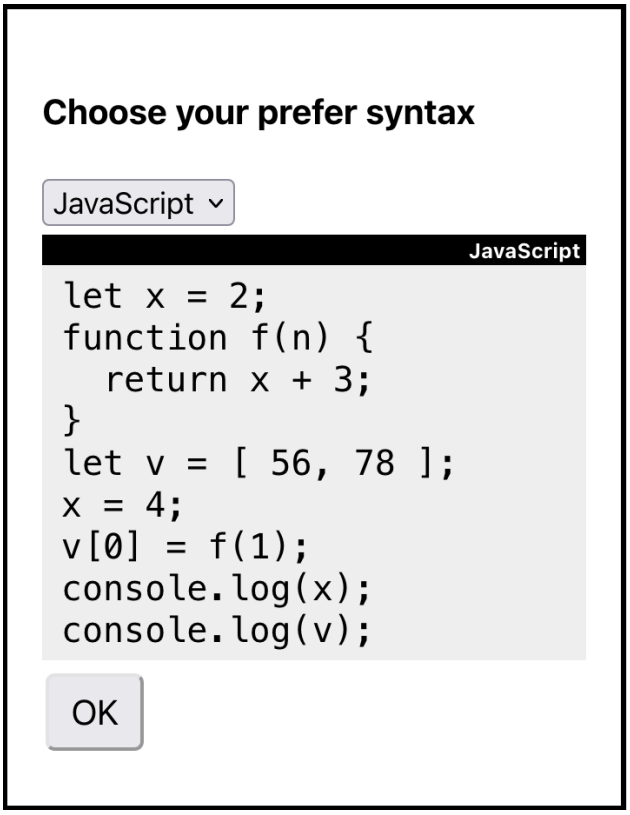


Table 4.1: SMoL Tutor topics and learning objectives

DEF{1-3}: variable and function definitions. • Define "blocks" • Evaluating an undefined variable is an error. • Variables are lexically scoped. • Define "scope" • Eager and sequential evaluation.
VECTORS{1-3}: mutable data structures. • Vectors can be aliased. • Define "heap" and "heap addresses". • Contrast heap and variable bindings.
MUTVARS{1-2}: mutable variables. • Variables are not aliased. • Contrast variable assignments and variable definitions, and clarify how variable mutation interacts with functions. • Define "environments", and contrast environments and blocks.
LAMBDA{1-3}: first-class functions and lambda expressions. • Functions are first-class values. • Define "closures", and clarify how first-class functions interact with environments. • Define "lambda expressions". • Clarify the relation between function definitions and lambda expressions.

4.1.2 Choosing a Tutorial

After selecting a syntax, users must also choose a tutorial. The Tutor covers five major topics, listed with their abbreviated learning objectives in Table 4.1. Each topic is further divided into 2-3 tutorials, named using labels such as DEF1. The intended duration for each tutorial was 20-30 minutes, but my data show that students typically spent between 9 and 30 minutes (median). Additionally, after each topic, the Tutor provides a review tutorial (POST{1-4}) covering previously learned concepts and an experimental tutorial (REFACTOR) exploring a new question format (Section 4.1.4).

The learning objectives of the tutorials are detailed in Appendix C. They extend SMoL Characteristics (Section 1.1) (e.g., detail the recursive look up of lexical scope), introduce concepts (e.g., heap, heap addresses, and environments), and contrast similar concepts (e.g., "Creating a vector does not inherently create a [variable] binding").

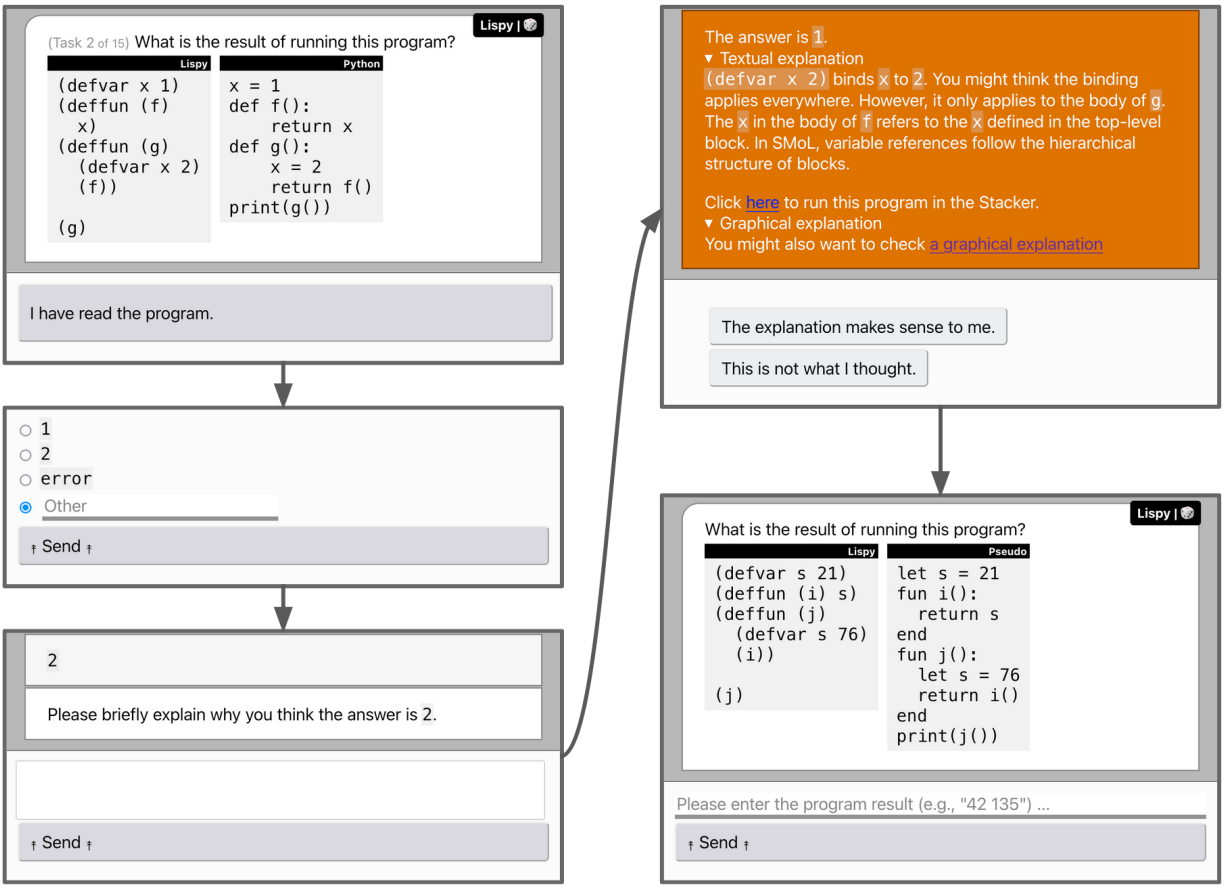
4.1.3 Interpreting Tasks

Each tutorial consists of a sequence of tasks, most of which are **interpreting tasks**. These tasks (illustrated in Figure 4.2) begin with multiple-choice questions (MCQs) designed to detect misconceptions.

The MCQs present programs in two syntaxes: the primary syntax, chosen through the process described in Section 4.1.1, and a randomly selected *secondary syntax*, which can be changed by clicking the black button displayed at the top right of the task.

In addition to answering the MCQs, students may be asked to justify their choices, as shown in the figure. After selecting an answer, they receive feedback. If a student answers incorrectly, they (a) receive an explanation addressing the misconception associated with their choice (or a generic explanation if no specific misconception applies), (b) evaluate the explanation by flagging if the explanation make sense to them (if it doesn't make sense, they can optionally provide comment), and (c) are then presented with a slightly modified

Figure 4.2: An interpreting task in the SMoL Tutor



follow-up question.

Each interpreting task has one role in the Tutor. It can be either a **warm-up** task, a **teaching** task, or a **post-test** task. Most tasks are teaching tasks. Warm-ups are designed to be simple and for introducing new syntax. There is one warm-up for each language construct. Post-tests are tasks in the POST tutorials.

Students are not always asked for a justification. I expect writing justification can be stressful and time-consuming for students. Therefore, the Tutor asks for justification only occasionally, and does so at most twice in each tutorial. The Tutor might ask for justification even when the answer is correct, but it is more likely to do so when the answer is incorrect. The Tutor does not collect justification in warm-ups and post-tests.

A misconception-targeting explanation is always presented as a *refutation text* (Sec-

tion 2.6), followed by a link to open a Stacker trace. Some explanations also include a link to a specialized version of the Stacker that contrasts the correct trace with an incorrect one.

The second question is deliberately designed to reinforce learning:

- The program remains semantically identical to the first but undergoes superficial changes (e.g., variable names, constants, and operators) to prevent students from relying on pattern recognition.
- 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 wanted to force reflection on the explanation just given, whereas with an MCQ, students could just guess. Second, I felt 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.

Consecutive interpreting tasks within each tutorial are presented in a randomized order.

4.1.4 Equivalence Tasks

Interpreting tasks ask students to evaluate programs. Although such tasks are commonly used to detect misconceptions, they lack authenticity: in real programming scenarios, people rarely evaluate programs mentally—there is usually no need, since they can simply run the code to observe its output. By contrast, it is not uncommon for programmers to compare two versions of code fragments during refactoring, code review, or similar activities. Equivalence tasks are designed to simulate these situations.

In addition to being more authentic, equivalence tasks also *require* students to actually read code. Unlike interpreting tasks, which involve complete programs, equivalence tasks

Figure 4.3: An equivalence task in the SMoL Tutor

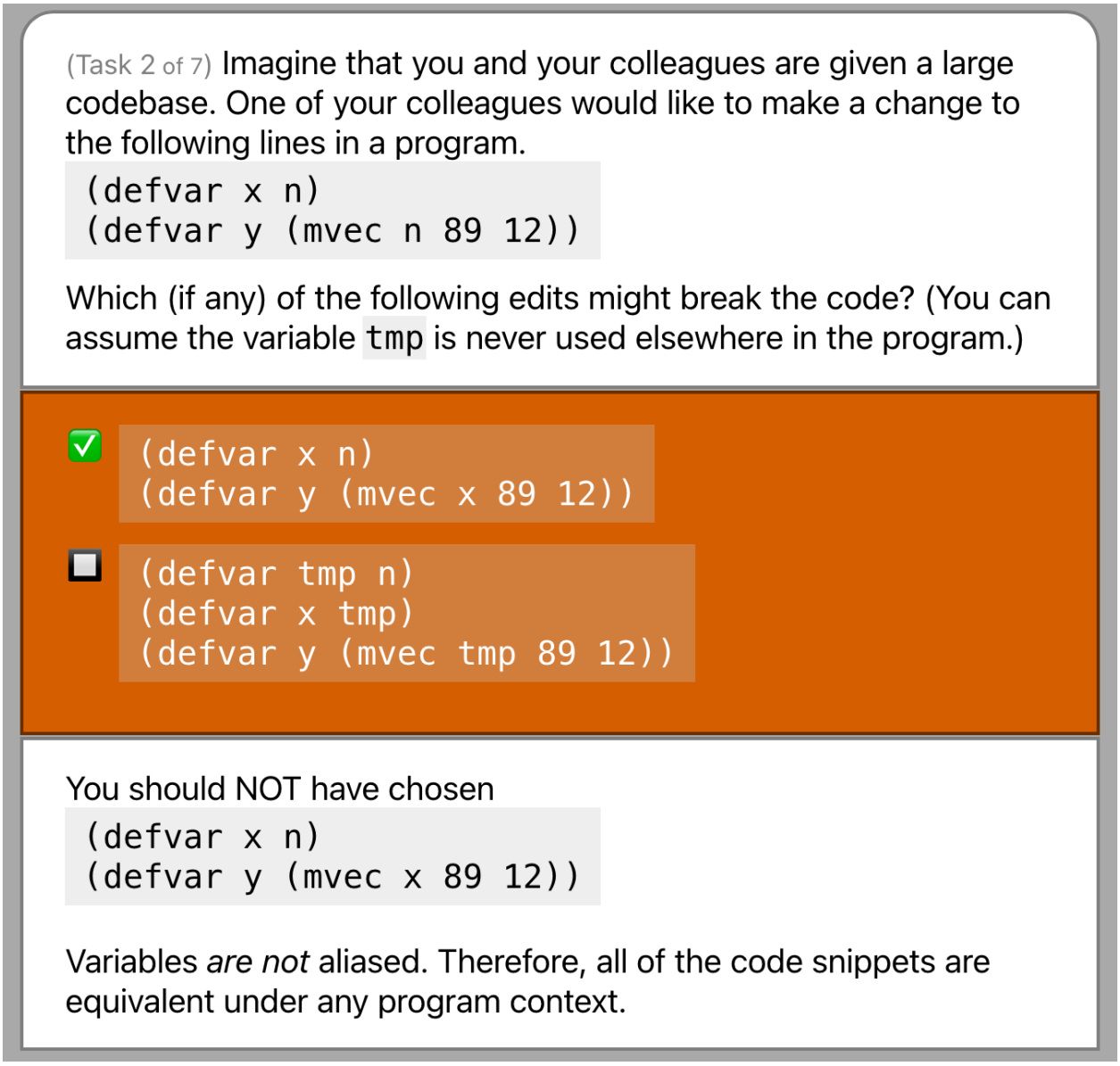
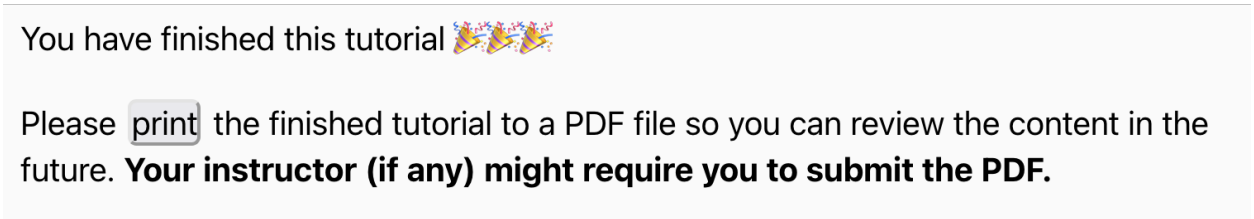


Figure 4.4: The message displayed by the SMoL Tutor at the end of each tutorial, prompting students to save a copy of their completed work.



focus on program fragments. As a result, students cannot simply run the code to determine the answer, which presumably encourages deeper reflection on the semantics of the fragments.

Figure 4.3 illustrates a completed equivalence task. Each task presents an *original* program fragment and a collection of *alternative* fragments, then asks students to select those that "*might break the code*". In the example shown, none of the alternatives break the code, assuming the `tmp` variable is not otherwise used. Thus, selecting the first option is incorrect, as indicated by the orange background. In such cases, the Tutor provides feedback: it first lists which options should or should not have been selected, then explains the reasoning behind the correct choices.

There are six equivalence tasks, each targeting a different aliasing misconception (Table 5.1). They are presented in random order within the REFACTOR tutorial.

Several prior studies have explored related ideas, but none specifically examine students' understanding of language behavior. Izu and Mirolo [37], like my work, asked students to judge whether program fragments were equivalent. However, their focus was on students' understanding of program *purpose*, rather than the behavior of specific language constructs. Both Gal-Ezer and Zur [29] and Krishnamurthi et al. [40] asked students to compare semantically equivalent fragments—some more performant than others—to investigate students' understanding of runtime costs. Oliveira, Keuning, and Jeuring [49] studied student-made errors while attempting to "improve" already-correct programs. Unlike other related work, their students could make arbitrary edits. The study revealed a wide range of misconceptions, from incorrect arithmetic to improper composition of control-flow constructs.

4.1.5 Other Components of the SMoL Tutor

The Tutor includes additional interactions to ensure a smooth learning experience:

- Each tutorial begins with a brief introduction outlining its purpose.
- Whenever a new language construct is introduced, the Tutor provides examples illustrating its typical usage.
- Learning objectives are presented at the *end* of each tutorial, for several reasons: (1) to avoid intimidating students with technical vocabulary upfront; (2) to prevent interference with students' existing conceptions (e.g., if they already associate the word "heap" with a different meaning); and (3) to potentially improve learning outcomes (see, for example, [68]).
- After completing a tutorial, students are encouraged to download a PDF copy for review (Figure 4.4). This feature was added in response to feedback from early student users, who wanted a way to revisit the material without redoing the entire tutorial.

The Tutor also includes special tasks beyond interpreting tasks (Section 4.1.3) and equivalence tasks (Section 4.1.4):

- In the vectors tutorials, students are shown programs and asked to determine their heap content.
- Some tutorials prompt students to reflect on their interpreting tasks and summarize their key takeaways.
- Other tutorials require students to analyze a Stacker trace and identify the trace state that best explains why the program produces the correct output rather than an incorrect one.

4.1.6 Hosting the SMoL Tutor

Instructors can host their own copy of the SMoL Tutor. Several collaborators have done so successfully without needing my help. When self-hosting, instructors have access to several configuration options:

Syntax Instructors can specify the suggested primary syntax and whether it should be enforced. If it is enforced, the dialog window described in Section 4.1.1 won't appear.

Available Tutorials Instructors can choose which tutorials are available to students.

Logging Instructors can insert a custom JavaScript function (or pick a predefined one) to log Tutor data to a spreadsheet or database. Instructions are provided for logging to a Google Sheet, and several collaborators have followed them successfully.

4.2 SMoL Tutor Versions and Studies

Multiple studies have been conducted using different versions of the SMoL Tutor. Some versions introduced significant updates (see Section 4.2.2) that make it difficult to compare results across versions. Therefore, this dissertation focuses on studies conducted using the two most recent versions: "Early 2024" and "Mid 2024".

4.2.1 Studies

All population names mentioned below are defined in Chapter 9.

The Mid 2024 Tutor was used in the PL course, which had 46 enrolled students. All completed the Tutor.

The same version was also deployed at the Belgium, where 126 students completed it.

The Early 2024 Tutor was used at US2 (13 students) and at the Sweden (56 students). For the Swedish population, Python was configured as the primary language.

4.2.2 Versions

The SMoL Tutor typically receives a major update about twice a year. I refer to each major version by its release window: "Mid 2022", "Early 2023", "Mid 2023", and so on. Minor updates—such as bug fixes, English refinements, and small tweaks to the interface or question wording—are made throughout the year. This section focuses on major updates.

Mid 2024

Introduced equivalence tasks (Section 4.1.4).

Early 2024

Substantially revised the question sets so that each incorrect answer maps to exactly one misconception (see Section 7.2).

Added randomization of question order (see Section 4.6.5 for a comparison with fixed order).

Introduced a two-language display mode. Previously, the Tutor displayed programs in only one language at a time, with a manual toggle. The new version shows programs in a fixed primary language alongside a randomly selected secondary language. I hope this design helps emphasize that the Tutor teaches cross-language concepts.

Added experimental support for Scala 3.

Previously, the Tutor did not require students to justify their answers. The current version does, allowing for deeper insights into whether students genuinely hold the anticipated misconceptions.

Mid 2023

Added language-switch buttons. Previously, the Tutor displayed programs only in the SMoL syntax. This change aligns with the reasoning provided in Chapter 3.

Introduced the web-based Stacker, allowing students to trace program execution by clicking a "Run" button. Previously, students had to copy programs into DrRacket and run them manually, which was less convenient.

Early 2023

Made numerous incremental improvements and addressed issues in the initial prototype.

Mid 2022

The initial release.

4.3 Evaluation: Coverage of Named Misconceptions

This section addresses two questions: How many wrong answers in interpreting tasks are associated with named misconceptions? And what patterns emerge among the remaining answers?

4.3.1 Association with Named Misconceptions

Table 4.2 shows that students answered correctly more than half the time. (See Chapter 9 for details on each population.) In most institutions—US2 being the exception—between 75% and 80% of wrong answers were associated with a named misconception. Readers should be cautious about interpreting the numbers for US2, as that group is much smaller than the

Table 4.2: Distribution of answers to interpreting tasks by conception type. Percentages may not sum to 100% due to rounding.

	Misconception-Associated	**Other Incorrect**	**Correct**
US1 (Mid 2024)	10%	2%	87%
Belgium (Mid 2024)	12%	3%	85%
US2 (Early 2024)	12%	29%	59%
Sweden (Early 2024)	17%	6%	77%

others (13 students vs. 46–126). A Fisher's exact test confirms that the difference between US2 and the other populations is not statistically significant ($p = 0.3329$).

4.3.2 Unassociated Wrong Answers

Wrong answers not linked to a named misconception may suggest new misconceptions. To explore this possibility, I examined a subset of the corresponding programs and students' justifications. The subset was selected using the following procedure:

1. Compute the frequency of each wrong answer.
2. If the most frequent answer occurs at least 1.5× as often as the second-most frequent, include it in the subset and repeat the process with the remaining answers. Otherwise, stop.

This process was applied to each population individually. However, the selected results are presented together, as many common wrong answers appeared across populations.

For each example below, all available student justifications are included. These are often sparse, as students were not always asked to explain their answers (see Section 4.1.3).

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

Some students selected `3`. Justifications included:

- *I am wrong it should error because y is out of scope in print statement*
- *Because y is not bound in the print call*

The first student likely selected the wrong answer by mistake. The second correctly noted that `y` is unbound, but still selected `3`—perhaps believing that unbound variables default to `0` or are silently ignored.

```
(defvar x 1)
(deffun (f)
  x)
(deffun (g)
  (defvar x 2)
  (f))
(g)
```

Some students selected `error`. Justification:

x in f undefined

The student may have overlooked the top-level definition of `x` or may have believed that functions cannot refer to variables defined outside their own block.

```
(defvar x 1)
(deffun (f y)
  (deffun (g)
    (defvar z 2)
    (+ x y z))
  (g))
(+ (f 3) 4)
```

Some students selected `error`. Justifications included:

- *in function (+ x y z) is not defined, it is binary operation*
- *because definition g does not have access to y*

The first justification does not seem to indicate a misconception, rather, it indicates a miscommunication between the Tutor and the student. The second justification might indicate

a misconception, but it is unclear to me what the misconception is due to the brevity of the justification.

```
(defvar v (mvec (mvec 58 43) (mvec 43 66)))
(defvar vr (vec-ref v 1))
(vec-ref vr 0)
```

Some students selected `58`, suggesting they treated index `1` as `0`. No justification was collected, as this was a warm-up task (see Section 4.1.3).

```
(defvar x (mvec 92 73))
(vec-set! x 0 67)
x
```

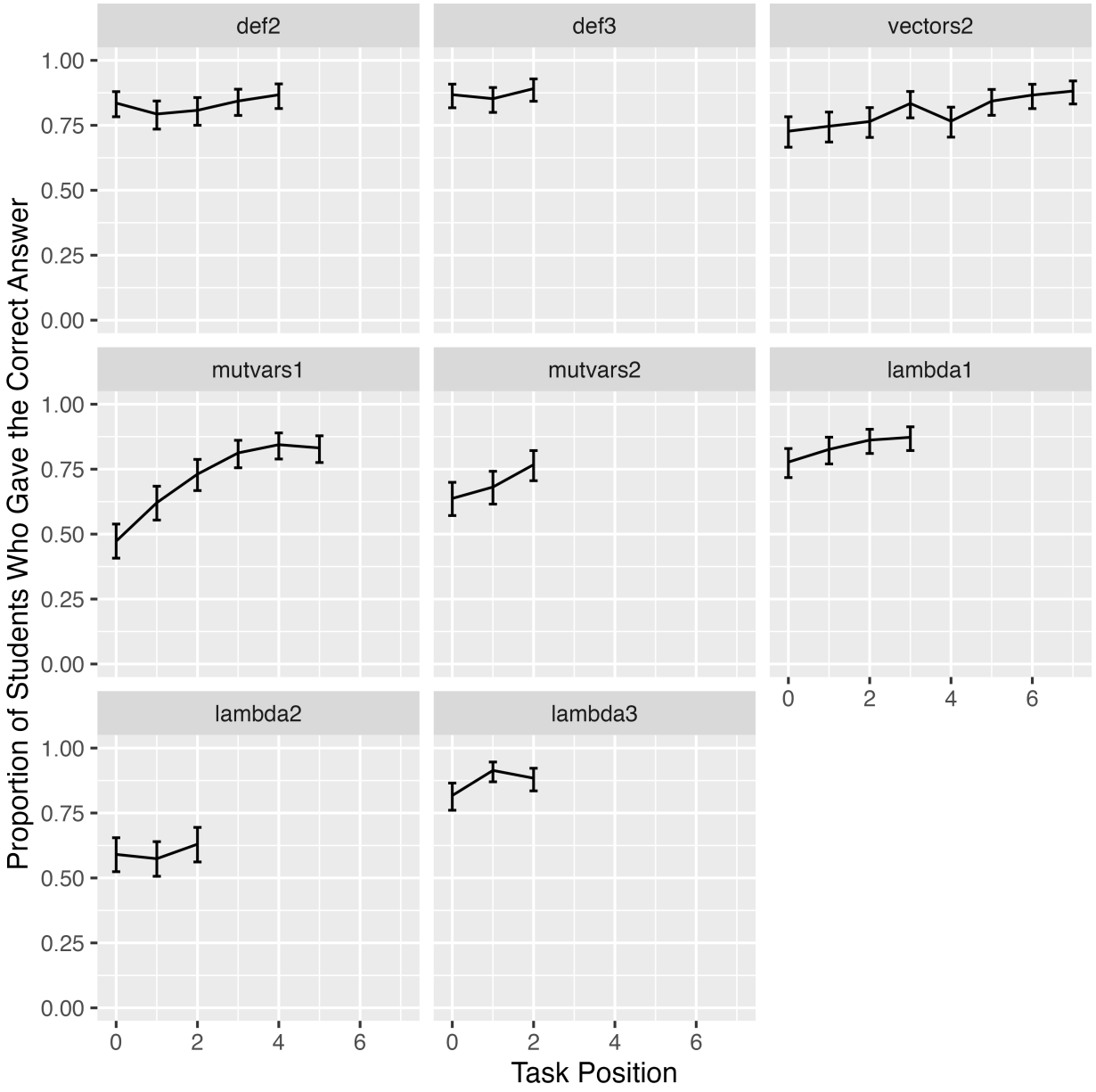
Some students selected `67 73` (omitting brackets) instead of the correct answer `#(67 73)`. Again, no justification was collected, as this was also a warm-up task.

4.4 Evaluation: How Effective is SMoL Tutor?

This section answers the question from two perspectives. If the Tutor is effective, then as students complete more tasks, they should (1) become more likely to give the correct answer, and (2) for each misconception, become less likely to select the corresponding incorrect answers. Technically, only the first perspective is necessary to demonstrate effectiveness. However, the second perspective provides especially useful insights:

- Ideally, the Tutor is effective because it helps correct misconceptions. It's possible, however, that students improve at giving correct answers without improving at avoiding misconception-related wrong answers—perhaps by selecting fewer "random" incorrect options. If this happens, the Tutor is not effective in the way we expect. The second perspective is needed to rule out this possibility.

Figure 4.5: Students seem more likely to give the correct answer in later tasks.



- The design of each Tutor task is based on the underlying misconceptions. The second perspective provides more detailed feedback on how well the Tutor addresses each misconception. For example, if the Tutor is generally effective except for a few misconceptions, we might suspect issues with the associated programs or refutation texts.

All analyses in this section aggregate data from the four populations.

4.4.1 Convergence to Correct Answers

Figure 4.5 shows how the proportion of students who gave the correct answer changes as they progress through each tutorial. It suggests that students improve. However, the improvement may not be clear in every tutorial. Therefore, I performed a statistical analysis to test for trends.

A straightforward approach would use logistic regression to model the relationship between task position and correctness. However, there are a few additional considerations:

1. As students complete more tasks, they may become more familiar with the UI. We need to verify that improvement is due to conceptual learning rather than just learning how to use the system.

2. The difficulty of interpreting tasks varies, and question order is randomized. If difficult tasks tend to appear earlier for most students, this could create a misleading impression of learning.

The first concern seems unlikely. If students were merely learning the UI, we would expect a clear upward trend *across tutorials* in Figure 4.5. However, no such trend is visible. In fact, logistic regression shows a slightly negative trend, with an effect size of -0.012 ($p < 0.001$).

The second concern—variation in task difficulty—remains. To address this, I compare three models for each tutorial:

Task-Specific Performance depends only on the task. This reflects the concern.

Position-Specific Performance depends only on task order. If this model is best, it suggests students improve over time regardless of task content.

Task + Position Performance depends on both task and position.

To find the most plausible model, I compute and compare the *Akaike Information Criterion (AIC)*. Intuitively, AIC measures how well a model fits the data while penalizing model complexity; lower values indicate a better fit. A common rule of thumb is that an AIC difference of at least 2 is needed to consider one model clearly better than another. What I am looking for are that the **Task-Specific** AIC is never clearly the best, and that the effect of position is positive and **clear** (i.e., statistically significant).

Table 4.3 confirms this: the Task-Specific model is never best, and five out of eight tutorials show a statistically significant positive trend.

The lack of improvement in DEF2 and DEF3 may be due to a ceiling effect—students performed well from the start (Figure 4.5).

The lack of improvement in LAMBDA2 is more interesting. This is the first tutorial introducing higher-order functions with interesting interactions with scope. It targets the FLATENV and DEEPCLOSURE misconceptions. These topics are known to be difficult, so a three-task tutorial may be too short. Given the suggestive positive trend (Table 4.3), this tutorial may benefit from additional tasks.

Table 4.3: Students are more likely to give the correct answer in later tasks. The improvement is clear (i.e., statistically significant) in five out of eight tutorials.

Tutorial	***AIC***			***Effect of Position***	
	Task	**Position**	**Task + Position**	**Size**	***p*-value**
DEF2	854.37	955.55	854.66	0.082	$= 0.19$
DEF3	467.90	484.99	469.14	0.130	$= 0.39$
VECTORS2	1574.47	1632.06	1546.42	**0.154**	< 0.001
MUTVARS1	1355.31	1394.82	1240.56	**0.450**	< 0.001
MUTVARS2	748.44	765.01	742.10	**0.316**	< 0.01
LAMBDA1	700.34	743.50	691.07	**0.298**	< 0.001
LAMBDA2	710.42	844.40	711.73	0.093	$= 0.41$
LAMBDA3	435.16	477.69	432.56	**0.326**	< 0.05

Figure 4.6: Students seem to be more likely to avoid the related wrong answer in later tasks.

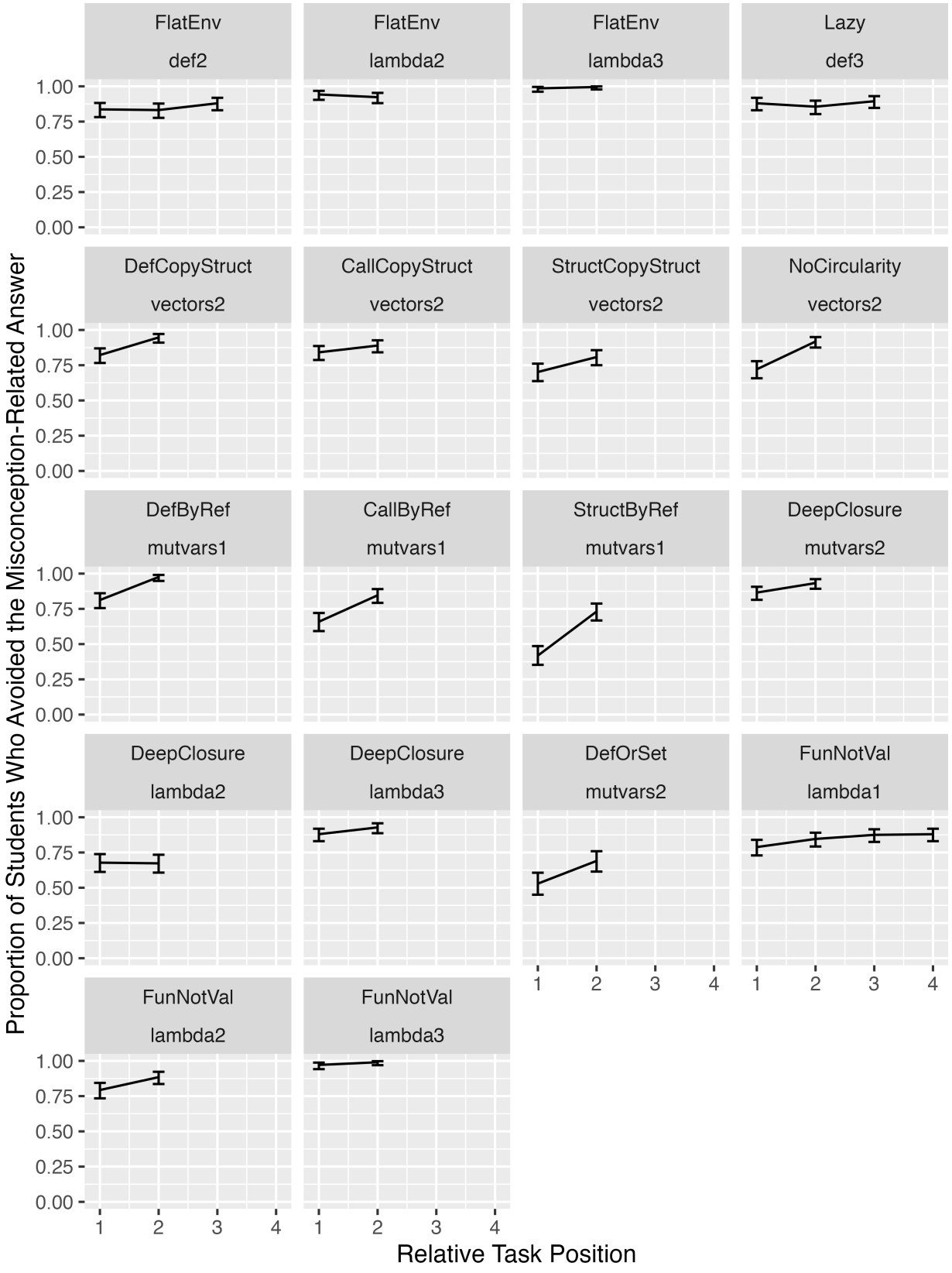


Table 4.4: Students are more likely to avoid the incorrect answer in later tasks. All clear (i.e., statistically significant) trends are improvement.

Misconception	Tutorial	*AIC*			*Effect of Position*	
		Task	Position	T + P	Size	*p*-value
FlatEnv	def2	441.47	531.40	440.80	0.248	$= 0.104$
FlatEnv	lambda2	177.25	208.57	178.52	-0.349	$= 0.392$
FlatEnv	lambda3	43.53	48.06	44.35	1.177	$= 0.312$
Lazy	def3	454.40	470.10	456.23	0.064	$= 0.675$
DefCopyStruct	vectors2	295.41	284.84	282.72	**1.287**	< 0.001
CallCopyStruct	vectors2	332.61	330.62	332.45	0.427	$= 0.144$
StructCopyStruct	vectors2	403.09	461.09	397.23	**0.699**	< 0.01
NoCircularity	vectors2	390.64	367.93	363.85	**1.483**	< 0.001
DefByRef	mutvars1	272.05	251.91	242.77	**2.212**	< 0.001
CallByRef	mutvars1	461.52	449.64	442.57	**1.084**	< 0.001
StructByRef	mutvars1	526.38	529.08	469.68	**1.748**	< 0.001
DeepClosure	mutvars2	257.09	270.94	253.38	**0.816**	< 0.05
DeepClosure	lambda2	440.18	528.34	442.17	-0.027	$= 0.907$
DeepClosure	lambda3	261.84	264.58	261.57	0.513	$= 0.137$
DefOrSet	mutvars2	410.19	410.19	403.31	**0.710**	< 0.01
FunNotVal	lambda1	663.73	707.55	655.51	**0.294**	< 0.01
FunNotVal	lambda2	353.60	364.76	349.44	**0.688**	< 0.05
FunNotVal	lambda3	77.91	80.93	77.99	1.070	$= 0.195$

4.4.2 Avoiding Misconception-Related Incorrect Answers

This section evaluates the Tutor's effectiveness from a second perspective: If the Tutor is effective, then for each misconception, students should become less likely to select misconception-related answers as they complete more tasks.

Figure 4.6 shows, for each misconception-tutorial pair, how the proportion of students who avoided the misconception changes over time.

The analysis includes a model comparison similar to Section 4.4.1. Here, I look for a negative and statistically **clear** effect—meaning students are increasingly avoiding misconception-related errors.

Table 4.4 presents the results. As expected, the Task-Specific model is never best. When the effect of position is statistically clear, it is always positive.

The unclear trends for FlatEnv–lambda3 and FunNotVal–lambda3 are likely due

to ceiling effects: students were already good at avoiding these misconceptions (Figure 4.6).

Ceiling effects don't explain the other weak trends. These cases likely require further investigation to understand how the Tutor can better address those misconception-topic combinations.

It is also worth noting that the "T + P" models are often clearly better than the Position-Specific model (i.e., the AIC of "T + P" is at least 2 points lower; see Section 4.4.1 for how to interpret AIC). This suggests that students are more likely to exhibit a misconception with some programs than others. See Section 10.2 for discussion of possible reasons.

4.5 Evaluation: What did students say about the explanations not working for them?

As illustrated in Figure 4.2, the Tutor collects student feedback to explanations in interpreting tasks. My hope was that the student feedback would be helpful for improving the Tutor, particular on its effectiveness (Section 4.4.2). This section checks if my hope is realistic by presenting some collected feedback.

As a case study, I investigate the feedback which most students flagged as *This is not what I thought.* In the four datasets, a total of 23 students did that. The second and third most flagged feedback each received 13 flags. The most flagged feedback corresponds to the wrong answer `1 1 1` to the following program or a slightly different version:

```
(deffun (foo)
  (defvar n 0)
  (deffun (bar)
    (set! n (+ n 1))
    n)
  bar)
(defvar f (foo))
```

```
(defvar g (foo))


(f)
(g)
(f)
```

The slightly different version swaps the order of the last two function calls, but does not affect the nature of the wrong answer: Either way, the wrong answer corresponds to the DeepClosure misconception. Students were given the following explanation for why their answer is wrong:

> You are right that `n` is bound to `0` when `bar` is bound to a function. You might think the function remembers the value `0`. However, `bar` does not remember the value of `n`. Rather, it remembers the environment and hence always refers to the latest value of `n`. `foo` is called twice, so two environments are created. `(f)` mutates the first, while `(g)` mutates the second. In SMoL, functions refer to the latest values of variables defined outside their definitions.

Eight of the 23 students provide meaningful response to the follow-up question "What is your thought? (Feel free to skip this question.)"

1. *n = 0 in foo() not reset the n to 0 when the second call to f is done? When f is bound to foo(), doesn't this also include the 'let n = 0'?*
2. *I thought calling bar() a second time would not mutate the n*
3. *The environment of foo resets after it returns. so the second call of f should return the same result as the first?*
4. *Intuitively, I do not understand why foo is stored in f, but when f is called n is not reset to 0.*

5. *I thought n in would be rebound to 0 once f was called for a second time.*

6. *foo returns the function bar, it doesnt call bar*

7. *Find it reall confusing that when calling f you do not activate the n = 0 again*

8. *I still do not understand how the values are transferred between the functions and why it is 1 2 1 instead of 1 2 3 when all three function calls should from the programs perspective be the same. There should in my opinion be an explanation of what "nonlocal" does to the variable. Because I assume that is what is making a difference here*

In Section 4.4.2, we found that DEEPCLOSURE is one of the least corrected misconception. The student responses provide some insights on why the misconception was not corrected well. Students 1, 4, 5, and 7 might have a misconception different from DEEPCLOSURE. They might believe that when `f` and `g` are called, the body of `foo` are re-run. I cannot make sense out of the other responses.

4.6 The Design Space Around SMoL Tutor

This section discusses the design space surrounding the SMoL Tutor, including past design, alternative designs that were considered but not implemented, and potential further exploration.

4.6.1 SMoL Quizzes, the Precursor of SMoL Tutor

Before developing the SMoL Tutor, this interactive self-paced tutorial, I created a set of quizzes (in the US sense: namely, a brief test of knowledge) that I call the SMoL Quizzes. There were three quizzes, ordered by linguistic complexity. The first consisted of only basic operators and first-order functions, corresponding to the DEF tutorials. The second added variable and structure mutation, corresponding to the VECTORS and MUTVARS tutorials. The

third added `lambda` and higher-order functions, corresponding to the LAMBDA tutorials. The entire instrument is presented in Appendix B.

Question orders were partially randomized. I wanted students to get some easy, warm-up questions initially, so those were kept at the beginning. Similarly, I 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.

In contrast, the current SMoL Tutor fixes the order of warm-up tasks, for which I see no clear downsides, and fully randomizes the order of other interpreting tasks. Many syntactically similar programs were lost or changed across multiple iterations of development (Section 5.4.2), so grouped randomization no longer carries much meaning, but it does hinder the interpretation of data. See Section 4.6.5 for further discussion of task ordering.

Students only received feedback after having completed a whole quiz. At the end of each quiz, they received both summative feedback *and* a document that explained every program that appeared in the quiz. It is unclear to what extent students read, understood, or internalized these. The SMoL Tutor gives immediate feedback, which I believe is much better.

4.6.2 Not Providing Refutation Texts

The SMoL Tutor's refutation texts come with costs: the Tutor's maintainers must write them manually, and students must read them, which could lead to fatigue as they progress through later tasks.

To determine whether providing refutation texts is beneficial, we need to assess their contribution to student learning. Ideally, this would involve a randomized controlled trial comparing the SMoL Tutor with a version that omits refutation texts. However, even without such an experiment, existing evidence suggests that refutation texts are effective. If they

were not, students would either ignore them or, even if they read them, would not care about their quality. As shown in Section 4.5, students read refutation texts and provide meaningful feedback when a text does not make sense to them. Furthermore, when a refutation text is unclear, students fail to make improvements.

It remains possible that refutation texts are either neutral or counterproductive but never truly helpful. However, I find this unlikely, given extensive prior research indicating that refutation texts support learning.

4.6.3 Enhancing Learning Retention

People forget knowledge over time, but repetition helps slow this process [22]. Delayed repetition is more effective than immediate repetition [8], and active recall—where learners apply knowledge—yields better retention than passive exposure [65].

The SMoL Tutor incorporates two strategies to support retention: (1) review tutorials (Section 4.1.2) and (2) encouraging students to download completed tutorials for easier review (Section 4.1.5).

(**Future Work:** Optimizing the Tutor to deliver review exercises at the ideal frequency and spacing for maximum retention.)

4.6.4 Dependencies on Prior Knowledge

The SMoL Tutor assumes that students have a basic understanding of the language constructs listed in Chapter 3 and can read programs that involve them. It also assumes familiarity with at least one of the supported languages.

Although the Tutor may appear to require more extensive prior knowledge—especially since it introduces technical terms like "environment" and "heap"—students are not expected to know these terms in advance. In fact, the Tutor aims to teach them, introducing such vocabulary only when necessary to provide precise explanations and establish a shared foundation for learning.

Adapting the Tutor for students with less prior knowledge is challenging. In particular, the refutation text approach on which the Tutor relies may not be well-suited for such learners. These students often *lack a conception* rather than *hold a misconception*. However, it is possible to adapt the Tutor in the opposite direction—toward teaching more advanced programming language content.

4.6.5 Task Ordering

The SMoL Tutor currently presents most interpreting tasks in random order (except for a few warm-up tasks) and randomizes all equivalence tasks.

It is likely that the effectiveness of the Tutor depends on the order of the tasks. However, optimizing the order for effectiveness is a difficult problem:

- Instructors often prefer to order tasks by difficulty, placing easier ones first to create a smoother learning curve. But difficulty depends on the misconceptions a student holds, which vary from student to student. Even if we could find an ordering that matches perceived difficulty, it may not maximize learning effectiveness. Easy tasks can be boring and may give students the impression that the Tutor is not relevant to them. This can reduce their engagement and, in turn, hurt their learning.

- Instructors might also want to group tasks by misconception, so students can focus on one misconception at a time. However, many tasks target multiple misconceptions, making such groupings infeasible for some tutorials. Even when such an ordering is possible, it may not be ideal: spaced repetition is often more effective than back-to-back repetition [8].

In general, theories about learning often conflict, and their effectiveness tends to be context-dependent. What I am certain of, however, is that fixing the question order limits our ability to interpret data (e.g., the analyses in Section 4.4 would be impossible), thereby limiting our ability to improve the Tutor. Unless we are highly confident in a specific ordering,

random order is likely the best default—it supports ongoing experimentation and continuous improvement, and it is less likely to accidentally harm students simply because I hold strong opinions about certain learning theories.

4.6.6 Handling Language Differences

The Tutor presents programs in multiple languages, which do not always agree on their outputs (Section 8.2.1). When such disagreements arise, a design decision must be made about which answer should be considered correct. Two natural options are:

1. Use the consensus answer. The SMoL Tutor adopts this approach and displays warnings—such as "JavaScript behaves differently"—when applicable.

2. Use the answer given by the primary syntax (Section 4.1.1).

I chose the first approach because I prioritize teaching the standard model of computation over the behavior of any particular language. This choice also proves practical in the context of the SMoL Tutor: such disagreements occur in only 4 out of the 35 interpreting tasks in the major tutorials (as listed in Figure 4.5).[1] Moreover, in all four cases, there is a clear consensus—only one language disagrees:

- JavaScript disagrees on whether to raise an error for division by zero (in two tasks).

- Python disagrees due to its identical syntax for variable assignment and definition, where expressions like `x = e` are permitted even when `x` has not been explicitly declared.

- Scala disagrees on whether function parameters can be mutated.

4.6.7 Providing Misconception-Aware Feedback

In the case of the SMoL Tutor, both the programs and the set of wrong answers are predefined, allowing feedback to be prepared in advance. I manually authored this feedback

[1]The non-major tutorials contain only warm-up tasks, where no disagreements occur, or post-test tasks, where the rate of disagreement is similarly low.

to tailor it to each specific program.

In more complex scenarios—such as those described by Chandra et al. [9]—programs and wrong answers may be arbitrary, requiring feedback to be computed dynamically. In such cases, misinterpreters are used to associate wrong answers with misconceptions, enabling the system to generate misconception-aware feedback.

CHAPTER 5

SMOL MISCONCEPTIONS

This chapter presents misconceptions about programming language behavior in SMoL. I begin by listing the misconceptions I identified (Section 5.1), followed by a discussion of misconceptions identified by others (Section 5.2). I then describe effective and ineffective approaches to identifying misconceptions (Section 5.3) and explain how I derived my collection of misconceptions (Section 5.4).

5.1 Misconceptions That I Identified

Tables 5.1 to 5.3 present the misconceptions supported by my data. Each table follows the same format: each row defines a misconception and provides a corresponding example program, along with the correct output and the incorrect output reflecting the misconception.

The **misinterpreters** (Chapter 7)—definitional interpreters that capture each misconception—are provided in Appendix A.

Table 5.1: Aliasing-related misconceptions

Misconception	**Example**

DefByRef: Variable definitions alias variables.	`(defvar x 12)`
	`(defvar y x)`
	`(set! x 0)`
	`y`
	`12`
	`0`

CallByRef: Function calls alias variables.	`(defvar x 12)`
	`(deffun (set-and-return y)`
	`  (set! y 0)`
	`  x)`
	`(set-and-return x)`
	`12`
	`0`

StructByRef: Data structures alias variables.	`(defvar x 3)`
	`(defvar v (mvec 1 2 x))`
	`(set! x 4)`
	`v`
	`#(1 2 3)`
	`#(1 2 4)`

DefCopyStructs: Variable definitions copy data structures.	`(defvar x (mvec 12))`
	`(defvar y x)`
	`(vec-set! x 0 345)`
	`y`
	`#(345)`
	`#(12)`

CallCopyStructs: Function calls copy data structures.	`(defvar x (mvec 1 0))`
	`(deffun (f y)`
	`  (vec-set! y 0 173))`
	`(f x)`
	`x`
	`#(173 0)`
	`#(1 0)`

StructCopyStructs: Constructing data structures copies input data structure(s).	`(defvar x (mpair 2 3))`
	`(set-right! x x)`
	`(left (right (right x)))`
	`2`
	`error`

Table 5.2: Scope-related misconceptions

Misconception	**Example**

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)`
	`error`
	`402`

DEEPCLOSURE: Closures copy the *values* of free variables.	`(defvar x 1)`
	`(defvar f`
	`  (lambda (y)`
	`  (+ x y)))`
	`(set! x 2)`
	`(f x)`
	`4`
	`3`

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`
	`error`
	`2`

Table 5.3: Miscellaneous misconceptions

Misconception	**Example**

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)`
	
	4
	error

Lazy: Expressions are only evaluated when their values are needed.[1]	`(defvar y (+ x 2))`
	`(defvar x 1)`
	`x`
	`y`
	error
	1 3

NoCircularity: Data structures can't (possibly indirectly) refer to themselves.	`(defvar x (mvec 1 0 2))`
	`(vec-set! x 1 x)`
	`(vec-len x)`
	3
	error

5.2 Misconceptions Identified by Others

Misconceptions related to scope, mutation, and higher-order functions have been widely documented across diverse populations—from CS1 students to graduate students to users of online forums—over many years (since at least 1986), in various programming languages (from Java to Racket), and across different countries (such as the USA and Sweden).

Two particularly notable efforts stand out. The curated inventory by Chiodini et al. [10] (latest version available at `https://progmiscon.org/`), discussed in Section 5.2.1, lists 90 misconceptions related to Python, Java, JavaScript, and Scratch. Sorva's dissertation, reviewed in Section 5.2.2, includes a table of 162 misconceptions, summarizing research up to 2012.

Additional related work is discussed in Section 5.2.3.

5.2.1 The Curated Inventory by Chiodini et al. [10]

Chiodini et al. [10] catalog a wide range of misconceptions related to Python, Java, JavaScript, and Scratch, based on several years of research. There is some overlap between their collection and mine:

- AssignmentCopiesObject (i.e., "assignment copies the object" rather than "a reference to the object") and VariablesHoldObjects (i.e., "a variable contains a whole object" rather than "a reference to an object") likely correspond to my DefCopyStructs. I cannot distinguish clearly between the two based on their descriptions.
- VariablesHoldExpressions appears to align with my Lazy.
- ReferenceToVariable is likely equivalent to my DefByRef.

At the time of writing, they do not report misconceptions corresponding to the remaining nine in my collection. Moreover, many of their misconceptions fall outside the scope of SMoL for several reasons:

Stylistic Differences Some entries seem to reflect stylistic preferences rather than actual misconceptions. For example, ComparisonWithBoolLiteral describes the belief that "to test whether an expression is `True` or `False`, one must compare it to `True` or `False`", rather than simply using the expression. It is unclear whether such students would misinterpret the behavior of programs that omit the comparison. If not, this belief does not qualify as a misconception in the sense used in this work.

Non-SMoL Features Several misconceptions involve features not supported by SMoL, such as object-oriented programming.

Syntax-Specific Issues Some misconceptions appear closely tied to surface syntax. For instance, AssignCompares—the incorrect belief that `=` compares values rather than assigns them—typically arises in languages where assignment and equality use similar syntax.

Artifact-Specific Other misconceptions may reflect issues with how they were elicited, rather than misunderstandings of program behavior. For example, ReturnUnwindsMultipleFrames refers to the belief that a Python `return` can unwind multiple call stack frames. This was identified using UML-like sequence diagrams. I suspect that this reflects confusion about the diagrams themselves rather than a misconception about Python semantics.

That said, several of their misconceptions—particularly those related to conditionals—appear relevant to my scope. I plan to include the following in future work: IfIsLoop, ConditionalIsSequence, OutsideInFunctionNesting, and NoShortCircuit.

5.2.2 The Dissertation of Sorva [74]

Appendix A of Sorva [74] provides an extensive inventory of misconceptions reported in the literature up to 2012. Like the list by Chiodini et al. [10], it includes syntax-dependent

misconceptions (e.g., Nos. 11 and 12), misconceptions involving non-SMoL features (e.g., object-oriented programming), and those unlikely to occur among students who have completed CS1 and CS2 (e.g., No. 7: "The machine understands English"). Some misconceptions relate more to implementation details than to observable language behavior (e.g., No. 22, first identified by Kaczmarczyk et al. [38]: "Unassigned variables of primitive type (in Java) have no memory allocated").

Nonetheless, there are notable overlaps:

- No. 51 corresponds to my Lazy.
- No. 66 (first reported by Ma [44]) corresponds to my DefCopyStructs.

Several other misconceptions from Sorva's list are particularly relevant, and I plan to include them in future work:

- Nos. 24–37: Misconceptions related to conditionals and loops, reported by various authors [19, 51, 60, 62, 72, 83].
- Nos. 47–49 (first reported by Fleury [27]): Misconceptions about scope.
- Nos. 52–53 (first reported by George [30]): Misconceptions about mutable variables.
- No. 67 (first reported by Ma [44]): Misconception about mutable data structures.
- No. 161: "Boolean values are just something used in conditionals and not data comparable to numbers or strings."

5.2.3 Other Related Work on Misconceptions

Table 5.4 summarizes other studies that are especially relevant to my work. Additionally, Tew and Guzdial [78] identify several cross-language difficulties, although they do not explicitly classify them as misconceptions.

Table 5.4: Similar misconceptions found in prior research.

Work	Population	Languages	Misconceptions
Fisler, Krishnamurthi, and Tunnell Wilson [26]	Third- and fourth-year undergrads	Java and Scheme	FlatEnv; CallByRef; CallsCopyStruct; DefByRef (See their Section 4)
Saarinen et al. [66]	CS2 students	Java	StructByRef (Their G2); DefByRef or DefCopyStructs (Their G3); CallsCopyStruct (Their G4).
Strömbäck et al. [76]	CS masters	Python	FlatEnv; CallByRef; DefsCopyStruct (See their section 4.2)
Strömbäck et al. [76]	CS undergrads	C++	FlatEnv; CallByRef; DefsCopyStruct (See their section 4.2)

5.3 How (Not) to Identify Misconceptions

5.3.1 Overcoming the Expert Blind Spot

As discussed in Section 5.2, several papers report student misconceptions related to different fragments of SMoL. However, the comprehensiveness of these collections is unclear. In many cases, the source of the example programs is not specified, and many appear to have been generated by experts.

The issue with expert-generated misconceptions is that they are often incomplete. Education researchers have documented the phenomenon of the *expert blind spot* [47]: experts frequently fail to anticipate the kinds of difficulties that learners actually face. As a result, we need methods for identifying misconceptions that go beyond expert intuition.

5.3.2 Ensuring Specific Evidence for Each Misconception

A key inspiration for my approach is the extensive literature on *concept inventories* [36], including several developed for computer science [77]. A concept inventory is typically

a multiple-choice instrument in which each question has one correct answer and several carefully constructed distractors. These distractors are not random: each is designed and validated to correspond to a specific misconception. If a student selects a particular wrong answer, it provides strong evidence of a specific misunderstanding.

For example, consider the question "What is `sqrt(4)`?" If a student answers `16`, it suggests confusion between square roots and squaring—a meaningful diagnostic. In contrast, an answer like `37` is less informative.

Concept inventories are powerful tools in instructional settings. Instructors can use them with clickers to gather real-time insights into student thinking. If multiple students choose a particular distractor, the instructor not only knows they are incorrect but also has a clear hypothesis about the underlying misconception—and can respond accordingly.

5.3.3 Precisely Defining Misconceptions

Misconceptions should be described as precisely as possible. A common practice in prior work, including that in Section 5.2, is to document misconceptions as short statements, occasionally accompanied by one or two illustrative programs. This makes it difficult to determine whether two misconceptions are actually the same or merely similar.

By contrast, when a misinterpreter is provided for a misconception—as described in Chapter 7—it becomes possible to generate many more example program-output pairs. Researchers who are comfortable reading interpreter code can even make exact comparisons. This precision supports reproducibility, clarity, and consistency across studies.

5.4 How I Identified the Misconceptions

The considerations discussed in Section 5.3 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 can generally associate wrong answers with specific misconceptions, approaching a concept inventory, and

3. We have a sense of what misconceptions students have.

Of course, I would also want to write a misinterpreter for each misconceptions that I identified.

5.4.1 Generating Problems Using Quizius

My main solution to the expert blind spot is to use the Quizius system [66]. 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, I 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 [39] 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 obtain a set of rationales.

[2]In the course (Chapter 9), 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.

After running Quizius in the course (Chapter 9), I took over as an expert. Determining which is the right answer is easy. Where expert knowledge is necessary 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 [66] finds that students do author programs that the experts did not imagine. In my case, I seeded Quizius with programs from prior papers (Section 5.2), 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.

5.4.2 Distilling Programs and Misconceptions

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

Misleading Variable Names For example:

```
(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 (producing `6`), but in fact, in many Lispy languages (and in the SMoL Language), `...` is a single variable, leading to

an arity error. Such programs do not reveal useful *behavior* misconceptions and were therefore filtered out.

Undefined Behavior Some programs relied on (or stumbled into) intentionally under-specified aspects of the SMoL Language, such as floating-point versus rational arithmetic. While important in general programming, I considered these outside the scope of SMoL Characteristics (due to their lack of standardization) and removed such programs.

Problems Specific to Lispy Syntax Some programs are likely would not produce diverse outputs if they were written in a non-Lispy syntax. For instance, the following program relies on interpreting the inequality correctly. It checks whether `3 > n` (i.e., whether `n < 3`), but some students presumably vocalized it incorrectly as "greater than 3, n?":

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

These programs were removed.

Hard to Execute Mentally Some programs produced diverse outputs simply because they were hard to parse or mentally trace. One example was a 17-line program with six similar-looking and similarly named functions. Such programs were removed or simplified.

Missing Certain Feature Combinations The existing programs did not cover all the concepts I wanted students to engage with. For example, some Quizius-generated programs demonstrated vector aliasing through variable definitions, but none illustrated aliasing via function calls. In such cases, I authored new programs to fill these conceptual gaps.

Conflating Misconceptions Some programs produced wrong (or even correct) answers that could be explained by multiple (mis)conceptions. In these cases, the program needed to be refined to be more discriminative, and misinterpreters (introduce in Chapter 7) were particularly helpful in this process (Section 7.2).

As a result, I manually curated the programs and misconceptions to address these issues. This curation must be an iterative process, typically due to the last reason—whenever I find new misconceptions, the question set may need to be refined—but also because some programs may not appear clearly problematic until sufficient data have been collected.

5.4.3 Confirming the Misconceptions With Cleaned-Up Programs

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

I delivered multiple-choice questions (MCQs) through various systems described in Chapter 4. Each MCQ presents a program and asks students to predict the program output. Students might choose from an available choice, or pick a special choice, "Other", which then allows students to enter an arbitrary answer.

This style of MCQs is related to concept inventories, which is discussed as a related work in Section 5.3. In a concept inventory, each option must either be the correct answer or correspond to exactly one misconception. In my case, the one-to-one mapping is mostly, but not entirely, preserved:

- An "Other" choice is presented.
- Additional random choices are presented to make it harder for students to find the right choice by elimination. Those options do not correspond to any misconceptions.

The reason for adding random options is as follows. The data often suggest very few wrong choices. Thus, in most cases, students have a considerable chance of just guessing the right answer or successfully using a process of elimination. By increasing the number of options, I 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, I added extra options as follows:

1. The “error” option is added if it is not already an option.

2. Collect *templates* of existing options. A template of an option is the option with all the number literals removed. For instance, The template of “`2 3 45`” is “`_ _ _`”.

3. Keep adding random options until we run out of random options or have at least eight distinct options. Each random option is generated by filling a template with number constants from the source program or existing options.

I 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.

CHAPTER 6

STACKER

Section 6.1 provides a guided tour of the Stacker, illustrating its typical usage and introducing its major components. Section 6.2 delves into the system's details. Section 6.3 explores the teaching instruments enabled by the Stacker. Section 6.4 compares the system with similar tools. Sections 6.5 and 6.6 present two user studies about the Stacker. Section 6.7 discusses design the design space, summarizing lessons learned from the tool comparison (Section 6.4) and the user studies (Sections 6.5 and 6.6).

6.1 A Guided Tour of Stacker

The Stacker is a tool for tracing SMoL programs. Figure 6.1 shows the UI when the Stacker is first opened. The UI consists of two main components, separated by a vertical gray bar:

(Program) Editor Panel For entering SMoL programs. This panel includes three bars at the top and a program editor, which supports common text-editing features like syntax highlighting. Section 6.2 provides more details.

Figure 6.1: The Stacker user interface

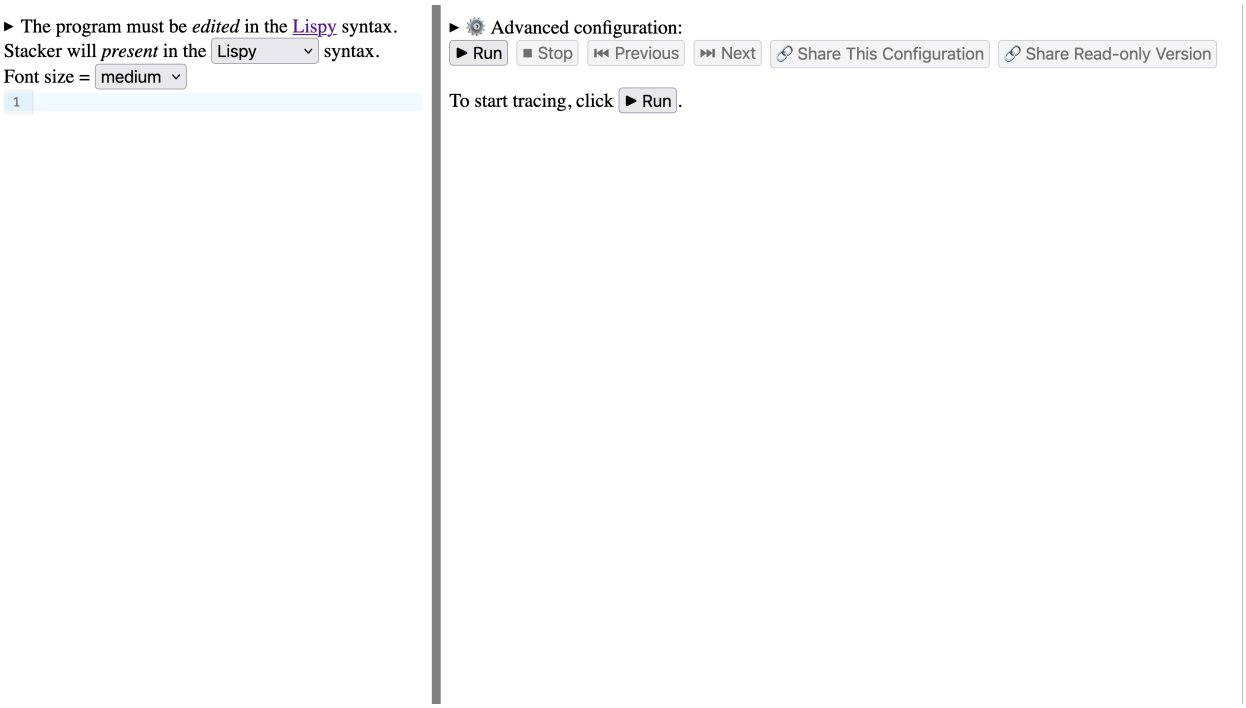
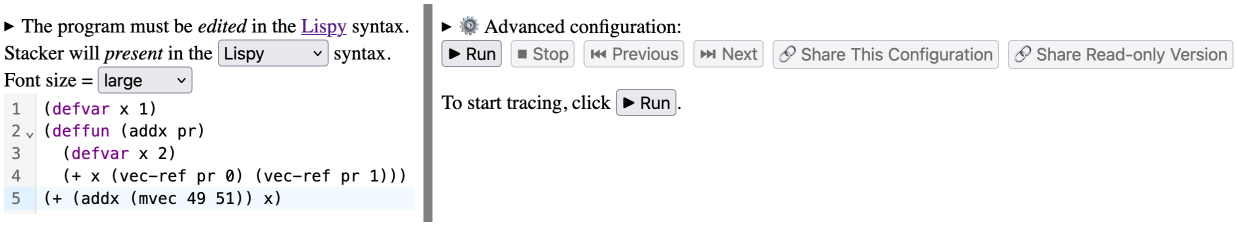


Figure 6.2: The Stacker user interface filled with an example program



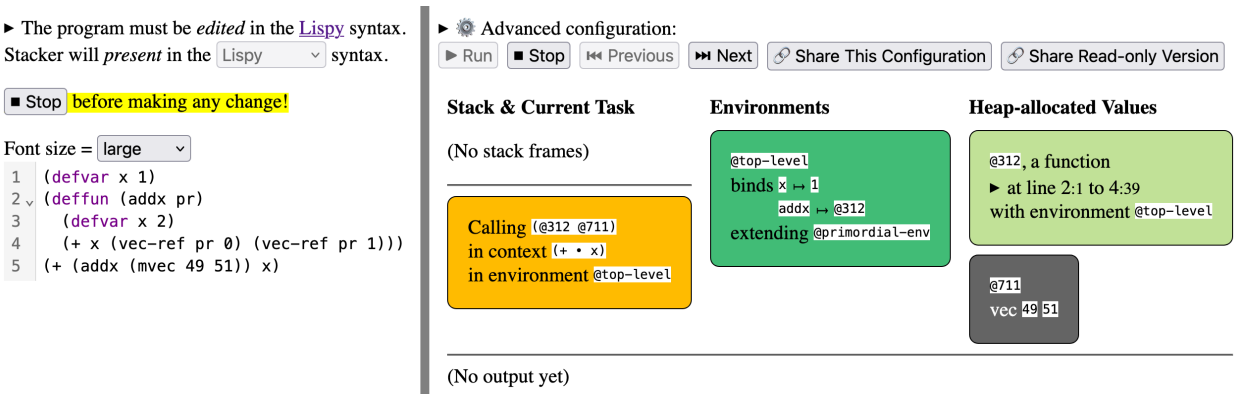
Trace Panel For starting, stopping, and navigating traces. This panel includes an advanced configuration bar, a trace control bar, and a display area. Advanced configurations are covered in Section 6.2.

The trace control bar enables buttons only when applicable. Initially, only the "Run" button is available, while the display area shows a hint (e.g., "To start tracing, …"). Once tracing starts, the display updates accordingly.

6.1.1 Example Program

Figure 6.2 shows the Stacker loaded with the following SMoL program:

Figure 6.3: An example Stacker trace (showing state 1 out of 5)




```
(defvar x 1)
(deffun (addx pr)
  (defvar x 2)
  (+ x (vec-ref pr 0) (vec-ref pr 1)))
(+ (addx (mvec 49 51)) x)
```

This program defines a variable `x` and a function `addx`, then prints the sum of `(addx (mvec 49 51))` and `x`. The function call `(addx (mvec 49 51))` defines a local variable `x`, then returns the sum of the local `x` and the first two elements of the function argument `pr`.

To maintain a reasonable length, this tour's example program does not cover all aspects of SMoL. However, it highlights some of the most important ones, including variable bindings, function calls, and compound data.

6.1.2 Running the Program

Clicking one of the "run" buttons in Figure 6.2 transitions the Stacker to tracing mode Figure 6.3, triggering the following changes:

- **Editor Panel**: Editing is disabled, and a reminder appears, instructing users to stop tracing before making changes. The Stacker disables editing while tracing to avoid showing information inconsistent on the editor panel and the trace panel.

- **Trace Control**
  - The “Run” button is disabled (since the program is already running).
  - The “Stop” button is enabled.
  - The “Previous” button remains disabled (no prior states exist).
  - The “Next” button is enabled (to advance the trace).
  - “Share” buttons are enabled (for sharing the current state).
- **Display Area** Now shows the current trace state, structured as follows:
  - Three columns (top): **the stack column** (“Stack & Current Task”), **the environment column** (“Environments”), and **the heap column** (“Heap-allocated Values”)
  - Program output (initially displaying “(No output yet)”).

The stack column, as its label suggests, includes two parts from top to bottom, split by a black horizontal line: **the (call) stack** and **the current task**.

The (call) stack remains empty in the current state (Figure 6.3), as indicated by the label “(No stack frames)”. Stack frames are inserted when a function call happens and removed when a function call returns.

The current task is represented by a box below the stack, which always includes three lines from top to bottom: a brief description of the current task, the context, and the environment of the current task. The box color depends on the kind of tasks. In the current state (Figure 6.3):

- The current task is to compute the function call `(@312 @711)`, where `@312` is the function and `@711` is the only argument.
- The current task occurs in `(+ • x)`, meaning that after the function call returns, the Stacker will compute `(+ • x)` with “•” replaced by the returned value.

- The current task occurs in the top-level environment, meaning the `x` in `(+ • x)` refers to the `x` defined in the top-level environment.

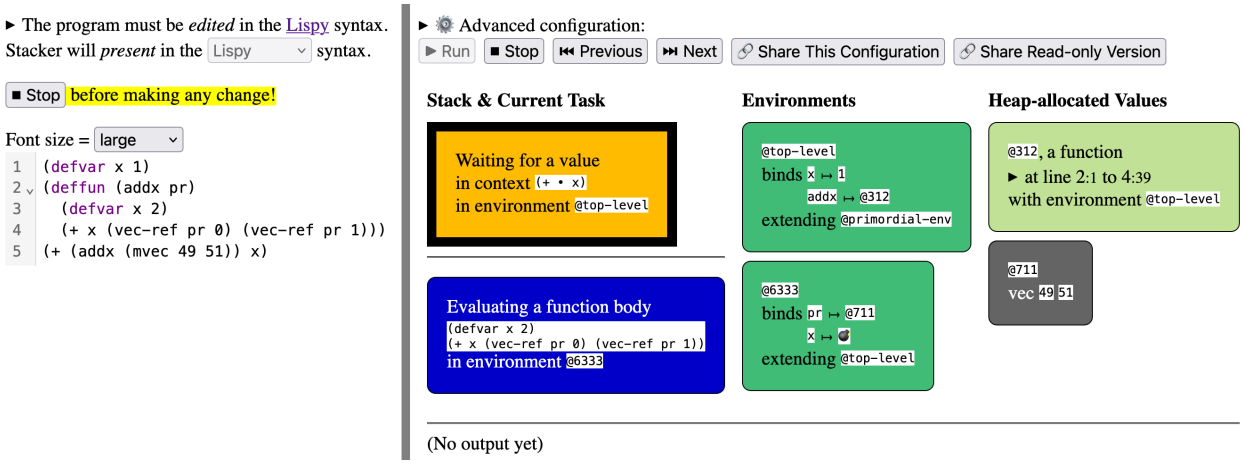
The context is a **local evaluation context**. A *context* is a piece of program with exactly one hole (in my case, represented by "•"), and an *evaluation context* is, roughly speaking, a context where the hole indicates the next smallest unit of meaningful computation [24]. In my case, the evaluation contexts are *local* because they describe only what happens within a function, without referring to the caller's context.[1]

Next, **the environment column** lists environments constructed so far in the trace. Environments are represented by dark green boxes, each containing three pieces of information from top to bottom: the address, the variable bindings, and the parent environment from which the environment extends. In the current state (Figure 6.3), the top-level is the only one constructed so far. This environment binds two top-level variables (`x` is bound to `1`, and `addx` is bound to the function `@312`) and extends from the primordial environment, where built-in constructs are defined.

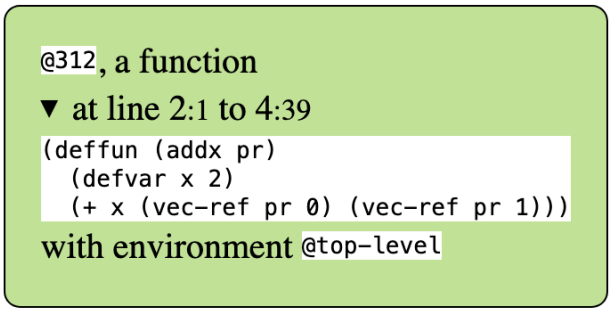
The heap column lists heap-allocated values constructed so far in the trace. These values are represented by boxes of varying colors, depending on the kind of the value. Functions are shown in light green boxes, whereas vectors are gray.

[1] We can obtain the standard form of a global evaluation context by composing local evaluation contexts into the holes of the preceding contexts on the stack. While doing so, we must also substitute variables using the corresponding environments before composition.

Figure 6.4: An example Stacker trace (showing state 2 out of 5)



A function box displays the function's source code location and the environment in which it was constructed. The environment is needed when the function's body refers to variables defined outside of it (i.e., free variables). Users can click the source code location to view the function body, for example:



Clicking the "Next" button in Figure 6.3 updates the Stacker to Figure 6.4. In Figure 6.3, the Stacker was about to compute a function call, so Figure 6.4 reflects the results of that call: a new environment is constructed to bind the function argument, the stack now contains a stack frame for the function call, and the current task has changed.

The new environment binds the function argument and declares the local variable `x`. The bomb symbol indicates an uninitialized binding. In a real language implementations, local variables might or might not reside in the same environment as function arguments. The Stacker presents a simplified view to avoid UI cluster.

Figure 6.5: An example Stacker trace (showing state 3 out of 5)

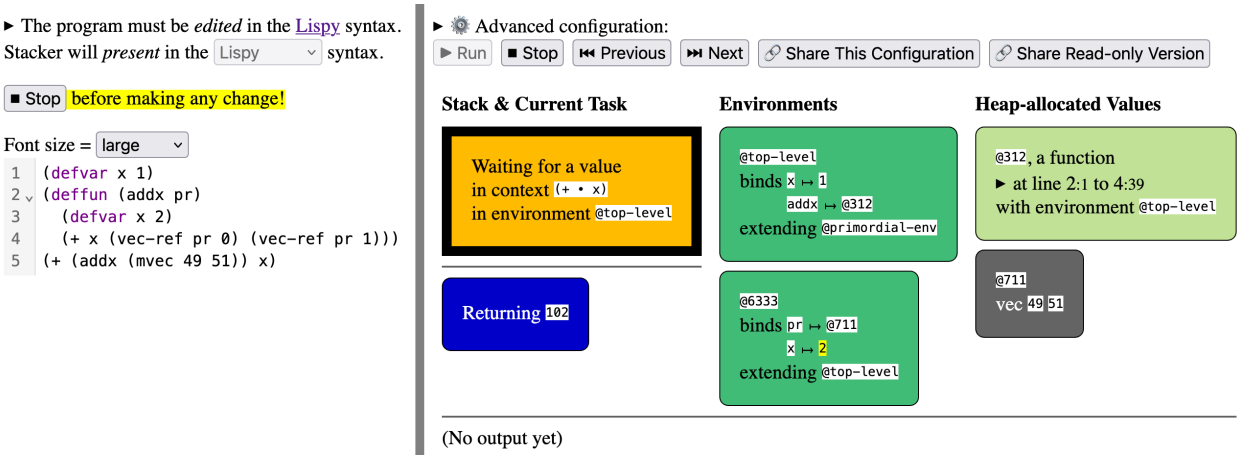
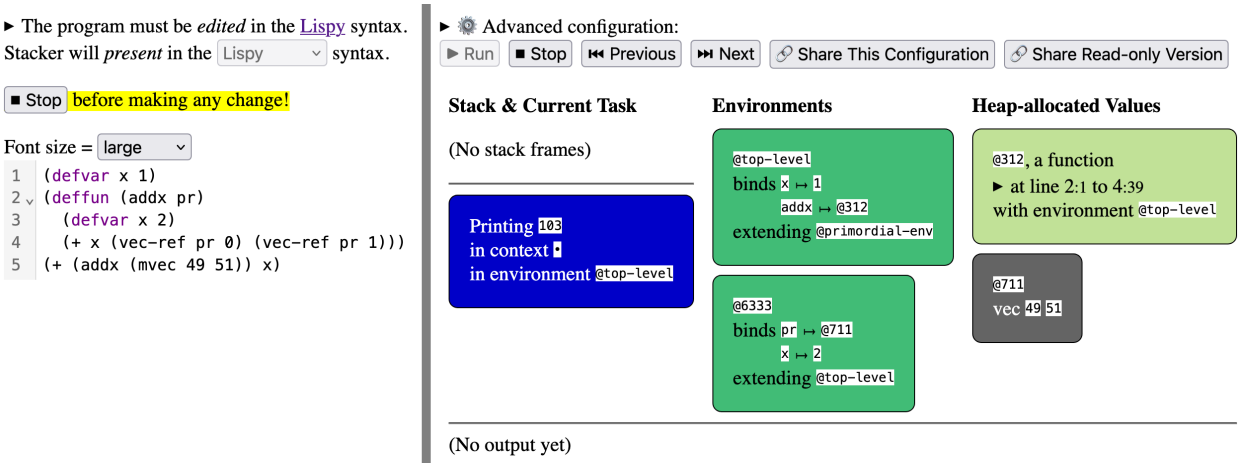


Figure 6.6: An example Stacker trace (showing state 4 out of 5)



Stack frames are presented as yellow boxes, the same color as function call tasks (Compare Figure 6.3 and Figure 6.4). This emphasizes the connection between function calls and stack frames. The content is nearly identical, except that a function call task says "Calling …", whereas a stack frame says "Waiting for a value".

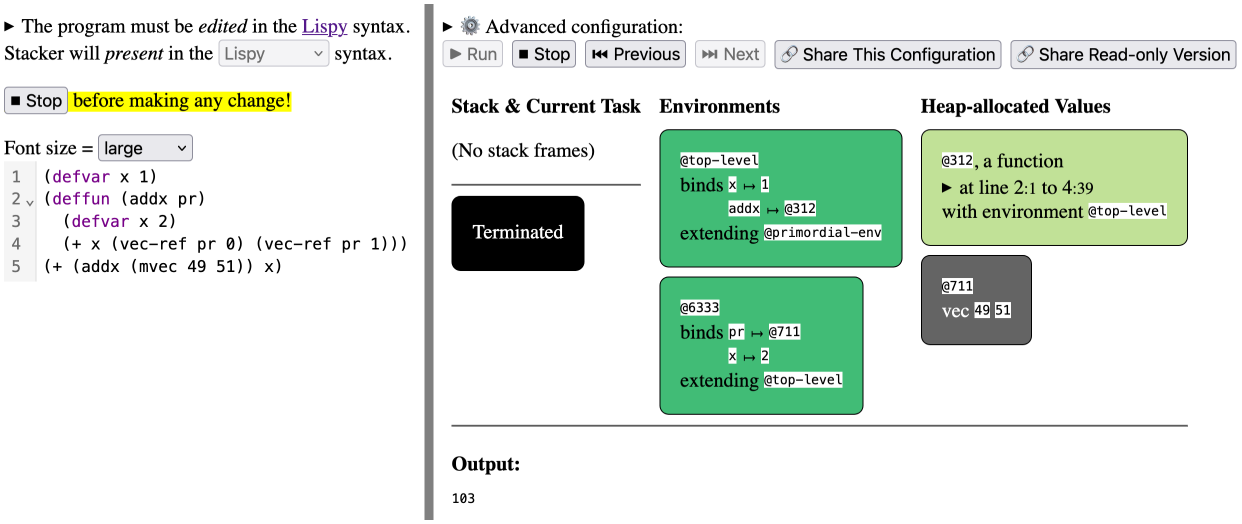
The current task is colored yellow only when displaying a function call, black when terminated, and blue otherwise. The kinds of current task are as follows:

1. The stacker is calling a function.

2. The stacker just called a function.

3. The stacker is returning from a function.

Figure 6.7: An example Stacker trace (showing state 5 out of 5)



4. The stacker is mutating a (variable) binding.

5. The stacker is mutating a data structure.

6. The stacker is printing a value.

7. The stacker just terminated.

The current task kinds determine all possible states. Designing a good collection of state kinds is non-trivial. I discuss this issue in Section 6.7.6.

The previous state (Figure 6.3) illustrates the first kind of current task, the current state (Figure 6.4) illustrates the second. The remaining trace states each demonstrate a different task kind:

- Figure 6.5 The stacker is returning from the function call.
- Figure 6.6 The stacker is printing the returned value.
- Figure 6.7 The stacker has terminated.

6.1.3 Key Features of Stacker

I conclude this section by highlighting several key features of the Stacker:

- The Stacker can display programs and traces in any language supported by the SMoL Translator (Chapter 8), but programs must be authored in the SMoL Language. (See Section 6.7.11 for a discussion of this design.)
- Every (non-final) state presents sufficient information to predict the next state.
- Users can generate a shareable URL for the current state, allowing others to view the exact same trace. (See Section 6.7.10 for a discussion of this design.)
- Stacker generates traces on the fly, so programs need not be terminating. (See Section 6.7.15 for a discussion of this design.)
- Memory addresses for environments and heap-allocated values are randomly generated, with a controllable random seed. (See Section 6.7.9 for a discussion of this design.)

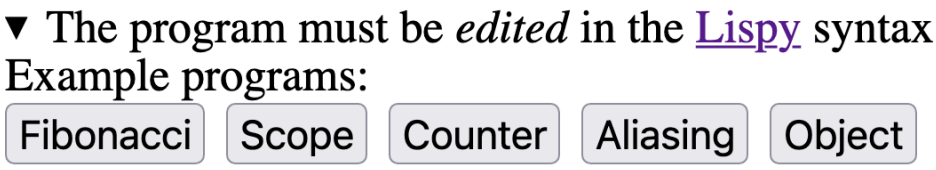
6.2 More Details on Using Stacker

This section presents more details on using the Stacker, complementing the guided tour (Section 6.1).

6.2.1 Editing Support

The UI element in the top-left corner offer guidance on writing SMoL programs. By default, it looks like

The underlined text “Lispy” links to a reference document summarizing the SMoL Language (essentially a shorter version of Chapter 3). Expending this element reveals a list of example programs:



Here is a brief description of the listed programs:

Fibonacci defines and calls a function to compute the N-th Fibonacci number

Scope scopes variables in a perhaps confusing way.

Count defines and tests a “counter” function that increases with each call.

Aliasing aliases data structures in a perhaps confusing way.

Object simulates an object using the SMoL Language.

6.2.2 Presentation Syntax

The following UI element allows users to choose the **syntax** for displaying programs and traces. Users are indeed choosing the *syntax* rather than the *language*: The Stacker always follow the semantics of the SMoL Language regardless of the chosen syntax. The “Lispy“ syntax stands for the syntax of the SMoL Language. All syntaxes from Section 8.1 are supported, except for Scala, which is still in progress. Choosing a syntax other than “Lispy” triggers a live translation in the trace panel (Figure 6.8).

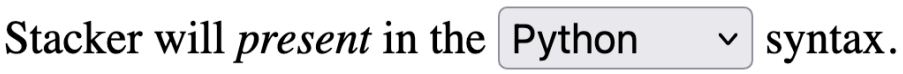
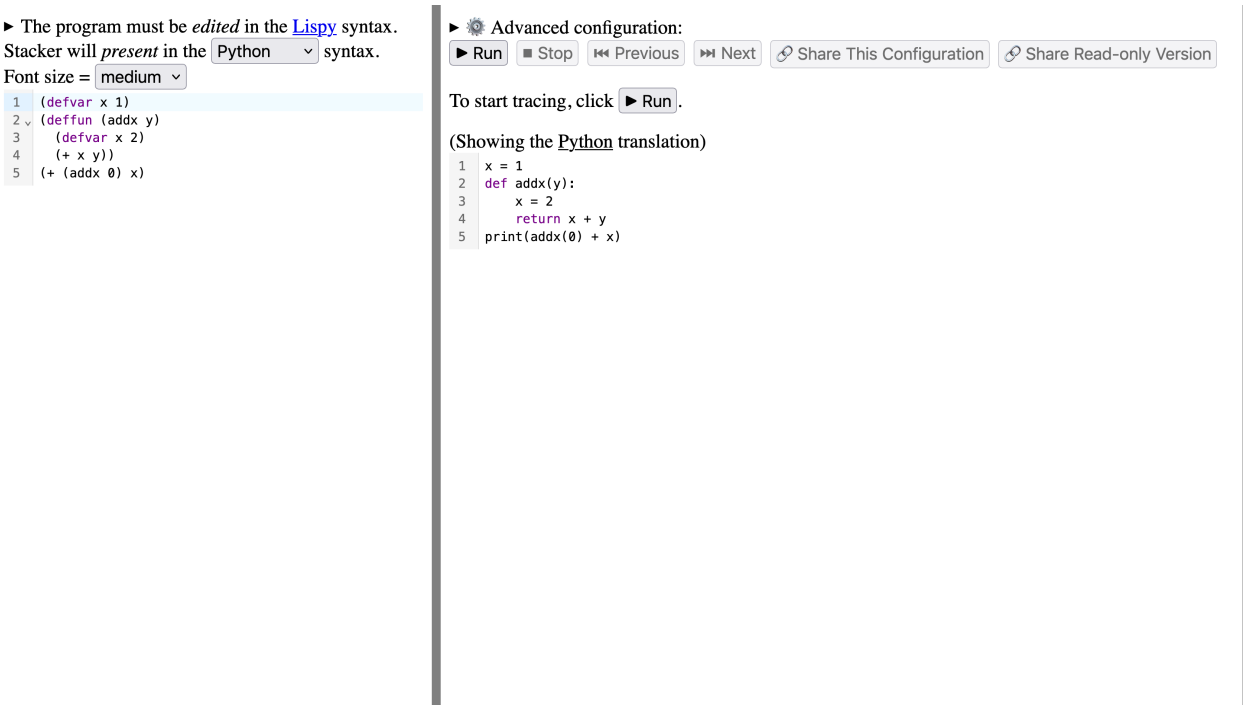
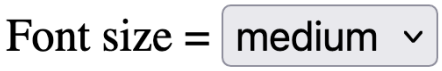


Figure 6.8: The live translation feature of the Stacker. When users are editing their programs and the presentation syntax is set to non-Lispy, a live translation of the program is shown on the trace panel.



6.2.3 Changing the Relative Font Size of the Editor



The above dropdown menu controls the font size of the editor, rather than the entire UI. This choice is intentional: since the Stacker is web-based, users can zoom in or out via their browser, but may occasionally need finer control over the editor's text size. For example:

- Instructors may reduce the font size to fit a long program's trace on a single screen.
- Users may increase the font size for readability when there is excessive whitespace.

6.2.4 Editor Features

The editor provides standard editing features, including:

- Autocompletion for SMoL Language keywords and parentheses
- Line numbers
- Multi-cursor editing
- Syntax highlighting

When a non-Lispy syntax is selected and the program is running, the editor displays the translated program with syntax highlighting.

The editor is based on CodeMirror [13], which makes the feature straightforward to implement.

6.2.5 Share Buttons

The following buttons generate permalinks that share the current trace state/configuration. The right button opens the Stacker in a simplified mode, hiding irrelevant details for easier trace navigation (Figure 6.9).

6.2.6 Advanced Configuration

The following UI element provides additional tracing configurations.

After expansion, it becomes

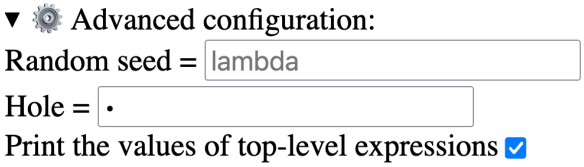
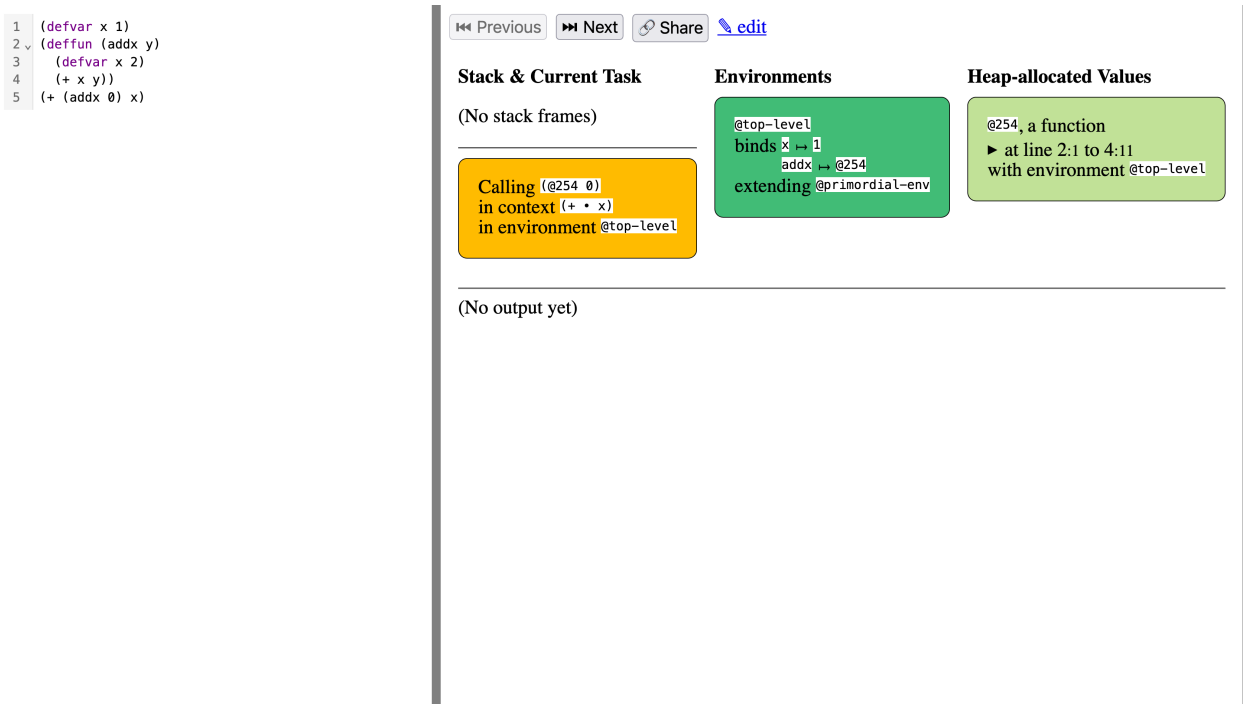


Figure 6.9: A read-only view of a Stacker trace



The "Random seed" is a string determining address generation. If unspecified, the Stacker selects a random seed (e.g., "lambda" in the example) and displays it in gray.

The "Hole" is a string representing holes in context, defaulting to "•" as seen in prior examples.

"Print the values of top-level expressions" controls whether top-level expressions are printed to output.

6.2.7 The Trace Display Area Highlights Replaced Values

The Stacker highlights changes caused by mutations:

- When a variable assignment occurs, the updated environment field is highlighted in yellow in the next state.
- When a structure mutation happens, the replaced field is also highlighted in yellow.

Table 6.1: All combinations of foreground and background colors in the Stacker

Foreground	Background	Example	Usage
white	`#000000`	Example	Terminated
white	`#0000c8`	Example	The default task color
white	`#646464`	Example	Data structure
black	`#41bc76`	Example	Environments
black	`#c1e197`	Example	Functions
black	`#ffbb00`	Example	Stack frames and function calls
black	`#ff7f79`	Example	Errors

6.2.8 The Trace Display Area Circles Referred Boxes

Hovering over an address reference thickens the referred box's border and changes the border color to red.

6.2.9 The Trace Display Area Color-Codes Boxes

The Stacker color-codes boxes according to Table 6.1

6.3 Educational Use Cases

The Stacker is designed for educational use. Instructors can step through traces to explain program execution or encourage students to explore the system independently. This section presents several perhaps more interesting use cases. While I believe these are effective, they are not empirically validated, so readers should consider them as instructional ideas rather than evidence-based recommendations.

6.3.1 Predicting the Next State

Students tend to learn more when they actively predict the next state before clicking "Next" rather than passively stepping through traces. While my observations are anecdotal, this aligns with active learning principles, which have been widely studied (see [58] for a review).

6.3.2 Contrasting Two Traces

Many SMoL misconceptions can be viewed as incorrectly assuming that different programs behave the same. For example, the DefCopyStructs misconception leads students to believe that the following two programs produce identical results:

```
;; Program 1
(defvar x (mvec 12))
(defvar y x)
(vec-set! x 0 345)
y
```

```
;; Program 2
(defvar x (mvec 12))
(defvar y (mvec 12))
(vec-set! x 0 345)
y
```

To address such misconceptions, we can instruct students to compare the two program traces and identify a pair of trace states that explain their differing behavior.

6.3.3 From State to Value

In the PL course (Chapter 9), students were asked to determine a program's output based on a given trace state (illustrated in Figure 6.10). These exercises essentially asked students to predict the final value from an intermediate state. I found them useful for assessing students' understanding of environments. Further analysis of the collected data remains future work.

6.3.4 From State to Program

In the PL course (Chapter 9), students were also instructed to construct programs that would produce a given trace state. Figure 6.11 illustrates one such state. Students were told that every solution must include a `pause` function that simply returns `0`:

```
(deffun (pause) 0)
```

Figure 6.10: A state-to-output question

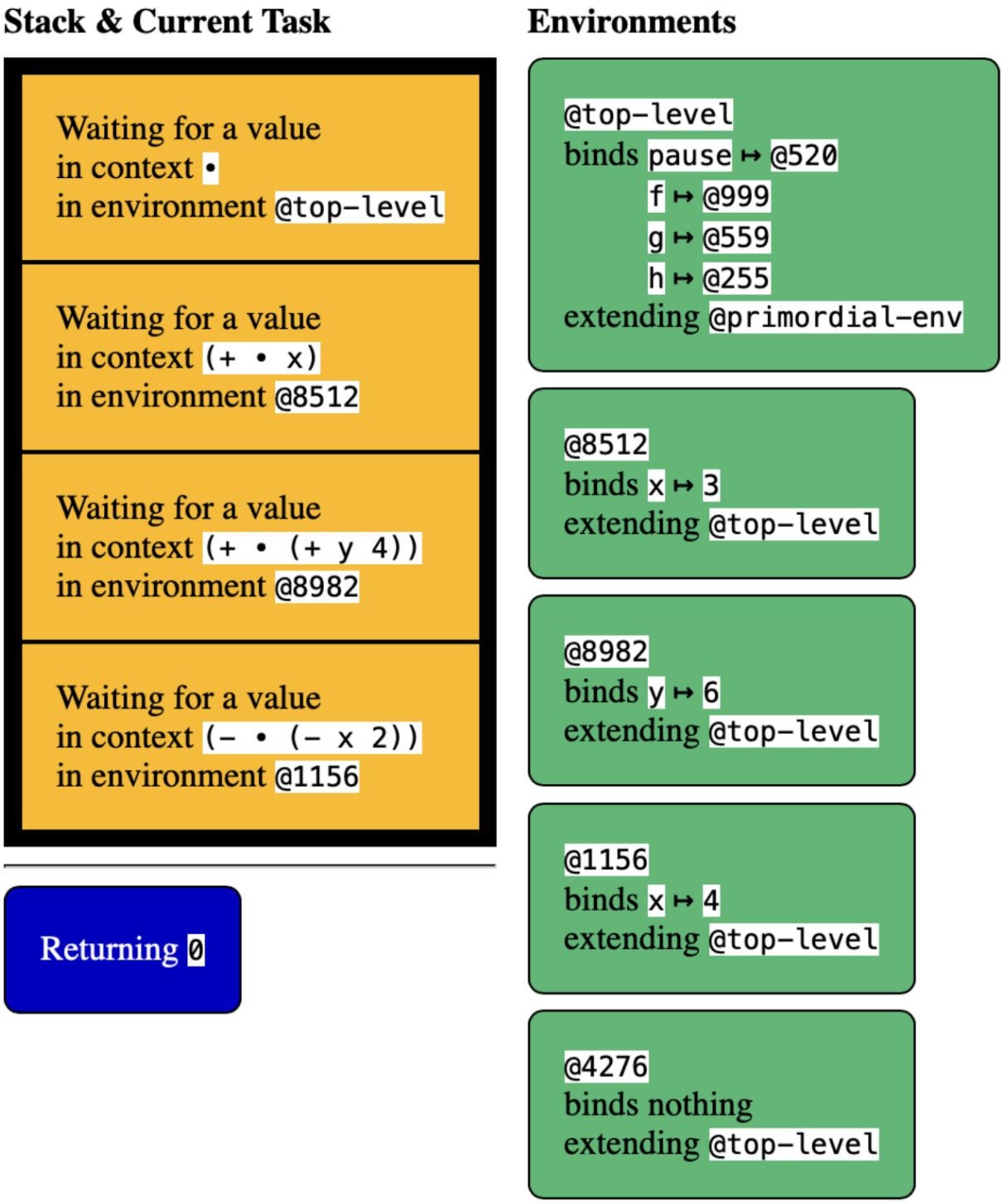
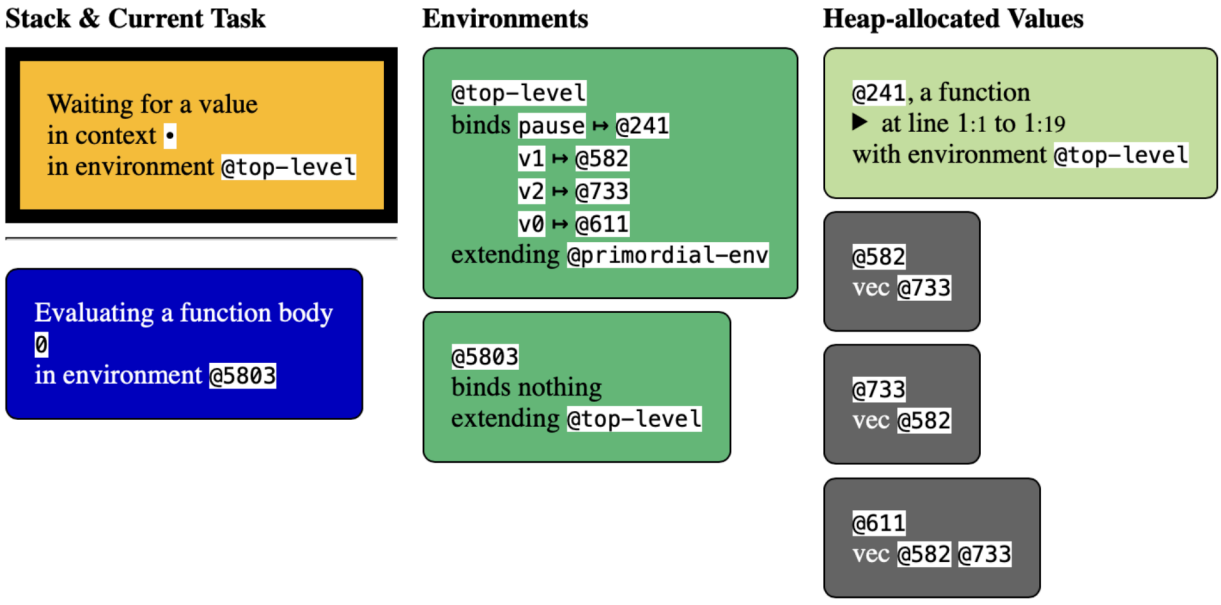


Figure 6.11: A state-to-program question



For the example shown, a correct answer is:

```
(deffun (pause) 0)
(defvar v1 (mvec 0))
(defvar v2 (mvec v1))
(defvar v0 (mvec v1 v2))
(vec-set! v1 0 v2)
(pause)
```

These exercises essentially asked students to reconstruct an initial program based on an arbitrary intermediate state. They turned out to be quite challenging, likely because students needed to consider *all* available language constructs. I observed that students learned a great deal through this process. Further analysis of the collected data remains future work.

6.3.5 Using Stacker to Teach Generators

The Stacker concepts can also help explain language features beyond SMoL, such as generators.

A generator behaves similarly to a stack frame—it has a context and an environment. However, while stack frames disappear once their context is empty, generators persist beyond

Figure 6.12: A Python program for the Stacker-generator instrument

```
def pause():
    return 0

def make_gen(n):
    def gen():
        yield n + 1
        yield n + 2
        yield n + 3
    return gen
g1 = make_gen(0)()
g2 = make_gen(10)()
pause()
print(next(g1))
print(next(g1))
print(next(g2))
pause()
```

execution. They can leave the stack either when their context becomes empty or when they yield, and they remain in memory until garbage collection determines they are no longer needed.

In the PL course (Chapter 9), students drew Stacker-like diagrams to visualize Python generator behavior. One such program is shown in Figure 6.12.

6.4 Comparing Stacker with Other Tools

This section compares the Stacker with several program trace visualization tools by running essentially the same program (Table 6.2) in each tool. All tools are configured to show the program state when the function `addx` has just been called, and the system is about to evaluate the first expression in the function body. The state as presented by the Stacker is shown in Figure 6.13.

This particular program state is chosen to simultaneously satisfy the following criteria:

- Avoid using language features that are unsupported by some of the surveyed tools,

Table 6.2: A program for tool comparison

Language	Program

SMoL	`(defvar x 1)`
	`(deffun (addx pr)`
	`  (defvar x 2)`
	`  (+ x (vec-ref pr 0) (vec-ref pr 1)))`
	`(+ (addx (mvec 49 51)) x)`

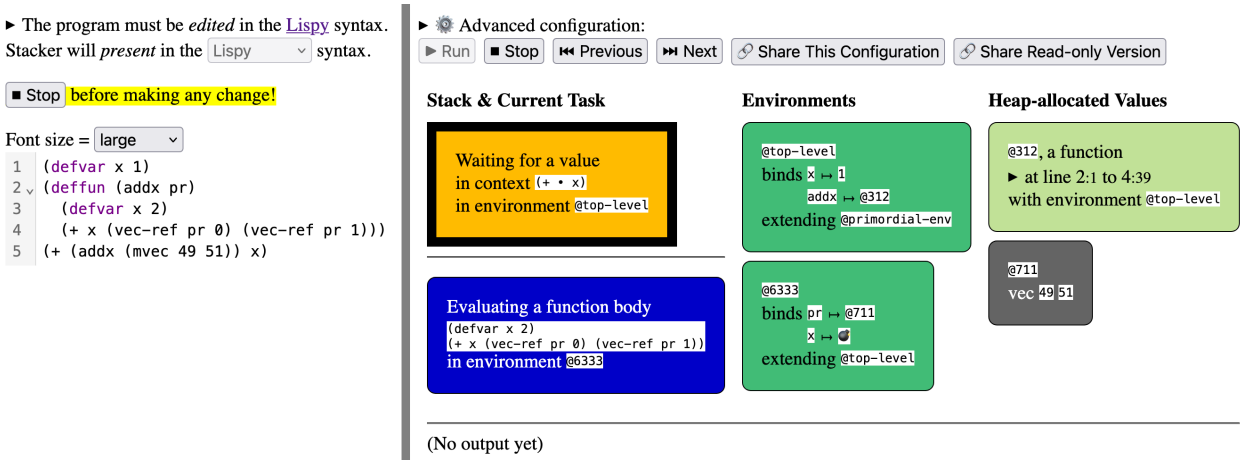
Python	`x = 1`
	`def addx(pr):`
	`    x = 2`
	`    return x + pr[0] + pr[1]`
	`addx([ 49, 51 ]) + x`

JavaScript	`let x = 1;`
	`function addx(pr) {`
	`  let x = 2;`
	`  return x + pr[0] + pr[1];`
	`}`
	`addx([ 49, 51 ]) + x`

Java	`public class Main {`
	`    static int x = 1;`
	`    public static int addx(int[] pr) {`
	`        int x = 2;`
	`        return x + pr[0] + pr[1];`
	`    }`
	`    public static void main(String[] args) {`
	`        int _ = addx(new int[]{49, 51}) + x;`
	`        return;`
	`    }`
	`}`

Racket	`(define x 1)`
	`(define (addx pr)`
	`  (local [(define x 2)]`
	`    (+ x (first pr) (second pr))))`
	`(+ (addx (list 49 51)) x)`

Figure 6.13: A Screenshot of the Stacker (a duplicate of Figure 6.4)



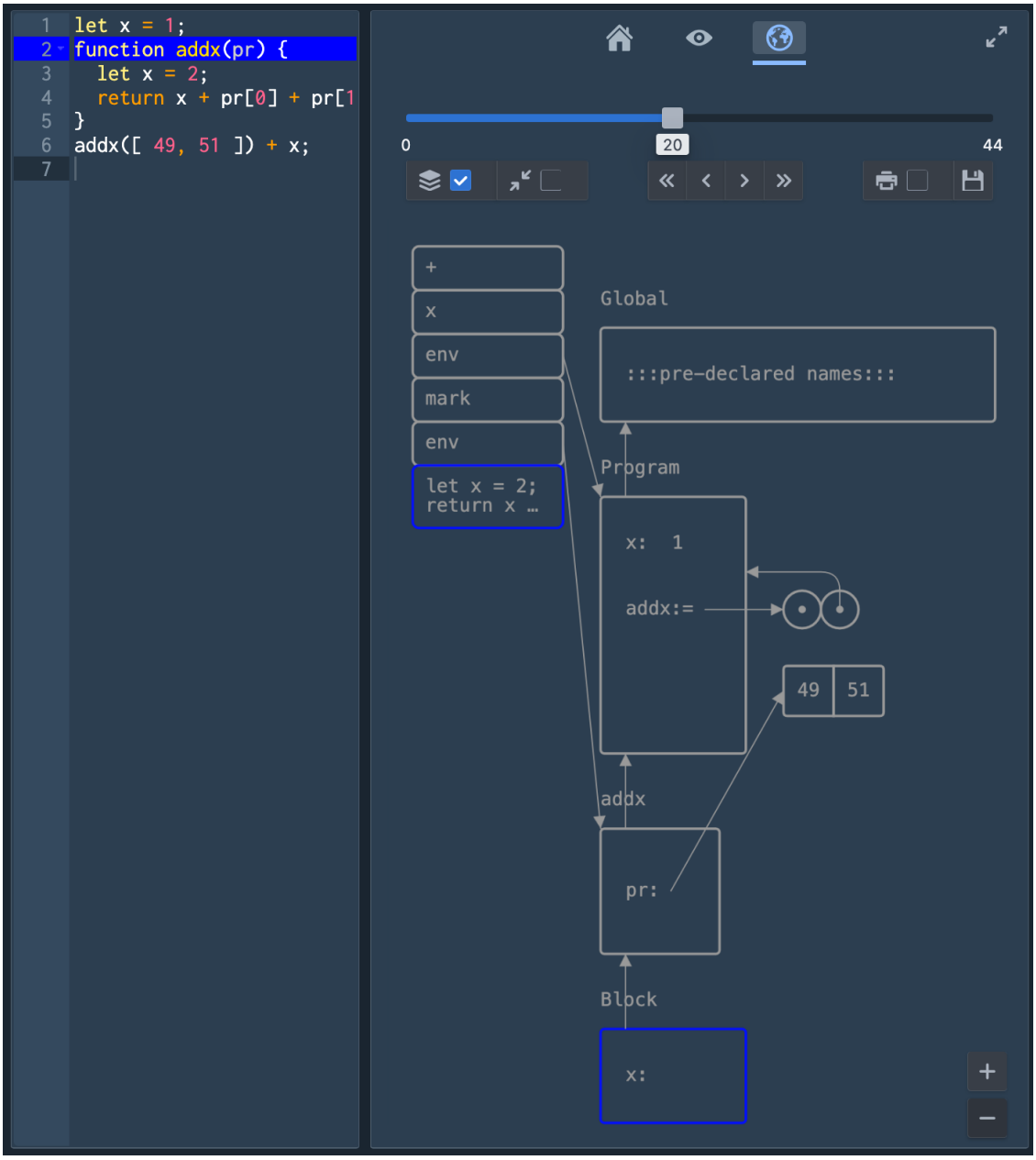
such as first-class functions and mutable state;

- Cover as many SMoL language features as possible;
- Ensure that each kind of visual element (stack frame, environments, functions, and data structures) is presented at least once.

6.4.1 The Surveyed Tools

The comparison includes the following tools:

Figure 6.14: A Screenshot of the Environment Model Visualization Tool



The Environment Model Visualization Tool (hereafter "the Env tool") Originally developed by Cai et al. [7] and later extended by Abad and Henz [1], this tool implements the environment model as presented in *Structure and Interpretation of Computer Programs* [2]. A screenshot of the Env tool is shown in Figure 6.14.

Figure 6.15: A Screenshot of the Online Python Tutor Running Python

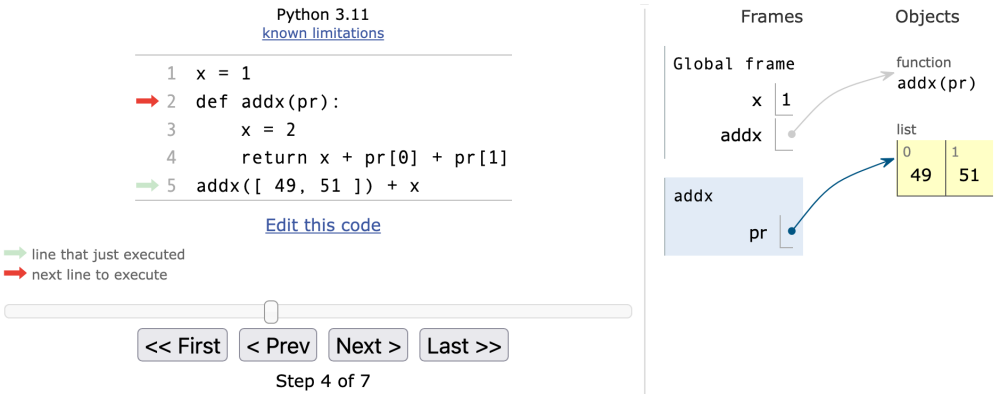


Figure 6.16: A Screenshot of the Online Python Tutor Running JavaScript

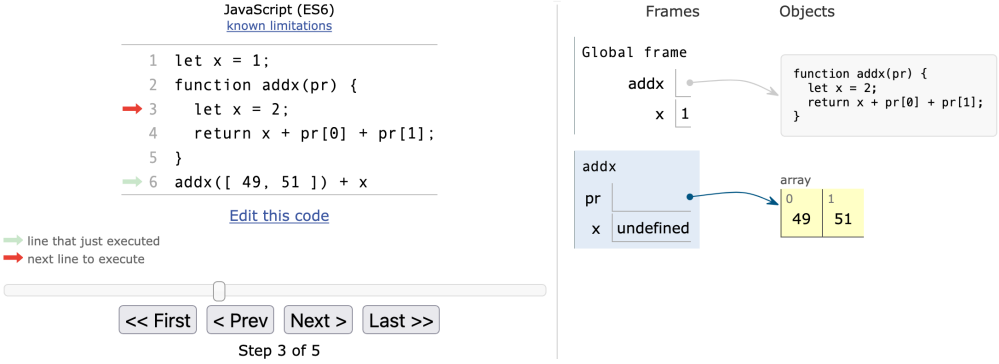
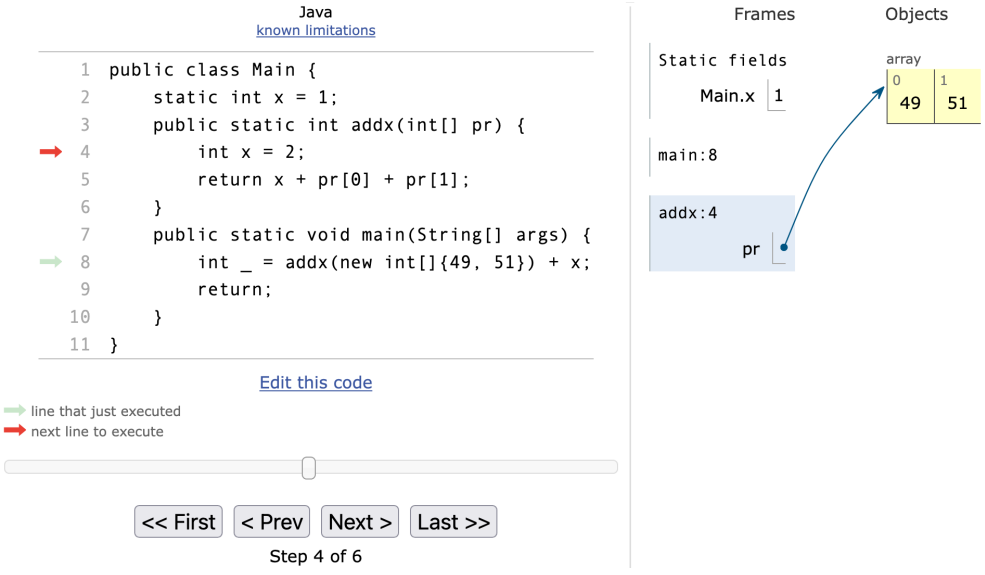
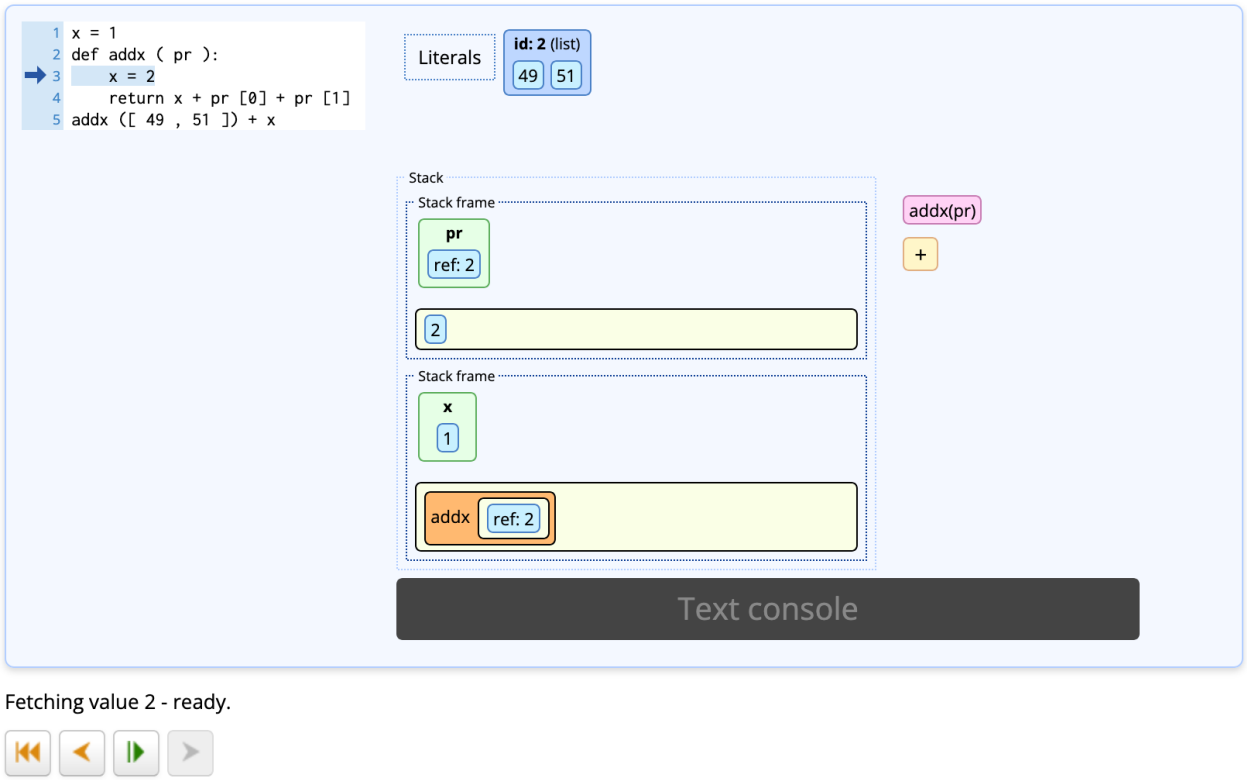


Figure 6.17: A Screenshot of the Online Python Tutor Running Java



Online Python Tutor (hereafter "OPT") Developed by Guo [33], this tool supports multiple language. Figures 6.15 to 6.17 show screenshots for different language configurations.

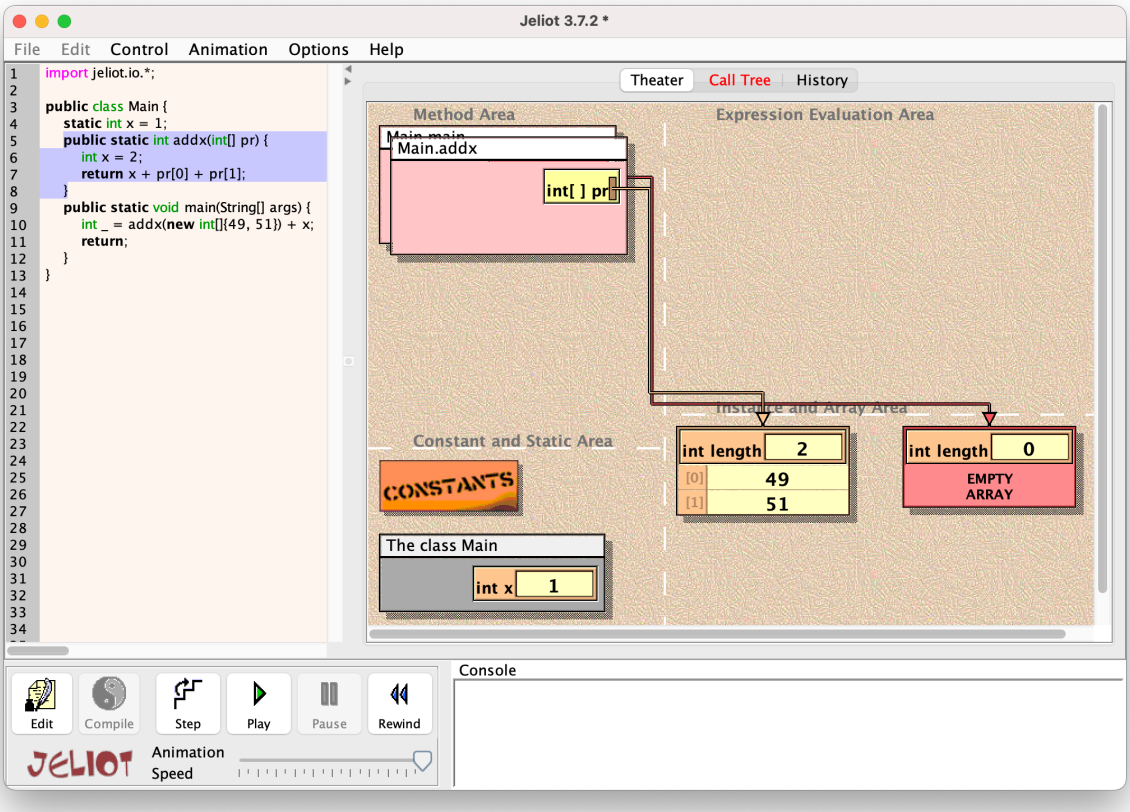
Figure 6.18: A Screenshot of the Jsvee & Kelmu



Jsvee & Kelmu (and their predecessor UUhistle) (hereafter "JK") According to UUhistle's official website (`http://www.uuhistle.org`), Jsvee & Kelmu [71] are successors to UUhistle [75]. Figure 6.18 shows a screenshot of JK.[2]

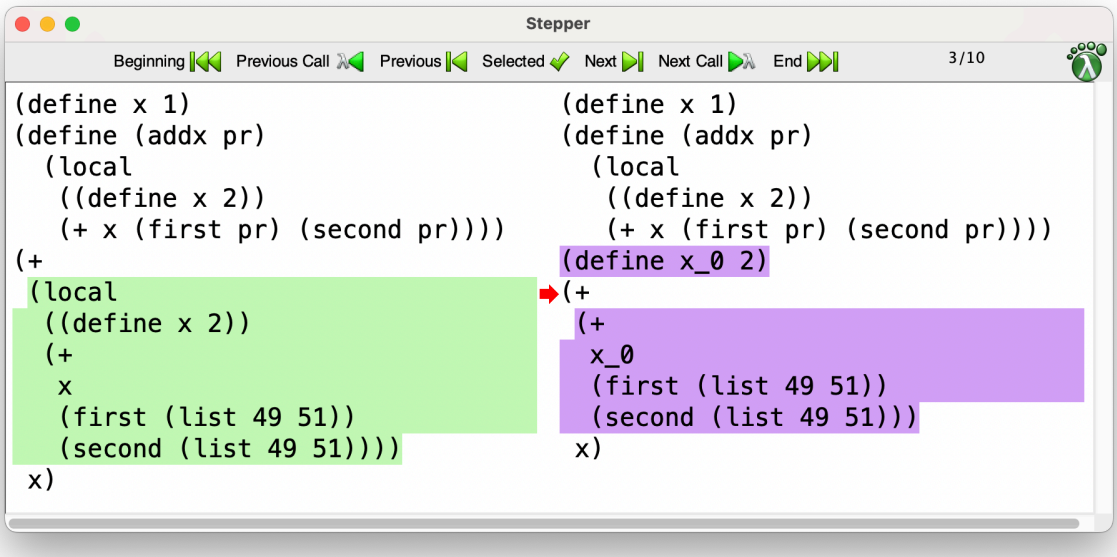
[2]Both systems offer more functionality than typical program trace visualization tools: UUhistle pioneered *visual program simulation*—a system that supports users in constructing program states as a way to practice their understanding—while Jsvee & Kelmu provide many features for *tailoring* program animations, such as adding highlights and explanations. In this work, I focus on their use as program trace visualization tools and do not consider their additional capabilities.

Figure 6.19: A Screenshot of Jeliot



Jeliot A series of visualization tools summarized by Ben-Ari et al. [5]. Figure 6.19 shows a screenshot of the latest version (Jeliot 3).

Figure 6.20: A Screenshot of Stepper



The Algebraic Stepper in DrRacket (previously known as DrScheme) (hereafter "Stepper") Developed by Clements, Flatt, and Felleisen [11]. Figure 6.20 shows a screenshot of Stepper.

There are many other similar tools. Given limited resources, I selected a subset of representative tools. Table 6.3 explains why they are selected. For a more comprehensive survey, see Sorva's dissertation [74].

6.4.2 Differences in Presented Information

Tables 6.4 to 6.7 compare these tools based on their presentation of the following aspects of program state:

The Current Task What the system is currently doing (e.g., "calling a function", "summing two numbers", "returning from a function"). (Table 6.4; See Section 6.7.1 for further discussion)

Table 6.3: Basic Information about the Compared Tools

Tool	Why Included?	Source URL	Languages
Env	A recently developed tool that—like the Stacker—emphasizes environments.	`https://sourceacademy.org/playground`	JavaScript, Scheme[a]
OPT	A widely used and actively developed tool that has drawn significant attention from both users and researchers.	`https://pythontutor.com/`	Python, JavaScript, Java, and non-SMoL languages (e.g., C, C++)
JK	A tool grounded in extensive research and exploration of the design space for trace visualizations.	`https://acos.cs.aalto.fi/jsvee-transpiler-python`	Python (limited: no nested or first-class functions)[b]
Jeliot	One of the earliest tools—a pioneer in program trace visualization.	`https://code.google.com/archive/p/jeliot3/`	Java[c]
Stepper	An unusual tool that uses substitution-based semantics to present evaluation steps.	`https://docs.racket-lang.org/stepper`	A pure subset of Racket[d]

[a]Scheme support appears to be broken at the time of writing. The system displays a long linked list that is not constructed by the program. My comparison is therefore based entirely on the JavaScript version.

[b]Based on private communication with the authors. They also note that the tool can animate first-class functions and other languages (e.g., Scala), but such animations cannot currently be generated entirely automatically.

[c]This may or may not be the "official" version of Jeliot, but it is the most accessible one I could find online. I found no significant differences between this version and the screenshots from published work on Jeliot 3 [5, 45].

[d]See the Stepper documentation for more details.

The Continuation What the system will do after the current task is complete. (Table 6.5; See Section 6.7.2 for further discussion)

Environments How the tool represents the bindings and scopes of variables. (Table 6.6)

The Heap How the tool shows heap-allocated values, including functions and data structures. (Table 6.7)

Sections 6.7.1 to 6.7.4 further discuss the differences, including proposing alternative designs and comparing the pros and cons of the design options.

6.4.3 Other Differences Among the Tools

This section discusses additional differences among the tools that are not covered in previous sections.

Neighboring States Among the surveyed tools, only the Stepper presents a *step*—that is, a state along with its immediate successor. Most tools present one state at a time.

Some tools do not present neighboring states in their entirety but still provide some information about previous states. OPT (Figures 6.15 to 6.17) marks the source code line of the previous current task with a light green arrow. JK (Figure 6.18) provides textual explanations about the previous state (e.g., "Fetching value 2" in the figure).

Addresses vs. Arrows Environments and values form references. Only the Stacker (Figure 6.13) and JK (Figure 6.18) present references as textual labels. The Env tool (Figure 6.14), OPT (Figures 6.15 to 6.17), and Jeliot (Figure 6.19) present references as arrows. JK also displays all related arrows when users hover their mouse over a reference. (This discussion does not apply to the Stepper.)

Table 6.4: A Comparison on the Presentation of the Current Task. See Section 6.7.1 for further discussion.

Stacker	**Env**	**OPT**	**JK**	**Jeliot**	**Stepper**
Shows the current term with its environment if evaluating a term. Otherwise, only relevant values are shown.	The corresponding line is highlighted with a blue background. The exact task is shown at the end of the continuation and is highlighted, along with its associated environment, using a blue border.	The corresponding line ("next line to execute") is indicated by a red arrow. The associated environment is highlighted with a light blue background.	The corresponding line is indicated by a blue arrow. Some values related to the current task are shown in the topmost stack frame, within a light yellow region.	The corresponding term is highlighted with a purple background. The associated environment appears as the topmost panel in the "Method Area".	The corresponding term is highlighted with a green background. (A purple background highlights the result of performing the task.)

Table 6.5: A Comparison on the Presentation of the Continuation. See Section 6.7.2 for further discussion.

Stacker	**Env**	**OPT**	**JK**	**Jeliot**	**Stepper**
The local continuation is shown as a local evaluation context (elided when empty). The call stack is displayed as a list of stack frames, each containing a local evaluation context and an environment.	No information about the continuation is shown without the extension by Abad and Henz [1]. The extension presents the entire continuation as a sequence of low-level instructions (e.g., `env` and `mark`).	Each "frame" consists solely of an environment—local evaluation contexts are not shown. The ordering of frames in the call stack is unclear.	Similar to the Stacker's presentation, but local evaluation contexts are not shown in full.	The call stack is displayed as a stack of panels in the "Method Area". No local evaluation context is shown.	The entire continuation is shown as an evaluation context, with variables replaced by their values.

Table 6.6: A Comparison on the Presentation of Environments. See Section 6.7.3 for further discussion.

Stacker	**Env**	**OPT**	**JK**	**Jeliot**	**Stepper**
Environments are shown separately from the call stack. The built-in ("primordial") environment is referenced but not displayed. The environment hierarchy is conveyed through the "extending" fields of the presented environments. Declared-but-uninitialized variables are represented using a bomb emoji.	Environments are shown separately from the call stack. The built-in environment (`Global`) is displayed on screen, but its contents are hidden. The environment hierarchy is represented by arrows pointing from child environments to their parents. Declared-but-uninitialized variables are shown as blank.	Environments are integrated into stack frames. The built-in environment is not shown. The presentation of the environment hierarchy varies by language: when tracing Python or JavaScript, the top-level environment is labeled as the "Global frame"; for Python, the hierarchy is indicated by annotations such as `[parent=<parent ID>]` in frame IDs when the parent is not the "Global frame"; for Java, environments are shown alongside static fields, but without any indication of the hierarchy. When running JavaScript, declared-but-uninitialized variables are shown as `undefined`; in other languages, they are omitted entirely.	Environments are integrated into stack frames. The built-in environment is not shown. The environment hierarchy is not displayed, which is understandable given the tool's language limitations: all meaningful environments extend directly from the top-level environment. Declared-but-uninitialized variables are not shown.	Environments constructed by method calls are integrated into stack frames. Other environments—such as static fields and instance fields—are displayed in separate areas (i.e., the "Constant and Static Area" and the "Instance and Array Area"). This separation, while complex, is perhaps unavoidable when tracing Java programs. Java has intricate rules for variable resolution, and variables are tightly coupled with objects and classes, which themselves form a separate hierarchy. As a result, presenting environments in Java tools is intrinsically complex. Declared-but-uninitialized variables are not shown.	Environments are implicitly presented by replacing free variables with their corresponding values.

Table 6.7: A Comparison on the Presentation of the Heap. See Section 6.7.4 for further discussion.

Stacker	**Env**	**OPT**	**JK**	**Jeliot**	**Stepper**
Functions and data structures are displayed in the same region. Function signatures and bodies can be revealed by clicking on their source code location.	Functions and data structures are displayed in the same region. Each function is represented as two horizontally adjacent disks: the right disk points to the associated environment, and the left disk reveals the function's signature and body when hovered over. Vector indices are also shown on hover.	Functions and data structures are displayed in the same region. However, each language mode omits some information—for example, the Java mode does not display functions at all.	Functions and data structures are presented in separate regions. Functions appear alongside the primitive operators used. This design may be related to the fact that all functions must be defined in the top-level block.	Functions are not presented.	Value contents are integrated into the continuation.

Granularity of Steps The Stacker typically produces shorter traces than other tools. Other tools often generate longer traces because they break down operations like variable definitions, vector construction, and delta reductions into finer-grained steps.

Stepper provides case-by-case control, allowing users to pick the step size using dedicated buttons.

Color Coding All tools except the Env tool (Figure 6.14) use background color coding to visually distinguish different types of elements.

Web Access and Permalinks All surveyed tools are web-based except for Jeliot and the Stepper.

Among the web-based tools, only the Stacker and OPT provide a button to generate permalinks for sharing the current state.

Multi-lingual Support and Translation The Stacker, the Env tool (Figure 6.14), and OPT appear to support multiple natural languages.

Among them, only the Stacker offers a translation feature, though it currently supports only unidirectional translation from the SMoL Language.

Looping Constructs Only the Stacker and the Stepper do not support looping constructs.

Smooth Transitions Only Jeliot and JK present transitions between states using smooth animations.

6.5 User Study 1: Stacker vs. Algebraic Stepper in DrRacket

During the early development of the Stacker, I conducted a study comparing it with the Algebraic Stepper in DrRacket. This study took place in the Accelerated Intro course (see Chapter 9 for details) in mid-2022. Students in this course may have had some prior

Figure 6.21: The programs used in the Stacker-vs-Stepper study (Section 6.5)

```
;; Program 1

(define (singles alon)
  (cond
    [(empty? alon) empty]
    [(cons? alon)
     (cons
      (first alon)
      (singles
       (remove-leading-run (first alon) (rest alon))))]))
(define (remove-leading-run val alon)
  (cond
    [(empty? alon) empty]
    [(cons? alon)
     (cond
       [(equal? val (first alon))
        (remove-leading-run val (rest alon))]
       [else alon])]))
(singles (list 1 1 2 1 1 1))


;; Program 2

(define (double-up ns)
  (cond
    [(empty? ns)
     empty]
    [(cons? ns)
     (cons (first ns)
           (cons (first ns)
                 (double-up (rest ns))))]))
(double-up (list 2 1 3))
```

Figure 6.22: The special-version Stacker used the Stacker vs Stepper study (Section 6.5)

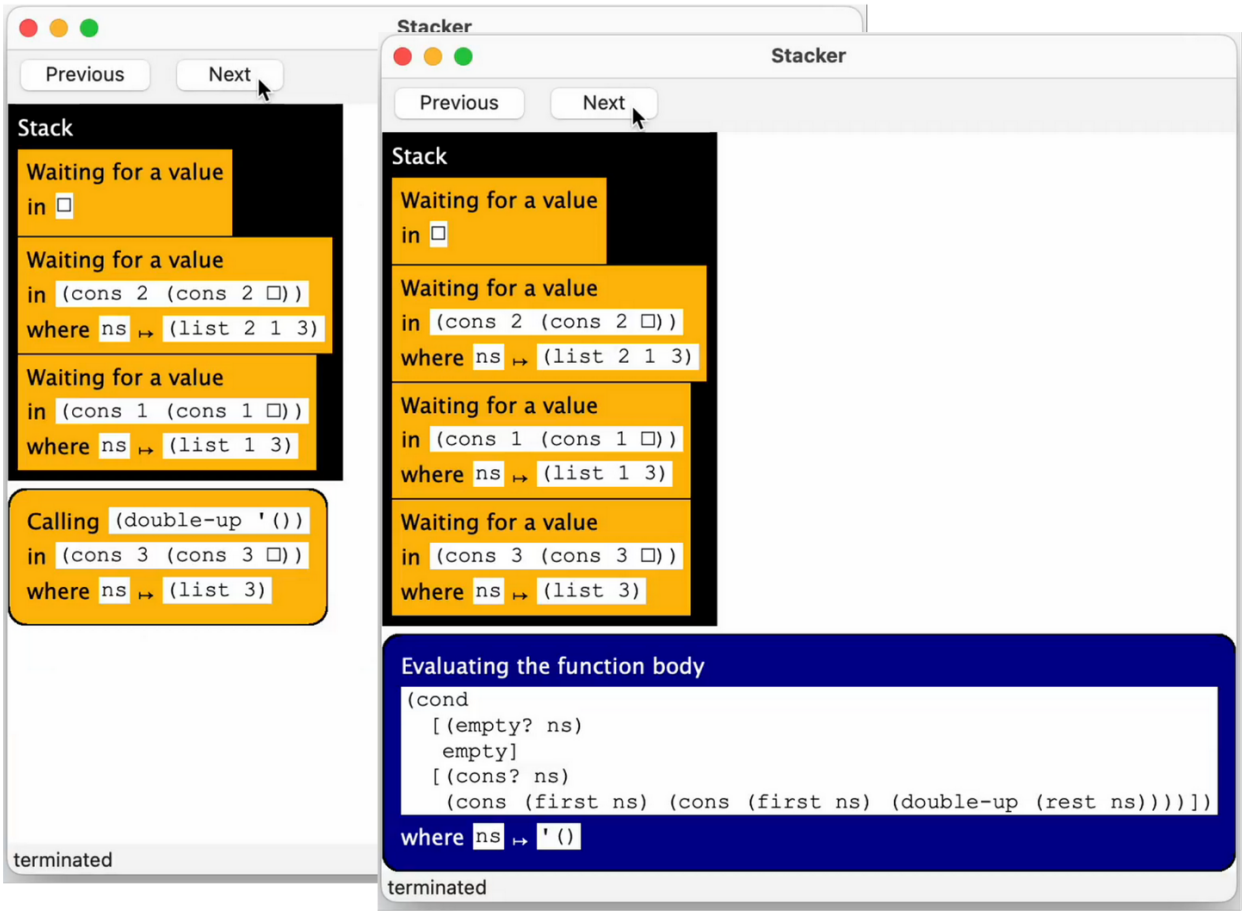
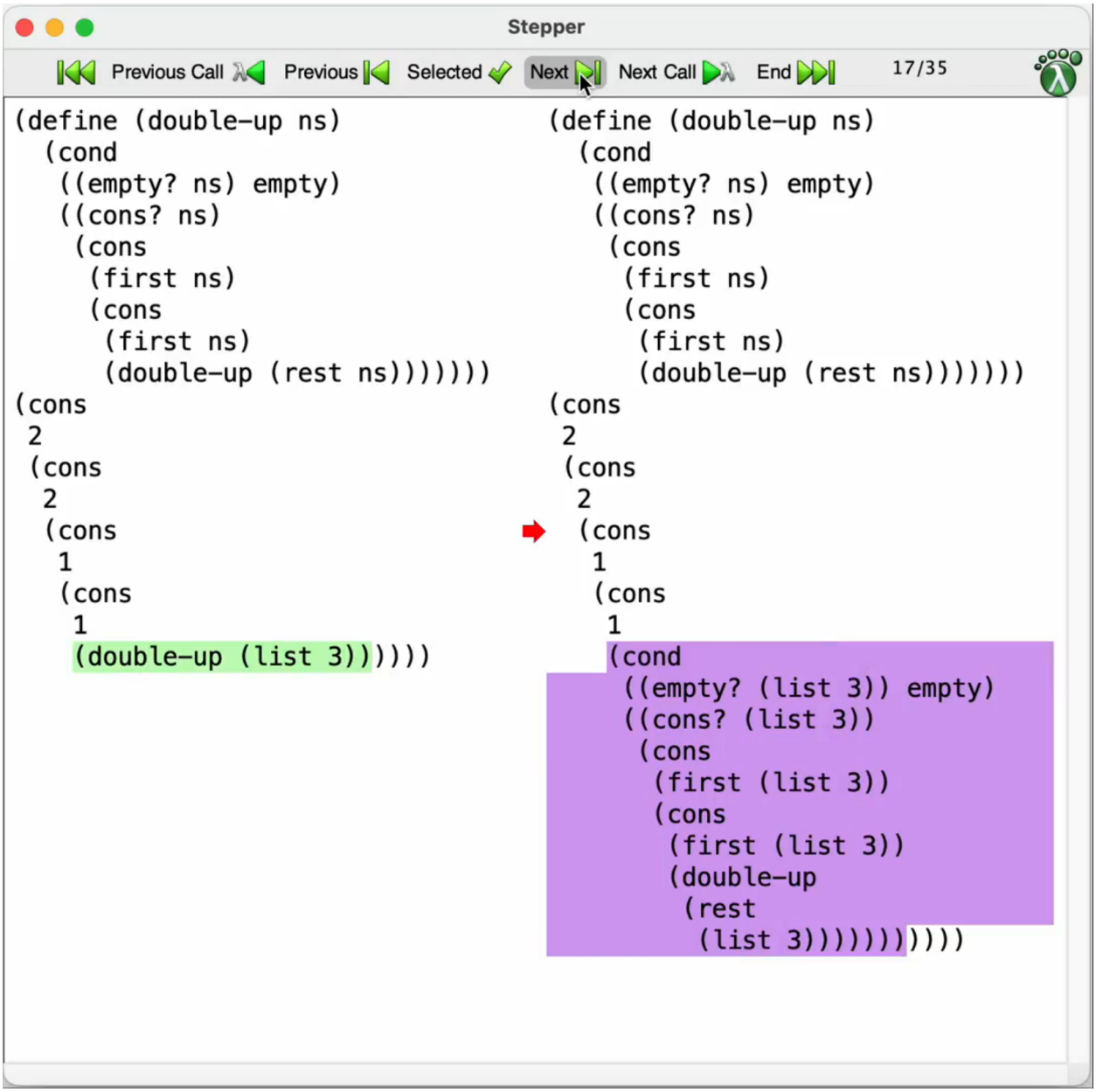


Figure 6.23: The Stepper tracing a program used the Stacker vs Stepper study (Section 6.5)



exposure to debuggers and limited familiarity with the Stepper, either through reading *How to Design Programs* (HtDP) or using DrRacket. However, they had not received formal instruction on using the Stepper. They had certainly never seen the Stacker and likely had never encountered evaluation contexts.

6.5.1 Key Tool Differences

This study uses a special version of the Stacker (shown in Figure 6.22). This version differs significantly from the current one. Notably:

- It presents only the first column, which includes the stack and the current task.
- Environments and heap contents are integrated into the call stack.
- Holes in evaluation contexts are rendered as "□" rather than "•".
- Racket is the only supported language.

Figure 6.23 illustrates the Stepper showing the "same" state as in the Stacker figure (Figure 6.22).

This study essentially compares two different approaches for presenting the current task, continuation, and environment:

Stacker presents the continuation as a sequence of stack frames, each representing a local evaluation context paired with an environment for resolving variable bindings. The current task is shown in a box below the stack.

Stepper presents the continuation as the entire program, using substitution to resolve variable bindings. The current task is highlighted in green.

One other notable difference is that the Stepper shows roughly twice as many intermediate states.

6.5.2 Study Objectives

The students likely had never encountered evaluation contexts before. This study examines how well they could comprehend the Stacker's novel presentation.

The Stepper (and indeed any substitution-based tool) is likely less effective when working with data structures. Data structure presentations are often significantly longer than variable names. Therefore, when substitution replaces variables bound to data structures, the visual state may change considerably, especially if the substituted structure is much larger than the original name.

6.5.3 Study Content

The study used two programs (Figure 6.21), each traced by both tools. The programs were carefully selected to manipulate "long" data structures: Program 1 defines recursive functions that consume long lists; Program 2 defines recursive functions that produce long lists.

The four traces were presented as videos. Students were asked to watch the videos and then complete tasks designed to assess their understanding of the tools and to gather their preferences between them.

6.5.4 Results

Students demonstrated a reasonable understanding of evaluation contexts. They were able to make connections between the evaluation contexts in the Stacker and the annotated programs in the Stepper:

- They recognized that the "□" in the bottom-most box of the Stacker corresponds to the green highlight in the Stepper.
- They also noticed that the evaluation contexts (i.e., the fields labeled "in") matched

the parts of the program excluding the highlighted expression.

Students also successfully related the environments in the Stacker (i.e., fields labeled "where") to the substitutions made by the Stepper. This is not surprising, as many students may have had experience using debugging tools.

The following student comments are particularly insightful:

- Some students observed that when debugging, one could use the Stacker to identify the problematic function call and then "zoom in" with the Stepper. (See Section 6.7.6 for further discussion.)
- Many students considered the Stacker more helpful for understanding recursion. Several noted that the environments in the Stacker made it easier to track the parameters of recursive calls.
- Several students said the Stacker helped them better understand "how much DrRacket has to do to compute the function". This perception may stem from UI differences: the Stacker breaks the continuation into distinct frames, making function calls more visually prominent. Each new frame visibly appears during execution, suggesting a cost model in which each function call allocates a new frame. In contrast, the Stepper presents the continuation as a single large evaluation context, with smaller and subtler UI updates during function calls.

Students also provided tool-specific feedback:

On the Stacker

- Some disliked that information such as "Calling `(double-up '())`" disappears once the current task becomes a stack frame. (See Section 6.7.2 for further discussion.)
- Several found the labels "in" and "where", along with the $x \mapsto \ldots$ notation, confusing. In response to this, the Stacker was updated to use more descriptive field names. For instance, "in" was renamed to "in context" and "where" to "in environment".

On the Stepper

- Students were confused by the distinction between "Next" and "Next Call".
- They felt there were too many intermediate steps, making it easy to get lost.
- As expected, some commented that function calls were replaced by large expressions, obscuring their structure.
- Others noted that the interface displayed too much text.

Overall, students expressed mixed preferences between the tools, though more students preferred the Stepper. Evaluation contexts themselves did not appear to be a problem, but the concise labels did. The substitution-related issue described in Section 6.5.2 was confirmed, though not as severe as anticipated.

It is worth noting that the results should be interpreted with caution for the following reasons:

- I am not entirely sure how deeply students understood the UI. There was no "deep" comprehension test, such as asking students to fill in a state template based on a program. Students may have inferred meaning from function names (e.g., `double-up`) and the source code itself.
- This student population was particularly selective and may have had prior experience with other tracing tools, such as `gdb`. As a result, they may have found labels like "stack" more meaningful than novice students would.

Nevertheless, I took this study as formative feedback for the UI design of the Stacker.

6.6 User Study 2: Stacker vs. Online Python Tutor

I conducted a study comparing the Stacker and the Online Python Tutor (OPT) to better understand the design space of tracing tools. The study was carried out in the PL course

population (Chapter 9) in late 2022.

Since then, the Stacker has evolved from a desktop application to a web-based tool, added support for non-Lispy syntaxes, and undergone minor UI tweaks. Nevertheless, I believe lessons I learned from the study still apply.

6.6.1 Survey Content

The study asks students to run essentially one program in the two tools. The program for the Stacker is written in the SMoL Language (Chapter 3):

```
(defvar make-counter
  (lambda (n)
    (defvar f
      (lambda ()
        (set! n (+ n 1))
           n))
    f))

(defvar c1 (make-counter 0))
(defvar c2 (make-counter 0))
(c1)
(c1)
```

The OPT version of the program is written in Python 3:

```
def make_counter(n):
    def f():
        nonlocal n
        n += 1
        return n
```

```
    return f


c1 = make_counter(0)
c2 = make_counter(0)
print(c1())
print(c1())
```

Both programs output `1` followed by `2`.

Students were instructed to run both tools in sequence, with the order randomly chosen by the survey system. The following list shows the prompts presented when the survey chose to ask about the Stacker before OPT. In the other version, the phrases "Stacker" and "Python Tutor" are simply swapped.

1. Below you are given a program to run in Stacker. As you run this program in Stacker, please record what you *notice* about the tool: _____
2. Once you are done running, please record what you *wonder* about the tool: _____
3. Below you are given an *equivalent* program to run in Python Tutor. As you run this program in Python Tutor, please record what you *notice* about the tool: _____
4. Once you are done running, please record what you *wonder* about the tool: _____
5. Now we would like you to compare and evaluate these two tools. In what follows, please ignore the programming languages; imagine both tools support the same language(s).
6. In what ways did you like Stacker more than Python Tutor and vice versa? Please make clear which tool you are referring to.
7. In an absolute sense (not relative to the other tool), what did you like and dislike about Stacker?

8. In an absolute sense (not relative to the other tool), what did you like and dislike about Python Tutor?

9. In what kinds of situations (if any) would you recommend a classmate use one tool rather than the other?

10. Do you have any other comments about these tools?

6.6.2 Results and Discussion

Students noticed and liked the column-based design shared by both tools. See Section 6.7.7 for further discussion.

Students preferred labeling all columns, as OPT did, whereas the version of the Stacker used in the study labeled only the first. As a result, the Stacker now has a label for every column. See Section 6.7.7 for further discussion.

For indicating the current execution state, OPT highlights the current corresponding line and the previous corresponding line in the source code. In contrast, the Stacker presents local evaluation contexts, which are program fragments with a hole that precisely indicate the currently executed expression. Students had mixed preferences regarding the two tools. They note that the Stacker is more accurate about the current execution state, and that OPT is simpler because it shows less information on the screen and because it makes it easier to relate the current execution state with the source program. See Section 6.7.1 for further discussion.

OPT includes buttons for jumping to the start or end of the execution trace, a feature absent in the Stacker. Students liked these buttons. See Section 6.7.15 for further discussion on this design choice.

OPT represents environment relationships using arrows, while the Stacker displays addresses. Students generally found arrows more readable, though some noted that an arrow-based UI could become overly cluttered as the relationships grow more complex. See Sec-

tion 6.7.8 for further discussion on this design choice.

Unlike OPT, which does not display declared-but-uninitialized variables, the Stacker represents them using a bomb emoji. Students preferred the Stacker's approach, with many commenting that it helped them realize such variables exist even before initialization. See Section 6.7.3 for further discussion.

OPT is not always explicit about parent environments: typically omitting them when the parent is the "global frame". In contrast, the Stacker consistently displays parent environments, a feature students preferred. See Section 6.7.3 for further discussion.

Students appreciated color-coded UI elements, which the Stacker uses extensively, but OPT only color-code heap-allocated values (i.e., "objects").

Overall, students noted that the Stacker presented more information and did so more accurately. However, they also found it initially more overwhelming to use. See Section 6.7.5 for further discussion.

6.7 The Design Space Around Stacker

This section discusses the design choices around the current Stacker, their respective pros and cons, and directions for future work. These choices include both design decisions made in surveyed tools (Section 6.4) and new ideas inspired by the survey and the user studies (Sections 6.5 and 6.6).

The discussion focuses primarily on two perspectives: the Programming Languages semantics perspective—such as whether a design presents sufficient information to distinguish programs with meaningfully different behaviors—and the UI perspective—such as accessibility and how easily users can locate relevant information in the interface.

Sorva's dissertation [74] surveys a broader collection of tools, including those that require students to actively participate in constructing traces. His discussion emphasizes cognitive dimensions more heavily. For example, Table 15.1 highlights various cognitive aspects of

design. Readers interested in a cognitive perspective or a broader collection of tools are encouraged to consult his dissertation, especially Chapter 15.

6.7.1 Presentation of the Current Task

Highlighting Related Source Code Region Many surveyed tools highlight a region of the source code, with either a distinct background color or an arrow. Students appreciate this feature (Section 6.6). Designers must choose between highlighting the *line*, as in the Env tool (Figure 6.14), OPT (Figures 6.15 to 6.17), and JK (Figure 6.18), or highlighting the *term*, as in Jeliot (Figure 6.19). Highlighting the term is preferable, as it is more precise and has no apparent drawbacks. (**Future Work:** Highlight the current term.)

Highlighting the Current Environment Many surveyed tools also highlight the environment associated with the current task. The Env tool (Figure 6.14) uses a distinct border color, while OPT (Figures 6.15 to 6.17) changes the background color. JV (Figure 6.18) always places the current environment within the top-most stack frame. Jeliot (Figure 6.19) displays the environment at the top of its "Method Area". I personally find this kind of highlighting helpful for identifying the current environment. (**Future Work:** Implement in the Stacker a design similar to the Env tool.)

Previewing the Outcome of the Current Task The Stepper (Figure 6.20) is the only surveyed tool that provides a preview of the current task's outcome. In fact, it displays the entire next state. While this is a useful feature, it must be designed carefully, especially when the outcome involves substantial information (e.g., when the current task is a function call).

6.7.2 Presentation of the Continuation

Breaking the Continuation into Segments The Stepper (Figure 6.20) and the Env tool (Figure 6.14) are the only surveyed tools that do not break the continuation into segments. This did not appear to be a problem for students in the Stacker-vs-Stepper study (Section 6.5). However, that study involved recursive functions with simple bodies, so the continuations were more regular than in most situations. It is conceivable that not segmenting the continuation would become problematic when the continuation structure is more complex.

A natural way to break the continuation into segments is to divide it into stack frames. All other surveyed tools (Figures 6.13 and 6.15 to 6.19) adopt this approach. I am not aware of any sensible alternative designs.

Evaluation Context vs. Instructions Details of the continuation—whether segmented or not—can be presented as an evaluation context, as in the Stepper (Figure 6.20) and the Stacker (Figure 6.13), or as a sequence of instructions, as in the Env tool (Figure 6.14). Unless learning the instruction set is itself a goal, the evaluation-context approach is preferable: the instruction-based approach introduces additional vocabulary, increasing the learning burden and obscuring the connection between the continuation and the source code.

Integrating Environments with Stack Frames Some surveyed tools, including OPT (Figures 6.15 to 6.17), JK (Figure 6.18), and Jeliot (Figure 6.19), integrate environments into stack frames. Others, including the Stacker (Figure 6.13) and the Env tool (Figure 6.14), present environments and stack frames as clearly separate entities.

Clements and Krishnamurthi [12] points out that integrating environments with stack frames may lead to a dynamic-scope misconception: the environment of the current frame may or may not extend from the environment of the "next" frame, while the typical stack arrangement visually suggests that it does. On the other hand, separating environments

from stack frames increases the number of visual elements. For tools that support a limited language where the environment hierarchy is trivial (e.g., restricting function definitions to the top-level block, as in JK and UUhistle), integration may be reasonable. But when the environment hierarchy can be more complex, clearly separating the two concepts seems to be the better design choice.

What to Include in a Stack Frame Presentation? A stack frame might present any subset of the following information:

- the local continuation, which may be presented completely (as in the Stacker, Figure 6.13) or partially (as in JK, Figure 6.18);
- the associated environment (presented by all tools);
- the reason the stack frame was created (not explicitly shown in any surveyed tool);

To support all the exercises listed in Section 6.3, a tool likely needs to show both the complete local continuation and the associated environment.

While no surveyed tool explicitly presents the reason a stack frame was created, they all implicitly convey this information by showing function calls. However, this information is lost once a function call becomes a stack frame. Students expressed dissatisfaction with this disappearance (Section 6.5).

Presenting this information risks breaking authenticity—real language implementations (unless in debugging mode) do not need to preserve the reason for each stack frame in order to compute the final result. I believe the ideal design should retain this information in a way that avoids misleading students into thinking that real implementations also preserve it. (**Future Work:** Add a question mark icon to each stack frame and reveal the corresponding function call when the user hovers over the icon.)

6.7.3 Presentation of Environments

Substitution vs. Environments Substitution, as used in the Stepper (Figure 6.20), does not require knowledge of environments or their hierarchical structure. As such, it may be more approachable for beginning students than environment-based semantics (used in all other surveyed tools).

However, substitution has several notable drawbacks:

- It has language limitations—it does not work well with mutable variables, which are ubiquitous in modern programming languages.
- It presents usability issues—substituting a variable with its value or a function call with its body often causes substantial changes to the program state.
- Substitution may promote the "calling copies values" misconception, which is one of the known misconceptions (CallCopyStructs in Table 5.1).
- Substitution lacks authenticity with respect to real-world language implementations.

Compared to substitution, environments are more helpful for tracking function calls (Section 6.5).

Environments are certainly the better choice for experienced students. Even for beginners, I am not convinced that the benefits of substitution outweigh its usability issues or justify the transition cost to an environment-based model (assuming these students eventually want to learn about mutable variables).

The Primordial Environment Both the Stacker (Figure 6.13) and the Env tool (Figure 6.14) acknowledge the existence of a built-in or primordial environment, but only the Env tool displays it.

I find it unhelpful to show the primordial environment, as it provides no useful information and occupies valuable screen space.

Declared-but-not-initialized Variables Many surveyed tools do not show declared-but-uninitialized variables in their environments. The exceptions are the Stacker (Figure 6.13), the Env tool (Figure 6.14), and OPT with JavaScript (Figure 6.16). Showing these variables is critical for understanding hoisting behavior, which is present in many modern languages (e.g., Python and JavaScript, but not Java).

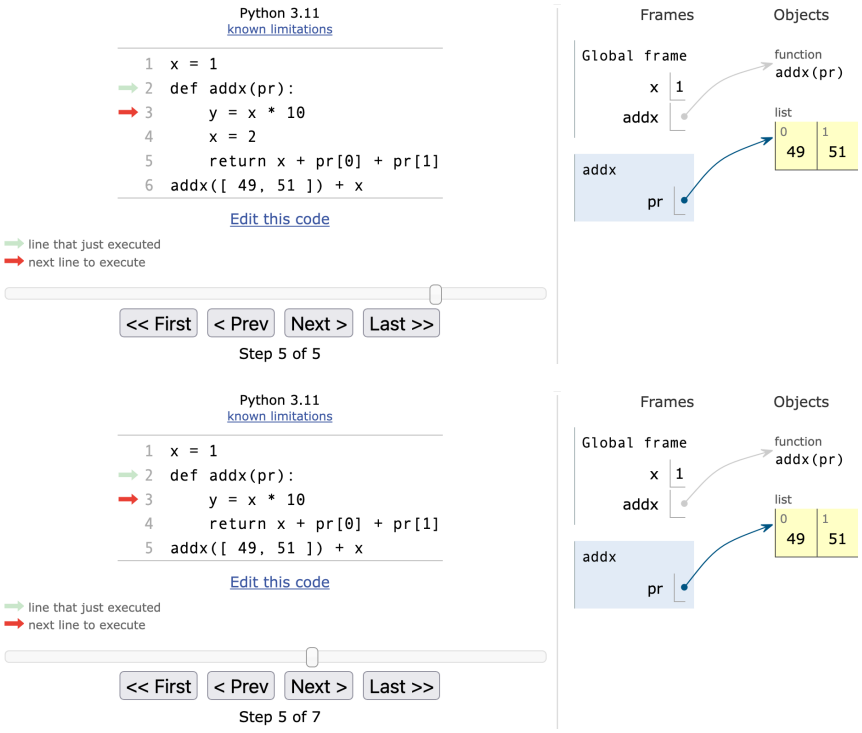
Consider the following two variants of the same program (used in Section 6.1 and Section 6.4). In the first, the line `y = x * 10` appears *before* `x = 2`:

```
x = 1
def addx(pr):
    y = x * 10
    x = 2
    return x + pr[0] + pr[1]
addx([ 49, 51 ]) + x
```

In the second variant, the new line replaces `x = 2`:

```
x = 1
def addx(pr):
    y = x * 10
    return x + pr[0] + pr[1]
addx([ 49, 51 ]) + x
```

The two programs treat the variable `y` very differently: the first raises an error because `y` refers to a variable that was declared but not initialized. The second binds `y` to `10`. Yet, the state just before `y = x * 10` appears identical in OPT:



Therefore, it is critical to show declared-but-uninitialized variables in languages that exhibit hoisting behavior. Moreover, students reported appreciating this information (Section 6.6).

The Environment Hierarchy Some surveyed tools, including the Stacker (Figure 6.13), the Env tool (Figure 6.14), and Jeliot (Figure 6.19), always make the environment hierarchy explicit. Others, like OPT with Python (Figure 6.15) and JavaScript (Figure 6.16), highlight only the top-level environment. Some tools, like JK (Figure 6.18), do not indicate the hierarchy, though users might infer it from the position of the bottom-most stack frame. I believe it is best to always be explicit about the environment hierarchy.

Separating Parameters and Local Variables The Stacker (Figure 6.13)—like all other surveyed tools (Figures 6.14 to 6.19) but unlike its predecessor [12]—does not distinguish between parameters and locally defined variables in environments. Clements and Krishnamurthi [12] separate the two: each environment has two sections, presenting parameters and local variables in their own areas without mixing them.

In my experience, making this distinction has not been helpful. Unless the tool targets a language where the distinction is meaningful (e.g., parameters are immutable), it is likely better not to separate the two.

6.7.4 Presentation of the Heap

Separating Different Kinds of Heap-allocated Values JK (Figure 6.18) separates function values and data structures. This makes sense for that particular tool, which imposes many restrictions on functions and effectively ensures that all function definitions must appear in the top-level environment.

Jeliot (Figure 6.19) separates classes and instances, which is appropriate for a tool designed for Java—a language in which classes and instances are key concepts.

When the language includes first-class functions, I believe it is most sensible not to separate functions from other heap-allocated values. Indeed, all applicable surveyed tools (Figures 6.13 to 6.16) follow this design.

Color-coding vs. Shape-coding The Env tool (Figure 6.14) uses different shapes to distinguish function values and data structures. The Stacker (Figure 6.13) and JK (Figure 6.18) use the same shape but different colors. OPT (Figures 6.15 to 6.17) uses both. Using both is probably the most usable and accessible approach. However, I find the simpler methods sufficient, and color-coding may be the easiest to implement.

Function Values A function value presentation might include any subset of the following information:

- the function name
- the parameter list
- the function body

- the associated environment
- the source code location

The three middle elements are essential for describing the program's behavior. The function name, shown by OPT with Python (Figure 6.15) and JK (Figure 6.18), and the source code location, shown by the Stacker (Figure 6.13), are also helpful because they assist users in connecting runtime state with the source program, which may aid understanding. Note that functions do not always have names—OPT with Python uses the symbol λ to represent `lambda` functions. (Anonymous functions are not supported in JK.)

Presenting function bodies can be tricky. Many surveyed tools do not present them at all. Even the ones that do (i.e., the Stacker and the Env tool) do not display them by default (Table 6.7). This is sensible: function bodies often contain a lot of text, which can make the UI overwhelming. Moreover, showing function bodies may lead to misconceptions. An earlier version of the Stacker *always* displayed function bodies, and some students (from the PL course described in Chapter 9) who used that version asked whether the heap stores copies of function bodies. That confusion seemed to disappear after the Stacker was updated.

Function environments are shown in the Stacker (Figure 6.13), the Env tool (Figure 6.14), and Jeliot (Figure 6.19)[3], but not in OPT (Figures 6.15 to 6.17) or JK (Figure 6.18). In JK's case, this might be reasonable since all environments are top-level due to its language restrictions (Table 6.3). For languages without such restrictions, I believe showing the environment is preferable, as omitting it risks reinforcing a dynamic-scope interpretation of functions.

The usefulness of presenting source code locations is unclear. The Stacker (Figure 6.13) is the only surveyed tool that includes them. Even as the main developer of the Stacker, I rarely find this information helpful for its intended purpose (relating function values back to their source code). It's difficult to map the location to a specific source code segment.

[3]Java objects/instances effectively serve as environments.

A better design might highlight the relevant source code when users hover over the source location.

Interestingly, while source code locations may not serve their intended purpose well, they are useful for telling whether two heap-allocated functions were constructed by the same code—if so, their source locations will be identical. That said, if the Stacker had always shown the function name and parameter list (as many other tools do), I probably wouldn't have needed to rely on source locations for this.

Overall, I believe function names and parameter lists should always be displayed. Tools might consider showing function bodies, but in a way that avoids the misconception that bodies are duplicated on the heap. Source code locations are less important. (**Future Work:** Change the Stacker so that function names and parameter lists are always shown, and revisit the usefulness of source code locations after this update.)

Vector Indices Vector indices are shown in the Env tool (Figure 6.14), OPT (Figures 6.15 to 6.17), and Jeliot (Figure 6.19). I think it's helpful to include indices. (**Future Work:** Present vector indices in the Stacker.)

6.7.5 The Total Amount of Information On the Screen

It is important to present states precisely and accurately, but also to control the total amount of information shown. In the Stacker-vs-OPT study (Section 6.6), many students felt that the Stacker displayed too much information—so much so that some preferred OPT, despite acknowledging the Stacker's greater accuracy.

Existing tools suggest strategies for gradually revealing information, which partially mitigate this issue. For example, the Env tool (Figure 6.14) hides most details about function values unless users hover over them (Table 6.7). The Stacker reveals those details only when users click on the function's source location (Section 6.1).

6.7.6 Number of States

What States to Present? As expected, the surveyed tools vary significantly in the states they present. I believe an ideal density should:

1. Avoid presenting states that are unlikely to be informative (e.g., reducing arithmetic expressions). For example, the Stacker omits states related to conditionals because students in the target population did not seem to struggle with them.

2. Ensure that each transition is easy to follow. For instance, advancing to the next state should not result in two items disappearing and three new items appearing—such a change would be difficult to track.

The Stacker presents many states related to function calls (see Section 6.1.2 for a full list of possible states). These often correspond to substantial changes in the "Stack & Current Task" column.

States right before mutations typically do not cause large UI changes, but are included because they relate closely to aliasing—a well-known source of misconceptions [26, 38, 80].

Similarly, states before print statements are also retained, despite not triggering major visual changes, because prior work Lu et al. [43] has shown that print behavior can be confusing.

(**Future Work:** Having identified new misconceptions (see Chapter 5), I plan to revisit which state types the Stacker presents. A notable omission in the current design is the state immediately before constructing heap-allocated values. Scope-related misconceptions seem connected to closure construction, while struct-copying misconceptions appear tied to data structure creation. I plan to update the Stacker to include these states.)

Other surveyed tools appear to include all the states that the Stacker presents, as well as additional ones—at least in the example program in Section 6.4. Given the consensus, I suggest that future tools present *at least* the states that the Stacker presents.

Multiple Levels of Detail In the Stacker-vs-Stepper study (Section 6.5), some students suggested that it would be useful to identify the problematic function call using the Stacker and then "zoom in" with the Stepper. The two tools differ significantly in the granularity of steps they present.

The Stepper (Figure 6.20) provides multiple "Next" buttons to support different step sizes. A key challenge in supporting multiple levels of density is communicating clearly what each level means. In the same study, some students expressed confusion about the distinction between the two "Next" buttons in the Stepper.

(**Future Work:** Add a button in the Stacker to support smaller step sizes.)

6.7.7 Major Areas

Arrangement of Areas The Stacker (Figure 6.13), the Env tool (Figure 6.14), OPT (Figures 6.15 to 6.17), and the Stepper (Figure 6.20) organize their visual elements into columns. JK (Figure 6.18) and Jeliot (Figure 6.19) also use rectangular regions, but these are not arranged strictly as columns.

I believe the column-based layout is superior. Students also prefer it (Section 6.6), and it aligns with the reading order of many natural languages.

Area Labels The Stacker (Figure 6.13), OPT (Figures 6.15 to 6.17), and Jeliot (Figure 6.19) label all their major areas. The Env tool (Figure 6.14) does not label any. I believe tools should include area labels—students appreciate them (Section 6.6), and there are no clear downsides.

6.7.8 Textual vs. Arrow References

References are represented as textual labels, arrows, or both in the surveyed tools (Section 6.4.3).

Labels necessarily occupy more screen space: each arrow must be replaced with one or

Figure 6.24: Online Python Tutor presenting a cyclic data structure

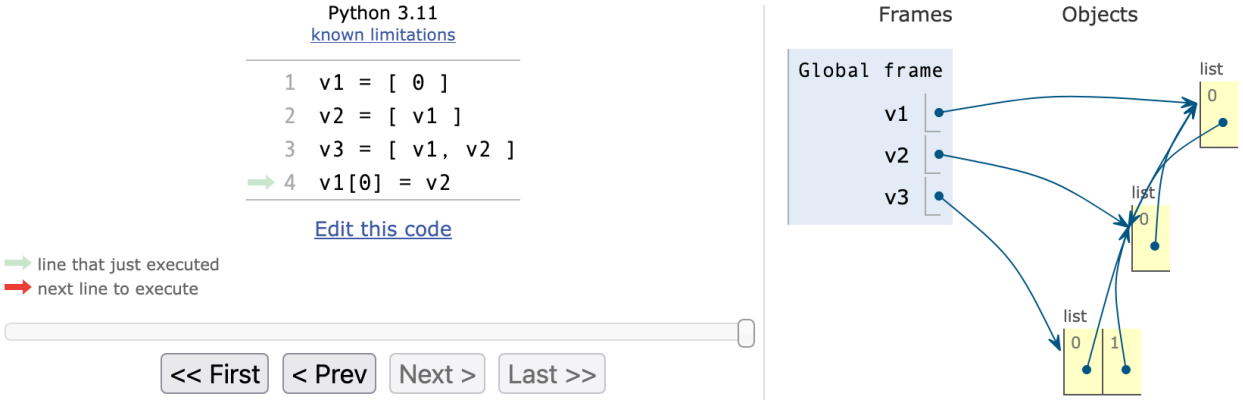
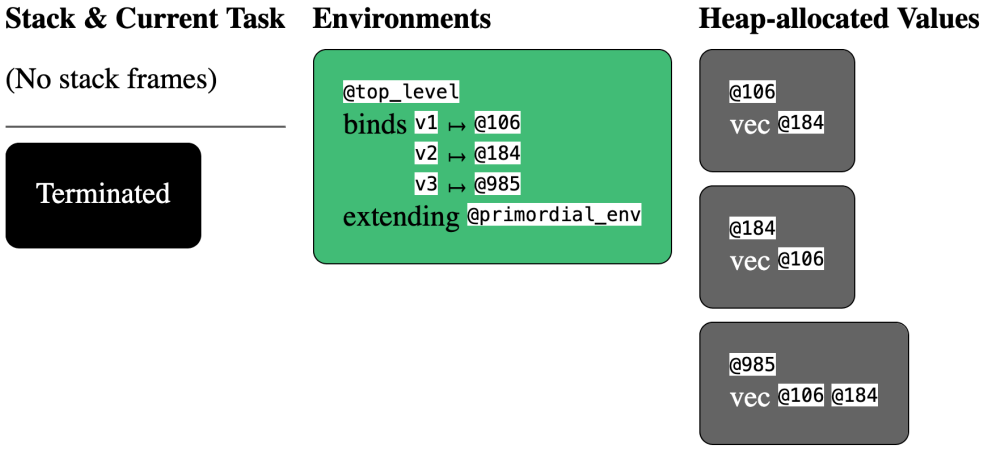


Figure 6.25: Stacker presenting a cyclic data structure



more labels. This increase in visual information—especially text—can raise the cognitive load for users. This might explain why some students found OPT more intuitive than the Stacker for simple references (Section 6.6).

Labels are also harder to follow than arrows. When users see a reference source and want to locate its target, an arrow provides a direct visual path. In contrast, labels require users to search across the screen to match identifiers.

However, the usability of arrows is highly sensitive to layout. Aesthetic aspects such as symmetry, edge crossings, and bends greatly impact graph readability [59]. Achieving a clean layout is non-trivial. For example, Figures 6.24 and 6.25 show OPT and the Stacker presenting essentially the same cyclic data structure—but the overlapping arrows in OPT

arguably make the references harder to follow than in the Stacker.

I conjecture that labels are less sensitive than arrows to reference complexity.

Promising hybrid approaches include:

Textual + Highlighting Highlight the referenced item when users hover over one of its labels.

Color-coding Labels Use background color or even emojis to differentiate labels.

Toggle Between Addresses and Arrows Let users switch between representations; default to arrows.

Show Arrows on Hover (JK's Approach) Display arrows only when the label is hovered over—reducing clutter while preserving clarity.

The Stacker originally avoided arrows because they were harder to implement on the web and because the authenticity of labels was seen as valuable in the PL course, where the Stacker was first deployed. After the Stacker-vs-OPT study (Section 6.6), I implemented the "Textual + Highlighting" approach in response to student feedback (Section 6.2.8). (**Future Work:** Implement the "Show Arrows on Hover" approach as well.)

6.7.9 Address Randomization

When (address) references are presented as labels, there is a choice between selecting labels from a clear sequence (e.g., 0, 1, 2) or using random labels, as in the Stacker.

I argue that random labels are preferable because they offer greater authenticity—real programming language implementations typically allocate resources in ways that appear unordered from the programmer's perspective.

However, random labels reduce reproducibility: two traces of the same program are unlikely to use the same labels in the same way, making structural comparisons across traces more difficult. This, in turn, complicates grading assignments involving the Stacker.

The Stacker addresses this grading challenge by controlling randomization through a seed. Permalinks (Section 6.2.5) preserve random seeds, ensuring consistent labeling. The seed can also be specified manually (Section 6.2.6). For many assignments (e.g., the one in Section 6.3.2), instructors can simply share permalinks with students when handing out assignments. In my experience, this approach suffices, but instructors may also ask students to use a specific seed.

The range of random addresses must also be chosen carefully. The Stacker uses natural numbers, which better reflect real machine addresses. It further restricts the range to 100–999 to avoid conflict with 0–99, which are convenient for illustrating programs involving arithmetic, while still keeping the randomized labels easy to read and discuss.

6.7.10 Web-based vs. Desktop

A web-based approach offers several advantages:

Easy Setup Students only need a browser to use the Stacker.

Built-in Accessibility Font sizes can be freely adjusted.

Simple Sharing Traces can be shared via URLs (as permalinks when no server state is required).

Editable Traces Since traces are HTML, users can edit them via browser developer tools.

The Stacker was originally a desktop tool. I moved it to the web for ease of setup—an unintended benefit was that permalinks turned out to be helpful in education.

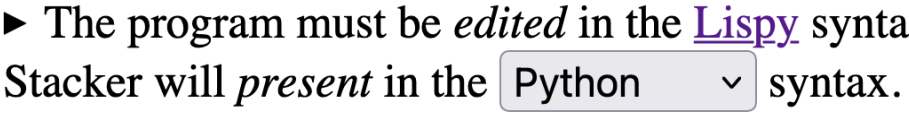
Web-based tools differ in their server cost models. OPT appears to be the most expensive [32]. In contrast, the Stacker operates with close to zero server cost:

- The only cost is serving the initial page load, which is static and therefore cacheable via CDNs.

- Evaluation happens entirely in the browser.
- Permalinks are self-contained `GET` requests, requiring no server-side state.[4]
- There is no persistent server state, so there are no ongoing space or time costs.

6.7.11 Supporting Multiple Languages

For multi-language tools that also support source-to-source translation, the UI design can be more complex. In the Stacker, users write programs in the SMoL Language, but can view traces in all supported languages. This distinction can be confusing. To mitigate this, the Stacker emphasizes the words "edit" and "present" in the UI:



Ideally, a multilingual tool should let users read and write in the same language. But this requires either separate runtimes for each language (as in OPT), or a core language plus compilers from each source language to the core. Maintaining multiple runtimes is a major undertaking. The core-language approach is also difficult—especially for traditional *textual* editors—because many language constructs fall outside the core, and even supported constructs may not map cleanly (e.g., Python's `nonlocal` and `global` keywords).

A multi-syntax *structural* editor might offer a more user-friendly editing experience while avoiding the engineering overhead of full multi-language support. (**Future Work:** I plan to add such a structural editor to the Stacker.)

6.7.12 Smooth Transitions Between States

State transitions are animated in JK (Figure 6.18) and Jeliot (Figure 6.19). Jeliot even provides a slider to control animation speed. Smooth transitions help users follow state

[4] `GET` requests typically have size limits ranging from 2 KB to 8 KB depending on the browser and server configuration. This bounds the size of programs that can be shared. In practice, however, this has not posed a problem.

updates, though they may be challenging to implement.

6.7.13 Accessibility Concerns

The Stacker's web-based architecture makes it easy to adjust font sizes.

The color palette (Table 6.1) is designed with accessibility in mind:

- Background colors were verified using Adobe Color [14] to be colorblind-friendly.
- Most foreground-background combinations meet WCAG 2.0 level AAA contrast standards, verified via WebAIM [88]. The only exception is white on #646464, which meets level AA.

6.7.14 Prerequisite Knowledge

There is a trade-off between faithfully representing program execution and minimizing the prerequisite knowledge required. Using standard terminology enhances authenticity, but may confuse students unfamiliar with certain concepts, requiring instructors to intervene.

How to balance this trade-off depends heavily on the teaching context; there is no universally optimal solution.

The Stacker currently avoids relying on system-level course knowledge by making a few minor adjustments:

- Using "current task" instead of "program counter".
- Displaying memory addresses as small random numbers (100–999) instead of hexadecimal.

6.7.15 Trace Navigation

First & Last Buttons

In the Stacker-vs-OPT study (Section 6.6), students expressed appreciation for OPT's "First" and "Last" buttons (Figures 6.15 to 6.17), which jump to the start and end of the trace.

A "First" button is also present in the Env tool (Figure 6.14), OPT, and JK (Figure 6.18). The Stacker effectively provides this functionality: users can click "Stop" and then "Run" to reach the first state. However, a single click would be more convenient. (**Future Work:** Add a "First" button to the Stacker.)

A "Last" button introduces a more nuanced design challenge. The final state of a trace may not exist if the program runs indefinitely.

Still, several tools provide a "Last" button, including the Env tool (Figure 6.14), OPT (Figures 6.15 to 6.17), and the Stepper (Figure 6.20). These tools handle long or infinite traces differently:

- The Env tool imposes a cap of 1000 steps. It appears to use heuristics to detect certain kinds of infinite loops. For example, if the source code includes `while (true)`, it shows a warning:
  The loop has encountered an infinite loop. It has no base case.

- OPT imposes a cap of 1000 steps. If the trace exceeds this, it warns:
  Stopped after running 1000 steps. Please shorten your code, since Python Tutor is not designed to handle long-running code.

- The Stepper pauses at the 500^{th} state and prompts the user to continue. Users can extend the limit incrementally or remove it entirely. If they decline, the last traced state becomes the final one.

I believe the Stepper's design is the most effective. (**Future Work:** Implement this design in the Stacker.)

Progress Bar

A progress bar for navigating the trace is available in the Env tool and OPT. Its utility is unclear. The Stacker-vs-OPT study (Section 6.6) did not find notable student preferences for this feature.

Further investigation is needed to determine whether progress bars are genuinely useful, and if not, how they might be redesigned to better support trace navigation.

6.7.16 Editable Traces

When visualizations are instructor-provided, it can be useful to annotate and customize them: adding text, arrows, and other highlights to match specific learning objectives, adjusting step granularity, or adding control buttons to allow students to skip certain steps.

Web-based tools naturally support editing to some extent (Section 6.7.10), but the level of support is highly dependent on the underlying technology. For example, states presented as backend-rendered images are much less editable than those rendered as HTML or SVG. Some editing tasks, such as adding control buttons, require in-depth knowledge of web technologies.

JK appears to be the only surveyed tool that allows users to add annotations to traces. It can even annotate traces generated by *other* tools, such as OPT, and, notably, the editing experience is WYSIWYG [71].

A future direction for the Stacker is to support exporting traces as HTML files with the `contenteditable` flag [15], allowing modern web browsers to function as WYSIWYG editors for the exported content.

CHAPTER 7

MISINTERPRETERS

7.1 Misinterpreters and Misconceptions

A **misinterpreter** is an interpreter that executes programs incorrectly. Misinterpreters are closely related to misconceptions, which are patterns of mistakes (Section 2.1). In the context of understanding program outputs, a mistake is an incorrect *pair* of a program and an output. A misconception, then, can be understood as a systematic pattern of such incorrect pairs—an incorrect *mapping* from programs to outputs.

Interpreters precisely define mappings from programs to outputs. By extension, misinterpreters provide a rigorous way to formalize misconceptions. A misinterpreter is essentially a definitional interpreter for a misconception.

7.2 The Value of Misinterpreters

7.2.1 Detecting Problematic Programs

Suppose I give students the following program:

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

This program outputs `12`. Now imagine a student predicts it will output `0`, because *"in `f`, `x` is set to `0`"*. What misconception underlies this prediction?

In fact, at least two distinct misconceptions could lead to this wrong answer:

CallByRef The parameter `x` aliases the top-level variable.

FlatEnv There is only one environment, so the parameter overwrites the top-level variable.

This example highlights a common challenge in misconception research: a single incorrect answer can stem from multiple, distinct misconceptions.

Failing to detect such ambiguity can be problematic. For example, if I overlook the possibility that FlatEnv also explains the wrong answer, I might incorrectly assume the student holds the CallByRef misconception. In that case, I might explain why `12` is correct in a way that aims to address CallByRef:

> You might think that the function call `(f x)` binds the parameter `x` to the top-level `x`, so changing the parameter would change the top-level variable. However, variables are bound to values, so the parameter `x` is bound to `12`, the value of the top-level `x`.

This explanation would likely not help a student who holds the FlatEnv misconception and might even be counterproductive. Therefore, it is crucial to reliably detect whether a wrong answer corresponds to multiple misconceptions.

However, detecting such ambiguity becomes time-consuming and error-prone as the number of misconceptions or programs grows. Misinterpreters help manage this complexity (as shown in Figure 7.1): by running a program through both the reference interpreter and each misinterpreter, we can check whether multiple misinterpreters produce the same wrong answer, suggesting ambiguity.

7.2.2 Fixing Test Programs

In addition to detecting ambiguity in test programs, misinterpreters also help tutorial designers develop better programs. For example, suppose we change the parameter name `x` and its references inside the function to `y`. Does this fix the problem? It is difficult to tell by manually interpreting the new program against all existing misconceptions. Running the modified version through the interpreters efficiently confirms that the wrong answer `0` no longer results from the FlatEnv misconception (Figure 7.2).

7.2.3 Maintaining a Growing Set of Misconceptions and Programs

Suppose we have gone through the laborious process of building an inventory of misconceptions and programs that unambiguously detect those misconceptions. What happens if we discover a new misconception? Each newly identified misconception may invalidate previously reliable programs. Therefore, it is critical to revalidate the entire inventory to check for any new ambiguity. Without misinterpreters, this revalidation would be tedious; with misinterpreters, the process can be largely automated.

Figure 7.1: Running a program through misinterpreters and the reference interpreter reveals that the wrong answer `0` can result from both the FlatEnv and CallByRef misinterpreters.

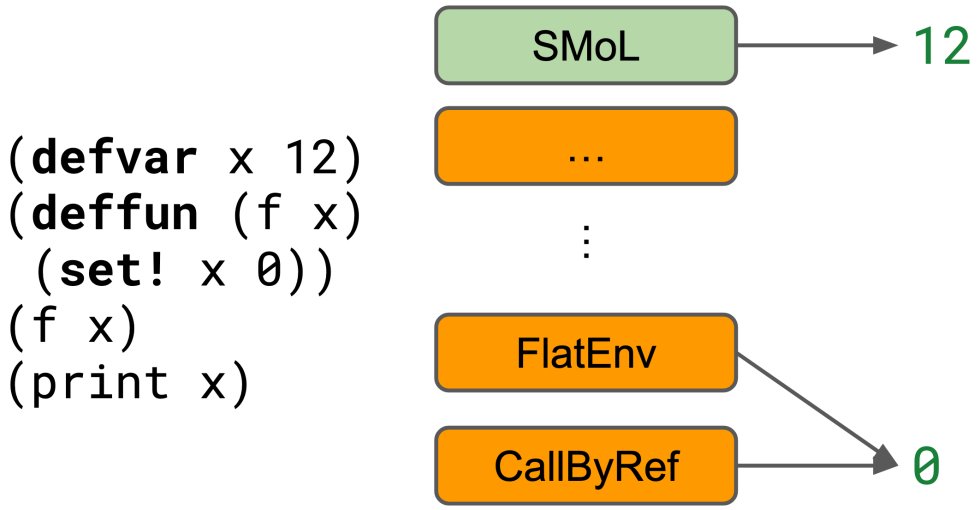
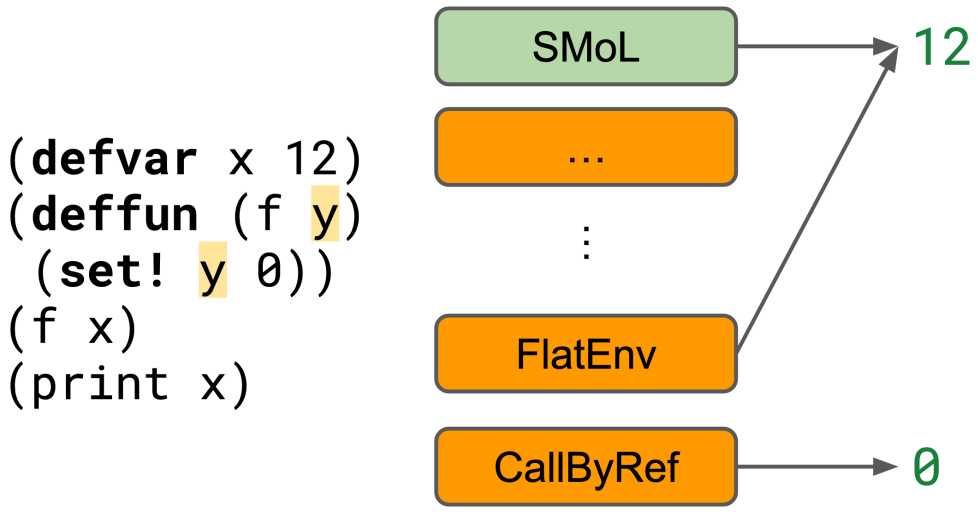


Figure 7.2: Running a modified program confirms that the wrong answer `0` cannot result from the FlatEnv misinterpreter.



7.3 Future Work

7.3.1 Synthesizing Diagnostic Programs

Section 7.2 illustrates how misinterpreters help instructors *manually* refine their test programs. This process could likely be performed *automatically* via *program synthesis*. The goal would be: given a reference interpreter R and a set of misinterpreters $\{M_i\}$, find programs p such that:

- At least one misinterpreter produces a wrong answer: $\exists i.\ M_i(p) \neq R(p)$
- No two misinterpreters produce the same wrong answer: $\neg\exists i, j.\ i \neq j \wedge M_i(p) = M_j(p) \neq R(p)$

7.3.2 Synthesizing Misinterpreters

A more ambitious application of program synthesis is to *synthesize misinterpreters themselves*, rather than test programs. Since misconceptions can be modeled by misinterpreters, and each misinterpreter corresponds to a cluster of mistakes, the task of identifying misconceptions could be reframed as: given a distribution of mistakes, synthesize a plausible collection of misinterpreters that explain them. Ideally, plausibility would be informed by cognitive models of human reasoning; in practice, minimizing the number of misinterpreters may serve as a reasonable heuristic.

7.3.3 Mystery Languages

The idea of misinterpreters is related to mystery languages [18, 56]. Both approaches use evaluators that represent alternative semantics for the same syntax. However, the two are complementary. In mystery languages, instructors design the semantic space with pedagogical intent, and students must create programs to explore that space. Misinterpreters, in

contrast, are driven by student input, while programs are provided by instructors. The two approaches also differ in their goals: mystery languages encourage students to experiment with language behavior, while misinterpreters aim to capture students' misconceptions.

It remains future work to investigate whether some misinterpreters could serve as effective mystery languages.

CHAPTER 8

MULTILINGUAL SUPPORT FOR SMOL

This chapter presents and discusses the **SMoL Translator**, a tool that translates SMoL programs into several commonly used programming languages. Section 8.1 describes the SMoL Translator and its key design choices. Section 8.2 provides a broader discussion.

8.1 The SMoL Translator

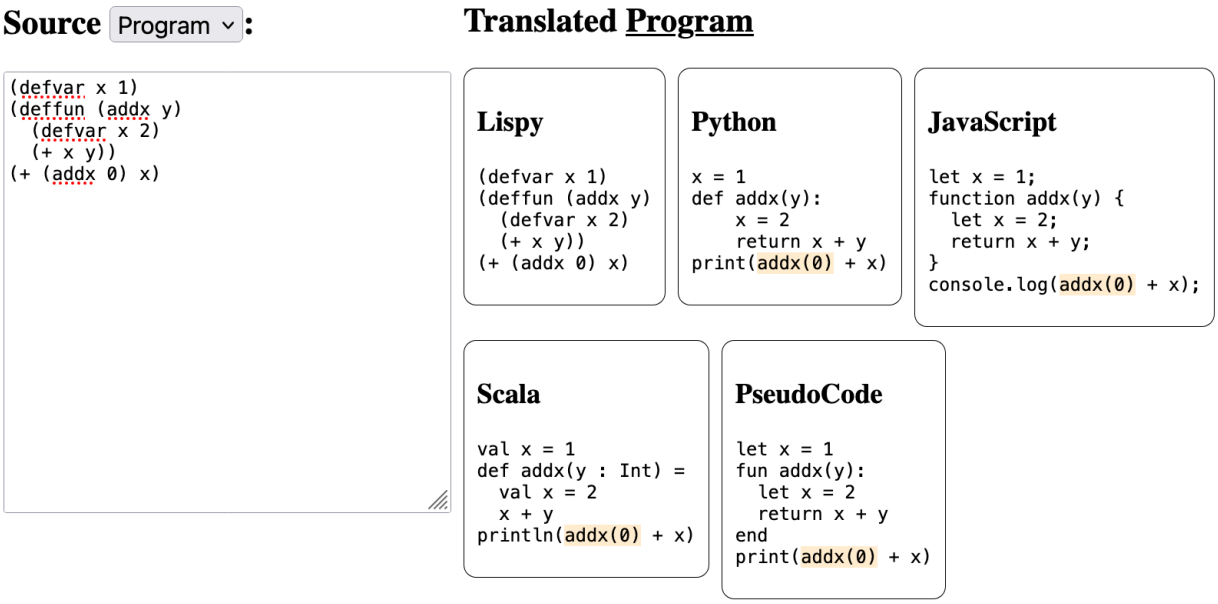
Figure 8.1 illustrates the UI of the **SMoL Translator**, which translates SMoL programs to JavaScript, Python, Scala 3 [WIP], and a pseudocode syntax.

The rest of this section presents key design decisions in the translation process. The presentation can be read as an experience report, which may interest readers looking to design similar tools. Perhaps more interestingly, it serves as an evaluation of how standard the SMoL Language is: if the SMoL Language were entirely standard, the translation would be straightforward, with few design decisions to make.

To make the content more digestible, I divide the discussion into the following subsections:

- Section 8.1.1 discusses design decisions that are broadly applicable, affecting many

Figure 8.1: The SMoL Translator highlighting the function call `(addx 0)`



supported target languages or commonly used programming languages.

- Section 8.1.2 focuses on design decisions likely specific to JavaScript.
- Section 8.1.3 focuses on design decisions likely specific to Python.
- Section 8.1.4 focuses on design decisions likely specific to Scala.

8.1.1 Widely Applicable Design Decisions

Infix Operations

The SMoL Language consistently uses prefix notation for all primitive operations. However, many languages offers certain primitive operations as infix expressions, such as "`a + b`". In such languages, parentheses are often need to disambiguate nested infix expressions, for example, differentiating between "`a - (b + c)`" and "`(a - b) + c`".

I considered the following solutions for handling nested infix expressions:

1. Parenthesize every infix operation.

2. Parenthesize an infix operation only when ambiguity exists.

3. Parenthesize every infix operation that appears immediately inside another infix operation.

The SMoL Translator implements solution 3. The issue with solution 1 is that it outputs programs like `f((a + b))`, which appear non-idiomatic.

Solution 2 is very demanding on engineering effort. Different languages have different precedence rules, resolution rules, etc. Therefore, the translator would have to know about every single language's rules. Furthermore, in some cases these rules are often somewhat embedded in their implementations, making it hard to reproduce exactly. Finally, they may change across versions.

In contrast, solution 3 works well in my experience and requires considerably less engineering effort than solution 2.

Alternative Syntax for `if` Statements

The SMoL Language uses a uniform syntax for `if` expressions. However, many languages provide an alternative syntax for `if` expressions when used as statements. Consider the following SMoL program

```
(defvar a 2)
(defvar b 3)
(defvar c (if (< a b) a b))
(if (< a b)
    (print c)
    (print (+ a b)))
```

A straightforward translation to JavaScript would use the conditional operator for both `if` expressions:

```
let a = 2;
```

```
let b = 3;
let c = (a < b) ? a : b;
(a < b) ? console.log(c) : console.log(a + b);
```

However, the output is likely more idiomatic if the second `if` expression is written as an `if` statement:

```
let a = 2;
let b = 3;
let c = (a < b) ? a : b;
if (a < b) {
    console.log(c);
} else {
    console.log(a + b);
}
```

This issue also applies to Python, although the concrete syntax is different:

```
a = 2
b = 3
c = a if a < b else b
if a < b:
    print(c)
else:
    print(a + b)
```

The SMoL Translator uses the statement-specific syntax whenever applicable to produce more idiomatic output.

Inserting "return"s

The SMoL Language dictates that functions return the value of the last expression in their body. However, many languages use explicit "return" statements (typically written as the keyword `return` followed by an expression, although the exact syntax may differ in some languages). When translating to such languages, the translator must insert appropriate "return" statements.

Inserting "return" keywords introduces a perhaps interesting interaction with conditionals: When the result of an `if` expression needs to be returned, and the expression is written in statement syntax, the "return" keyword must be inserted into the branches of the conditional.

The insertion of "return" also introduces a perhaps interesting interaction with assignment expressions. Consider the following SMoL program:

```
(deffun (swap pr)
  (defvar tmp (vec-ref pr 0))
  (vec-set! pr 0 (vec-ref pr 1))
  (vec-set! pr 1 tmp))
```

A straightforward JavaScript translation would be:

```
function swap(pr) {
    let tmp = pr[0];
    pr[0] = pr[1];
    return pr[1] = tmp;
}
```

While the translation is authentic, it is not ideal: Relying on the value of an assignment expression is widely considered poor practice, except for a few specific use cases (e.g., `x = y = e`), so the output program is not idiomatic; In this case, the translation also fail to preserve the semantics because, in the SMoL Language, assignments return the `None` value rather than

the more meaningful result.

To address this, the SMoL Translator inserts an empty "return" when the expression to be returned is an assignment.

Inserting "print"s

In the SMoL Language, expressions written in top-level blocks are automatically printed, following the tradition of Lispy languages. However, in many other languages, top-level expressions are not printed by default.

Currently, the SMoL Translator inserts a "print" for a top-level expression only if it is not an assignment. However, this heuristic does not always work as intended. Consider the following SMoL program:

```
(deffun (inc-first ns)
  (vec-set! ns 0 (+ (vec-ref ns 0) 1)))
(defvar my-numbers (mvec 1 2 3))
(inc-first my-numbers)
```

In this case, the translator would insert a `print` for `(inc-first my-numbers)`, but this is incorrect. I see two possible solutions to address this issue:

1. Change the semantics of the SMoL Language so that it does not automatically print top-level expressions;

2. Infer the types of top-level expressions and print only those whose type is not `None`.

I plan to implement solution 1 in the future.

Variable Naming Styles and Restrictions

The SMoL Language uses kebab-case for naming (e.g., `vec-len`). However, in many programming languages the hyphen (`-`) character is not allowed in variable names. These

languages typically use underscores (e.g., `vec_len`) or capitalization (e.g., `vecLen` or `VecLen`) to separate meaningful components.

In addition to character restrictions, many languages also have reserved names, such as `var` in JavaScript or `nonlocal` in Python.

One simple solution is to replace invalid variable names with generic ones, such as `x1`. However, the SMoL Translator takes a more sophisticated approach by outputting names that resemble the original names. For example, it translates names like `abc-foobar` to `abc_foobar` when targeting Python, and to `abcFoobar` when targeting JavaScript or Scala. If a natural translation results in collision with a reserved name, the translator prefixes the variable name (e.g., `$var`) to avoid conflicts.

Data Types

The SMoL Language assumes no type system. However, many languages, including a supported language, Scala, do have type systems. When translating to a typed language, preserving semantics can be challenging because many dynamic errors become static errors. This affects the program output, as a static error prevents any code from being executed, including prints that "happens before" the error. Furthermore, if the type system is not sufficiently expressive, some SMoL programs might not be typeable, resulting in no translation; On the other hand, if the type system is too sophisticated, type inference becomes difficult.

Currently, the SMoL Translator infers types using a unification-based algorithm, which can infer a type for the whole program if it can be "simply typed" in the sense of *Simply-Typed Lambda Calculus*. There is no strong rationale behind this design. An alternative design could involve making the SMoL Language typed, which is discussed in Section 8.2.

Mutability of Variables

In the SMoL Language, all variables are defined in the same way, regardless of whether the program mutates them or not. However, many languages distinguish between mutable

and immutable variables, and it is more idiomatic in those languages to specify whether a variable is (im)mutable.

To produce the most idiomatic translation, the translator should specify the mutability of variables. However, this approach may not always be desirable in an educational context, where the goal is sometimes to let students figure out which variables are mutated.

The SMoL Translator uses a heuristic that has worked well in practice: When the source program involves any mutation, the translator declares all variables as mutable; otherwise, all variables are declared immutable.

8.1.2 Design Decisions Specific to JavaScript

JavaScript is known for its reluctance to raise errors. Consider the following SMoL program, which produces an out-of-bounds error:

```
(defvar x (vec-ref (mvec 1 2 3) 9))
x
```

A straightforward JavaScript translation is:

```
let x = [ 1, 2, 3 ][9];
console.log(x);
```

The JavaScript version does *not* raise an error. In general, JavaScript does not throw an error when accessing an array out of bounds, nor when dividing by zero. See [87] for a perhaps amusing collection of non-erroring programs in JavaScript and other languages.

The SMoL Translator opts for the straightforward translation, even at the cost of occasionally not preserving the original semantics.

8.1.3 Design Decisions Specific to Python

Lambda Must Contain Exactly One Expression

Consider the following SMoL program, which prints `42`:

```
(defvar f (lambda (x)
            (defvar y 1)
            (+ x y)))
(f 41)
```

This program does not have a straightforward Python translation because, in Python, `lambda` bodies must contain exactly one expression. A workaround is to declare the local variable as a keyword argument:

```
f = lambda x, y=1: x + y
print(f(41))
```

However, this approach does not work if a local variable depends on parameters (e.g., `(defvar y (+ x 1))`).

The SMoL Translator takes an easier approach: It refuses to translate lambdas that involve definitions or multiple expressions to Python.

Scoping Rules

Python does not have a syntax for variable declaration or definition. When Python programmers want to define a variable, they do so by assigning a value to it. Python disambiguates between definition and assignment using the `nonlocal` and `global` keywords. Roughly speaking, a variable `x` is considered locally defined in the current function body (or the top-level block if there is no enclosing function body) unless it is declared as `nonlocal` or `global`. If declared `nonlocal`, the variable is considered defined in the nearest applicable function body, following typical lexical scoping rules; if declared `global`, the variable is

considered defined in the top-level block. The SMoL Translator translates both `(defvar x e)` and `(set! x e)` to `x = e`, inserting `nonlocal` or `global` declarations as needed to resolve ambiguity. However, some programs contain ambiguity that cannot be resolved. For example, the following SMoL program results in an error because it mutates a variable before defining it:

```
(set! x 2)
```

There is no Python translation that preserves the same semantics. In such cases, the translator produces translations that may not preserve semantics. For the example above, the translator outputs:

```
x = 2
```

8.1.4 Design Decisions Specific to Scala

In Scala, when declaring or using a nullary function, it is idiomatic to omit the empty argument list if and only if the function has no side effects. Currently, the SMoL Translator omits the empty argument list only if the entire program has no side effects. A more precise translation, which considers individual function side effects, is planned for future work.

8.2 Discussion

The SMoL Language (Chapter 3) and the SMoL Translator were developed to present programs illustrating SMoL Characteristics across multiple programming languages. This section evaluates how well the system (i.e., the SMoL Language and the SMoL Translator) serves this purpose and discusses potential improvements.

8.2.1 Desired Goals for Multilingual Presentation

Several goals are desirable for multilingual presentation:

Straightforwardness There is a clear one-to-one correspondence between terms.

Totality A program exists in every language of interest.

Semantics preservation The programs produce essentially the same output.

Idiomaticity Each program looks idiomatic in its target language.

The SMoL Translator achieves these goals in many cases; however, there are situations where some properties are compromised:

- **Totality** is compromised when the target language is typed or has restrictions on `lambda`s (see data types in Section 8.1.1 and Python `lambda`s in Section 8.1.3).
- **Semantics preservation** is compromised when the target language does not automatically print top-level expressions, is typed, exhibits unusual behavior with certain primitive operations, or has nonstandard scoping rules (see "prints" and data types in Section 8.1.1, JavaScript error issues in Section 8.1.2, and Python scope in Section 8.1.3).
- **Idiomaticity** is compromised when the target language uses infix syntax, declares mutable and immutable variables differently, has nonstandard scoping rules, or uses nonstandard conventions for writing argument lists (see infix operations and mutability in Section 8.1.1, Python scope in Section 8.1.3, and Scala's argument list conventions in Section 8.1.4).

Straightforwardness is always preserved, partly because the translation is mostly a structurally recursive algorithm, which maintains this property at low cost, and partly because I consider it critical for a multilingual learning experience.

8.2.2 Supported Constructs

As future work, I plan to include looping constructs, which are similarly widespread, as well as generators and async functions, which are becoming increasingly important.

8.2.3 Type Systems

Introducing a type system to the SMoL Language could improve totality and better preserve semantics when translating to typed languages. However, this comes at the cost of limiting the programs we can express, and would surely worsen semantics preservation when translating to untyped languages. Therefore, I do not plan to pursue this as a future direction. However, there could be a separate discussion on whether we should have a standard model of *typed* languages that cover both dynamics and statics (i.e., type systems) (Section 10.1).

8.2.4 Statements

The SMoL Language could become more similar to other languages, making the translation process easier, by adding statement-specific syntax for `if` expressions or using explicit "return" statements. However, these changes do not seem to offer significant benefits for presenting programs but would complicate the SMoL Language. Therefore, I do not plan to implement these changes.

8.2.5 Automatic Printing of Top-Level Expressions

Disabling automatic printing of top-level expressions in the SMoL Language could help preserve semantics for a broader range of programs, with minimal downside—other than requiring programs to print explicitly. This is a potential direction for future work.

CHAPTER 9

STUDIED POPULATIONS

The majority of this work was conducted with students in a "Principles of Programming Languages" course at a selective, private U.S. university (also referred to as **the PL course** or **US1**). The course uses *Programming Languages: Application and Interpretation* (PLAI) [70] as its textbook and enrolls approximately 70–75 students per year. It is not a required course; students take it by choice. Virtually all are computer science majors, most in their third or fourth year of undergraduate study, with around 10% being graduate students. All have taken at least one semester of imperative programming, and most have had significantly more. Most students have also completed close to a semester of functional programming.

One study was conducted in an accelerated introductory course at the same university as US1 (also referred to as the **Accelerated Intro course**). Most students in this course had some prior programming experience. To enroll, students were required to read parts of *How to Design Programs* (HtDP) [23] and successfully complete a series of Racket programming exercises.

Several studies were replicated with other populations:

PL Course at another U.S. University A primarily Hispanic public university in the U.S.

(also referred to as **US2**). Its student demographics differ significantly from those at US1. The Tutor was deployed in a third-year programming languages course in Spring 2023, taken by 12 students. Students were required to have completed two introductory programming courses focused on C++.

PL Course at a Belgian University A research university in Belgium (also referred to as **Belgium**). This course closely mirrors the PL course at US1: it also uses PLAI and reuses most of the same assignments. In addition, it covers JavaScript and asynchronous control flow, Rust and ownership/borrowing, TypeScript and gradual typing, and provides a brief introduction to OCaml (presented as an industry-strength counterpart to the PLAI teaching language). A total of 136 students took the course.

Project-Based PL Course at a Swedish University A research institution in Sweden (also referred to as **Sweden**). Students in this course had previously completed a Python course. The course focuses on creating simple programming languages in Ruby, with a preparatory course covering fundamental theory (e.g., lexing and parsing). During the project course, students attend four lectures on compiler structuring topics such as grammars, lexical analysis, and variable scoping. The study was mandatory, published at the beginning of the course, and students were encouraged to complete it by the midpoint of the semester, before beginning their implementations.

These additional populations differ in important ways from the original group and help assess whether the problems identified are specific to US1 or generalize across contexts.

CHAPTER 10

DISCUSSION

10.1 Extending SMoL with Types

Many modern programming languages include a type system. This section explores the possibility of defining a standard model similar to SMoL but incorporating a type system.

Modern type systems are generally expressive enough to assign types to many SMoL programs. In particular, they typically support:

- product-like types (e.g., tuples and structures),
- sum-like types (e.g., enums),
- types for recursive data structures (e.g., algebraic data types and recursive types), and
- parametric polymorphism (e.g., generics).

However, there is little consensus on how these types should be represented, making it difficult to draw connections between languages. Consider the following program, which constructs a self-referencing value:

```
(defvar x (mvec 1 0 2))
(vec-set! x 1 x)
(vec-len x)
```

There are numerous approaches to assign types to this program. For example:

1. Languages with a "top" type (e.g., `Object` and `Any`) allow programmers to declare the array elements as the "top" type.

2. Languages with union types (e.g., TypeScript and Flow) permit the array elements to be a union of numbers and arrays of the same kind.

3. Languages with inheritance (e.g., Java) enable programmers to define a base interface or class implemented by two specific subclasses—one wrapping an array and the other wrapping a number—allowing the array elements to be of the base type.

4. Languages with algebraic data types (ADTs) (e.g., OCaml) let programmers define an ADT with two variants—one for arrays and one for numbers—so the array elements belong to the ADT type.

While approaches 1 and 2 may seem similar, the others result in significantly different programs.

It remains unclear how to define a standard model that accommodates these variations in type systems. Even if such a model were possible, the substantial differences between languages make it challenging to teach type systems as part of a unified standard model.

10.2 Why Are Students More Likely to Make Mistakes in Certain Cases?

In Section 4.4, we observed that some questions are more difficult than others, even when they cover the same topics and are designed to detect the same misconception. It

is unsurprising that question difficulty varies within the same tutorial: some questions can reveal more misconceptions, offering students more opportunities to make mistakes. What is more interesting, however, is that the same misconception is more likely to be triggered by certain questions than by others.

One possibility is that unknown or latent misconceptions overlap with the targeted ones, providing more pathways to errors. **Knowledge in Pieces (KiP)** offers an alternative perspective that may explain this phenomenon (see [17] for an accessible introduction). A key concept in KiP is that of **p-prims**, which diSessa [17] defines as:

> P-prims are elements of intuitive knowledge that constitute people's "sense of mechanism"—their sense of which happenings are obvious, plausible, or implausible, and how one can explain or refute real or imagined possibilities.

diSessa [17] provides several examples of p-prims in physics:

1. Increased effort yields greater results;
2. The world is full of competing influences, where the stronger one "gets its way", even if accidental or natural "balance" sometimes exists;
3. The shape of a situation determines the shape of the action within it (e.g., orbits around square planets are recognizably square).

and in mathematics:

1. Multiplication makes numbers bigger;
2. A change in a given quantity generally implies a similar change in a related quantity;
3. Negative numbers cannot apply to the real world.

P-prims have several notable characteristics [17]:

- They typically—but not universally—work.

- They are context-sensitive: whether a p-prim is activated depends on how a task is framed and the person's "frame of mind" at the time.
- They cannot be consciously considered or rejected, as they are not tied to explicit reasoning and are often applied without awareness.

Unlike misconceptions, p-prims are "both smaller and more general …, conceived of as involved in and contributing to both naive and expert understanding" [34]. Sherin, Krakowski, and Lee [69] provides more examples of how p-prims may assemble to form (potentially incorrect) conceptions.

It is likely that some of the misconceptions I observed are rooted in underlying p-prims. Because p-prims are context-sensitive, a misconception that is more easily triggered by one program than another likely reflects the activation of a p-prim. It appears that nearly all the misconceptions studied fall into this category (Section 4.4.2). Identifying the underlying p-prims remains future work.

One potential p-prim I have identified is: "Definitions can be ignored until the variables are referenced." This appears to be a p-prim because it feels intuitive, often yields correct results, is context-sensitive (more likely to be applied when the expression is complex, less likely in step-by-step reasoning), is typically applied unconsciously (in my own experience), and is used even by experts (e.g., myself).

I suspect this p-prim underlies the Lazy, StructByRef, DeepClosure, and potentially all struct-copying misconceptions (see Section 5.1 for definitions). Consider the following example of StructByRef, repeated from Table 5.1:

```
(defvar x 3)
(defvar v (mvec 1 2 x))
(set! x 4)
v
```

A student might reach the misconception-related wrong answer `#(1 2 4)` through reasoning influenced by this p-prim:

1. `x` is initially bound to `3`;
2. `v` is defined using the expression `(mvec 1 2 x)`, but the student ignores the definition for now;
3. `x` is then updated to `4`;
4. When finally evaluating `v`, the student revisits the ignored expression `(mvec 1 2 x)` and uses the current value of `x`, yielding `#(1 2 4)`.

A similar pattern appears in the DeepClosure misconception:

```
(deffun (foo)
  (defvar n 0)
  (deffun (bar)
    (set! n (+ n 1))
    n)
  bar)
(defvar f (foo))
(defvar g (foo))

(f)
(g)
(f)
```

Here, a student might reach the incorrect answer `1 1 1` by similarly ignoring definitions until use:

1. `f` is defined as the result of evaluating `(foo)`, but the student ignores this evaluation for now;
2. `g` is defined similarly;
3. When evaluating `f`, the student evaluates `(foo)` and concludes that it returns a `bar` function with a fresh `n` initialized to `0`, so `(f)` returns `1`;
4. The same reasoning is applied again to `g`, and then again to `f`.

A promising direction for future work is to investigate whether people actually exhibit this tendency when reading programs (e.g., through an experiment similar to [16]), whether these misconceptions can be unified, and whether the corresponding refutation texts can be improved based on a deeper understanding of the underlying p-prims.

10.3 How Well Do Misinterpreters Model Misconceptions?

(Mis)interpreters are easier to write when the semantics are inductively defined over syntactic structure. As a result, modeling misconceptions as misinterpreters inherently biases the representation toward misconceptions that are consistent across contexts and independent of syntactic details. However, students' misconceptions do not always follow these patterns. For example, when reasoning about variable lookup, a student might hold a misconception such as, "I read backwards until I find the variable I'm looking for."[1] Such misconceptions are difficult to model with interpreters, because the student's reading strategy does not necessarily align with the syntactic structure that interpreters typically operate on.

Moreover, a misinterpreter applies its incorrect interpretation uniformly across all programs, whereas students appear more likely to exhibit a misconception in some programs than in others (Section 10.2).

Therefore, misinterpreters may capture some misconceptions poorly and may fail to capture others entirely. Nonetheless, as a model for representing misconceptions, they remain useful: misinterpreters provide a clear and accessible way to define misconceptions and support the development of high-quality assessments. As George Box said, *all models are wrong but some are useful* [6].

[1] Credit to Will Crichton.

10.4 Modifying SMoL Tutor to Teach a Different Semantics

The Tutor is designed to be compatible with languages that are not exactly SMoL but are semantically close. See Section 4.6.6 for a discussion of how the Tutor handles differences among the supported languages.

In principle, the SMoL Tutor can be adapted to teach a substantially different semantics. However, since it was not originally designed with this level of flexibility, such changes would likely require significant modifications to the source code.

For example, the R language exhibits copy-on-write behavior, which more closely resembles a combination of the CopyStructs misconceptions than standard SMoL. Adapting the Tutor for R would involve several key steps:

1. Replacing the current reference interpreter with one that reflects the new semantics.
2. Reclassifying the old reference interpreter as a misinterpreter.
3. Re-running all tutorial programs to ensure that each incorrect answer still maps to at most one misinterpreter.

In addition, the Tutor's explanations must be revised to align with the terminology and conceptual framework of the new semantics.

10.5 Misconceptions and Programming Language Design

Many authors have argued for designing programming languages with human factors in mind. Several notable groups of authors have advanced this perspective, including the Natural Programming Project [46, 48, 50], Hanenberg et al. (e.g., [25, 52]), and Stefik et al. (e.g., [61, 82]).

However, no matter how a programming language is designed, it must have a consistent semantics. Tunnell Wilson, Pombrio, and Krishnamurthi [81] found that people often

disagree about language behavior—and sometimes even with themselves. My own work on misconceptions reveals further examples of such inconsistencies (Chapters 4 and 5).

Even when there is some consistency, language design should not be driven solely by the popularity of particular conceptions. For example, dynamic scope (including the FlatEnv misconception) is widely regarded as problematic due to the difficulty of managing variable access, which can lead to security vulnerabilities and other issues. As a result, a language should avoid behaving like FlatEnv, no matter how widespread the misconception is.

As another example, a native implementation of CallCopyStruct frequently copies data structures, which is computationally expensive. Smarter strategies—such as the copy-on-write mechanism used in R—do exist, but they introduce a more complex runtime and result in less predictable execution times. Given these trade-offs, it may not be ideal for a language to always behave like CallCopyStruct. Nevertheless, offering both behaviors—with SMoL-style semantics as the default—could be a reasonable compromise.

10.6 Grading Tutor Assignments

Instructors can access student solutions through two mechanisms. First, the Tutor can log student data (Section 4.1.6). Second, the Tutor prompts students to save completed tutorials as PDFs (Figure 4.4), both to support review and to provide instructors with evidence of completion.

In either case, instructors must do additional work to view summary statistics—such as how students performed on each question or how individual students fared across tutorials. A planned improvement is to offer a dashboard tool to simplify the process of accessing such summaries.

However, I strongly recommend that instructors grade students based on *completion*, not *performance*. Grading by performance can discourage students from making mistakes and may even incentivize cheating (e.g., by using the Stacker to find correct answers). The

Tutor is explicitly designed to help students confront and correct misconceptions—mistakes are expected. In fact, if students make no mistakes, the Tutor is likely being used with the wrong population.

10.7 Textual vs. Graphical Explanation

The Tutor primarily explains program behavior using text (i.e., refutation texts), while the Stacker provides graphical explanations. Although the Tutor includes links to the Stacker, few students click them (in the US1 population, 12 students clicked zero times, 25 clicked 1–5 times, and 9 clicked more than 5 times). It remains future work to determine which format is more effective, and whether combining them creates a synergistic effect. Depending on the outcome, I may revise how the Tutor presents explanations to further enhance learning.

10.8 What Courses Does the SMoL Tutor Fit?

The Tutor's knowledge prerequisites (Section 4.6.4) align with concepts typically covered in CS1 or CS2. While some CS1 and CS2 students may not have encountered anonymous functions, they can still complete all earlier tutorials.

The Tutor does support a limited set of languages (Section 4.1.1). This is not a problem when a course's primarily language is supported. In addition, the limited set of supported languages is also generally not a problem in Programming Languages courses, where a central goal is to reason about semantics independently of syntax.

However, instructors should note that the SMoL Tutor is not a JavaScript Tutor, a Python Tutor, or a Scala Tutor. The correct answers in the Tutor occasionally do not match a specific language (see Section 4.6.6). More importantly, these languages likely include many features that are outside the scope of SMoL and behave surprisingly [31, 54, 55, 87]. Therefore, when the goal is teaching one of the language, I recommend that instructors

supplement the SMoL Tutor with teaching materials like WatChat by Chandra et al. [9] to address language-specific misconceptions.

Overall, although my studies have primarily focused on programming language courses, I believe the Tutor is also valuable for less advanced courses—even in CS1 or CS2 if the relevant language features have been covered.

10.9 Notable Limitations of SMoL Tutor's Content

The Tutor does not address several important aspects of programming language behavior that are relevant in real-world programming:

- *Syntactic misconceptions.* For example, misconceptions such as confusing `x = e` with `x == e` are explicitly excluded from the scope of the SMoL Tutor.
- *Implementation-specific behavior and undefined behavior.*
- *Language-specific behavior,* as discussed in Section 10.8.

CHAPTER 11

CONCLUSION

To repeat, here is my thesis statement:

> Misconceptions about SMoL behavior are widespread, but a tutoring system with carefully designed questions and feedback can effectively correct them.

I claim to have fulfilled this thesis by building a tutoring system—the SMoL Tutor (Chapter 4)—that embodies the goals of the thesis.

The Tutor is designed to correct the misconceptions identified in Chapter 5, presents programs in multiple languages (Chapters 3 and 8), relies on the Stacker (Chapter 6) and refutation text to deliver feedback, has been deployed across multiple populations (Chapter 9), and is effective at correcting most misconceptions without causing clear negative effects on the rest (Section 4.4).

APPENDIX A

INTERPRETERS

This appendix presents source code related to the interpreters. There is one definitional interpreter for the SMoL Language (Chapter 3), and one misinterpreter for each misconception from Chapter 5. Appendix A.1 defines the syntax of the Language. All interpreters depend on this module. Appendix A.2 presents the source code of the definitional interpreter for the Language. The remaining sections present the source code *differences* between the definitional interpreter and each misinterpreter:

CallByRef Appendix A.3

CallCopyStructs Appendix A.4

DefByRef Appendix A.5

DefCopyStructs Appendix A.6

StructByRef Appendix A.7

StructCopyStructs Appendix A.8

DeepClosure Appendix A.9

DefOrSet Appendix A.10

FlatEnv Appendix A.11

FunNotVal Appendix A.12

Lazy Appendix A.13

NoCircularity Appendix A.14

All the source code are written in the plait language [53].

A copy of the source code is also available at the dissertation's webpage:

github.com/LuKuangChen/dissertation

A.1 Syntax of The SMoL Language

```
#lang plait


(define-type Constant
  (logical [l : Boolean])
  (numeric [n : Number]))
(define-type Statement
  (expressive [e : Expression])
  (definitive [d : Definition]))
(define-type Expression
  (Con [c : Constant])
  (Var [x : Symbol])
  (Set! [x : Symbol] [e : Expression])
  ;; `and` and `or` are translated to `if`
```

```
  (If [e_cnd : Expression] [e_thn : Expression] [e_els :
     Expression])
  (Cond [ebs : (Listof (Expression * Body))] [ob : (Optionof Body
     )])
  (Begin [es : (Listof Expression)] [e : Expression])
  (Lambda [xs : (Listof Symbol)] [body : Body])
  (Let [xes : (Listof (Symbol * Expression))] [body : Body])
  ;; `let*` and `letrec` are translated to`let`
  ;; primitive operations are supported by defining the operators
      as variables
  (App [e : Expression] [es : (Listof Expression)]))

(define (And es)
  (cond
    [(empty? es) (Con (logical #t))]
    [else (If (first es)
              (And (rest es))
              (Con (logical #f)))]))

(define (Or es)
  (cond
    [(empty? es) (Con (logical #f))]
    [else (If (first es)
              (Con (logical #t))
              (Or (rest es)))]))

(define (Let* xes b)
```

```
  (cond
    [(empty? xes) (Let (list) b)]
    [else (Let (list (first xes))
               (pair (list) (Let* (rest xes) b)))]))

(define (Letrec xes b)
  (Let (list)
       (pair (append (map xe->definitive xes) (fst b))
             (snd b))))
(define (xe->definitive xe)
  (definitive (Defvar (fst xe) (snd xe))))

(define-type Definition
  (Defvar [x : Symbol] [e : Expression])
  (Deffun [f : Symbol] [xs : (Listof Symbol)] [b : Body]))
(define-type-alias Body ((Listof Statement) * Expression))
(define-type-alias Program (Listof Statement))
(define-type PrimitiveOperator
  ;; + - * /
  (Add)
  (Sub)
  (Mul)
  (Div)

  ;; < <= > >=
  (Lt)
  (Le)
```

```
  (Gt)
  (Ge)

  (VecNew) ;; mvec
  (VecLen)
  (VecRef)
  (VecSet)

  (PairNew) ;; mpair
  (PairLft)
  (PairRht)
  (PairSetLft)
  (PairSetRht)

  (Eq) ;; =
  )
(define (make-the-primordial-env load)
  (list
   (hash
    (list
      (pair '+ (load (Add)))
      (pair '- (load (Sub)))
      (pair '* (load (Mul)))
      (pair '/ (load (Div)))
      (pair '< (load (Lt)))
      (pair '> (load (Gt)))
      (pair '<= (load (Le)))
```

```
      (pair '>= (load (Ge)))
      (pair 'mvec (load (VecNew)))
      (pair 'vec-len (load (VecLen)))
      (pair 'vec-ref (load (VecRef)))
      (pair 'vec-set! (load (VecSet)))
      (pair 'mpair (load (PairNew)))
      (pair 'left (load (PairLft)))
      (pair 'right (load (PairRht)))
      (pair 'set-left! (load (PairSetLft)))
      (pair 'set-right! (load (PairSetRht)))
      (pair '= (load (Eq)))
  ))))
```

A.2 The Definitional Interpreter

```
#lang plait

(require smol-interpreters/syntax)
(require
  (typed-in racket
    [append-map : (('a -> (Listof 'b)) (Listof 'a) -> (Listof 'b)
       )]
    [hash-values : ((Hashof 'a 'b) -> (Listof 'b))]
    [count : (('a -> Boolean) (Listof 'a) -> Number)]
    [for-each : (('a -> 'b) (Listof 'a) -> Void)]
    [string-join : ((Listof String) String -> String)]
    [displayln : ('a -> Void)]
```

```
    [list->vector : ((Listof 'a) -> (Vectorof 'a))]
    [vector->list : ((Vectorof 'a) -> (Listof 'a))]
    [number->string : (Number -> String)]
    [check-duplicates : ((Listof 'a) -> Boolean)]))

(define-syntax for
  (syntax-rules ()
    [(for ([x xs]) body ...)
     (for-each (lambda (x) (begin body ...)) xs)]))

(define-type Tag
  (TNum)
  (TStr)
  (TLgc)
  (TFun)
  (TVec))

(define-type Value
  (unit)
  (embedded [c : Constant])
  (primitive [o : PrimitiveOperator])
  (function [xs : (Listof Symbol)] [body : Body] [env :
     Environment])
  (vector [vs : (Vectorof Value)]))

(define (value-eq? v1 v2)
  (cond
```

```
    [(and (unit? v1) (unit? v2)) #t]
    [(and (embedded? v1) (embedded? v2)) (equal? v1 v2)]
    [(and (primitive? v1) (primitive? v2)) (equal? v1 v2)]
    [else (eq? v1 v2)]))

(define (as-logical v)
  (type-case Value v
    [(embedded c)
     (type-case Constant c
       [(logical b) b]
       [else (error 'smol "expecting a boolean")])]
    [else
     (error 'smol "expecting a boolean")]))
(define (from-logical [v : Boolean])
  (embedded (logical v)))

(define (as-numeric v)
  (type-case Value v
    [(embedded c)
     (type-case Constant c
       [(numeric n) n]
       [else (error 'smol "expecting a number")])]
    [else
     (error 'smol "expecting a number")]))
(define (from-numeric [n : Number])
  (embedded (numeric n)))
```

```
(define (as-vector v)
  (type-case Value v
    [(vector v)
     v]
    [else
     (error 'smol "expecting a vector")]))
(define (as-pair v)
  (let ([v (as-vector v)])
    (if (= (vector-length v) 2)
        v
        (error 'smol "expecting a pair"))))

(define (as-one vs)
  (cond
    [(= (length vs) 1)
     (first vs)]
    [else
     (error 'smol "arity-mismatch, expecting one")]))
(define (as-two vs)
  (cond
    [(= (length vs) 2)
     (values (first vs) (first (rest vs)))]
    [else
     (error 'smol "arity-mismatch, expecting two")]))
(define (as-three vs)
  (cond
    [(= (length vs) 3)
```

```
     (values (first vs) (first (rest vs)) (first (rest (rest vs))
        ))]
    [else
     (error 'smol "arity-mismatch, expecting three")]))

(define-type-alias EnvironmentFrame (Hashof Symbol (Boxof (
   Optionof Value))))
(define-type-alias Environment (Listof EnvironmentFrame))

(define (load f)
  (box (some (primitive f))))
(define the-primordial-env
  (make-the-primordial-env load))

(define (cmp f vs) (from-logical (cmp-helper f vs)))
(define (cmp-helper f vs) : Boolean
  (cond
    [(empty? vs) #t]
    [else
     (local ((define (rec v vs)
               (cond
                 [(empty? vs) #t]
                 [else
                  (and (f v (first vs))
                       (rec (first vs) (rest vs)))])))
       (rec (first vs) (rest vs)))]))
```

```
(define (delta p vs)
  (type-case PrimitiveOperator p
    [(Add) (from-numeric (foldl (lambda (m n) (+ m n)) 0 (map
      as-numeric vs)))]
    [(Sub)
     (let ([vs (map as-numeric vs)])
       (from-numeric
        (foldl (lambda (m n) (- n m))
               (first vs)
               (rest vs))))]
    [(Mul) (from-numeric (foldl (lambda (m n) (* m n)) 1 (map
      as-numeric vs)))]
    [(Div)
     (let ([vs (map as-numeric vs)])
       (from-numeric
        (foldl (lambda (m n) (/ n m))
               (first vs)
               (rest vs))))]
    [(Eq) (cmp value-eq? vs)]
    [(Lt) (cmp (lambda (a b) (< (as-numeric a) (as-numeric b)))
      vs)]
    [(Gt) (cmp (lambda (a b) (> (as-numeric a) (as-numeric b)))
      vs)]
    [(Le) (cmp (lambda (a b) (<= (as-numeric a) (as-numeric b)))
      vs)]
    [(Ge) (cmp (lambda (a b) (>= (as-numeric a) (as-numeric b)))
      vs)]
```

```
[(VecNew) (vector (list->vector vs))]
[(VecLen)
 (local ((define v (as-one vs)))
   (from-numeric (vector-length (as-vector v))))]
[(VecRef)
 (local ((define-values (v1 v2) (as-two vs))
         (define vvec (as-vector v1))
         (define vnum (as-numeric v2)))
   (vector-ref vvec vnum))]
[(VecSet)
 (local ((define-values (v1 v2 v3) (as-three vs))
         (define vvec (as-vector v1))
         (define vnum (as-numeric v2)))
   (begin
     (vector-set! vvec vnum v3)
     (unit)))]
[(PairNew)
 (local ((define-values (v1 v2) (as-two vs)))
   (vector (list->vector vs)))]
[(PairLft)
 (local ((define v (as-one vs)))
   (vector-ref (as-pair v) 0))]
[(PairRht)
 (local ((define v (as-one vs)))
   (vector-ref (as-pair v) 1))]
[(PairSetLft)
 (local ((define-values (vpr vel) (as-two vs)))
```

```
        (begin
          (vector-set! (as-pair vpr) 0 vel)
          (unit)))]
     [(PairSetRht)
      (local ((define-values (vpr vel) (as-two vs)))
        (begin
          (vector-set! (as-pair vpr) 1 vel)
          (unit)))]))

(define (make-env env xs)
  (begin
    (when (check-duplicates xs)
      (error 'smol "can't define a variable twice in one block"))
    (local [(define (allocate x)
              (pair x (box (none))))]
      (cons (hash (map allocate xs)) env))))
(define (env-frame-lookup f x)
  (hash-ref f x))
(define (env-lookup-location env x)
  (type-case Environment env
    [empty
     (error 'smol "variable undeclared")]
    [(cons f fs)
     (type-case (Optionof '_) (env-frame-lookup f x)
       [(none) (env-lookup-location fs x)]
       [(some loc) loc])]))
(define (env-lookup env x)
```

```
  (let ([v (env-lookup-location env x)])
    (type-case (Optionof '_) (unbox v)
      [(none) (error 'smol "refer␣to␣a␣variable␣before␣assign␣it␣
        a␣value")]
      [(some v) v])))
(define (env-update! env)
  (lambda (x v)
    (let ([loc (env-lookup-location env x)])
      (set-box! loc (some v)))))

(define (declared-Symbols [ts : (Listof Statement)])
  (local [(define (xs-of-t [t : Statement])
            : (Listof Symbol)
            (type-case Statement t
              [(expressive e) (list)]
              [(definitive d)
               (type-case Definition d
                 [(Defvar x e) (list x)]
                 [(Deffun f xs b) (list f)])]))]
    (append-map xs-of-t ts)))

(define (eval-def env d)
  (type-case Definition d
    [(Defvar x e)
     (let ([v ((eval-exp env) e)])
       ((env-update! env) x v))]
    [(Deffun f xs b)
```

```
     (let ([v (function xs b env)])
       ((env-update! env) f v))]))

(define (eval-body env xvs b)
  (local [(define vs (map snd xvs))
          (define xs
            (append (map fst xvs)
                    (declared-Symbols (fst b))))]
    (let ([env (make-env env xs)])
      (begin
        ; bind arguments
        (for ([xv xvs])
          ((env-update! env) (fst xv) (snd xv)))
        ; evaluate starting terms
        (for ([t (fst b)])
          ((eval-statement env) t))
        ; evaluate and return the result
        ((eval-exp env) (snd b))))))

(define (eval-exp env)
  (lambda (e)
    (type-case Expression e
      [(Con c) (embedded c)]
      [(Var x) (env-lookup env x)]
      [(Lambda xs b) (function xs b env)]
      [(Let xes b)
       (local [(define (ev-bind xv)
```

```
            (let ([v ((eval-exp env) (snd xv))])
              (pair (fst xv) v)))]
   (let ([xvs (map ev-bind xes)])
     (eval-body env xvs b)))]
[(Begin es e)
 (begin
   (map (eval-exp env) es)
   ((eval-exp env) e))]
[(Set! x e)
 (let ([v ((eval-exp env) e)])
   (begin
     ((env-update! env) x v)
     (unit)))]
[(If e_cnd e_thn e_els)
 (let ([v ((eval-exp env) e_cnd)])
   (let ([l (as-logical v)])
     ((eval-exp env)
      (if l e_thn e_els))))]
[(Cond ebs ob)
 (local [(define (loop ebs)
           (type-case (Listof (Expression * Body)) ebs
             [empty
              (type-case (Optionof Body) ob
                [(none) (error 'smol "fall␣throught␣cond")]
                [(some b)
                 (eval-body env (list) b)])]
             [(cons eb ebs)
```

```
                    (let ([v ((eval-exp env) (fst eb))])
                      (let ([l (as-logical v)])
                        (if l
                            (eval-body env (list) (snd eb))
                            (loop ebs))))]))]
          (loop ebs))]
        [(App e es)
         (let ([v ((eval-exp env) e)])
           (let ([vs (map (eval-exp env) es)])
             (type-case Value v
               [(function xs b env)
                (if (= (length xs) (length vs))
                    (eval-body env (map2 pair xs vs) b)
                    (error 'smol "(arity-mismatch␣(length␣xs)␣(
                       length␣vs))"))]
               [(primitive p)
                (delta p vs)]
               [else
                (error 'smol "(type-mismatch␣(TFun)␣v)")])))]))))

(define (eval-statement env)
  (lambda (t)
    (type-case Statement t
      [(definitive d)
       (begin
         (eval-def env d)
         (unit))]
```

```
      [(expressive e)
       ((eval-exp env) e)])))

(define (constant->string [c : Constant])
  (type-case Constant c
    [(logical l) (if l "#t" "f")]
    [(numeric n) (number->string n)]))

(define (self-ref [i : Number]) : String
  (foldr string-append
         ""
         (list "#" (number->string i) "#")))
(define (self-def [i : Number]) : String
  (foldr string-append
         ""
         (list "#" (number->string i) "=")))

(define (value->string visited-vs)
  (lambda (v)
    (type-case Value v
      [(unit) "#<void>"]
      [(embedded c) (constant->string c)]
      [(primitive o) "#<procedure>"]
      [(function xs body env) "#<procedure>"]
      [(vector vs)
       (type-case (Optionof (Boxof (Optionof Number))) (hash-ref
          visited-vs vs)
```

```
[(none)
 (let ([visited-vs (hash-set visited-vs vs (box (none)))
    ])
   (let* ([s (foldr string-append
                    ""
                    (list
                     "#("
                     (string-join
                      (map (value->string visited-vs) (
                         vector->list vs))
                      "␣")
                     ")"))]
          [boi (some-v (hash-ref visited-vs vs))])
     (type-case (Optionof Number) (unbox boi)
       [(none) s]
       [(some i) (string-append (self-def i) s)]))))]
[(some boi)
 (type-case (Optionof Number) (unbox boi)
   [(none)
    (let ([i (count some? (map unbox (hash-values
       visited-vs)))])
      (begin
        (set-box! (some-v (hash-ref visited-vs vs)) (
           some i))
        (self-ref i)))]
   [(some i)
    (self-ref i)])
```

```
          ])])))
(define (print-value v)
  (type-case Value v
    [(unit) (void)]
    [else (displayln ((value->string (hash (list))) v))]))

(define (exercute-terms-top-level env)
  (lambda (ts) : Void
    (type-case (Listof Statement) ts
      [empty (void)]
      [(cons t ts)
       (type-case Statement t
         [(definitive d)
          (begin
            (eval-def env d)
            ((exercute-terms-top-level env) ts))]
         [(expressive e)
          (begin
            (print-value ((eval-exp env) e))
            ((exercute-terms-top-level env) ts))])])))

(define (evaluate [p : Program])
  (local [(define xs (declared-Symbols p))
          (define the-top-level-env (make-env the-primordial-env
            xs))]
    ((exercute-terms-top-level the-top-level-env) p)))
```

A.3 The CallByRef Misinterpreter

```
--- systems/smol-interpreters/Definitional.rkt  2025-02-08
   11:00:46
+++ systems/smol-interpreters/misinterpreters/CallByRef.rkt
   2025-02-11 10:38:14
@@ -93,11 +93,11 @@
      [else
       (error 'smol "arity-mismatch, expecting three")]))

-(define-type-alias EnvironmentFrame (Hashof Symbol (Boxof (
   Optionof Value))))
+(define-type-alias EnvironmentFrame (Hashof Symbol (Boxof (
   Optionof (Boxof Value)))))
 (define-type-alias Environment (Listof EnvironmentFrame))

 (define (load f)
-  (box (some (primitive f))))
+  (box (some (box (primitive f)))))
 (define the-primordial-env
   (make-the-primordial-env load))

@@ -213,10 +213,10 @@
   (type-case Definition d
     [(Defvar x e)
      (let ([v ((eval-exp env) e)])
-       ((env-update! env) x v))]
```

```
+       ((env-update! env) x (box v)))]
      [(Deffun f xs b)
       (let ([v (function xs b env)])
-       ((env-update! env) f v))]))
+       ((env-update! env) f (box v)))]))

 (define (eval-body env xvs b)
   (local [(define vs (map snd xvs))
@@ -233,16 +233,22 @@
           ((eval-statement env) t))
         ; evaluate and return the result
         ((eval-exp env) (snd b))))))
+
+(define (eval-ref env)
+  (lambda (e)
+    (type-case Expression e
+      [(Var x) (env-lookup env x)]
+      [else (box ((eval-exp env) e))])))

 (define (eval-exp env)
   (lambda (e)
     (type-case Expression e
       [(Con c) (embedded c)]
-      [(Var x) (env-lookup env x)]
+      [(Var x) (unbox (env-lookup env x))]
       [(Lambda xs b) (function xs b env)]
       [(Let xes b)
```

```
         (local [(define (ev-bind xv)
-                  (let ([v ((eval-exp env) (snd xv))])
+                  (let ([v ((eval-ref env) (snd xv))])
                     (pair (fst xv) v)))]
           (let ([xvs (map ev-bind xes)])
             (eval-body env xvs b)))]
@@ -253,7 +259,7 @@
        [(Set! x e)
         (let ([v ((eval-exp env) e)])
           (begin
-            ((env-update! env) x v)
+            (set-box! (env-lookup env x) v)
             (unit)))]
        [(If e_cnd e_thn e_els)
         (let ([v ((eval-exp env) e_cnd)])
@@ -277,14 +283,14 @@
           (loop ebs))]
       [(App e es)
        (let ([v ((eval-exp env) e)])
-         (let ([vs (map (eval-exp env) es)])
+         (let ([vs (map (eval-ref env) es)])
            (type-case Value v
              [(function xs b env)
               (if (= (length xs) (length vs))
                   (eval-body env (map2 pair xs vs) b)
                   (error 'smol "(arity-mismatch (length xs) (
                     length vs))"))]
```

```
               [(primitive p)
-               (delta p vs)]
+               (delta p (map unbox vs))]
               [else
                (error 'smol "(type-mismatch (TFun) v)")])))])))
```

A.4 The CallCopyStructs Misinterpreter

```
--- systems/smol-interpreters/Definitional.rkt  2025-02-08
   11:00:46
+++ systems/smol-interpreters/misinterpreters/CallCopyStructs.rkt
          2025-02-11 11:44:20
@@ -33,6 +33,11 @@
   (function [xs : (Listof Symbol)] [body : Body] [env :
      Environment])
   (vector [vs : (Vectorof Value)]))

+(define (copy-value v)
+  (type-case Value v
+    [(vector v) (vector (list->vector (vector->list v)))]
+    [else v]))
+
 (define (value-eq? v1 v2)
   (cond
     [(and (unit? v1) (unit? v2)) #t]
@@ -281,7 +286,7 @@
            (type-case Value v
```

```
               [(function xs b env)
                (if (= (length xs) (length vs))
-                   (eval-body env (map2 pair xs vs) b)
+                   (eval-body env (map2 pair xs (map copy-value
   vs)) b)
                    (error 'smol "(arity-mismatch (length xs) (
                       length vs))"))]
               [(primitive p)
                (delta p vs)]
```

A.5 The DefByRef Misinterpreter

```
--- systems/smol-interpreters/Definitional.rkt  2025-02-08
   11:00:46
+++ systems/smol-interpreters/misinterpreters/DefByRef.rkt
   2025-02-11 11:52:39
@@ -93,11 +93,11 @@
     [else
      (error 'smol "arity-mismatch, expecting three")]))

-(define-type-alias EnvironmentFrame (Hashof Symbol (Boxof (
   Optionof Value))))
+(define-type-alias EnvironmentFrame (Hashof Symbol (Boxof (
   Optionof (Boxof Value)))))
 (define-type-alias Environment (Listof EnvironmentFrame))

 (define (load f)
```

```
-   (box (some (primitive f))))
+   (box (some (box (primitive f)))))
 (define the-primordial-env
    (make-the-primordial-env load))

@@ -212,10 +212,10 @@
 (define (eval-def env d)
    (type-case Definition d
      [(Defvar x e)
-      (let ([v ((eval-exp env) e)])
+      (let ([v ((eval-ref env) e)])
        ((env-update! env) x v))]
      [(Deffun f xs b)
-      (let ([v (function xs b env)])
+      (let ([v (box (function xs b env))])
        ((env-update! env) f v))]))

 (define (eval-body env xvs b)
@@ -227,18 +227,24 @@
        (begin
          ; bind arguments
          (for ([xv xvs])
-           ((env-update! env) (fst xv) (snd xv)))
+           ((env-update! env) (fst xv) (box (snd xv))))
          ; evaluate starting terms
          (for ([t (fst b)])
            ((eval-statement env) t))
```

```
          ; evaluate and return the result
          ((eval-exp env) (snd b))))))

+(define (eval-ref env)
+  (lambda (e)
+    (type-case Expression e
+      [(Var x) (env-lookup env x)]
+      [else (box ((eval-exp env) e))])))
+
 (define (eval-exp env)
   (lambda (e)
     (type-case Expression e
       [(Con c) (embedded c)]
-      [(Var x) (env-lookup env x)]
+      [(Var x) (unbox (env-lookup env x))]
       [(Lambda xs b) (function xs b env)]
       [(Let xes b)
        (local [(define (ev-bind xv)
@@ -253,7 +259,7 @@
       [(Set! x e)
        (let ([v ((eval-exp env) e)])
          (begin
-           ((env-update! env) x v)
+           (set-box! (env-lookup env x) v)
            (unit)))]
       [(If e_cnd e_thn e_els)
        (let ([v ((eval-exp env) e_cnd)])
```

A.6 The DefCopyStructs Misinterpreter

```
--- systems/smol-interpreters/Definitional.rkt  2025-02-08
   11:00:46
+++ systems/smol-interpreters/misinterpreters/DefCopyStructs.rkt
           2025-02-11 11:49:52
@@ -33,6 +33,11 @@
   (function [xs : (Listof Symbol)] [body : Body] [env :
      Environment])
   (vector [vs : (Vectorof Value)]))

+(define (copy-value v)
+  (type-case Value v
+    [(vector v) (vector (list->vector (vector->list v)))]
+    [else v]))
+
 (define (value-eq? v1 v2)
   (cond
     [(and (unit? v1) (unit? v2)) #t]
@@ -213,7 +218,7 @@
   (type-case Definition d
     [(Defvar x e)
      (let ([v ((eval-exp env) e)])
-       ((env-update! env) x v))]
+       ((env-update! env) x (copy-value v)))]
     [(Deffun f xs b)
      (let ([v (function xs b env)])
```

```
         ((env-update! env) f v))]))
@@ -243,7 +248,7 @@
        [(Let xes b)
         (local [(define (ev-bind xv)
                   (let ([v ((eval-exp env) (snd xv))])
-                    (pair (fst xv) v)))]
+                    (pair (fst xv) (copy-value v))))]
           (let ([xvs (map ev-bind xes)])
             (eval-body env xvs b)))]
        [(Begin es e)
```

A.7 The StructByRef Misinterpreter

```
--- systems/smol-interpreters/Definitional.rkt  2025-02-08
   11:00:46
+++ systems/smol-interpreters/misinterpreters/StructByRef.rkt
   2025-02-11 11:59:22
@@ -31,9 +31,11 @@
   (embedded [c : Constant])
   (primitive [o : PrimitiveOperator])
   (function [xs : (Listof Symbol)] [body : Body] [env :
      Environment])
-  (vector [vs : (Vectorof Value)]))
+  (vector [vs : (Vectorof (Boxof Value))]))

 (define (value-eq? v1 v2)
+  (value-eq-helper? (unbox v1) (unbox v2)))
```

```
+(define (value-eq-helper? v1 v2)
   (cond
     [(and (unit? v1) (unit? v2)) #t]
     [(and (embedded? v1) (embedded? v2)) (equal? v1 v2)]
@@ -52,7 +54,7 @@
   (embedded (logical v)))


 (define (as-numeric v)
-  (type-case Value v
+  (type-case Value (unbox v)
     [(embedded c)
      (type-case Constant c
       [(numeric n) n]
@@ -63,7 +65,7 @@
   (embedded (numeric n)))


 (define (as-vector v)
-  (type-case Value v
+  (type-case Value (unbox v)
     [(vector v)
      v]
     [else
@@ -93,11 +95,11 @@
     [else
      (error 'smol "arity-mismatch, expecting three")]))


-(define-type-alias EnvironmentFrame (Hashof Symbol (Boxof (
```

```
   Optionof Value))))
+(define-type-alias EnvironmentFrame (Hashof Symbol (Boxof (
   Optionof (Boxof Value)))))
 (define-type-alias Environment (Listof EnvironmentFrame))

 (define (load f)
-  (box (some (primitive f))))
+  (box (some (box (primitive f)))))
 (define the-primordial-env
   (make-the-primordial-env load))

@@ -143,7 +145,7 @@
      (local ((define-values (v1 v2) (as-two vs))
              (define vvec (as-vector v1))
              (define vnum (as-numeric v2)))
-       (vector-ref vvec vnum))]
+       (unbox (vector-ref vvec vnum)))]
     [(VecSet)
      (local ((define-values (v1 v2 v3) (as-three vs))
              (define vvec (as-vector v1))
@@ -156,10 +158,10 @@
        (vector (list->vector vs)))]
     [(PairLft)
      (local ((define v (as-one vs)))
-       (vector-ref (as-pair v) 0))]
+       (unbox (vector-ref (as-pair v) 0)))]
     [(PairRht)
```

```
       (local ((define v (as-one vs)))
-        (vector-ref (as-pair v) 1))]
+        (unbox (vector-ref (as-pair v) 1)))]
      [(PairSetLft)
       (local ((define-values (vpr vel) (as-two vs)))
         (begin
@@ -212,11 +214,10 @@
 (define (eval-def env d)
   (type-case Definition d
     [(Defvar x e)
-     (let ([v ((eval-exp env) e)])
+     (let ([v (box ((eval-exp env) e))])
        ((env-update! env) x v))]
     [(Deffun f xs b)
-     (let ([v (function xs b env)])
-       ((env-update! env) f v))]))
+     ((env-update! env) f (box (function xs b env)))]))

 (define (eval-body env xvs b)
   (local [(define vs (map snd xvs))
@@ -227,18 +228,24 @@
       (begin
         ; bind arguments
         (for ([xv xvs])
-          ((env-update! env) (fst xv) (snd xv)))
+          ((env-update! env) (fst xv) (box (snd xv))))
         ; evaluate starting terms
```

```
          (for ([t (fst b)])
            ((eval-statement env) t))
          ; evaluate and return the result
          ((eval-exp env) (snd b))))))

+(define (eval-ref env)
+  (lambda (e)
+    (type-case Expression e
+      [(Var x) (env-lookup env x)]
+      [else (box ((eval-exp env) e))])))
+
 (define (eval-exp env)
   (lambda (e)
     (type-case Expression e
       [(Con c) (embedded c)]
-      [(Var x) (env-lookup env x)]
+      [(Var x) (unbox (env-lookup env x))]
       [(Lambda xs b) (function xs b env)]
       [(Let xes b)
        (local [(define (ev-bind xv)
@@ -253,7 +260,7 @@
       [(Set! x e)
        (let ([v ((eval-exp env) e)])
          (begin
-           ((env-update! env) x v)
+           (set-box! (env-lookup env x) v)
            (unit)))]
```

```
        [(If e_cnd e_thn e_els)
         (let ([v ((eval-exp env) e_cnd)])
@@ -277,11 +284,11 @@
           (loop ebs))]
        [(App e es)
         (let ([v ((eval-exp env) e)])
-          (let ([vs (map (eval-exp env) es)])
+          (let ([vs (map (eval-ref env) es)])
            (type-case Value v
              [(function xs b env)
               (if (= (length xs) (length vs))
-                  (eval-body env (map2 pair xs vs) b)
+                  (eval-body env (map2 pair xs (map unbox vs)) b
   )
                   (error 'smol "(arity-mismatch (length xs) (
                     length vs))"))]
              [(primitive p)
               (delta p vs)]
@@ -328,7 +335,7 @@
                                (list
                                 "#("
                                 (string-join
-                                 (map (value->string visited-vs) (
  vector->list vs))
+                                 (map (value->string visited-vs) (
  map unbox (vector->list vs)))
                                 " ")
```

```
                                        ")"))]
                        [boi (some-v (hash-ref visited-vs vs))]))
```

A.8 The StructCopyStructs Misinterpreter

```
--- systems/smol-interpreters/Definitional.rkt  2025-02-08
   11:00:46
+++ systems/smol-interpreters/misinterpreters/StructCopyStructs.
   rkt     2025-02-08 11:00:46
@@ -3,6 +3,7 @@
 (require smol-interpreters/syntax)
 (require
   (typed-in racket
+    [vector-map : (('a -> 'b) (Vectorof 'a) -> (Vectorof 'b))]
     [append-map : (('a -> (Listof 'b)) (Listof 'a) -> (Listof 'b
        ))]
     [hash-values : ((Hashof 'a 'b) -> (Listof 'b))]
     [count : (('a -> Boolean) (Listof 'a) -> Number)]
@@ -33,6 +34,11 @@
   (function [xs : (Listof Symbol)] [body : Body] [env :
      Environment])
   (vector [vs : (Vectorof Value)]))

+(define (copy-value v)
+  (type-case Value v
+    [(vector v) (vector (vector-map copy-value v))]
+    [else v]))
```

```
+
 (define (value-eq? v1 v2)
   (cond
     [(and (unit? v1) (unit? v2)) #t]
@@ -135,7 +141,7 @@
     [(Gt) (cmp (lambda (a b) (> (as-numeric a) (as-numeric b)))
        vs)]
     [(Le) (cmp (lambda (a b) (<= (as-numeric a) (as-numeric b)))
        vs)]
     [(Ge) (cmp (lambda (a b) (>= (as-numeric a) (as-numeric b)))
        vs)]
-    [(VecNew) (vector (list->vector vs))]
+    [(VecNew) (vector (list->vector (map copy-value vs)))]
     [(VecLen)
      (local ((define v (as-one vs)))
        (from-numeric (vector-length (as-vector v))))]
@@ -149,7 +155,7 @@
                (define vvec (as-vector v1))
                (define vnum (as-numeric v2)))
        (begin
-         (vector-set! vvec vnum v3)
+         (vector-set! vvec vnum (copy-value v3))
          (unit)))]
     [(PairNew)
      (local ((define-values (v1 v2) (as-two vs)))
@@ -163,12 +169,12 @@
     [(PairSetLft)
```

```
       (local ((define-values (vpr vel) (as-two vs)))
         (begin
-          (vector-set! (as-pair vpr) 0 vel)
+          (vector-set! (as-pair vpr) 0 (copy-value vel))
           (unit)))]
      [(PairSetRht)
       (local ((define-values (vpr vel) (as-two vs)))
         (begin
-          (vector-set! (as-pair vpr) 1 vel)
+          (vector-set! (as-pair vpr) 1 (copy-value vel))
           (unit)))]))

 (define (make-env env xs)
```

A.9 The DeepClosure Misinterpreter

```
--- systems/smol-interpreters/Definitional.rkt  2025-02-08
   11:00:46
+++ systems/smol-interpreters/misinterpreters/DeepClosure.rkt
   2025-02-11 10:34:48
@@ -3,6 +3,7 @@
 (require smol-interpreters/syntax)
 (require
   (typed-in racket
+    [hash->list : ((Hashof 'a 'b) -> (Listof ('a * 'b)))]
     [append-map : (('a -> (Listof 'b)) (Listof 'a) -> (Listof 'b
       ))]
```

```
     [hash-values : ((Hashof 'a 'b) -> (Listof 'b))]
     [count : (('a -> Boolean) (Listof 'a) -> Number)]
@@ -208,14 +209,31 @@
                  [(Defvar x e) (list x)]
                  [(Deffun f xs b) (list f)])]))]
     (append-map xs-of-t ts)))
+
+;; Copy the location only if it has been initialized
+(define (maybe-copy-box bo)
+  (if (none? (unbox bo))
+      bo
+      (box (unbox bo))))
+(define (copy-xbov f)
+  (lambda (x)
+    (pair x
+          (maybe-copy-box (some-v (hash-ref f x))))))
+(define (copy-env-frame frm)
+  (hash (map (copy-xbov frm) (hash-keys frm))))
+(define (copy-env env)
+  (map copy-env-frame env))

+(define (build-function xs b env)
+  (function xs b (copy-env env)))
+
 (define (eval-def env d)
   (type-case Definition d
     [(Defvar x e)
```

```
       (let ([v ((eval-exp env) e)])
         ((env-update! env) x v))]
      [(Deffun f xs b)
-      (let ([v (function xs b env)])
+      (let ([v (build-function xs b env)])
         ((env-update! env) f v))]))

 (define (eval-body env xvs b)
@@ -239,7 +257,7 @@
      (type-case Expression e
        [(Con c) (embedded c)]
        [(Var x) (env-lookup env x)]
-       [(Lambda xs b) (function xs b env)]
+       [(Lambda xs b) (build-function xs b env)]
        [(Let xes b)
         (local [(define (ev-bind xv)
                   (let ([v ((eval-exp env) (snd xv))])
@@ -281,7 +299,7 @@
             (type-case Value v
               [(function xs b env)
                (if (= (length xs) (length vs))
-                   (eval-body env (map2 pair xs vs) b)
+                   (eval-body (copy-env env) (map2 pair xs vs) b)
                    (error 'smol "(arity-mismatch (length xs) (
                       length vs))"))]
               [(primitive p)
                (delta p vs)]
```

A.10 The DefOrSet Misinterpreter

```
--- systems/smol-interpreters/Definitional.rkt  2025-02-08
   11:00:46
+++ systems/smol-interpreters/misinterpreters/DefOrSet.rkt
   2025-02-11 11:56:29
@@ -3,6 +3,7 @@
 (require smol-interpreters/syntax)
 (require
   (typed-in racket
+    [hash-has-key? : ((Hashof 'a 'b) 'a -> Boolean)]
     [append-map : (('a -> (Listof 'b)) (Listof 'a) -> (Listof 'b
        ))]
     [hash-values : ((Hashof 'a 'b) -> (Listof 'b))]
     [count : (('a -> Boolean) (Listof 'a) -> Number)]
@@ -93,13 +94,13 @@
     [else
      (error 'smol "arity-mismatch, expecting three")]))

-(define-type-alias EnvironmentFrame (Hashof Symbol (Boxof (
   Optionof Value))))
+(define-type-alias EnvironmentFrame (Boxof (Hashof Symbol (
   Optionof Value))))
 (define-type-alias Environment (Listof EnvironmentFrame))

 (define (load f)
-  (box (some (primitive f))))
```

```
+  (some (primitive f)))
 (define the-primordial-env
-  (make-the-primordial-env load))
+  (map box (make-the-primordial-env load)))

 (define (cmp f vs) (from-logical (cmp-helper f vs)))
 (define (cmp-helper f vs) : Boolean
@@ -171,68 +172,44 @@
          (vector-set! (as-pair vpr) 1 vel)
          (unit)))]))

-(define (make-env env xs)
-  (begin
-    (when (check-duplicates xs)
-      (error 'smol "can't define a variable twice in one block")
   )
-    (local [(define (allocate x)
-              (pair x (box (none))))]
-      (cons (hash (map allocate xs)) env))))
-(define (env-frame-lookup f x)
-  (hash-ref f x))
-(define (env-lookup-location env x)
-  (type-case Environment env
-    [empty
-     (error 'smol "variable undeclared")]
-    [(cons f fs)
-     (type-case (Optionof '_) (env-frame-lookup f x)
```

```
-       [(none) (env-lookup-location fs x)]
-       [(some loc) loc])]))
+(define (make-env env)
+  (cons (box (hash (list))) env))
 (define (env-lookup env x)
-  (let ([v (env-lookup-location env x)])
-    (type-case (Optionof '_) (unbox v)
-      [(none) (error 'smol "refer to a variable before assign it
    a value")]
-      [(some v) v])))
+  (let ([v (hash-ref (unbox (first env)) x)])
+    (type-case (Optionof (Optionof Value)) v
+      [(none) (env-lookup (rest env) x)]
+      [(some ov)
+       (type-case (Optionof Value) ov
+         [(none) (error 'smol "refer to a variable before assign
    it a value")]
+         [(some v) v])])))
 (define (env-update! env)
-  (lambda (x v)
-    (let ([loc (env-lookup-location env x)])
-      (set-box! loc (some v)))))
-
-(define (declared-Symbols [ts : (Listof Statement)])
-  (local [(define (xs-of-t [t : Statement])
-            : (Listof Symbol)
-            (type-case Statement t
```

```
-               [(expressive e) (list)]
-               [(definitive d)
-                (type-case Definition d
-                  [(Defvar x e) (list x)]
-                  [(Deffun f xs b) (list f)])])]
-    (append-map xs-of-t ts)))
+  (lambda (x mk-v)
+    (let ([f (first env)])
+      (begin
+        (unless (hash-has-key? (unbox f) x)
+          (set-box! f (hash-set (unbox f) x (none))))
+        (set-box! f (hash-set (unbox f) x (some (mk-v)))))))))

 (define (eval-def env d)
   (type-case Definition d
     [(Defvar x e)
-     (let ([v ((eval-exp env) e)])
+     (let ([v (lambda () ((eval-exp env) e))])
        ((env-update! env) x v))]
     [(Deffun f xs b)
-     (let ([v (function xs b env)])
+     (let ([v (lambda () (function xs b env))])
        ((env-update! env) f v))]))

 (define (eval-body env xvs b)
-  (local [(define vs (map snd xvs))
-          (define xs
```

```
-            (append (map fst xvs)
-                    (declared-Symbols (fst b)))))]
-    (let ([env (make-env env xs)])
-      (begin
-        ; bind arguments
-        (for ([xv xvs])
-          ((env-update! env) (fst xv) (snd xv)))
-        ; evaluate starting terms
-        (for ([t (fst b)])
-          ((eval-statement env) t))
-        ; evaluate and return the result
-        ((eval-exp env) (snd b))))))
+  (let ([env (make-env env)])
+    (begin
+      ; bind arguments
+      (for ([xv xvs])
+        ((env-update! env) (fst xv) (lambda () (snd xv))))
+      ; evaluate starting terms
+      (for ([t (fst b)])
+        ((eval-statement env) t))
+      ; evaluate and return the result
+      ((eval-exp env) (snd b)))))

 (define (eval-exp env)
   (lambda (e)
@@ -251,10 +228,9 @@
          (map (eval-exp env) es)
```

```
          ((eval-exp env) e))]
       [(Set! x e)
-       (let ([v ((eval-exp env) e)])
-         (begin
-           ((env-update! env) x v)
-           (unit)))]
+       (begin
+         ((env-update! env) x (lambda () ((eval-exp env) e)))
+         (unit))]
       [(If e_cnd e_thn e_els)
        (let ([v ((eval-exp env) e_cnd)])
          (let ([l (as-logical v)])
@@ -366,6 +342,5 @@
             ((exercute-terms-top-level env) ts))])])))

 (define (evaluate [p : Program])
-  (local [(define xs (declared-Symbols p))
-          (define the-top-level-env (make-env the-primordial-env xs))]
+  (local [(define the-top-level-env (make-env the-primordial-env))]
     ((exercute-terms-top-level the-top-level-env) p)))
```

A.11 The FlatEnv Misinterpreter

```
--- systems/smol-interpreters/Definitional.rkt  2025-02-08 11:00:46
```

```
+++ systems/smol-interpreters/misinterpreters/FlatEnv.rkt
   2025-02-08 11:00:46
@@ -3,6 +3,7 @@
 (require smol-interpreters/syntax)
 (require
   (typed-in racket
+    [hash-has-key? : ((Hashof 'a 'b) 'a -> Boolean)]
     [append-map : (('a -> (Listof 'b)) (Listof 'a) -> (Listof 'b
       ))]
     [hash-values : ((Hashof 'a 'b) -> (Listof 'b))]
     [count : (('a -> Boolean) (Listof 'a) -> Number)]
@@ -94,12 +95,13 @@
      (error 'smol "arity-mismatch, expecting three")]))

 (define-type-alias EnvironmentFrame (Hashof Symbol (Boxof (
   Optionof Value))))
-(define-type-alias Environment (Listof EnvironmentFrame))
+(define-type-alias Environment (Boxof EnvironmentFrame))

+
 (define (load f)
   (box (some (primitive f))))
-(define the-primordial-env
-  (make-the-primordial-env load))
+(define (the-primordial-env)
+  (box (first (make-the-primordial-env load))))
```

```
 (define (cmp f vs) (from-logical (cmp-helper f vs)))
 (define (cmp-helper f vs) : Boolean
@@ -171,23 +173,25 @@
          (vector-set! (as-pair vpr) 1 vel)
          (unit)))]))


+
 (define (make-env env xs)
   (begin
     (when (check-duplicates xs)
       (error 'smol "can't define a variable twice in one block")
          )
-    (local [(define (allocate x)
-              (pair x (box (none))))]
-      (cons (hash (map allocate xs)) env))))
+    (set-box! env
+      (foldl (lambda (x env)
+               (if (hash-has-key? env x)
+                   env
+                   (hash-set env x (box (none)))))
+        (unbox env)
+        xs))
+    env))
 (define (env-frame-lookup f x)
   (hash-ref f x))
 (define (env-lookup-location env x)
-  (type-case Environment env
```

```
-    [empty
-     (error 'smol "variable undeclared")]
-    [(cons f fs)
-     (type-case (Optionof '_) (env-frame-lookup f x)
-       [(none) (env-lookup-location fs x)]
-       [(some loc) loc])]))
+  (type-case (Optionof '_) (env-frame-lookup (unbox env) x)
+    [(none) (error 'smol "variable undeclared")]
+    [(some loc) loc]))
 (define (env-lookup env x)
   (let ([v (env-lookup-location env x)])
     (type-case (Optionof '_) (unbox v)
@@ -367,5 +371,5 @@

 (define (evaluate [p : Program])
   (local [(define xs (declared-Symbols p))
-          (define the-top-level-env (make-env the-primordial-env
-    xs))]
+          (define the-top-level-env (make-env (the-primordial-
+   env) xs))]
     ((exercute-terms-top-level the-top-level-env) p)))
```

A.12 The FunNotVal Misinterpreter

```
--- systems/smol-interpreters/Definitional.rkt  2025-02-08
   11:00:46
+++ systems/smol-interpreters/misinterpreters/FunNotVal.rkt
```

```
   2025-02-08 11:00:46
@@ -32,6 +32,11 @@
   (primitive [o : PrimitiveOperator])
   (function [xs : (Listof Symbol)] [body : Body] [env :
      Environment])
   (vector [vs : (Vectorof Value)]))
+(define (assert-not-fun [v : Value])
+  (type-case Value v
+    [(primitive o) (error 'smol "can't pass functions around")]
+    [(function _xs _body _env) (error 'smol "can't pass
   functions around")]
+    [else v]))

 (define (value-eq? v1 v2)
   (cond
@@ -234,8 +239,14 @@
         ; evaluate and return the result
         ((eval-exp env) (snd b))))))

+(define (eval-fun env)
+  (lambda (e)
+    ((eval-exp-helper env) e)))
 (define (eval-exp env)
   (lambda (e)
+    (assert-not-fun ((eval-exp-helper env) e))))
+(define (eval-exp-helper env)
+  (lambda (e)
```

```
     (type-case Expression e
       [(Con c) (embedded c)]
       [(Var x) (env-lookup env x)]
@@ -276,7 +287,7 @@
                                (loop ebs))))]))]
          (loop ebs))]
       [(App e es)
-        (let ([v ((eval-exp env) e)])
+        (let ([v ((eval-fun env) e)])
          (let ([vs (map (eval-exp env) es)])
            (type-case Value v
              [(function xs b env)
```

A.13 The Lazy Misinterpreter

```
--- systems/smol-interpreters/Definitional.rkt  2025-02-08
   11:00:46
+++ systems/smol-interpreters/misinterpreters/Lazy.rkt
   2025-02-26 14:56:33
@@ -26,14 +26,29 @@
   (TFun)
   (TVec))

+(define-syntax-rule (delay e)
+  (let ([cache (box (none))])
+    (lambda () : Value
+      (type-case (Optionof Value) (unbox cache)
```

```
+          [(some v) v]
+          [(none)
+           (let ([v e])
+             (begin
+               (set-box! cache (some v))
+               v))])))))
+(define (force v) : Value
+  (v))
+
 (define-type Value
    (unit)
    (embedded [c : Constant])
    (primitive [o : PrimitiveOperator])
    (function [xs : (Listof Symbol)] [body : Body] [env :
       Environment])
-   (vector [vs : (Vectorof Value)]))
+   (vector [vs : (Vectorof (-> Value))]))

 (define (value-eq? v1 v2)
+  (value-eq-helper? (force v1) (force v2)))
+(define (value-eq-helper? v1 v2)
    (cond
      [(and (unit? v1) (unit? v2)) #t]
      [(and (embedded? v1) (embedded? v2)) (equal? v1 v2)]
@@ -52,7 +67,7 @@
    (embedded (logical v)))
```

```
 (define (as-numeric v)
-  (type-case Value v
+  (type-case Value (force v)
     [(embedded c)
      (type-case Constant c
        [(numeric n) n]
@@ -63,7 +78,7 @@
   (embedded (numeric n)))

 (define (as-vector v)
-  (type-case Value v
+  (type-case Value (force v)
     [(vector v)
      v]
     [else
@@ -93,11 +108,11 @@
     [else
      (error 'smol "arity-mismatch, expecting three")]))

-(define-type-alias EnvironmentFrame (Hashof Symbol (Boxof (
-  Optionof Value))))
+(define-type-alias EnvironmentFrame (Hashof Symbol (Boxof (
+  Optionof (-> Value)))))
 (define-type-alias Environment (Listof EnvironmentFrame))

 (define (load f)
-  (box (some (primitive f))))
```

```
+  (box (some (delay (primitive f)))))
 (define the-primordial-env
   (make-the-primordial-env load))

@@ -143,7 +158,7 @@
      (local ((define-values (v1 v2) (as-two vs))
              (define vvec (as-vector v1))
              (define vnum (as-numeric v2)))
-       (vector-ref vvec vnum))]
+       (force (vector-ref vvec vnum)))]
     [(VecSet)
      (local ((define-values (v1 v2 v3) (as-three vs))
              (define vvec (as-vector v1))
@@ -156,10 +171,10 @@
        (vector (list->vector vs)))]
     [(PairLft)
      (local ((define v (as-one vs)))
-       (vector-ref (as-pair v) 0))]
+       (force (vector-ref (as-pair v) 0)))]
     [(PairRht)
      (local ((define v (as-one vs)))
-       (vector-ref (as-pair v) 1))]
+       (force (vector-ref (as-pair v) 1)))]
     [(PairSetLft)
      (local ((define-values (vpr vel) (as-two vs)))
        (begin
@@ -212,11 +227,11 @@
```

```
 (define (eval-def env d)
   (type-case Definition d
     [(Defvar x e)
-     (let ([v ((eval-exp env) e)])
+     (let ([v (delay ((eval-exp env) e))])
        ((env-update! env) x v))]
     [(Deffun f xs b)
      (let ([v (function xs b env)])
-       ((env-update! env) f v))]))
+       ((env-update! env) f (delay v)))]))

 (define (eval-body env xvs b)
   (local [(define vs (map snd xvs))
@@ -238,11 +253,11 @@
   (lambda (e)
     (type-case Expression e
       [(Con c) (embedded c)]
-      [(Var x) (env-lookup env x)]
+      [(Var x) (force (env-lookup env x))]
       [(Lambda xs b) (function xs b env)]
       [(Let xes b)
        (local [(define (ev-bind xv)
-                 (let ([v ((eval-exp env) (snd xv))])
+                 (let ([v (delay ((eval-exp env) (snd xv)))])
                    (pair (fst xv) v)))]
          (let ([xvs (map ev-bind xes)])
            (eval-body env xvs b)))]
```

```
@@ -251,7 +266,7 @@
           (map (eval-exp env) es)
           ((eval-exp env) e))]
        [(Set! x e)
-        (let ([v ((eval-exp env) e)])
+        (let ([v (delay ((eval-exp env) e))])
           (begin
             ((env-update! env) x v)
             (unit)))]
@@ -277,7 +292,7 @@
           (loop ebs))]
        [(App e es)
         (let ([v ((eval-exp env) e)])
-          (let ([vs (map (eval-exp env) es)])
+          (let ([vs (map (lambda (e) (delay ((eval-exp env) e))) es)])
             (type-case Value v
               [(function xs b env)
                (if (= (length xs) (length vs))
@@ -328,7 +343,7 @@
                                (list
                                 "#("
                                 (string-join
-                                 (map (value->string visited-vs) (vector->list vs))
+                                 (map (value->string visited-vs) (map force (vector->list vs)))
```

```
                                  " ")
                                 ")")))]
                      [boi (some-v (hash-ref visited-vs vs))]))
```

A.14 The NoCircularity Misinterpreter

```
--- systems/smol-interpreters/Definitional.rkt  2025-02-08
   11:00:46
+++ systems/smol-interpreters/misinterpreters/NoCircularity.rkt
   2025-02-08 11:00:46
@@ -3,6 +3,7 @@
 (require smol-interpreters/syntax)
 (require
   (typed-in racket
+    [ormap : (('a -> Boolean) (Listof 'a) -> Boolean)]
     [append-map : (('a -> (Listof 'b)) (Listof 'a) -> (Listof 'b
        ))]
     [hash-values : ((Hashof 'a 'b) -> (Listof 'b))]
     [count : (('a -> Boolean) (Listof 'a) -> Number)]
@@ -113,6 +114,17 @@
                     (and (f v (first vs))
                          (rec (first vs) (rest vs)))])))
         (rec (first vs) (rest vs))])))
+
+(define (checked-vector-set! the-v i e)
+  (local [(define (occur? x)
+            (type-case Value x
```

```
+               [(vector v)
+                (or (eq? v the-v)
+                    (ormap occur? (vector->list v)))]
+               [else #f]))]
+    (if (occur? e)
+        (error 'smol "vector can't contain itself")
+        (vector-set! the-v i e))))

 (define (delta p vs)
   (type-case PrimitiveOperator p
@@ -149,7 +161,7 @@
              (define vvec (as-vector v1))
              (define vnum (as-numeric v2)))
         (begin
-          (vector-set! vvec vnum v3)
+          (checked-vector-set! vvec vnum v3)
           (unit)))]
     [(PairNew)
      (local ((define-values (v1 v2) (as-two vs)))
@@ -163,12 +175,12 @@
     [(PairSetLft)
      (local ((define-values (vpr vel) (as-two vs)))
        (begin
-         (vector-set! (as-pair vpr) 0 vel)
+         (checked-vector-set! (as-pair vpr) 0 vel)
          (unit)))]
     [(PairSetRht)
```

```
      (local ((define-values (vpr vel) (as-two vs)))
        (begin
-         (vector-set! (as-pair vpr) 1 vel)
+         (checked-vector-set! (as-pair vpr) 1 vel)
          (unit)))]))

 (define (make-env env xs)
```

APPENDIX B

SMOL QUIZZES

This appendix presents the SMoL Quizzes instruments:

1. Appendix B.1 presents the content of the first quiz, smol/fun;
2. Appendix B.2 presents the content of the second quiz, smol/state;
3. Appendix B.3 presents the content of the third quiz, smol/hof;

B.1 The smol/fun Quiz

This quiz keeps ‘arithmetic operators’ and ‘0 as condition’ at the beginning and randomizes the order of all remaining questions.

smol/fun

How to read this file?

```
(a smol/fun program)
```

- **Correct answer**
- **Other answer 1**
- **Other answer 2**

Why the correct answer is correct

arithmetic operators

```
(deffun (f o) (o 1 1))
(f +)
```

- **Error**
- **Syntax error**
- **2**

The correct answer is Error because smol/fun doesn't allow programmers to pass functions as arguments. 2 would have been correct if smol/fun permits higher-order functions, i.e. functions that consume or produce functions, such as f.

0 as condition

```
(if 0 #t #f)
```

- **#t**
- **#f**

In smol/fun, every value other than #f is considered "true". You might find this confusing if you are familiar with Python or C.

redeclare var using defvar

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

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

- **Error**
- **2; 0**
- **0; 0**
- **Nothing is printed**

You can't redeclare x in the same scope level (the global, in this case).

expose local defvar

```
(defvar x 42)
(deffun (create)
  (defvar y 42)
  y)

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

- **Error**
- **42; #t**

The variable y is declared locally. You can't use it outside of the create function.

pair?

```
(pair? (pair 1 2))
(pair? (ivec 1 2))
(pair? '#(1 2))
(pair? '(1 2))
```

- **#t #t #t #f**
- **#t #f #t #f**
- **#t #t #t #t**

In smol, pair is a special-case of ivec. The last vector-like expression is Racket's way of writing list 1, 2.

let* and let

```
(let* ([v 1]
       [w (+ v 2)]
       [y (* w w)])
  (let ([v 3]
        [y (* v w)])
    y))
```

- **3**
- **9**
- **27**

When the inner y is created with (* v w), the v is the outer v.

defvar and let

```
(defvar x 3)
(defvar y (let ([y 6] [x 5]) x))

(* x y)
```

- **15**
- **25**
- **9**

The global y is defined to be equal to the local x, which is 5.

fun-id equals to arg-id

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

- **5**
- **Error**

The parameter f shadows the function name f.

scoping rule of let

```
(let ([x 4]
      [y (+ x 10)])
  y)
```

- **Error**
- **14**

The let expression binds x and y simultaneously, so y cannot see x. If you replace let with let*, the program will produce 14.

the right component of ivec

```
(right (ivec 1 2 3))
```

- **Error**
- **2**

The documentation of `right` says that the input must be of type Pair, and an ivec of size 3 can't be a Pair. If the smol/fun does not check that its parameter is a pair, however, this will return 2.

identifiers

```
(defvar x 5)

(deffun (reassign var_name new_val)
  (defvar var_name new_val)
  (pair var_name x))

(reassign x 6)
x
```

- **'#(6 5) 5**
- **'#(6 6) 5**
- **'#(6 6) 6**
- **Error**
- **Nothing is printed**

The inner defvar declare var_name locally, which shadows the parameter var_name. Neither var_name has anything to do with x, which is defined globally.

defvar, deffun, and let

```
(defvar a 1)
(deffun (what-is-a) a)

(let ([a 2])
  (ivec
    (what-is-a)
    a))
```

- **'#(1 2)**
- **'#(2 2)**

The function what-is-a is defined globally. When it uses the variable a, it looks up in the global scope.

syntax pitfall

```
(deffun (f a b) a + b)
(f 5 10)
```

- **10**
- **15**
- **5**
- **Error**

It is easy to forget smol/fun uses prefix parenthetical syntax. To do the right thing, the deffun should be `(deffun (f a b) (+ a b))`. This program produces 10 because when smol/fun computes the value of `(f 5 10)`, it computes a, and computes +, and finally computes b and returns b's value, which is 10.

B.2 The smol/state Quiz

This quiz randomizes the order of all questions.

smol/state

How to read this file?

```
(a smol program)
```

- **Correct answer**
- **Other answer 1**
- **Other answer 2**

Why the correct answer is correct

circularity

```
(defvar x (mvec 2 3))
(set-right! x x)
(set-left! x x)
x
```

- **x='#(x x) or something similar. Both (left x) and (right x) are x itself.**
- **'#(#(2 #(2 3)) #(2 3))**
- **Error**

set-right! makes x a pair whose left is 2 and whose right is the pair itself. set-left! makes a pair whose both components are x itself.

eval order

```
(defvar x 0)
(ivec x (begin (set! x 1) x) x)
```

- **'#(0 1 1)**
- **'#(0 1 0)**
- **'#(1 1 1)**

When computing the value of `(ivec …)`, we first compute x, which is 0 at that moment, then `(begin …)`, which mutates x to 1 and returns 1, and finally the last x, which is now 1.

mvec as arg

```
(defvar x (mvec 1 2))
(deffun (f x)
  (vset! x 0 0))
(f x)
x
```

- `'#(0 2)`
- **`'#(1 2)`**

f was given the *same* mutable vector. When two mutable values are the same, updates to one are visible in the other.

var as arg

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

- `12`
- **`0`**

The global variable x and the parameter x are different variables. Changing the binding of the parameter x will not change the binding of the global x.

seemingly aliasing a var

```
(defvar x 5)
(deffun (set1 x y)
  (set! x y))
(deffun (set2 a y)
  (set! x y))
(set1 x 6)
x
(set2 x 7)
x
```

- `5; 7`

- **6; 7**
- **5; 5**

Similar to the last question (var as arg), calling the function set1 will not change the global x. The other function (set2), however, is using the global x.

mutable var in vec

```
(defvar x 3)
(defvar v (mvec 1 2 x))
(set! x 4)
v
x
```

- **'#(1 2 3); 4**
- **'#(1 2 4); 4**

The mutable vector stores the *value* of x (i.e. 3) rather than the *binding* (i.e. the information that x is mapped to 3). So the later set! doesn't affect v.

aliasing mvec in mvec

```
(defvar v (mvec 1 2 3 4))
(defvar vv (mvec v v))
(vset! (vref vv 1) 0 100)
vv
```

- **'#(#(100 2 3 4) #(100 2 3 4))**
- **Error**
- **'#(#(1 2 3 4) #(1 2 3 4))**
- **'#(#(1 2 3 4) #(100 2 3 4))**

Both components of vv are identical to v. That is, the left of vv, the right of vv, and v are the same vector.

vset! in let

```
(defvar x (mvec 123))
(let ([y x])
  (vset! y 0 10))
x
```

- `'#(10)`
- **`'#(123)`**

y and x are bound to the same vector.

set! in let

```
(defvar x 123)
(let ([y x])
  (set! y 10))
x
```

- **`123`**
- **10**

y and x are different variables. The set! re-binds y to 10. This won't affect x.

seemingly aliasing a var again

```
(defvar x 10)
(deffun (f y z)
  (set! x z)
  y)
(f x 20)
x
```

- **`10; 20`**
- `20; 20`

At first, x is bound to 10. When f is called, y is bound to the value of x, which is 10, and z is bound to the value of 20, which is 20 itself. Then f re-binds x to the value of z, which is 20. After that f returns the value of y, which is 10. Finally, the program computes the value of x, which is now 20 because f has rebound x.

B.3 The smol/hof Quiz

This quiz organizes questions into question groups. The order of groups is randomized. Questions within the same group are presented in a row but in a randomized order. Questions named as ‘fun and state i/4‘ are in the same group. Questions named as ‘eq? fun fun i/3‘ are in another group. Each one of the remaining questions has its own group.

smol/hof

How to read this file?

```
(a smol program)
```

- **Correct answer**
- **Other answer 1**
- **Other answer 2**

Why the correct answer is correct

fun returns lambda

```
(deffun (f x)
  (lambda (y) (+ x y)))

((f 2) 1)
```

- **3**
- **Error**

The lambda expression is created in the scope of x, so it can use x.

filter gt

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

- **'(1 2)**
- **'(4 5)**

This program keeps numbers that 3 is greater than (not that is greater than 3).

fun and state 1/4

```
(defvar x 1)
(defvar f
  (lambda (y)
    (+ x y)))
(set! x 2)
```

```
(f x)
```

- **4**
- **3**

Every time f is called, it looks up the value of x again.

fun and state 2/4

```
(defvar x 1)
(deffun (f y)
  (+ x y))
(set! x 2)
(f x)
```

- **4**
- **3**

Same as fun and state 1/4

fun and state 3/4

```
(defvar x 1)
(defvar f
  (lambda (y)
     (+ x y)))
(let ([x 2])
  (f x))
```

- **3**
- **4**

The x in the definition of f is the global x, which is a variable different from the x in let.

fun and state 4/4

```
(defvar x 1)
(deffun (f y)
  (+ x y))
(let ([x 2])
  (f x))
```

- **3**
- **4**

Same as fun and state 3/4.

eval order

```
(deffun (f x) (+ x 1))
(deffun (new-f x) (* x x))

(f (begin
     (set! f new-f)
     10))
```

- **11**
- **100**
- **Error**

Function application first computes the value first computes the value of the operator, then the values of operands (i.e. actual parameters) from left to right. So when the set! happened, f had been resolved to its initial value.

counter

```
(deffun (make-counter)
  (let ([count 0])
    (lambda ()
      (begin
        (set! count (+ count 1))
        count))))
(defvar f (make-counter))
(defvar g (make-counter))

(f)
(g)
(f)
(f)
(g)
```

- **1; 1; 2; 3; 2**

- **1; 1; 1; 1; 1**
- **1; 1; 2; 3; 4**

Every time the function make-counter is called, it returns a function that returns 1 when called the first time, 2 the second time, etc. Each (lambda () …) has its own local variable count.

hof + set!

```
(defvar y 3)
(+ ((lambda (x) (set! y 0) (+ x y)) 1)
   y)
```

- **1**
- **7**
- **4**
- **Error**

Because function applications compute their parameters from left to right, set! happened before resolving the last y.

filter

```
(defvar l (list (ivec) (ivec 1) (ivec 2 3)))
(filter (lambda (x) (vlen x)) l)
```

- **'(#() #(1) #(2 3))**
- **‘(0 1 2)**
- **'(#(1) #(2 3))**
- **Error**

Recall that all values other than #f are considered truthy. The filter is effectively creating a copy of l.

eq? fun fun 1/3

```
(eq? (λ (x) (+ x x))
     (λ (x) (+ x x)))
```

- **#f**
- **#t**

Lambda expressions are similar to mvec in the sense that everytime we compute the value of a lambda expression, a new value is created. eq? returns true only when the two values are the same (i.e. identical).

eq? fun fun 2/3

```
(deffun (f x) (+ x x))
(deffun (g x) (+ x x))
(eq? f g)
```

- #f
- **#t**

(deffun (f x) …) can be viewed as (defvar f (lambda (x) …)).

eq? fun fun 3/3

```
(deffun (f x) (+ x x))
(deffun (g) f)
(eq? f (g))
```

- #t
- **#f**

The function value associated with f is computed exactly once when f is defined. (g) is just looking up the value of f.

equal? fun fun

```
(deffun (f) (lambda () 1))
(equal? (f) (f))
```

- #f
- **#t**

Two function values are equal if and only if they are eq. f computes (lambda () …) everytime it is called. So (f) is not equal to another (f).

APPENDIX C

SMOL TUTOR

This chapter presents the learning objectives for each tutorial in the SMoL Tutor (Appendix C.1) and evaluates these objectives as Rules of Program Behavior (RPBs) (Appendix C.2).

C.1 Learning Objectives

Each section lists the learning objectives of the corresponding tutorial. When a tutorial includes multiple learning objectives, they are placed in different subsections.

C.1.1 def1

Introducing "Blocks"

We have two kinds of places where a definition might happen: the top-level **block** and function bodies (which are also **blocks**). A block is a sequence of definitions and expressions.

Blocks form a tree-like structure in a program. For example, we have four blocks in the following program:

```
(defvar n 42)
(deffun (f x)
  (defvar y 1)
  (+ x y))
(deffun (g)
  (deffun (h m)
    (* 2 m))
  (f (h 3)))
(g)
```

The blocks are:

- the top-level block, where the definitions of `n`, `f`, and `g` appear
- the body of `f`, where the definition of `y` appears, which is a sub-block of the top-level block
- the body of `g`, where the definition of `h` appears, which is also a sub-block of the top-level block, and
- the body of `h`, where no local definition appears, which is a sub-block of the body of `g`

Evaluating Undefined Variables

It is an error to evaluate an undefined variable.

C.1.2 def2

Lexical Scope

Variable references follow the hierarchical structure of blocks.

If the variable is defined in the current block, we use that declaration.

Otherwise, we look up the block in which the current block appears, and so on recursively. (Specifically, if the current block is a function body, the next block will be the block in which the function definition is; if the current block is the top-level block, the next block will be the **primordial block**.)

If the current block is already the primordial block and we still haven't found a corresponding declaration, the variable reference errors.

The primordial block is a non-visible block enclosing the top-level block. This block defines values and functions that are provided by the language itself.

Introducing "Scope"

The **scope** of a variable is the region of a program where we can refer to the variable. Technically, it includes the block in which the variable is defined (including sub-blocks) *except* the sub-blocks where the same name is re-defined. When the exception happens, that is, when the same name is defined in a sub-block, we say that the variable in the sub-block **shadows** the variable in the outer block.

We say a variable reference (e.g., "the `x` in `(+ x 1)`") is **in the scope of** a declaration (e.g., "the `x` in `(defvar x 23)`") if and only if the former refers to the latter.

C.1.3 def3

Variables are bound to values. Specifically, every variable definition evaluates the expression immediately and binds the variable to the value, even if the variable is not used later in the program; every function call evaluates the actual parameters immediately and binds the values to formal parameters, even if the formal parameter is not used in the function.

Every block evaluates its definitions and expressions in reading order (i.e., top-to-bottom and left-to-right).

C.1.4 vectors1

(This section defines no learning objects.)

C.1.5 vectors2

vector aliasing

A vector can be referred to by more than one variable and even by other vectors (including itself). Referring to a vector does not create a copy of the vector; rather, they share the same vector. Specifically

1. Binding a vector to a new variable does not create a copy of that vector.
2. Vectors that are passed to a function in a function call do not get copied.
3. Creating new vectors that refer to existing ones does not create new copies of the existing vectors.

The references share the same vector. That is, vectors can be **aliased**.

The heap

In SMoL, each vector has its own unique **heap address** (e.g., `@100` and `@200`). The mapping from addresses to vectors is called the **heap**.

(**Note:** we use `@ddd` (e.g., verb|@123|, `@200`, and `@100`) to represent heap addresses. Heap addresses are *random*. The numbers don't mean anything.)

The heap and (variable) bindings

Creating a vector does not inherently create a binding.

Creating a binding does not necessarily alter the heap.

C.1.6 vectors3

(This section defines no learning objects.)

C.1.7 mutvars1

Variable assignments change *only* the mutated variables. That is, variables are not aliased.

(**Note:** some programming languages (e.g., C++ and Rust) allow variables to be aliased. However, even in those languages, variables are not aliased by default.)

C.1.8 mutvars2

Mutable variables

Variable assignments mutate existing bindings and do not create new bindings.

Functions refer to the latest values of variables defined outside their definitions. That is, functions do not remember the values of those variables from when the functions were defined.

Environments

Environments (rather than blocks) bind variables to values.

Similar to vectors, environments are created as programs run.

Environments are created from blocks. They form a tree-like structure, respecting the tree-like structure of their corresponding blocks. So, we have a primordial environment, a top-level environment, and environments created from function bodies.

Every function call creates a new environment. This is very different from the block perspective: every function corresponds to exactly one block, its body.

C.1.9 lambda1

Functions are (also) *first-class* citizens of the value world. Specifically,

- Variables (notably parameters) can be bound to functions,
- Functions can return functions, and
- Vectors can refer to functions.

C.1.10 lambda2

Functions remember the environment in which they are defined. That is, function bodies are "enclosed" by the environments in which the function values are created. So, function values are called **closures**.

C.1.11 lambda3

Introducing Lambda Expressions

Lambda expressions are expressions that create functions.

The following program illustrates how to create a function without giving it a name and then call it immediately.

```
((lambda (n)
   (+ n 1))
 2)
```

The function is created by a lambda expression. This function increases it parameter by one.

```
(lambda (n)
  (+ n 1))
```

Lambdas

```
(deffun (f x y z) body)
```

is a "shorthand" for

```
(defvar f (lambda (x y z) body))
```

C.2 Evaluating the Learning Objectives as Rules of Program Behavior

This section evaluates the SMoL Tutor's learning objectives (Appendix C) and SMoL Characteristics against the 16 evaluation criteria proposed by Duran, Sorva, and Seppälä [21].

Because the learning objectives are essentially a refined version of the SMoL Characteristics (Section 1.1), the discussion applies to both unless stated otherwise.

Cultural Fit An unusual aspect of my rules is that they are not designed for learners with little or no programming background. Instead, they assume users already possess basic programming knowledge, including familiarity with concepts like vectors, mutation, and higher-order functions.

Accuracy Because my rules are designed to be multilingual, they cannot be 100% accurate due to variations across languages. However, each rule is correct for many languages. Some learning objectives intentionally sacrifice accuracy for pedagogical simplicity. For example, in the DEF tutorials, recursive lexical scope lookup is explained directly in terms of blocks, rather than environments. Later, in the MUTVARS module, environments are introduced, and the lexical scope rules are revisited. Without mutation, introducing environments earlier would be unjustified, making this staged introduction necessary.

Coverage My rules cover mutable variables, mutable vectors, first-class functions, and their interactions.

Simplicity The notion of simplicity in [21] concerns the number of concepts and the complexity of their dependencies. Evaluating SMoL Characteristics by this standard is less meaningful: they are simple but perhaps overly so for detailed reasoning. The learning objectives, however, are explicitly designed with simplicity in mind. Early tutorials (DEF) intentionally omit mutation; concepts like the heap and heap addresses are introduced only when needed (e.g., after introducing mutable vectors), and environments are introduced only after mutable variables are covered.

Consistency The consistency of my rules is compromised whenever a documented language behavior diverges from the rules. (If undocumented, it instead affects accuracy.) My rules explicitly note assumptions such as eager and sequential evaluation within a single thread, and caution that aliasing behavior differs between mutable data structures and mutable variables.

Granularity The learning objectives are intended to be fine-grained enough for learners to mechanically evaluate any program covered by SMoL. The SMoL Characteristics are somewhat coarser.

Abstraction The learning objectives are abstract enough to avoid unnecessary implementation details. Similarly, the SMoL Characteristics are abstract enough to describe behaviors common across many modern programming languages.

Expressibility Duran, Sorva, and Seppälä [21] defines expressibility as "how feasible it is to construct a student-facing form of the RPBs suitable for the target audience". By this measure, the learning objectives are highly expressible: the SMoL Tutor presents them

directly to students. Although I believe this presentation is suitable, I acknowledge that evaluating this point objectively is difficult.

Notational Fit Following [21], notational fit refers to how well the rules are phrased in terms of syntactic constructs. Explaining scope in terms of blocks achieves high notational fit but compromises **accuracy**, as discussed earlier. Conversely, introducing heaps and heap addresses reduces notational fit but is necessary to explain the behavior of mutable data structures correctly.

Generality The rules are designed to be highly general, remaining largely consistent across a wide range of programming languages.

Transferability This criterion seems intended for RPBs that target a single language, and is likely not applicable to my rules.

Sensitivity to Conceptions Duran, Sorva, and Seppälä [21] describe this criterion as addressing how RPBs account for learners' prior knowledge, common misconceptions, and difficult content. My rules assume basic programming experience but no system-level knowledge. Although the rules themselves were not designed specifically around misconceptions, the development of refutation texts consistently drew contrasts between misconceptions and the learning objectives. The rules also attend to difficult concepts by sequencing topics carefully: first-class functions, a difficult topic, are introduced only after students have a firmer grasp of earlier material.

Other Criteria It is unclear how to evaluate my rules against the criteria of **Wieldiness**, **Assessability**, **Implementability**, and **Clarity of Writing**.

BIBLIOGRAPHY

[1] Kyriel Abad and Martin Henz. *Beyond SICP – Design and Implementation of a Notional Machine for Scheme*. Dec. 2024. DOI: 10.48550/arXiv.2412.01545. arXiv: 2412.01545 [cs]. (Visited on 04/01/2025).

[2] Harold Abelson and Gerald Jay Sussman. *Structure and Interpretation of Computer Programs*. The MIT Press, 1996. URL: https://library.oapen.org/handle/20.500.12657/26092 (visited on 04/02/2025).

[3] Louis Alfieri, Timothy J. Nokes-Malach, and Christian D. Schunn. "Learning Through Case Comparisons: A Meta-Analytic Review". In: *Educational Psychologist* 48.2 (Apr. 2013), pp. 87–113. ISSN: 0046-1520, 1532-6985. DOI: 10.1080/00461520.2013.775712. (Visited on 10/05/2023).

[4] John R. Anderson and Brian J. Reiser. "The LISP Tutor". In: *Byte* 10.4 (1985), pp. 159–175. URL: http://act-r.psy.cmu.edu/wordpress/wp-content/uploads/2012/12/113TheLISPTutor.pdf (visited on 10/10/2023).

[5] Mordechai Ben-Ari, Roman Bednarik, Ronit Ben-Bassat Levy, Gil Ebel, Andrés Moreno, Niko Myller, and Erkki Sutinen. "A Decade of Research and Development on Program Animation: The Jeliot Experience". In: *Journal of Visual Languages & Computing*

22.5 (Oct. 2011), pp. 375–384. ISSN: 1045-926X. DOI: 10.1016/j.jvlc.2011.04.004. (Visited on 04/14/2025).

[6] George EP Box. "Robustness in the Strategy of Scientific Model Building". In: *Robustness in Statistics*. Elsevier, 1979, pp. 201–236. URL: https://www.sciencedirect.com/science/article/pii/B9780124381506500182 (visited on 04/30/2025).

[7] Kaian Cai, Martin Henz, Kok-Lim Low, Xing Yu Ng, Jing Ren Soh, Kyn-Han Tang, and Kar Wi Toh. "Visualizing Environments of Modern Scripting Languages." In: *CSEDU (1)*. 2023, pp. 146–153. URL: https://www.scitepress.org/Papers/2023/117667/117667.pdf (visited on 04/01/2025).

[8] Nicholas J. Cepeda, Edward Vul, Doug Rohrer, John T. Wixted, and Harold Pashler. "Spacing Effects in Learning: A Temporal Ridgeline of Optimal Retention". In: *Psychological Science* 19.11 (Nov. 2008), pp. 1095–1102. ISSN: 0956-7976, 1467-9280. DOI: 10.1111/j.1467-9280.2008.02209.x. (Visited on 04/02/2025).

[9] Kartik Chandra, Katherine M. Collins, Will Crichton, Tony Chen, Tzu-Mao Li, Adrian Weller, Rachit Nigam, Joshua Tenenbaum, and Jonathan Ragan-Kelley. *WatChat: Explaining Perplexing Programs by Debugging Mental Models*. Oct. 2024. DOI: 10.48550/arXiv.2403.05334. arXiv: 2403.05334 [cs]. (Visited on 04/03/2025).

[10] Luca Chiodini, Igor Moreno Santos, Andrea Gallidabino, Anya Tafliovich, André L. Santos, and Matthias Hauswirth. "A Curated Inventory of Programming Language Misconceptions". In: *Proceedings of the 26th ACM Conference on Innovation and Technology in Computer Science Education V. 1*. ITiCSE '21. New York, NY, USA: Association for Computing Machinery, June 2021, pp. 380–386. ISBN: 978-1-4503-8214-4. DOI: 10.1145/3430665.3456343. (Visited on 03/20/2025).

[11] John Clements, Matthew Flatt, and Matthias Felleisen. "Modeling an Algebraic Stepper". In: *Programming Languages and Systems*. Ed. by Gerhard Goos, Juris Hartmanis, Jan Van Leeuwen, and David Sands. Vol. 2028. Berlin, Heidelberg: Springer

Berlin Heidelberg, 2001, pp. 320–334. ISBN: 978-3-540-41862-7 978-3-540-45309-3. DOI: 10.1007/3-540-45309-1_21. (Visited on 04/11/2025).

[12] John Clements and Shriram Krishnamurthi. "Towards a Notional Machine for Runtime Stacks and Scope: When Stacks Don't Stack Up". In: *Proceedings of the 2022 ACM Conference on International Computing Education Research - Volume 1*. Vol. 1. ICER '22. Lugano and Virtual Event Switzerland: ACM, Aug. 2022, pp. 206–222. ISBN: 978-1-4503-9194-8. DOI: 10.1145/3501385.3543961. (Visited on 01/30/2025).

[13] *CodeMirror*. URL: http://codemirror.net/ (visited on 02/18/2025).

[14] *Color Wheel, a Color Palette Generator | Adobe Color*. URL: https://color.adobe.com/create/color-wheel (visited on 02/18/2025).

[15] *Contenteditable - HTML: HyperText Markup Language | MDN*. Apr. 2025. URL: https://developer.mozilla.org/en-US/docs/Web/HTML/Reference/Global_attributes/contenteditable (visited on 04/25/2025).

[16] Will Crichton, Maneesh Agrawala, and Pat Hanrahan. "The Role of Working Memory in Program Tracing". In: *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*. 2021, pp. 1–13. (Visited on 04/30/2025).

[17] Andrea A. diSessa. "A Friendly Introduction to "Knowledge in Pieces": Modeling Types of Knowledge and Their Roles in Learning". In: *Invited Lectures from the 13th International Congress on Mathematical Education*. Ed. by Gabriele Kaiser, Helen Forgasz, Mellony Graven, Alain Kuzniak, Elaine Simmt, and Binyan Xu. Cham: Springer International Publishing, 2018, pp. 65–84. ISBN: 978-3-319-72169-9 978-3-319-72170-5. DOI: 10.1007/978-3-319-72170-5_5. (Visited on 04/05/2025).

[18] Amer Diwan, William M. Waite, Michele H. Jackson, and Jacob Dickerson. "PL-detective: A System for Teaching Programming Language Concepts". In: *Journal on Educational Resources in Computing* 4.4 (Dec. 2004), 1–es. ISSN: 1531-4278. DOI: 10.1145/1086339.1086340. (Visited on 04/24/2023).

[19] Benedict Du Boulay. “Some Difficulties of Learning to Program”. In: *Journal of Educational Computing Research* 2.1 (Feb. 1986), pp. 57–73. ISSN: 0735-6331. DOI: 10.2190/3LFX-9RRF-67T8-UVK9. (Visited on 01/30/2025).

[20] Benedict Du Boulay, Tim O’Shea, and John Monk. “The Black Box inside the Glass Box: Presenting Computing Concepts to Novices”. In: *International Journal of man-machine studies* 14.3 (1981), pp. 237–249. URL: https://www.sciencedirect.com/science/article/pii/S0020737381800569 (visited on 01/30/2025).

[21] Rodrigo Duran, Juha Sorva, and Otto Seppälä. “Rules of Program Behavior”. In: *ACM Transactions on Computing Education* 21.4 (Nov. 2021), 33:1–33:37. DOI: 10.1145/3469128. (Visited on 03/23/2023).

[22] Hermann Ebbinghaus. “Memory: A Contribution to Experimental Psychology”. In: *Annals of neurosciences* 20.4 (2013), p. 155. URL: https://pmc.ncbi.nlm.nih.gov/articles/PMC4117135/ (visited on 04/02/2025).

[23] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. *How to Design Programs: An Introduction to Programming and Computing*. Mit Press, 2018. (Visited on 04/17/2025).

[24] Matthias Felleisen and Robert Hieb. “The Revised Report on the Syntactic Theories of Sequential Control and State”. In: *Theoretical Computer Science* 103.2 (1992), pp. 235–271. ISSN: 03043975. DOI: 10.1016/0304-3975(92)90014-7. (Visited on 04/23/2025).

[25] Lars Fischer and Stefan Hanenberg. “An Empirical Investigation of the Effects of Type Systems and Code Completion on API Usability Using TypeScript and JavaScript in MS Visual Studio”. In: *Proceedings of the 11th Symposium on Dynamic Languages*. DLS 2015. New York, NY, USA: Association for Computing Machinery, Oct. 2015, pp. 154–167. ISBN: 978-1-4503-3690-1. DOI: 10.1145/2816707.2816720. (Visited on 04/29/2025).

[26] 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. New York, NY, USA: Association for Computing Machinery, 2017, pp. 213–218. ISBN: 978-1-4503-4698-6. DOI: 10.1145/3017680.3017777.

[27] Ann E. Fleury. "Parameter Passing: The Rules the Students Construct". In: *ACM SIGCSE Bulletin* 23.1 (1991), pp. 283–286. ISSN: 0097-8418. DOI: 10.1145/107005.107066.

[28] FMc. *Why Can a Function Modify Some Arguments as Perceived by the Caller, but Not Others?* Forum Post. Sept. 2023. URL: https://stackoverflow.com/q/575196 (visited on 04/28/2025).

[29] Judith Gal-Ezer and Ela Zur. "The Efficiency of Algorithms—Misconceptions". In: *Computers & Education* 42.3 (Apr. 2004), pp. 215–226. ISSN: 0360-1315. DOI: 10.1016/j.compedu.2003.07.004. (Visited on 04/30/2025).

[30] Carlisle E. George. "EROSI—Visualising Recursion and Discovering New Errors". In: *ACM SIGCSE Bulletin* 32.1 (Mar. 2000), pp. 305–309. ISSN: 0097-8418. DOI: 10.1145/331795.331875. (Visited on 04/22/2025).

[31] Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. "The Essence of JavaScript". In: *Proceedings of the 24th European Conference on Object-oriented Programming*. ECOOP'10. Berlin, Heidelberg: Springer-Verlag, June 2010, pp. 126–150. ISBN: 978-3-642-14106-5. (Visited on 10/15/2023).

[32] Philip Guo. "Ten Million Users and Ten Years Later: Python Tutor's Design Guidelines for Building Scalable and Sustainable Research Software in Academia". In: *The 34th Annual ACM Symposium on User Interface Software and Technology*. UIST '21. New York, NY, USA: Association for Computing Machinery, Oct. 2021, pp. 1235–1251. ISBN: 978-1-4503-8635-7. DOI: 10.1145/3472749.3474819. (Visited on 04/24/2023).

[33] 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. New York, NY, USA: Association for Computing Machinery, Mar. 2013, pp. 579–584. ISBN: 978-1-4503-1868-6. DOI: 10.1145/2445196.2445368. (Visited on 03/24/2023).

[34] David Hammer. "Misconceptions or P-Prims: How May Alternative Perspectives of Cognitive Structure Influence Instructional Perceptions and Intentions?" In: *The Journal of the Learning Sciences* 5.2 (1996), pp. 97–127. JSTOR: 1466772. URL: http://www.jstor.org/stable/1466772.

[35] Robert Harper. *Practical Foundations for Programming Languages*. Cambridge University Press, 2016. (Visited on 01/31/2025).

[36] David Hestenes, Malcolm Wells, and Gregg Swackhamer. "Force Concept Inventory". In: *The Physics Teacher* 30.3 (Mar. 1992), pp. 141–158. ISSN: 0031-921X, 1943-4928. DOI: 10.1119/1.2343497. (Visited on 10/10/2023).

[37] Cruz Izu and Claudio Mirolo. "Comparing Small Programs for Equivalence: A Code Comprehension Task for Novice Programmers". In: *Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education*. Trondheim Norway: ACM, June 2020, pp. 466–472. ISBN: 978-1-4503-6874-2. DOI: 10.1145/3341525.3387425. (Visited on 04/30/2025).

[38] Lisa C. Kaczmarczyk, Elizabeth R. Petrick, J. Philip East, and Geoffrey L. Herman. "Identifying Student Misconceptions of Programming". In: *Proceedings of the 41st ACM Technical Symposium on Computer Science Education*. Milwaukee Wisconsin USA: ACM, Mar. 2010, pp. 107–111. ISBN: 978-1-4503-0006-3. DOI: 10.1145/1734263.1734299. (Visited on 01/31/2025).

[39] Michael N. Katehakis and Arthur F. Veinott. "The Multi-Armed Bandit Problem: Decomposition and Computation". In: *Mathematics of Operations Research* 12.2 (May

1987), pp. 262–268. ISSN: 0364-765X, 1526-5471. DOI: 10.1287/moor.12.2.262. (Visited on 10/10/2023).

[40] Shriram Krishnamurthi, Anika Bahl, Benjamin Lee, and Steven Sloman. "Problematic and Persistent Post-Secondary Program Performance Preconceptions". In: *Proceedings of the 22nd Koli Calling International Conference on Computing Education Research*. Koli Calling '22. New York, NY, USA: Association for Computing Machinery, Nov. 2022, pp. 1–6. ISBN: 978-1-4503-9616-5. DOI: 10.1145/3564721.3564722. (Visited on 04/30/2025).

[41] Shriram Krishnamurthi and Kathi Fisler. "13 Programming Paradigms and Beyond". In: *The Cambridge handbook of computing education research* (2019). URL: https://books.google.com/books?hl=en&lr=&id=vmAwEAAAQBAJ&oi=fnd&pg=PA377&dq=Programming+paradigms+and+beyond&ots=1muxLT8WbI&sig=kOuflHESzpWeEa9w0LxLsL52r8c (visited on 02/27/2025).

[42] *Lambda Expressions (The Java™ Tutorials > Learning the Java Language > Classes and Objects)*. URL: https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html (visited on 01/23/2025).

[43] Kuang-Chen Lu, Shriram Krishnamurthi, Kathi Fisler, and Ethel Tshukudu. "What Happens When Students Switch (Functional) Languages (Experience Report)". In: *Proceedings of the ACM on Programming Languages* 7.ICFP (Aug. 2023), pp. 796–812. ISSN: 2475-1421. DOI: 10.1145/3607857. (Visited on 04/12/2025).

[44] Linxiao Ma. "Investigating and Improving Novice Programmers' Mental Models of Programming Concepts". PhD thesis. Citeseer, 2007. URL: https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=3c8efb0c95325ac2f6bb38bd3d56fdbe900e4892 (visited on 04/22/2025).

[45] Andrés Moreno, Niko Myller, Erkki Sutinen, and Mordechai Ben-Ari. "Visualizing Programs with Jeliot 3". In: *Proceedings of the Working Conference on Advanced Visual*

Interfaces. Gallipoli Italy: ACM, May 2004, pp. 373–376. ISBN: 978-1-58113-867-2. DOI: 10.1145/989863.989928. (Visited on 04/02/2025).

[46] Brad A. Myers, John F. Pane, and Amy J. Ko. "Natural Programming Languages and Environments". In: *Communications of the ACM* 47.9 (Sept. 2004), pp. 47–52. ISSN: 0001-0782, 1557-7317. DOI: 10.1145/1015864.1015888. (Visited on 03/21/2025).

[47] 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*. Vol. 644648. 2001, pp. 644–648. (Visited on 10/10/2023).

[48] *Natural Programming*. URL: https://www.cs.cmu.edu/~NatProg/ (visited on 04/29/2025).

[49] Eduardo Oliveira, Hieke Keuning, and Johan Jeuring. "Student Code Refactoring Misconceptions". In: *Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V. 1*. Turku Finland: ACM, June 2023, pp. 19–25. ISBN: 9798400701382. DOI: 10.1145/3587102.3588840. (Visited on 04/22/2025).

[50] John F. Pane and Brad A. Myers. "More Natural Programming Languages and Environments". In: *End User Development*. Ed. by John Karat, Jean Vanderdonckt, Henry Lieberman, Fabio Paternò, and Volker Wulf. Vol. 9. Dordrecht: Springer Netherlands, 2006, pp. 31–50. ISBN: 978-1-4020-4220-1 978-1-4020-5386-3. DOI: 10.1007/1-4020-5386-X_3. (Visited on 04/29/2025).

[51] Roy D. Pea. "Language-Independent Conceptual "Bugs" in Novice Programming". In: *Journal of Educational Computing Research* 2.1 (Feb. 1986), pp. 25–36. ISSN: 0735-6331, 1541-4140. DOI: 10.2190/689T-1R2A-X4W4-29J2. (Visited on 04/22/2025).

[52] Pujan Petersen, Stefan Hanenberg, and Romain Robbes. "An Empirical Comparison of Static and Dynamic Type Systems on API Usage in the Presence of an IDE: Java vs. Groovy with Eclipse". In: *Proceedings of the 22nd International Conference on*

Program Comprehension. ICPC 2014. New York, NY, USA: Association for Computing Machinery, June 2014, pp. 212–222. ISBN: 978-1-4503-2879-1. DOI: 10.1145/2597008.2597152. (Visited on 04/29/2025).

[53] *Plait Language*. URL: https://docs.racket-lang.org/plait/ (visited on 04/25/2025).

[54] 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. New York, NY, USA: Association for Computing Machinery, Oct. 2012, pp. 1–16. ISBN: 978-1-4503-1564-7. DOI: 10.1145/2384577.2384579. (Visited on 01/23/2025).

[55] Joe Gibbs Politz, Alejandro Martinez, Mae Milano, Sumner Warren, Daniel Patterson, Junsong Li, Anand Chitipothu, and Shriram Krishnamurthi. "Python: The Full Monty". In: *SIGPLAN Not.* 48.10 (Oct. 2013), pp. 217–232. ISSN: 0362-1340. DOI: 10.1145/2544173.2509536. (Visited on 01/23/2025).

[56] 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/ (visited on 10/14/2023).

[57] George J. Posner, Kenneth A. Strike, Peter W. Hewson, and William A. Gertzog. "Toward a Theory of Conceptual Change". In: *Science education* 66.2 (1982), pp. 211–227. (Visited on 10/10/2023).

[58] Michael Prince. "Does Active Learning Work? A Review of the Research". In: *Journal of Engineering Education* 93.3 (2004), pp. 223–231. ISSN: 2168-9830. DOI: 10.1002/j.2168-9830.2004.tb00809.x. (Visited on 02/20/2025).

[59] Helen Purchase. "Which Aesthetic Has the Greatest Effect on Human Understanding?" In: *Graph Drawing*. Ed. by Gerhard Goos, Juris Hartmanis, Jan Van Leeuwen, and

Giuseppe DiBattista. Vol. 1353. Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, pp. 248–261. ISBN: 978-3-540-63938-1 978-3-540-69674-2. DOI: 10.1007/3-540-63938-1_67. (Visited on 04/15/2025).

[60] Ralph T. Putnam, D. Sleeman, Juliet A. Baxter, and Laiani K. Kuspa. "A Summary of Misconceptions of High School Basic Programmers". In: *Journal of Educational Computing Research* 2.4 (Nov. 1986), pp. 459–472. ISSN: 0735-6331, 1541-4140. DOI: 10.2190/FGN9-DJ2F-86V8-3FAU. (Visited on 04/24/2025).

[61] Timothy Rafalski, P. Merlin Uesbeck, Cristina Panks-Meloney, Patrick Daleiden, William Allee, Amelia Mcnamara, and Andreas Stefik. "A Randomized Controlled Trial on the Wild Wild West of Scientific Computing with Student Learners". In: *Proceedings of the 2019 ACM Conference on International Computing Education Research*. ICER '19. New York, NY, USA: Association for Computing Machinery, July 2019, pp. 239–247. ISBN: 978-1-4503-6185-9. DOI: 10.1145/3291279.3339421. (Visited on 04/29/2025).

[62] Noa Ragonis and Mordechai Ben-Ari. "A Long-Term Investigation of the Comprehension of OOP Concepts by Novices". In: *Computer Science Education* 15.3 (Sept. 2005), pp. 203–221. ISSN: 0899-3408, 1744-5175. DOI: 10.1080/08993400500224310. (Visited on 04/22/2025).

[63] Teemu Rajala, Mikko-Jussi Laakso, Erkki Kaila, and Tapio Salakoski. "VILLE: A Language-Independent Program Visualization Tool". In: *Proceedings of the Seventh Baltic Sea Conference on Computing Education Research-Volume 88*. Citeseer, 2007, pp. 151–159. URL: https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=e16256a156b54db655641080f8f8e56b3501ffc8 (visited on 04/02/2025).

[64] Anthony Robins, Janet Rountree, and Nathan Rountree. "Learning and Teaching Programming: A Review and Discussion". In: *Computer Science Education* 13.2 (June 2003), pp. 137–172. ISSN: 0899-3408, 1744-5175. DOI: 10.1076/csed.13.2.137.14200. (Visited on 01/30/2025).

[65] Henry L. Roediger and Jeffrey D. Karpicke. "Test-Enhanced Learning: Taking Memory Tests Improves Long-Term Retention". In: *Psychological Science* 17.3 (Mar. 2006), pp. 249–255. ISSN: 0956-7976, 1467-9280. DOI: 10.1111/j.1467-9280.2006.01693.x. (Visited on 04/02/2025).

[66] 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*. Minneapolis MN USA: ACM, 2019, pp. 606–612. ISBN: 978-1-4503-5890-3. DOI: 10.1145/3287324.3287504.

[67] Noah L. Schroeder and Aurelia C. Kucera. "Refutation Text Facilitates Learning: A Meta-Analysis of Between-Subjects Experiments". In: *Educational Psychology Review* 34.2 (June 2022), pp. 957–987. ISSN: 1040-726X, 1573-336X. DOI: 10.1007/s10648-021-09656-z. (Visited on 10/10/2023).

[68] Daniel L. Schwartz, Catherine C. Chase, Marily A. Oppezzo, and Doris B. Chin. "Practicing versus Inventing with Contrasting Cases: The Effects of Telling First on Learning and Transfer." In: *Journal of educational psychology* 103.4 (2011), p. 759. URL: https://psycnet.apa.org/fulltext/2011-18448-001.html (visited on 04/29/2025).

[69] Bruce L. Sherin, Moshe Krakowski, and Victor R. Lee. "Some Assembly Required: How Scientific Explanations Are Constructed during Clinical Interviews". In: *Journal of Research in Science Teaching* 49.2 (2012), pp. 166–198. ISSN: 1098-2736. DOI: 10.1002/tea.20455. (Visited on 03/20/2025).

[70] Shriram Krishnamurthi. *Programming Languages: Application and Interpretation*. 3rd. URL: https://www.plai.org/ (visited on 02/14/2025).

[71] Teemu Sirkiä. "Jsvee & Kelmu: Creating and Tailoring Program Animations for Computing Education". In: *Journal of Software: Evolution and Process* 30.2 (Feb. 2018), e1924. ISSN: 2047-7473, 2047-7481. DOI: 10.1002/smr.1924. (Visited on 04/02/2025).

[72] D. Sleeman, Ralph T. Putnam, Juliet Baxter, and Laiani Kuspa. "Pascal and High School Students: A Study of Errors". In: *Journal of Educational Computing Research* 2.1 (Feb. 1986), pp. 5–23. ISSN: 0735-6331, 1541-4140. DOI: 10.2190/2XPP-LTYH-98NQ-BU77. (Visited on 04/24/2025).

[73] Juha Sorva. "Notional Machines and Introductory Programming Education". In: *ACM Trans. Comput. Educ.* 13.2 (July 2013), 8:1–8:31. DOI: 10.1145/2483710.2483713. (Visited on 01/30/2025).

[74] Juha Sorva. "Visual Program Simulation in Introductory Programming Education". PhD thesis. Aalto University, 2012. URL: https://aaltodoc.aalto.fi/handle/123456789/3534.

[75] Juha Sorva and Teemu Sirkiä. "UUhistle: A Software Tool for Visual Program Simulation". In: *Proceedings of the 10th Koli Calling International Conference on Computing Education Research*. Koli Finland: ACM, Oct. 2010, pp. 49–54. ISBN: 978-1-4503-0520-4. DOI: 10.1145/1930464.1930471. (Visited on 04/02/2025).

[76] 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. New York, NY, USA: Association for Computing Machinery, Jan. 2023, pp. 39–48. ISBN: 978-1-4503-9941-8. DOI: 10.1145/3576123.3576128.

[77] C. Taylor, D. Zingaro, L. Porter, K.C. Webb, C.B. Lee, and M. Clancy. "Computer Science Concept Inventories: Past and Future". In: *Computer Science Education* 24.4 (Oct. 2014), pp. 253–276. ISSN: 0899-3408. DOI: 10.1080/08993408.2014.970779. (Visited on 04/10/2023).

[78] Allison Elliott Tew and Mark Guzdial. "The FCS1: A Language Independent Assessment of CS1 Knowledge". In: *Proceedings of the 42nd ACM Technical Symposium on*

Computer Science Education. Dallas TX USA: ACM, Mar. 2011, pp. 111–116. ISBN: 978-1-4503-0500-6. DOI: 10.1145/1953163.1953200. (Visited on 02/14/2025).

[79] Christine D. Tippett. "REFUTATION TEXT IN SCIENCE EDUCATION: A REVIEW OF TWO DECADES OF RESEARCH". In: *International Journal of Science and Mathematics Education* 8.6 (Dec. 2010), pp. 951–970. ISSN: 1571-0068, 1573-1774. DOI: 10.1007/s10763-010-9203-x. (Visited on 04/22/2025).

[80] Preston Tunnell Wilson, Kathi Fisler, and Shriram Krishnamurthi. "Student Understanding of Aliasing and Procedure Calls". In: *SPLASH Education Symposium*. 2017. URL: https://par.nsf.gov/servlets/purl/10067510 (visited on 04/12/2025).

[81] Preston Tunnell Wilson, Justin Pombrio, and Shriram Krishnamurthi. "Can We Crowdsource Language Design?" In: *Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. Vancouver BC Canada: ACM, Oct. 2017, pp. 1–17. ISBN: 978-1-4503-5530-8. DOI: 10.1145/3133850.3133863. (Visited on 04/08/2025).

[82] Phillip Merlin Uesbeck and Andreas Stefik. "A Randomized Controlled Trial on the Impact of Polyglot Programming in a Database Context". In: *OASIcs, Volume 67, PLATEAU 2018* 67 (2019). Ed. by Titus Barik, Joshua Sunshine, and Sarah Chasins, 1:1–1:8. ISSN: 2190-6807. DOI: 10.4230/OASICS.PLATEAU.2018.1. (Visited on 04/29/2025).

[83] Vesa Vainio. "Opiskelijoiden Mentaaliset Mallit Ohjelmien Suorituksesta Ohjelmoinnin Peruskurssilla". In: *Master's Thesis. University of Helsinki, Helsinki, Finland* (2006). URL: http://www.vesavainio.fi/pro/Pro_Gradu_Vesa_Vainio.pdf (visited on 04/24/2025).

[84] Vallabha. *Modifying a Copy of a JavaScript Object Is Causing the Original Object to Change*. Forum Post. Mar. 2021. URL: https://stackoverflow.com/q/29050004 (visited on 04/28/2025).

[85] Kurt VanLehn. "The Behavior of Tutoring Systems". In: *International Journal of Artificial Intelligence in Education* 16.3 (Jan. 2006), pp. 227–265. ISSN: 1560-4292. URL: https://content.iospress.com/articles/international-journal-of-artificial-intelligence-in-education/jai16-3-02.

[86] Veera. *How Do I Copy an Object in Java?* Forum Post. Dec. 2017. URL: https://stackoverflow.com/q/869033 (visited on 04/28/2025).

[87] *Wat*. URL: https://www.destroyallsoftware.com/talks/wat (visited on 01/23/2025).

[88] *WebAIM: Contrast Checker*. URL: https://webaim.org/resources/contrastchecker/ (visited on 02/18/2025).